Fur rendering with shells, fins, and order-independent transparency

Odd-Erik Frantzen

2023
April

1 Introduction

I have chosen to work on fur rendering, using a variant of NVIDIA's shells and fins method. I've also implemented a form of order-independent transparency, as described in the paper Weighted Blended Order-Independent Transparency. I've also implemented a skybox, I use textures and object files, I use multiple shaders, and I use a custom framebuffer. You can move around the scene with WASD + Shift + Space, and you can turn the camera with the arrow keys.

You can find my github repo here: `https://github.com/odderikf/fur_project`
Make sure you run the binary from a subdirectory, like ./build or ./binaries.

2 Weighted Blended Order-Independent Transparency

Hair and fur notably have very fine detail, often sub-pixel, and therefore frequently need some form of blending to avoid jittery aliasing. Trying to depth sort hundreds or thousands of strands is of course, not trivial, so I've gone for a form of order-independent shading.

Specifically, I've implemented a weighted order-independent 3-pass method, where the first pass is plainly rendered opaque geometry to the main framebuffer, the second pass does order-independent blending of semitransparent elements to its own framebuffer, and the third pass blends the transparent elements into the main framebuffer. A 4th pass can be done after this to add UI elements.

This is based on the paper by Morgan McGuire and Louis Bavoil [1]. An article on LearnOpenGL has been somewhat useful here [2], as have the blog posts by one of the original authors of the paper [3, 4, 5]. The LearnOpenGL article on framebuffers was very useful [6], as this method requires more complex framebuffer objects.

The semitransparency pass has three channels:

Accumulation stores the colored lit contribution to the blended product, based on their alpha, and their weight (a function of their distance in clip-space). It adds them to a separate accumulation buffer, with src=one, dst=one, so that the final result is the sum of all fragments. This means that blending happens through the in-shader weighting (alpha is multiplied into color for this). The alpha channel of accumulation stores their alpha times their weight, so that in the end, we can scale the accumulated color such that their alpha equals one.

The revealage channel determines how much of the opaque scene to reveal through the semitransparent elements. This channel therefore clears to 1, fully revealing the opaque scene by default, and is then blended with src = 0, dst = 1-src, such that a visible fragment will lower the final revealage of the pass, hiding the opaque scene more. In the shader, the written value is simply the normal alpha of the object.

Finally, the modulation output filters color out of the color channel from the opaque scene. This essentially lets color filters function as filters. This works by outputting (1 - color), adjusted for that fragments contribution (accumulation's alpha), the color being the fragment's material color, and not it's illuminated product. The blending mode is set to src=zero, dst= 1 - srccolor, so the modulation subtracts from the opaque scene's colors.

The accumulated color is composited onto the color buffer, using the accumulation channel for color data, and the revealage channel to determine the alpha of this composite layer.

Because OpenGL does not let you modify the default framebuffer, this solution creates its own framebuffer, and copies its final output into the default framebuffer at the end for display.

3 Fur rendering through Shells and Fins, using geometry shaders

Rendering individual fur strands requires an obscene amount of vertices, so shortcuts and cheats are naturally needed. One such cheat is the shell method, where you draw individual fur strands as dots on a 2d texture, on a shell around the model. By progressively stacking shells outwards, these dots form lines that can pass as strands of fur.

The shell method works pretty great head-on, and you can imitate ambient shadows by just darkening the roots, but when viewing at an angle, you are able to see between the layers, which breaks the illusion. This can be alleviated by having fur cards, textured rectangles, that display fur strands. We can elect to generate these exclusively at the silhouette, to cover for the weak spot of the shell method.

This method was described in a paper from 2001 [7] and a geometry-shader based approach was described by NVIDIA in 2007 [8, 9]

I've mainly gone by intuition on these two techniques, other than that I've mainly looked at the NVIDIA paper.

