Realistic Rendering of Ice and Crack Propagations

AMIN GHAFARI, University of California, Berkeley
JULIAN PARK, University of California, Berkeley

ABSTRACT Ice rendering is a challenging subject in path tracing which requires special care to have decently rendered pictures. There has been some effort in past projects to make this happen, and they were successful to get good results. However, the appearance of the surface of most of these ices is unrealistically clear and pristine. Here, a class called MicrofacetGlass is defined to perturb the normal surfaces at the hit point in order to blend the properties of a glass and a surface out of the rendering. Our random bubble generator which generates bubbles within range of specified radius inside the ice, along with physical fractures, makes our ice cubes look highly realistic.

Additional Key Words and Phrases: Path tracing, Ice, Microfacet, BSDF

ACM Reference format:
Amin Ghafari and Julian Park. 2017. Realistic Rendering of Ice and Crack Propagations. *ACM Trans. Graph.* , , Article (May 2017), 4 pages.
DOI: 0000001.0000001_2

1 TECHNICAL APPROACH

Despite all effort to get rendered ice close to natural ice, past outcomes were more close to glass, which is too specular. This immediately gave the impression that it is not actual ice. Also, most of the rendered ice samples were missing the white color that an ice has in its center, which again makes the pictures less appealing. Recent efforts has been successful to get the white part of the ice to some good level [Yi Lang 2006] but the surface of the produced ice made it more like glass than ice, which has some diffuse property on the surface.

In this paper, we render ice with realistic texture and structure. In addition to internal bubbles and subsurface scattering, this includes rendering of propagated ice inside the ice geometry and making an animation of a falling cube while it impacts a surface breaks to pieces from the previously generated crack and fracture piece of ice.

The first step toward getting a realistic ice is to get a surface to represent ice. Although displacement mapping (or bump mapping) is an option to get variation in the normal surfaces of the mesh, it requires a very fine mesh with thousands of triangle primitives which make the rendering process very slow. This computational power can be saved and used to render thousands of bubbles inside the ice cube. This will significantly help to get a more realistic ice than bump mapping. Hence, our solution to making a good surface representation of ice is a microfacet surface which represents a diffuse glass, named as MicrofacetGlass in our code.

Starting with some ray tracing code from previous projects, we introduce a new BSDF class called MicrofacetGlass which generates a new normal to the surface of the material when a ray hits it. This distribution is the same as for a microfacet material, called a Beckmann normal distribution function (NDF) which perturbs the normal slightly according to the value of α; this factor determines that the surface is more diffuse for larger values and more glassy for smaller values. Then, having this new normal and the direction of the ray we assume that the surface is a glass with refraction coefficient of ice to determine if a ray reflects or refracts from the surface of the ice. Since the normal is not in the z-direction of the local coordinate, we added a function to calculate the reflection and refraction direction of the ray for a general vector to the surface; these functions are called generalreflect() and generalrefract(). This algorithm is shown in Algorithm 1.

Furthermore, in order to have some white "clouds" inside the ice for a better rendering, we introduced a cloud of bubbles inside the ice using a random generation of bubble positions inside the ice

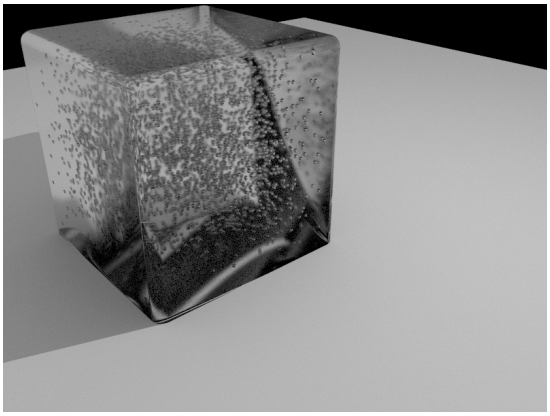
ALGORITHM 1: MicrofacetGlass::sample_f

Input: wo(outgoing direction)
Output: Spectrum(reflected or refracted spectrum), wi(incoming direction), pdf(probability distribution function)

