# Graphics Paper

Asier Bilbao

December 2019

## 1 Abstract

This paper is explains a method to render fur using the techniques detailed on Fur (using Shells and Fins) by Sarah Tariq. It takes advantage of the Geometry Shader and 2D Texture Array to fulfill its job.

Model with Shells and Fins

## 2 Motivation

A lot of games tend to have 3D models of furry animals. One of the problems is how to render that fur. Normally, applying a texture may look good enough. But if doesn't, there a lot of ways of rendering fur to give it a more realistic look. One of them being the one introduced by this paper: Fur (using Shells and Fins)

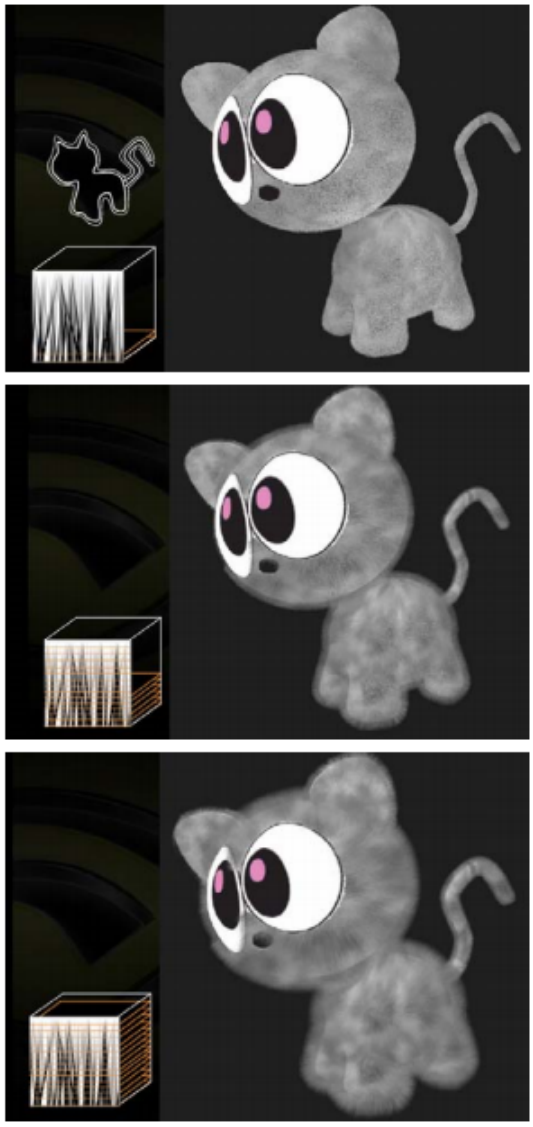
## 3 Implementation

This technique, as the name says, it's actually a mix of both: Shells and Fins. Note that all variables that have the same name are exactly the same, both for Shells and Fins. For areas that have no fur, like eyes or paws,the texture should have the alpha channel set to zero, to later on be discarded on the fragment shaders. First, I'll start with Shells.

### 3.1 Shells

This method consists on rendering the mesh multiples times, each time extruding along the normal by an amount proportional to the index of the shell.

Each shell is textured with a texture taken from an array of textures. While we are rendering these shells, alpha blending and depth writing are disabled.
In the next pages you have a visual representation of what I have explained and some code snippets for the shaders:




Shell rendering[1]

[1]From "Fur (using Shells and Fins)"

```
void main()
{
    vec4 color = texture(Diffuse,aTexCoords);
    float lengthFraction = color.a;
    if(lengthFraction < 0.2)
        lengthFraction = 0.2;

    vec3 pos = aPosition.xyz + (aNormal + CombVector) * shellIncrement * shell * lengthFraction;

    gl_Position =  proj * view * model * vec4(pos,1);
    outTexCoords = aTexCoords
}
```

Vertex Shader

```
void main()
{
    vec4 outputColor;

    vec2 TexCoords = vertexTexCoords * textureTilingFactor;
    vec4 tangentAndAlpha = texture(furTextureArray,vec3(TexCoords, shellNumber));
    vec4 offset = texture(furOffsetArray,vec3(TexCoords, shellNumber));

    TexCoords -= (offset.xy - 0.5f) * 2.0f;
    TexCoords /= textureTilingFactor;
    outputColor = texture(Diffuse, TexCoords);

    if(outputColor.a < 0.01)
        discard;


    outputColor.a = tangentAndAlpha.a * offset.a;

    FragColor = outputColor;

}
```

Fragment Shader

*CombVector* is an additional vector used to extrude the mesh along a direction, which can be used to give an effect of moving fur. *shellIncrement* is the increment between two shells and *shell* is the current index of the shell you are on. *shellNumber* is the number of total shells (textures) that can be rendered.