3.1 Shells implementation

First, a fully opaque base is rendered, in the opaque pass. Then, we render the base again, but with a dedicated shader program. This shader program has a geometry shader, that takes in triangles, and generates several triangles displaced outwards. The local direction and length are determined by the color and alpha channels, respectively, of a texture, in addition to a uniform float for control of overall fur length. Using textures at a vertex level does of course, bottleneck the effective resolution of these textures by vertex density. This combing texture is in tangent space, so a TBN table must be used.

The generated shells are then linearly spaced along this direction, and the usual preparations for lighting are calculated for each generated vertex. Additionally, thinning of hair is simulated by lowering alpha as you leave the scalp. So the normalized displacement as a third texture coordinate for 3d-textures.

Additionally, as proof of concept, I've implemented a very basic wind displacement through uniform, that maintains length, but shifts fur directions somewhat along the wind.

The fragment shader uses the blending method described above, uses phong lighting like most other elements in the scene, but slight rudimentary self-shadowing is implemented, by slightly darkening the roots and brightening the tips. Fur shells have their own normal map (blurred copy of the original, to soften perceived edges), and share their base's roughness map and color map.

The final alpha of a pixel of shell is determined by its texture, a tip-thinning simulated by lowering alpha, and a turbulence/dots/fur density texture that determines which UV coordinates form strands. The python file gen_fur_density.py generates these dots textures.

3.2 Fins implementation

The fins are also implemented through a geometry shader. The NVIDIA paper suggests using adjacency input to find edges that form silhouettes, but I've opted to instead just use triangles input, and test every edge. This does lead to duplicates.

For every triangle, it checks every edge. For each edge, if that edge's averaged vertex-normal is roughly perpendicular to the view direction, then generate a fur-card. A wide treshold is used, so that the cards can fade instead of pop in and out. These cards are also displaced along the fur direction, see the shells implementation, and can be generated with multiple vertices for deformation e.g. for wind or interaction.

These silhouette fur-cards are rendered without back-face culling, as they cull themselves within reason, and this saves worrying about winding order and facing The fragment shader shares most of its details with the shells shader, though notably the fins do not use normal maps, roughness maps, or a dots texture.

4 Skybox

A basic skybox is set up. It is drawn as the first opaque element, has a box mesh, uses position for UVs, isn't lit, and has gl_Position set to xyww, so that it is always at the maximum distance in clipspace. Depth testing uses less-than-equal because of this. I've used a free skybox texture named Yokohama2 [10]. The tutorial of cubemaps and skyboxes from LearnOpenGL was useful here [11]

5 Workflow, filetypes, libraries

The blender directory in the github repo (not included in code zip) contains a couple of blender projects. They have various setup done. Ricky.blend contains the mouse, and has shaders set up to emulate some of the rendering techniques, for more representative texture painting. Ricky features a normal map baked from a displacement map, which was done in blender.

fur_project.blend contains the grass-covered terrain, the broad terrain, and an export script in python.

For object loading, I've used tinyobjloader. For PNG loading, I've used the given utility file, which uses lodepng. For OpenGL support, I've used GLAD, the python GLAD generator, GLFW window loader, and GLM maths library. For string processing, I've used fmt, a text formatting library. All of these libraries are included as git submodules, and are linked, included, and compiled through my CMakeLists file.

6 Troubles and tribulations

It took me a while to figure out a good weighting for the blending operator, as the paper is somewhat cryptic, and no other resources I found made things much clearer. I ended up picking formula 10, which uses fragment coordinates, and tweaking the results until the fur behaved. You can tell that the pads become less transparent as you get far away from them, I imagine there's more tweaking I can do to perfect these weights, but these weights and the difficulty/impossibility of perfecting them for arbitrary scenes are also a flaw of this method.

I decided to implement the color modulation/transmission blending variant described in the third blog post eventually, thinking it would solve a problem I had with blending, when really the main problem there was the weight function. I've still gotten it roughly working, I think, but it ended up being mostly unnecessary work.