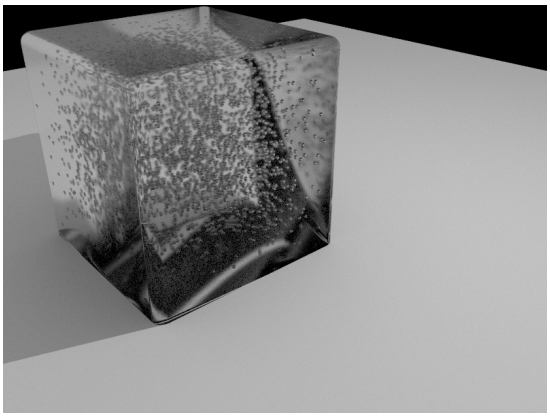
```
r = get a sample 2d Vector;
theta_h = atan(alpha*sqrt(-log(1.0-r.x)));
phi_h = 2.0*PI*r.y;
if
 (theta_h ≤ 0.0ortheta_h ≥ PI/2.0orphi_h ≤ 0.0orphi_h ≥ 2.0 ∗ PI)
 then
    *pdf = 0.0;
    return Spectrum();
end
h.x = cos(phi_h)*sin(theta_h);
h.y = sin(phi_h)*sin(theta_h);
h.z = cos(theta_h);
if (!generalrefract((wo, h, wi, ior)),(if refraction does not happen)) then
    generalreflect(wo, h, wi);
    *pdf = 1.0;
    return reflectance/abs_cos_theta(*wi);
else
    double R_0 = pow((ior-1.0)/(ior+1.0),2.0); double R = R_0 +
     (1.0-R_0)*pow((1.0-abs(dot(h,wo))),5.0); double _eta;
    if (wo.z¿=0.0) then
        _eta = 1.0/ior;
    else
        _eta = ior;
    end
    if (coin_flip(R)) then
        generalreflect(wo, h, wi);
        *pdf = R;
        return R*reflectance/abs_cos_theta(*wi);
    else
        generalrefract(wo, h, wi, ior);
        *pdf = 1.0-R; return
         (1.0-R)*transmittance/abs_cos_theta(*wi)/pow(_eta,2.0);
    end
end
```

using a normal distribution of the position – the same idea utilized to generate the radius of the bubbles. Then, these numbers have been checked to see if they are in a valid range. Afterward, all of these generated bubbles and their positions are manually added to the DAE file of the ice. The algorithm of this code is not explained here for conciseness.

Combining these steps, we have successfully created a fairly realistic simulation of ice as shown in Fig. 1. The first ice cube is rendered when there is only one set of random distribution of bubbles in the middle, and the second one has two distributions where one of them is concentrated at the center with high concentration and the other spread across the whole domain. This helps to make the resultant rendered ice seem more realistic.

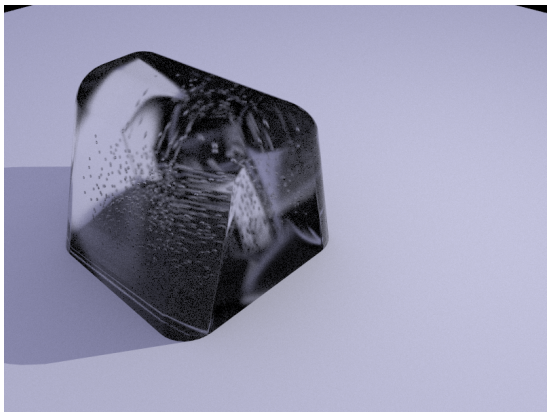


(a) An ice cube with only normal distribution of the position inside the cube

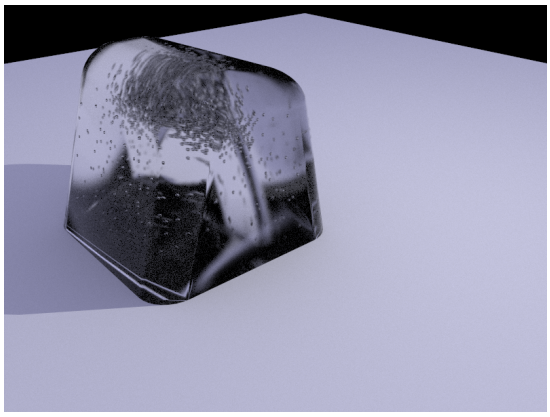


(b) An ice cube with two distributions, one highly concentrated in the middle and one all over the cube with lower concentration

Fig. 1. Ice cube rendered with bubbles inside



(a) A melting ice with bubbles across whole volume



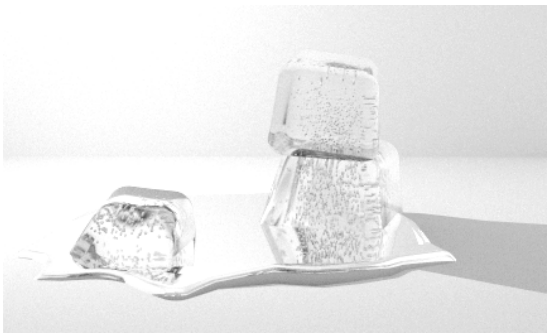
(b) A melting ice with one high concentration distribution of bubbles in the middle and one with low all over the shape

Fig. 2. Melting ice

For another case, we used a shape simulating a half-through melted ice (Fig. 2). Again, it is shown how a bubble distribution helps make the rendering better. In the second picture, it is clear how the high concentration of bubbles in the middle makes the region look as opaque as real ice.