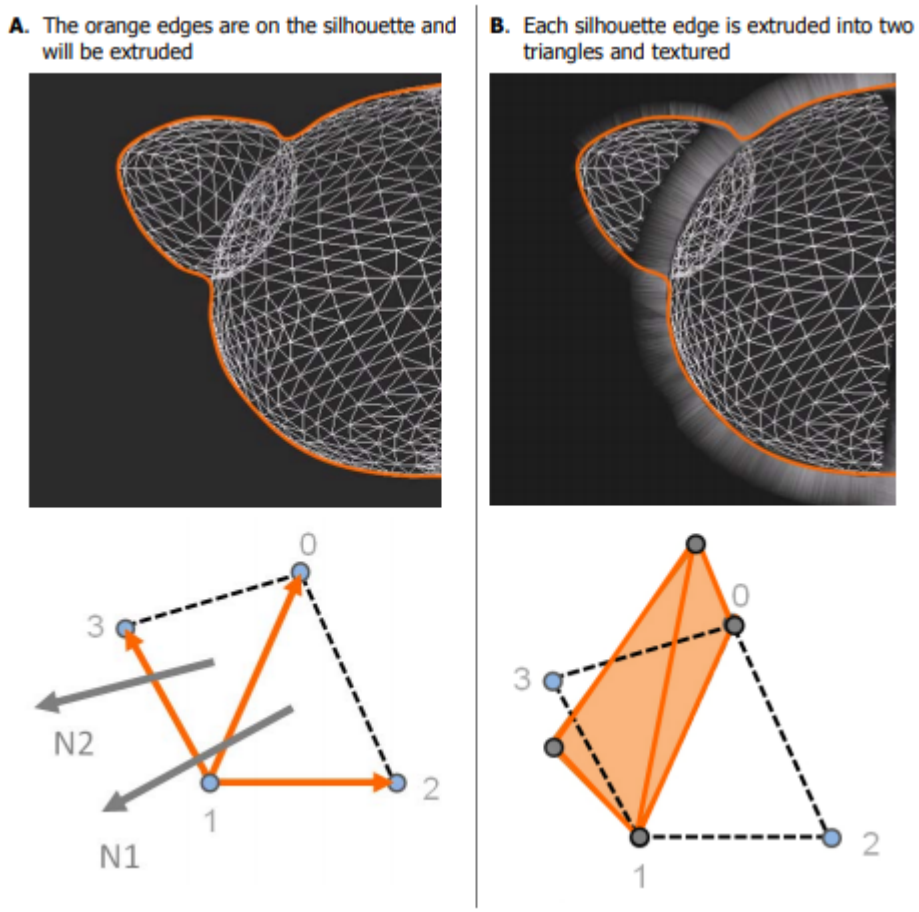
After applying this, you should get something like this figure:

Final result:Cat with Shells

### 3.2 Fins

"Fins are rendered to preserve the illusion of fur along the silhouette edge"[2]. In this method we take advantage of the Geometry Shader. First, we want to know which are silhouette edges and which are not. To determine if an edge is a silhouette edge or no, simply check if one of the triangles is facing the viewer, and the other one is facing away.

