Simulating Stained Glass in PBRT

JAMISON ROSE
Western Washington University
jamison@jamisonrose.com

Abstract
This paper covers a basic method for use by a ray-tracing program to generate images which can be reused in turn in a ray-tracer to create a realistic representation of stained glass.

I. INTRODUCTION
The motivation behind this project was to be able to create stained glass in a pbrt scene without explicitly defining it and instead by providing a description of the stained glass similar to how pbrt converts the description of a scene into a 2d image through ray-tracing. By modifying pbrt it was possible to come close to this goal and the results look good, but there is still many things that might make the system better.

II. METHODS
There are two main methods used to recreate a scene as stained glass. First an image for the glass must be generated, and then it needs to be accurately represented in a separate scene as glass.

I. Generating a Glass Image
Generating an image for use as strained glass consisted of several steps.
- Creating a flat color image
- Outlining objects
- Separating shading into steps
- Outlining color borders
- Overlaying a panel like grid

I.1 Creating a flat color image
Creating a flat color image was one of the more difficult steps and pbrt is not well suited to creating such images. The goal for this step was to have each object in the pbrt world to have a single color so it could be represented by a single or few pieces of colored glass. Pbrts main objective is to create photo realistic images so modifying it to only output an objects color would require significant changes through a variety of classes.

In order to create a realistic image pbrt uses several levels of abstraction to encapsulate methods that recreate a realistic image. An object in the pbrt world has a material that describes the surface of it. This material can have several methods applied to it to reduce aliasing in the final image. These would need to overwritten because the color needs to be constant over the surface of the image. The Material is then sampled by a scenes sampler. This would need to be overwritten, otherwise pixels would have data from several objects encoded in them which is not possible when the pixels are treated as glass. Then, a surface integrator uses the samples to model how the color changes over the surface of the objects which should be constant in this case. Finally the film the image is recorded on can modify the color based on the samples it receives to make a more accurate representation of the image as it would be stored on film. Each of these steps would need to be overwritten to some degree which would require a lot of time and work.

Luckily one of pbrrts default renderers provides a simple solution to this. Pbrts sampler renderer class has a parameter to shade objects based on their object ids. To do this it hashes the objects id into a unique color and whenever the main intersection in a pixel is an object, it
uses that objects id hash as the color rather then the color returned by the samples of the sampler. The downside to this approach is that only the structure of the original scene is preserved and not the basic color of objects. After this step a image such as this can be generated:

I.2 Outlining objects

Outlining objects is simplified since we are guaranteed to have each object be a different color at this point. When writing an image we just have to look to see if the next pixel horizontally or vertically is a different color and if it is then write black instead of the color that should have been on the pixel. An outline of this kind can look like this:

If it was not guaranteed that each object would be a different color then pbrets sampler would need to be modified so that when a pixel contained samples from more than one object it would need to make that pixel black.

I.3 Separating shading into steps

Real stained glass is made out of several panels of glass each having it own color. Because of this it is difficult for to have accurate shading in a stained glass picture. To replicate this in pbret it was important to separate the shading into several unique steps rather then a smooth gradient. There are several ways of achieving this, but the method used was to calculate the brightness at a given pixel and run it through a step function producing a gray-scale image representing the shading. Four levels of lightness were left over after this process. One problem with this technique is that if the image had similar brightness across the whole image it would be almost entirely one shade after going through the step function. To correct for this the shading function was modified to first find the minimum and maximum brightness and modify the step functions so that the steps occurred evenly within this area of brightness rather then across the whole scale. After this step a image layer would look like this:
I.4 Outlining color borders

After the shading is finished a new problem is introduced. Objects can now have multiple colors again which is not possible with single colored glass. To rectify this the borders between shades need to be outlined. Similar to the process of outlining shapes we already have an image that is guaranteed to have unique color transitions were we want to outline, namely the shading image from the previous step. Using the same technique as the object outlining step except on the shading image produces a result such as this:

I.5 Overlaying a panel like grid

Finally the image should be divide up more that it currently is. Although each panel is currently a unique color satisfying that criterion, they are rather large so we need a method to break them up into more manageable sizes. An easy way to do this is to overlay a grid generated using Worley noise. For this image we used manhattan distance to calculate the boundaries, but it also has support for using euclidian distance. The results look like this:

One thing to notice is that some panels end up having all black, but this is alright since sometimes stained glass ends up like this anyway.

To get the image into a pbtr scene first all the layers from the previous parts needed to be combined. This was simple since there was a single color image and several gray-scale images. To get the final stained glass image the color was multiplied by each of the overlays. This process consisted of treating each color (RGB) as a value from zero to one. Then by multiplying them by the gray-scale value of zero to one a final result would be produced. For the shading step this would lighten or darken the image, but for the other steps the overlays were all either black (0) or white (1) meaning that they simply blackened areas as appropriate. This produces the final image like so:
II. Representing the image in a scene

The object used to display the image in pbtr is a simple plane made up of two uv mapped triangles in a *trianglemesh* object. The plane was then transformed to fit into the scene as appropriate. To get it to act like glass a glass material was chosen and it’s Kr and Kt values were set to an *imagetexture* made from the generated image. The Kr parameter of the glass surface describes the reflectivity of the glass. This allows the light that is see reflecting from the glass to be the color that it is. The Kt parameter describes the transmissivity of the glass which is what causes the project to have the colors that it does.

The default *directlightning* used by pbtr was not sufficient to make the glass project the image on it like would happen with real stained glass. Instead when using direct lightning the glass would be treated as a solid opaque object and occlude any light that should have gone through it. To counteract this the lighting was changed to a *photomap*. The photon map was set to send out 1000000 photons in order to estimate the effect of light passing through the glass. While this produces a good effect close inspection shows that it does not look entirely accurate.

III. Results

The results of the total process looks like this:

As can be seen in the image the glass accurately projects an image onto the ground next to the stain glass pane. Since the veins between panels are pure black light is not let through so the veins are visible in the projection. Also when observing the back face of the pane the veins stick out more since there is such a lack of light. Observed from the front they are also very visible since the background does not shine through at all in those areas while the colored panels show much of the background.
IV. Discussion and Future Work

In the future it there are a number of improvements that could be made to the system. By modifying each of the classes explained in the flat image section it would be possible to get accurate color of objects. A simple technique for this would be to simply average the color over the entire surface of an object and then make the whole object that color.

The glass pane in the final image currently looks very flat so in the future it would be nice to make the glass pane be a high density mesh so that the black veins could be used to disturb the geometry of the glass to make it look more three dimensional.

The glass is currently very pure so adding a volumetric integrator to it while defining the glass’ internal structure would make for much nicer looking glass, but would require more of an understanding of the physics present in the glass than pbtr provides.

Finally, the most disappointing part of the render is that when fog was added to the picture it did not light up like would be expected in a real world scenario. Modifying the volume integrator for constant fog would produce striking results, but would probably be the most difficult in terms of development time and physical understanding. An example of this disappointing effect can be seen in this image:

References

[Pharr and Humphreys, 2010] Physically Based Rendering, Second Edition: From Theory To Implementation. 2nd ed. N.p.: Morgan Kaufmann