Glints Rendering

Xuantong Liu, Weili Liu, Frank Cai, Ray Gong

May 2021

1 Introduction

Glitter objects exist widely in our daily lives, such as fairy dust, stars, Christmas lights, etc. It is impractical for complex specular surfaces under sharp point lighting to use Monte Carlo pixel sampling to render them because energy is concentrated on tiny highlights. The problem is essential because we would like to generate more realistic images. We mainly worked on adding texture supports and mipmapping to our existing path tracer projects. Then, we added glint effects by using high-resolution normal maps. The glint rendering on complex specular surfaces is just like what we see in the real world.

2 Methods

Our glint rendering implementation can be divided into two major parts: adding texture support to our existing pathtracer from project 3, and further adding the ability to apply normal mapping, which is what we needed to render glints.

2.1 Texture Mapping

We experimented with three types of texture mapping: constant-color mapping, fixed-pattern mapping and image texture mapping. The most challenging and time-consuming task in this part is to understand the project 3 code and the places we needed to modify to make texture mapping work.

2.1.1 Constant-Color Texture Mapping

In order to add texture support, we started from the most basic one: applying a constant-color texture to a primitive. The reason we started from something this simple is that adding features onto project3 has proven to be more of a challenge than we originally expected. This allows us to figure out which part of the code we need to modify in order to add texture support without having to worry too much about the implementation details. We present our result in figure 1, we apply the color red to the left wall and yellow to the left sphere. Since everything will be assigned a single color, we do not need UV mapping in this part.

2.1.2 Fixed-Pattern Texture Mapping

Then, we moved onto something slightly more complicated: fixed-pattern textures. In particular, we experimented with the checkerboard pattern. In order to achieve this effect, we need to alternative between two colors. As shown in algorithm 1, we make use of the hit point where the ray intersects

a primitive. By using the fact that sin alternates between 1 and -1 for a range of inputs, we'll be able to achieve this pattern by multiplying the sin of each coordinate of the hit point. If the result is less than 0, it takes one color, otherwise, it takes the other color. Again, here we only make use of world coordinates to calculate the color, so we don't need to perform any uv mapping. We show some example pictures in figure 2, after applying the checkerboard texture, we still see the right color bleeding onto the sphere.

2.1.3 Image Texture Mapping

Finally, we implemented texture mapping using images. For this, we read in an image file and map the colors onto a primitive based on uv mapping. In order to calculate the correct uv coordinates on a sphere, we utilized algorithm 2. We first get the vector from the sphere center to the intersection point on the surface of the sphere. Then, by using the arctengent function as well as the x and z directions of the vector, we can get the angle. We further divide the angle by 2π and add 0.5 so that the angle is between 0 and 1. This is our u coordinate. The v coordinate is simply the y direction of the vector. Again, we need to shift it to be within the range of 0 and 1. Then, we need to shift the u and v coordinates by the texture image width and height to retrieve the rbg value of the correct pixel. We will now use this value instead of a constant albedo value for the diffuse material as we did in project 3-1. Again, we show some example results in figure 3. In both images, the sphere on the left is mapped to an image texture. The original texture image are also shown along with the rendered results.

Algorithm 1 Checkerboard Pattern Texture Mapping

```
1: procedure GETCOLOR(hitpoint, color1, color2)
2:     sines ← sin(hitpoint.x) * sin(hitpoint.y) * sin(hitpoint.z)
3:     if sines < 0 then return color1
4:     else return color2
```

Algorithm 2 UV Mapping for Spheres

```
1: procedure GETUV(hitpoint, origin)
2:     n ← hitpoint − origin
3:     u ← atan2(n.x, n.z)/(2 * π) + 0.5
4:     v ← n.y * 0.5 + 0.5
```

3 Normal Mapping

Normal maps are defined as a function $n : R^2 \rightarrow D$ from points $u = (u, v)$, which locates in texture space to normals $s = (s, t)$. The Jacobian of $n(u)$, $J(n(u))$, plays an really important role in determine the highlight areas. To get normal mapping to work we need a per-fragment normal. Similar to what we did with diffuse and specular maps we can use a 2D texture to store per-fragment normal data. This way we can sample a 2D texture to get a normal vector for that specific fragment.