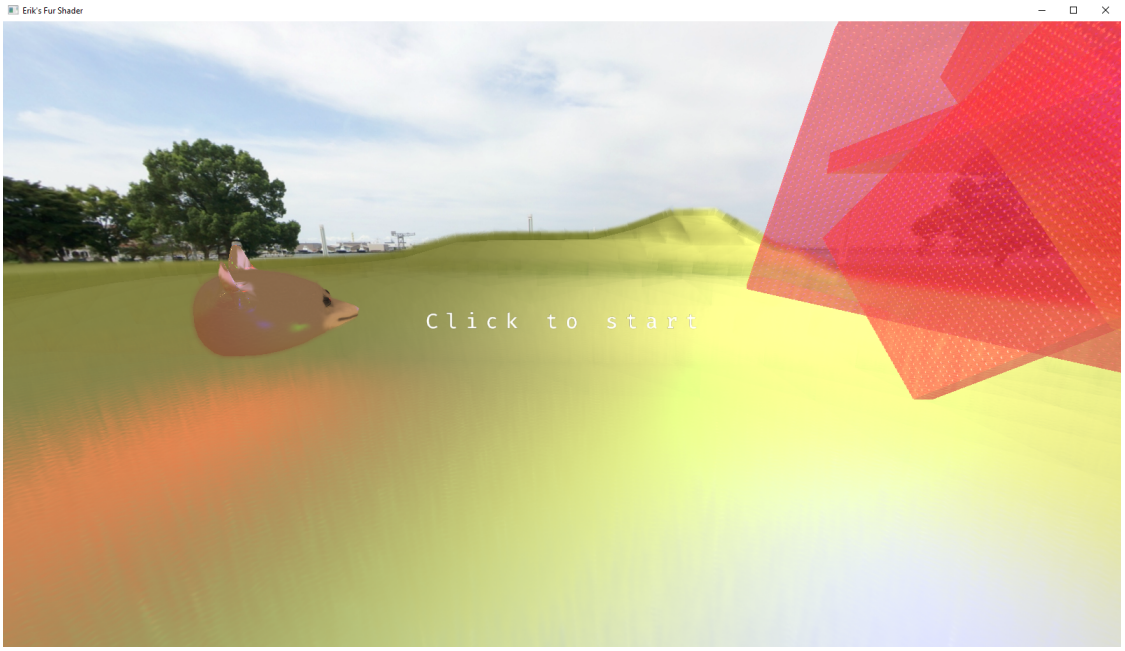
I ultimately wished to make a new animal model, as ricky the rat is neither pretty nor well mapped, but decided to focus on technical things instead. Might make something for the presentation though.

For ages, the way I generated tangents was flawed, and it took me an embarassing amount of time to even fully realize. Ultimately the problem seems to have been a lack of normalization.