[2]Fur (using Shells and Fins) by Sarah Tariq




Detecting silhouette edges with two triangles [3]

Once we know that it is a silhouette edge, we simply extrude by the normal and render it with a different texture.
Here you have some code snippets for the shaders:

[3] From "Fur (using Shells and Fins)"

```
/*
out vec2 vert_TexCoords;
out vec3 vert_Normal;

void main()
{
    vert_TexCoords = aTexCoords;
    vert_Normal = aNormal;
    gl_Position = vec4(aPosition,1);
}

*/
```

Vertex Shader

```
// Compute face normals
vec4 nT = ComputeFaceNormal(p0,p2,p4);
float eyeDotnT = dot(nT.xyz,eyeVec);


//Check if it is a silhouette edge
vec3 eyeVec1 = normalize( Eye - p0 );
vec4 n0 = ComputeFaceNormal(p0,p1,p2);
float eyeDotn0 = dot(n0.xyz,eyeVec1);

    if(eyeDotn0 * eyeDotnT <= 0)
        MakeFin(0,2,maxOpacity);
//Else it might be almost there, so to avoid weird popping we render it anyways
    else if(abs(eyeDotn0) < finThreshold)
        MakeFin(0,2, (finThreshold
    - abs(eyeDotn0))*(maxOpacity/finThreshold));
```

Geometry Shader

The first check is to see if one of the adjacent triangles is back facing. If it's not, we check if it's almost there, comparing it with a threshold. This is to avoid weird or sudden popping that might happen. Also keep in mind that this should be done three times if you are using a Triangle Adjacency list, as you have three adjacenct triangles.

```
void MakeFin(int e0, int e1, float opacity)
{
    vec2 texcoord = {1,0.1};
    float furLength[2];
    vec4 color = texture(Diffuse,vert_TexCoords[e0]);
    furLength[0] = color.a;
    color = texture(Diffuse,vert_TexCoords[e1]);
    furLength[1] = color.a;


    gl_Position = MVP * gl_in[e0].gl_Position;
    textureMesh = vert_TexCoords[e0];
    out_opacity = opacity;
    textureFin  = vec2(0,texcoord[0]);
    EmitVertex();

    gl_Position = MVP * (gl_in[e0].gl_Position + vec4(normalize(vert_Normal[e0])
    + combStrength*combVector,0)*numShells*shellIncrement*furLength[0]);
    textureMesh = vert_TexCoords[e0];
    out_opacity = opacity;
    textureFin  = vec2(0,texcoord[1]);
    EmitVertex();

    gl_Position = MVP * gl_in[e1].gl_Position;
    textureMesh = vert_TexCoords[e1];
    out_opacity = opacity;
    textureFin  = vec2(1,texcoord[0]);
    EmitVertex();

    gl_Position = MVP * (gl_in[e1].gl_Position + vec4(normalize(vert_Normal[e1])
    + combStrength*combVector,0)*numShells*shellIncrement*furLength[1]);
    textureMesh = vert_TexCoords[e1];
    out_opacity = opacity;
    textureFin  = vec2(1,texcoord[1]);
    EmitVertex();

    EndPrimitive();
}
```

Geometry Shader

As you can see the MakeFin function basically creates a quad from the two vertices that compose the silhouette edge.