In order to measure the quality of our rendering in a metric based on real life, we compared our rendered images with a real picture of a few ice cubes on some surface water, Fig.3. The whole scene is modeled in Blender and put inside a Cornell Box. It is then exported and rendered in our own path tracer. The water on the surface is modeled as a variant of a glass material with refraction coefficient of 1.33, and the ice cubes are filled with bubble distributions. As you can see in Fig. 3. the resultant picture is very close to a real-life case of melting ice cubes.

(a) Real picture of ice



(b) Our rendered ice

Fig. 3. Real-life comparison

2 CRACK GENERATION

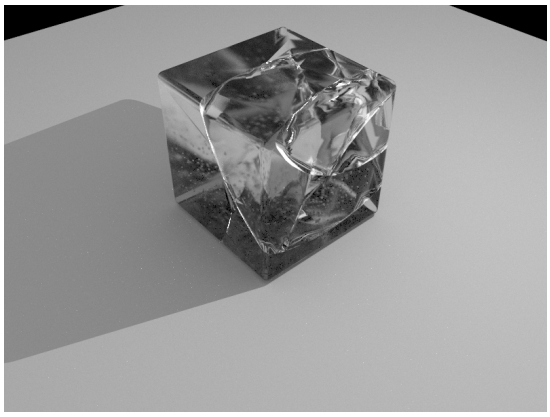
The next step was to generate cracks and fractures inside the ice cube to get an even better rendering result. Fig. 4 shows some high-quality rendered ices with bubbles and cracks. The first idea behind this was to manipulate the generated mesh from the DAE inputs file by adding some extra features to the meshEdit files. However, we realized that this work can be easily done in Blender through fracture libraries that break any geometry into pieces. So instead of spending time coming up with our own fracture-generation function, we focused on how to manipulate and render such fractures more realistically. By applying physics-based forces to each broken piece, we obtained a dynamic animation of a falling cube and its explosion.

After crack generation, we found out that it is possible to manipulate the DAE files of ice to change the position of each piece of ice. So we created a MATLAB program explained in Algorithms 2 and 3 to update the position vector of each piece per time step. We use an initial text file which is a copy of a DAE file of an ice cube with all of its pieces in order to update position of each piece and generate a new DAE file corresponding to that position for rendering. We applied gravity in the Z direction and initial velocity downward until the point of collision at a time step when the center of mass of the cube is 1 unit above the surface. Afterwards, an upward velocity smaller than the impact velocity as well as a radial velocity is introduced in order to get the exploding animation for the pieces of ice. The integration and updated location of each piece is obtained using an explicit Euler method. Then these files are rendered to generate a dynamic GIF animation.

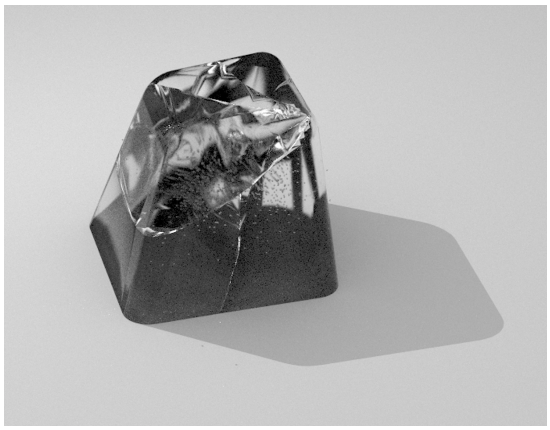
ALGORITHM 2: Updating positions

Input: .txt file
Output: .dae files
% Get a .dae file as an input file Pos_text = fileread('position.txt'); % Find the position matrix of every piece of the ice that is shattered;
% Apply the gravity with an initial speed in the desired direction and speed to each piece;
% Update the position of each piece by Euler integration for each time step;
% Replace the old position with the new one and generate a new DAE file for each time step;
% Using this new position matrix find the next position in time loop;

Fig 5 shows a frame-by-frame sequence of a fractured ice cube falling downwards due to gravity and each piece exploding outwards after collision with the horizontal plane as intended in our MATLAB code.