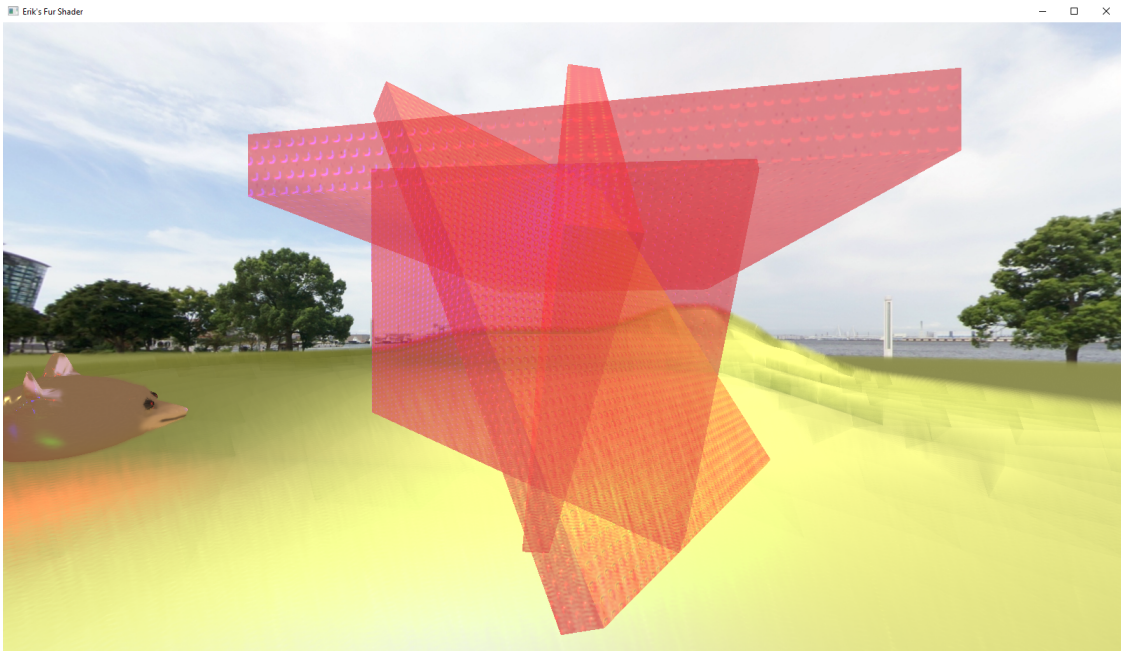
Appendix

Cool JPGs

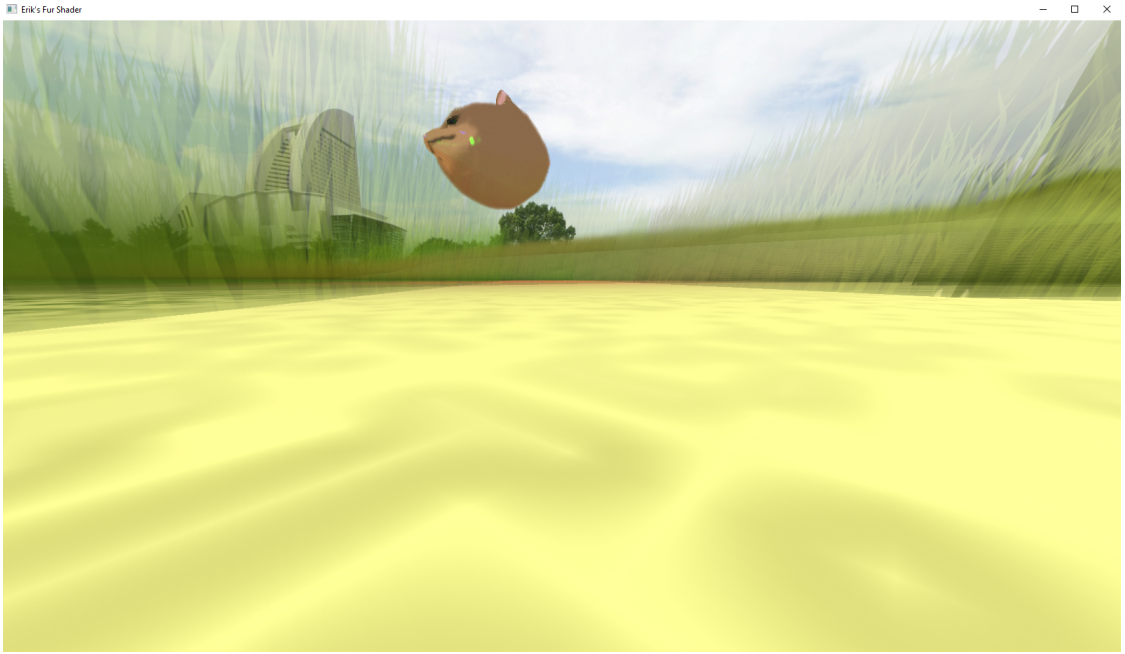
Here's the full scene. You can see intersecting, overlapping, textured, normal-mapped, roughness mapped semitransparent paddles on the right, showing off the blending. Below the paddles is a grass/fur covered field. Behind the field and hill is a skybox, rendered on top of all this is a text overlay, and to the left is the fur-covered rat. The scene has three colored lights (alas, no visual representation of their sources), as well as a fourth high in the sky with very high intensity, to act like the sun. The sun should probably be a dirlight instead of a pointlight, but I ended up never making the time for the implementation.