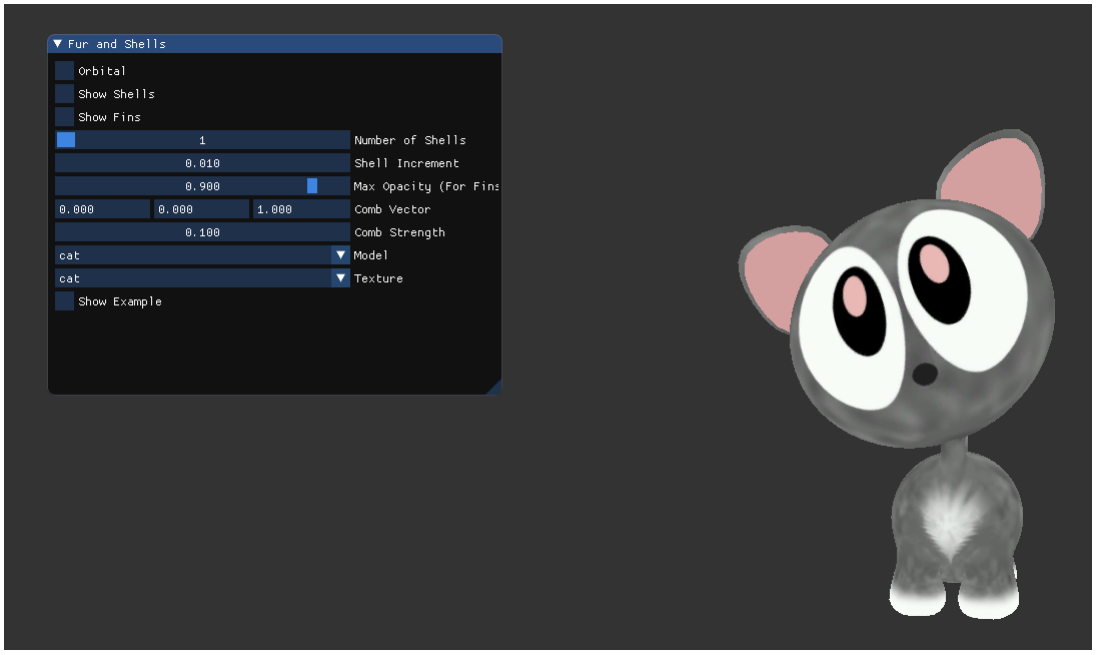
```
out vec4 FragColor;

in vec2 textureMesh;
in vec2 textureFin;
in float out_opacity;


layout(binding = 0) uniform sampler2D Diffuse;
layout(binding = 1) uniform sampler2D finTexture;
layout(binding = 2) uniform sampler2D finOffset;

void main()
{
    vec4 finOpacity = texture(finTexture,textureFin);
    vec4 offset = texture(finOffset,textureFin);

    vec2 texCoords = textureMesh;
    texCoords -= (offset.xy - 0.5f) * 2.0f;
    vec4 color = texture(Diffuse,texCoords);

    color.a = finOpacity.a * out_opacity;

    FragColor = color;
}
```

Fragment Shader

Here we have two additional textures, one from which we extract the form of the Fin, and the other one the current texture coordinate. We then use the final alpha as the mix between the opacity and the texture itself. The resukt that you get, with only Fins, should look like the figure of the next page:

Final result: Cat with Fins

# 4 Explanation on how to use the demo




Demo Window

When opening the demo, you are met with a small IMGUI window and the model of a cat. The controls of the camera, while maintaining the right button of the mouse, are WASD to move around and CTRL/Space Bar to go up and down. You can also activate an orbital mode, which you can rotate while maintaining the right button of the mouse and zoom in/out with the wheel.For default, Shells and Fins are not show. To shown then, tick their corresponding box. You can increase and decrease the number of shells by sliding it. You can also set the shell increment you want, along with the comb vector and its strength.

There is also additional models and textures you can switch to. If you change the model, be aware that you will also have to change the shell increment, since each mesh need its own number. Lastly, you can tick the "Show Example", which will render five spheres with different textures.

# 5 Improvements or other approaches

The simplest approach would be to render all hair by its own geometry. This would result in ultra-realistic fur, but at a high cost. Another one would be to use NVIDIA Hairworks, which is a combination of rendering techniques for

fur,hair,grass and the likes.

Originally, the paper uses a Line Adjacency list as the input for the Geometry Shader. In my case, I used a Triangle Adjacency list, since this way in don't have to also transform it. You can also take advantage of this, and instead of doing a separate pass for the mesh, render it when calculating the silhouette edges.

# 6 Problems encountered

One of the problems that I found was that when doing a conversion from a Triangle list to a Triangle Adjacency list and then to a Line adjacency list, it would take a lot of time to load additional meshes. Not really a problem, but you need pre-generated textures for all the steps, so I recommend to find some before starting or create an algorithm to auto-generated them.

# 7 Conclusion

In conclusion, Fur using Shells and Fins is a technique that is easy to implement and easy to use. Most important logic is done on shader, so your code doesn't need big changes to adapt. It requires pre-generated textures, but that can be arranged by doing it yourself or creating an algorithm for that.

# 8 Bibliography

1-Fur (using Shells and Fins). Sarah Tariq.
http://developer.download.nvidia.com/SDK/10/direct3d/Source/Fur/doc/FurShellsAndFins.pdf