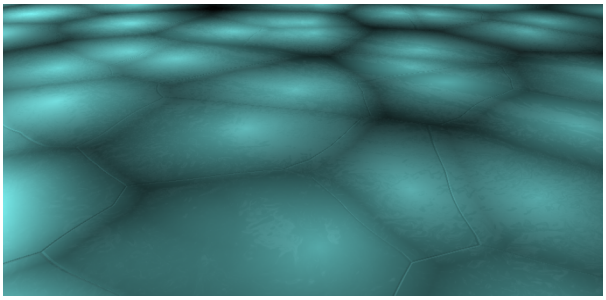
Infinite-Resolution Real-Time Dynamic Procedural Ice-Crack Texture

Etienne Salimbeni

June 2020

1 Introduction

Procedural textures provide an alternative to image-based textures. They are expensive in terms of computation , but require much less memory , can produce infinite resolution (since each pixel is computed) and be easily dynamic (ex: shadows of the cracks , zoom effect).



final render

We implemented this project using Regl.js a library that simplify WebGL.
This is a project done for the course : Introduction to computer graphics at EPFL.

2 Crack pattern (Worley noise)

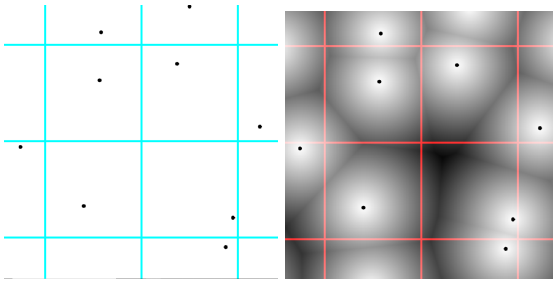
To mimic crack pattern , Worley noise is an obvious choice. Briefly Worley noise , take as an input a list of points P, then each pixel value is the distance between the pixel coordinate to the closest point p in P.

2.1 optimisation for scale

Worley noise creation has two main issues :

- each pixel has to iterate through all points in P .
- you have to pass the dynamic array of points P to the shader .

Our approach (which uses $theBookOfShader$ as a reference) solves both problems with one stone. To get the points , instead of passing them to the shader , we generate them with a pseudo-random function that given a vec2 return a random vec2. This function always return the same vectors , this is important because between each frame the Worley diagram should not change. We divide the texture in a grid , each corner coordinate of a box will be used as an input to the pseudo-random function. This way when end up having a grid with a point in each box (see figure).
Finally each pixel only generate the 9 closest points (the 8 one in its neighboor boxes + its own) in the fragment shader. Since the closest point must be one of those 9 points , the resulting diagram is a correct Worley noise.



grid with generated points

2.1.1 other implementations

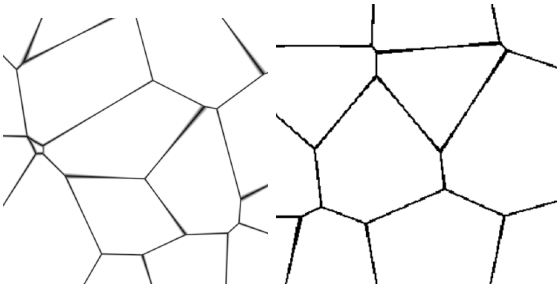
With this solution , there is very little control over the points locations. You can only change the grid shape (scaling it for example will create a zoom effect) and modify the pseudo-random function (this can be used as a seed) . Another approach would consist in statically having an array of points on the fragment shader , but this would force us to recompile the shader each this array is modified. More practical , we can pass the array of points via a Texture , but then we would have to sort the array and the number of points would not be infinite. There is an interesting 3rd option , rendering cones from the the top , fast but we would lose a lot in resolution since the cones would have to be approximated by a mesh.

2.2 border

So far the Worley noise uses the distance from the pixel to the closest point. But to get the crack effect we can compute the distance from the pixel to the closest border. This is simply done by finding the 2 closest points (since the 2 closest points are within the 9 boxes around the pixel no need to change the previous algo). the final distance is then the distance to the 2nd point - distance to the closest one.