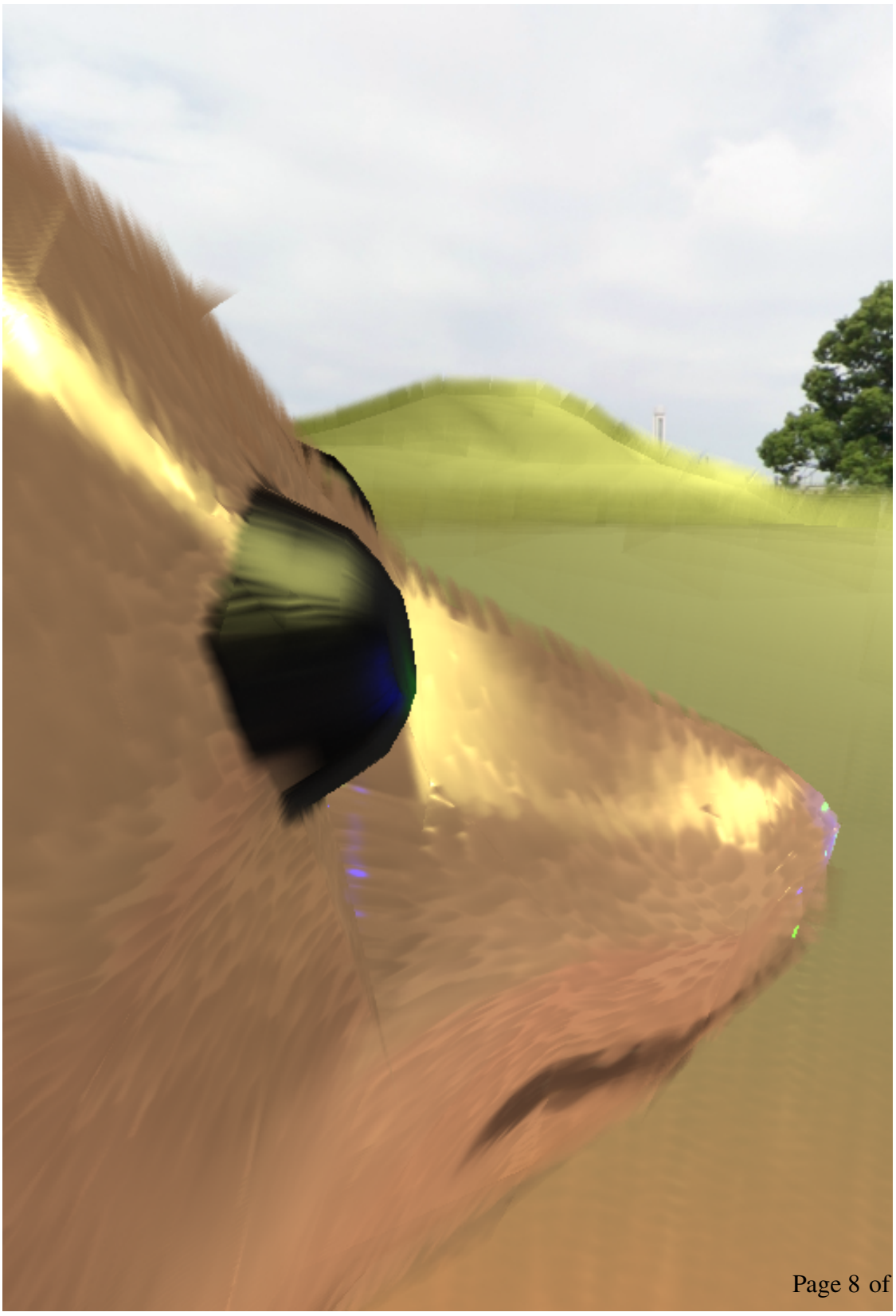
Here's a closeup of the overlapping pads. You can definitely see some blending artifacts with the grass here, but tuning the weights to never have any of that happen is difficult.