(a) Ice cube and bubbles and cracks



(b) Melting ice cube with cracks

Fig. 4. Realistic rendering

ALGORITHM 3: Movement function ::f_move

Input: X(initial position matrix), dX(displacement vector), dTh(rotation vector)

Output: M(final displacement matrix)

```
% Displacement and rotation matrix D = [1 0 0 dX(1); 0 1 0 dX(2); 0 0 1
  dX(3); 0 0 0 1];
cosx = cos(dTH(1));
sinx = sin(dTH(1));
Rx = [1 0 0 0; 0 cosx -sinx 0; 0 sinx cosx 0; 0 0 0 1;];
cosy = cos(dTH(2));
siny = sin(dTH(2));
Ry = [cosy 0 siny 0; 0 1 0 0; -siny 0 cosy 0; 0 0 0 1];
cosz = cos(dTH(3));
sinz = sin(dTH(3));
Rz = [cosz -sinz 0 0; sinz cosz 0 0; 0 0 1 0; 0 0 0 1];
X = D*X;
% positive and negative displacement for rotation around axis;
Xop = eye(size(X));
Xop(1:3,4) = X(1:3,4);
Xon = eye(size(X));
Xon(1:3,4) = -X(1:3,4);
M = Xop*Rz*Ry*Rx*Xon*X;
```

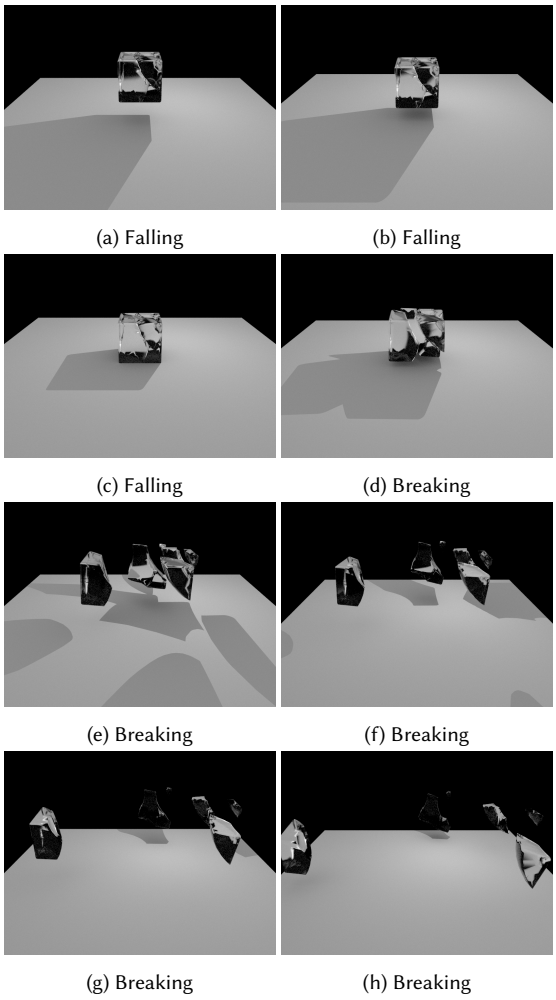
3 CONCLUSION

In this project, we successfully rendered various geometries of ice by combining many of ice's unique material properties: glassy surface, internal bubbles, melting shapes, and sharp cracks. As seen in our results throughout this paper, such as in Fig. 4(b), we can confidently say that our rendered ice possesses those properties in a highly realistic manner.

While creating multiple DAE files via Blender, then rendering them on our customized path tracer and our MicrofacetGlass class, then adding a normal distribution of bubbles and fractures, we resolved many challenges like finding the right probabilistic balance between refraction and reflection for our MicrofacetGlass class (by incorporating an alpha factor for realistic blurriness and transparency), or figuring out a way to make our bubble distributions look less artificial (by overlaying multiple random distributions). We tackled all our tasks, as described throughout our paper, with one goal in mind: make our ice rendering as realistic as possible. At a high level, we learned that even a seemingly obvious material like ice takes significant computational effort to accurately render on screen. We often take ice's physical structure and texture for granted in our daily lives, but alike many other natural materials, realistically rendering ice entailed many features to be engineered.

4 CONTRIBUTIONS

- Amin Ghafari: wrote the code for the new Microfacet Glass materials in path tracer code, wrote the code for random generation of bubbles using MATLAB, manipulated the resultant DAE files to add material properties, wrote the code for updating the position of fracture pieces of ice, rendered majority of pictures.
- Julian Park: made all the DAE files modeling ice in Blender, made the half-through melting ice with rounded edges, developed the random fractures of ice to the DAE files, modeled the cube pictures similar to real-life, rendered some pictures, made the video and presentation.



(a) Falling (b) Falling (c) Falling (d) Breaking (e) Breaking (f) Breaking (g) Breaking (h) Breaking

Fig. 5. Animation of a falling ice breaking into pieces

REFERENCES

Joyce Yi Lang. 2006. Rendering an Ice Cube. (2006). http://web.archive.org/web/20080207010024/http://www.808multimedia.com/winnt/kernel.htm

Received April 2017