2.2.1 aliasing

This is more of a general comment , while developing procedural textures. You must avoid as mush as possible the use of *step* (this is simply changing the color depending on a threshold) , but use *smoothstep* to reduce the aliasing effect that is caused while comparing float values.



with and without aliasing

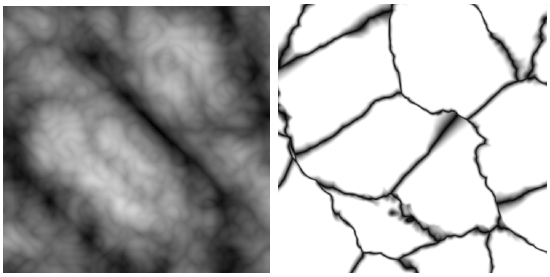
2.2.2 Voronoi noise

A problem with this computation of the borders , is that in some configurations output incorrect borders This is because we are not computing the correct distance to the border but an approximation. To correct this we can use Voronoi noise with use the correct distance formula , but since our points are evenly spaced due to the grid , this effect is rarely noticeable.

2.3 distortion

Worley noise generate straight line , which is too perfect for realistic ice crack. To make the borders less regular we translate the input of the worley noise (which is the pixel coordianate) with the turbulance noise :

$$worleyNoise(position + turbulanceNoise(position))$$



turbulence noise — distorted borders

3 Lighting and environment

Now lets focus more on the fragment shader.

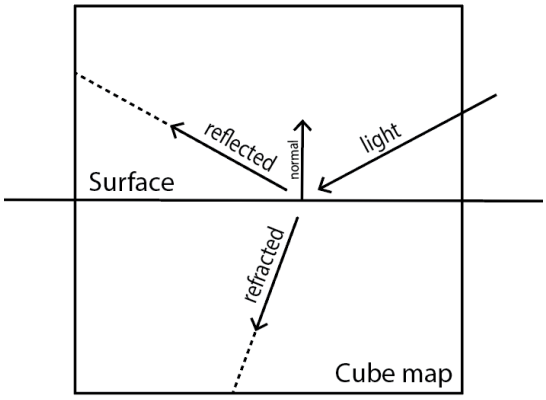
3.1 basic lighting

We use the classic combination of ambient + diffuse + specular light. The only particularity is that we take the material color computing it with the worly noise explained is section1 , and we use the normals presented in the next section.

3.2 reflection - refraction

Ice is both transparent and reflective. To achieve reflections , we compute the reflected light direction

vector ($reflect()$ do it for you) Refraction is very similar ($refract()$ do it for you) Once you have the direction of the refected/refracted light you can use this vector to sample a color in a cubemap texture. Finally you mix this color with the ambient + diffuse + specular color based on how much refected/refracted degree you want.

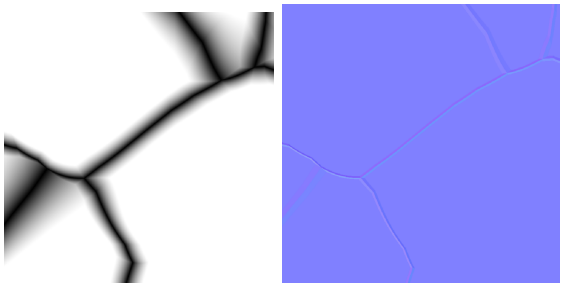


4 Relief

There are two types of relief on our texture : cracks and the roughness of the ice surface. For both effect we computed normal maps from a bump map. To convert from bump to normal , we simply take 3 close points to the point of interest. Create 2 vectors from those 3 points , the cross product of the 2 is the normal at that point of interest.

4.1 crack relief

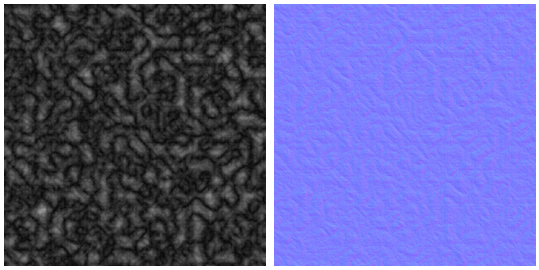
To get the crack bump map we used Worley distorted borders , closer to the edge.



bump and normal map for cracks

4.2 surface roughness

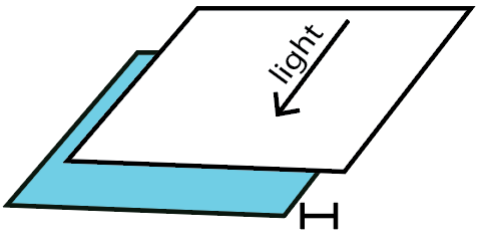
For the surface we used a high density turbulence noise. Bump map easily get a strong gradient from white to black , which result in drastic normal changes. To smooth the normals , we scale the bump map values to have lower differences (the bump map is now practically all black).