Here's a view of below the grass shells, so you pretty much just see the fins. The ground has very big polygons, so the fins struggle to actually cover them well. There could definitely be some more graceful fading in and out of fins vs shells here.



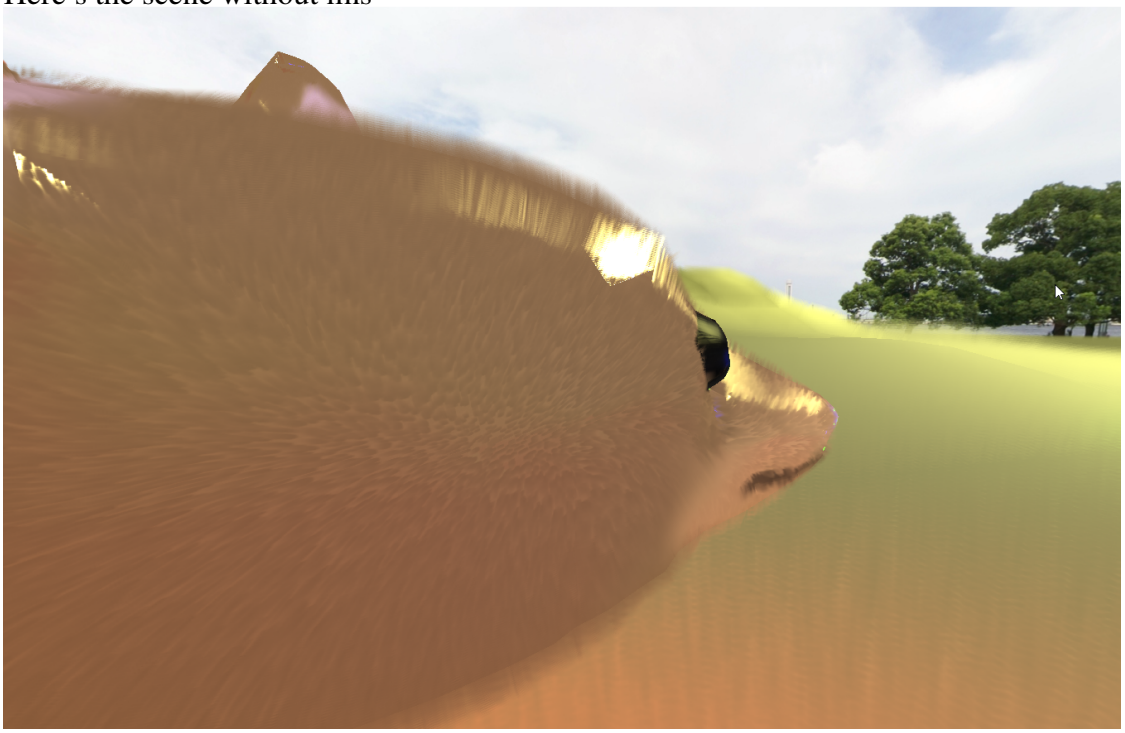
Here's a closeup of the rat, you can see that the fur is fairly groomed, and parts like the eyes and the nose have no fur on them, since the alpha of the fur texture is zero there.



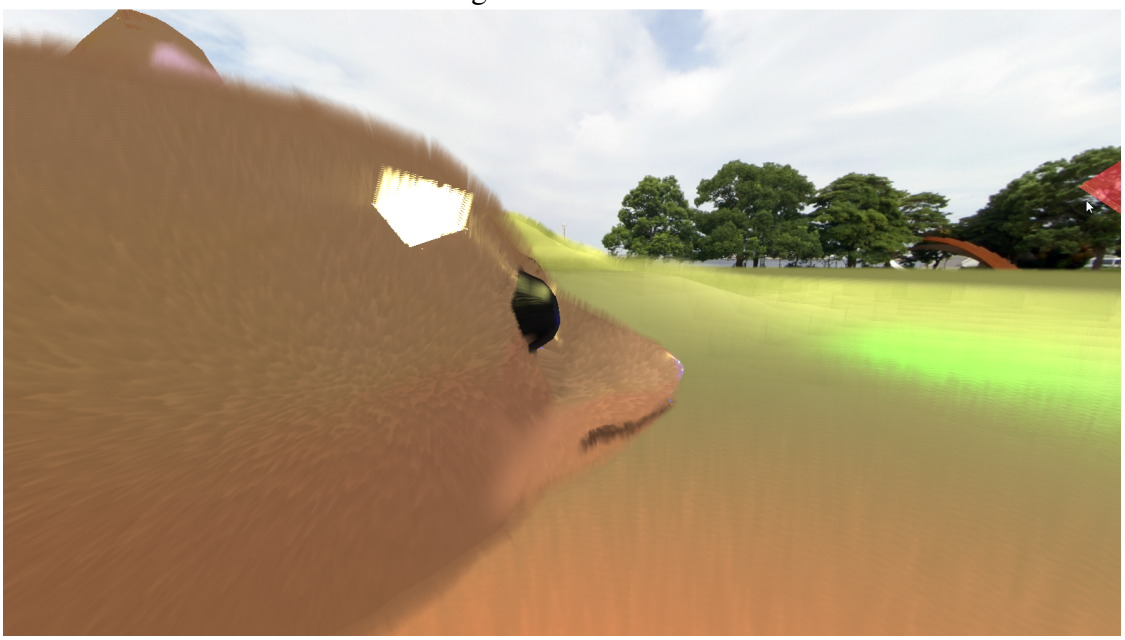
As you can see here, blended things are still occluded by the opaque bases.

Here's the scene without shells

Here's the scene without fins



Here's the scene with both enabled again



References

[1] M. McGuire and L. Bavoil, "Weighted blended order-independent transparency," Journal of Computer Graphics Techniques (JCGT), vol. 2, pp. 122–141, December 2013. http://jcgt.org/published/0002/02/09/.

[2] M. H. Moghaddam, "Weighted blended." https://learnopengl.com/Guest-Articles/2020/OIT/Weighted-Blended.

[3] M. McGuire, "Weighted, blended order-independent transparency." http://casual-effects.blogspot.com/2014/03/weighted-blended-order-independent.html, Mar 2014.

[4] M. McGuire, "Implementing weighted, blended order-independent transparency." http://casual-effects.blogspot.com/2015/03/implemented-weighted-blended-order.html, Mar 2015.

[5] M. McGuire, "Fast colored transparency." http://casual-effects.blogspot.com/2015/03/colored-blended-order-independent.html, Mar 2015.

[6] J. de Vries, "Framebuffers." https://learnopengl.com/Advanced-OpenGL/Framebuffers.

[7] J. E. Lengyel, E. Praun, A. Finkelstein, and H. Hoppe, "Real-time fur over arbitrary surfaces," in 2001 ACM Symposium on Interactive 3D Graphics, pp. 227–232, Mar. 2001.

[8] "White paper - fur (using shells and fins)." https://developer.download.nvidia.com/SDK/10.5/direct3d/Source/Fur/doc/FurShellsAndFins.pdf, Feb 2007.

[9] "Nvidia direct3d sdk 10 code samples." https://developer.download.nvidia.com/SDK/10/direct3d/samples.html#Fur, Jan 2008.

[10] E. H. Persson, "Yokohama2." https://humus.name/index.php?page=Textures&ID=138. licensed under CC BY 3.0.

[11] J. de Vries, "Cubemaps." https://learnopengl.com/Advanced-OpenGL/Cubemaps.