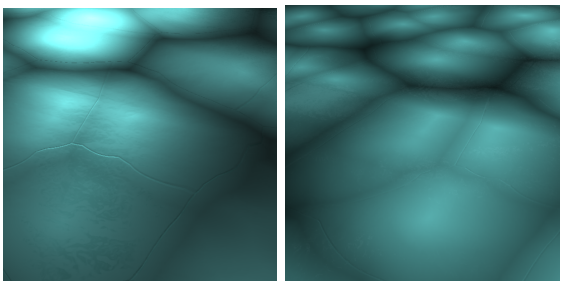
bump and normal map for surface

5 Shadows

To add more realism to the ice floe , we implemented the shadows of the ice cracks. (we could imagine an ice on a lake and the shadow would be casted on the lake floor). To achieve this we assume the ice surface is flat , and use some sort of parallax effect to move the texture depending on the light direction. We offset the texture coordinate by the .xy values of the light direction vector.



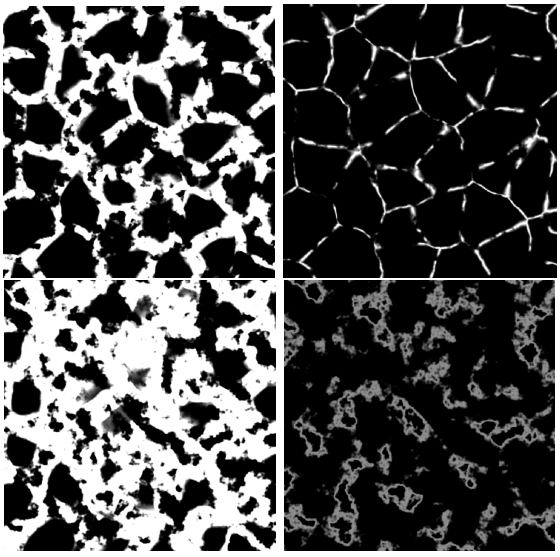
offset of the parallax effect



renders with different light angles

6 Snow

In our quest of realness , we implemented snow on the ice surface. The snow should follow the cracks , to achive this we masked a perlin noise with the Worley edge diagram. See Figure (perlin noise , Worley edge , resut) this makes the snow very customisable since you can change how close to the cracks the snow stays and how dispersed the snow is.



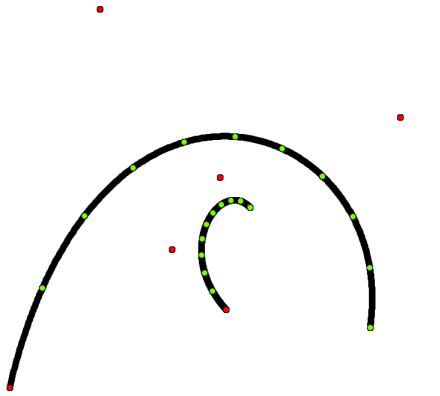
different levels of snow

7 Bezier curves

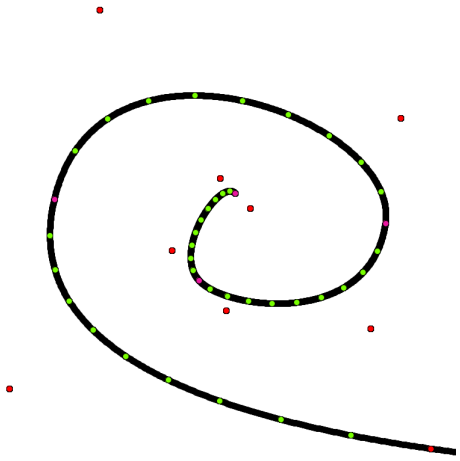
This is not really for the texture per-se , but to show it off. We implemented Bezier curves to move the light source smoothly , as well as the camera.

To create a continuous bezier curve from a list of points , we first add to this list of points all the points between them. Then 4 points at a time , we compute the cubic bezier curve. If we do not add the middle points to the list of points the curves looks fragmented.

To get the point value on that curve at a given time , we divide the time by 4 it to get the index of which pair of 4 point the curve is at. Doing modulo on the time gives the [0,1] index used in the cubic bezier formula to know which point on the curve to compute.



without middle points



with middle points

8 Future work

- improve snow : so far the snow is simply a white stain on the surface with no shadow or peculiar reflectance
- dynamic cracks : make the texture cracks updatable from an external source , ex : new cracks in a specific area
- more optimisation : we are often using the same noise function , recomputing always the same value.

9 References

https://thebookofshaders.com/12/
https://www.algosome.com/articles/continuous-bezier-curve-line.html
http://regl.party/api
https://www.youtube.com/watch?v=pP2zmOTGfgA