if we note deltaPos1 and deltaPos2 two edges of our triangle, and deltaUV1 and deltaUV2 the corresponding differences in UVs, we can express our problem with the following equation :

$$\delta_{pos1} = x_{\delta_{uv1}} * T + y_{\delta_{uv1}} * B$$

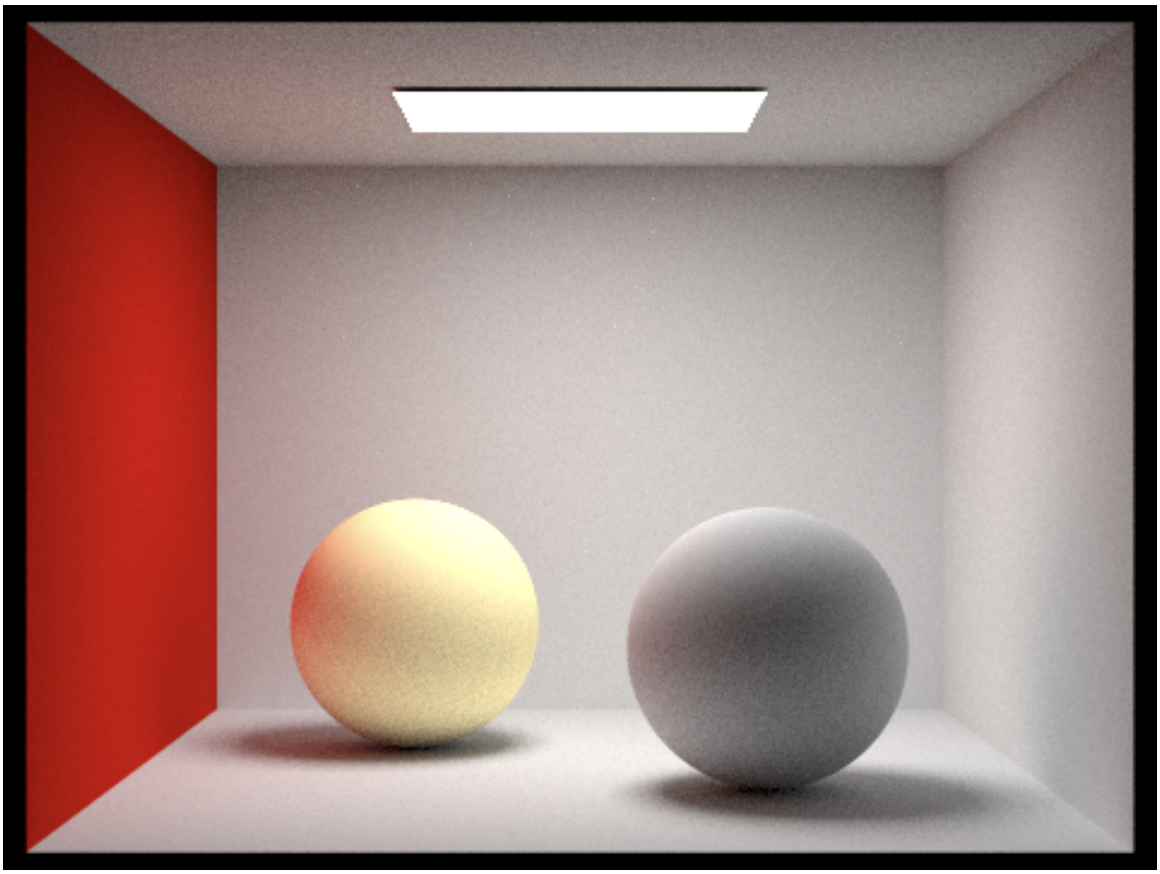

Figure 1: Constant-Color Texture Mapping Result

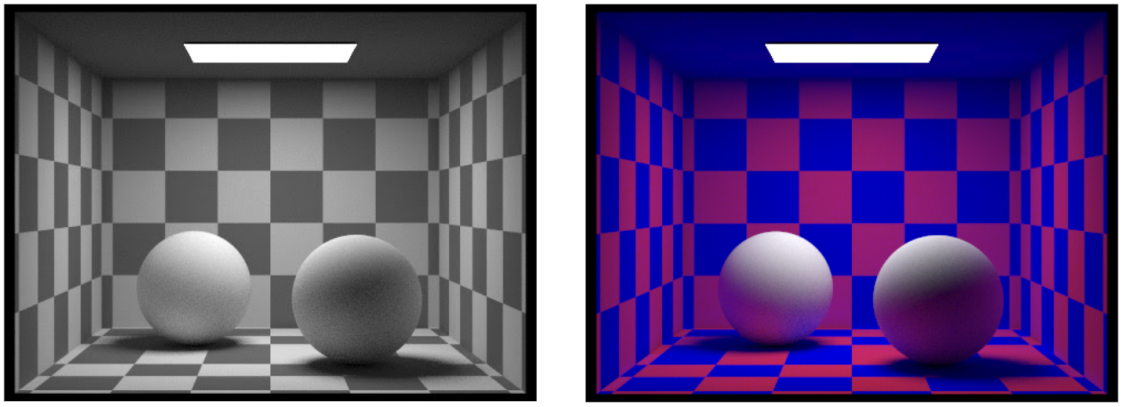

Figure 2: Checkerboard-Pattern Texture Mapping Result

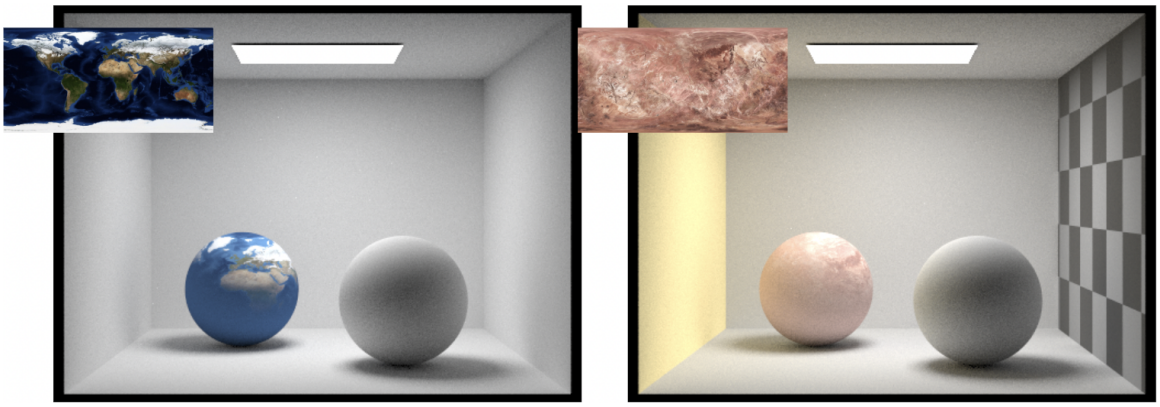

Figure 3: Image Texture Mapping Result

$$\delta_{pos2} = x_{\delta_{uv2}} * T + y_{\delta_{uv2}} * B$$

By solving these equations we can obtain the T and B. After successfully solving T, B as well as N, we can use the following matrix to transform from tangent space to model space:

$$\begin{bmatrix} T_x & B_x & N_x \\ T_y & B_y & N_y \\ T_z & B_z & N_z \end{bmatrix}$$

With this TBN matrix, we can transform normals (extracted from the texture) into model space.

4 Experiments Results

By adding proposed normal map to specific objects in the scene rendering process, we can successfully observe glint effect in the original texture rendering surface on the object. Comparison results are shown in figure 4.

5 Conclusion

In this project, we mainly research on glint rendering. Based on our assignment 3, we have successfully made some improvements on original codes and enable it to rendering surface with arbitrary textures which are stored in image files. Based on this we have dive deeper into normal map algorithm and implement it to our glint rendering pipeline. By adding glint features to original objects, the objects' surface become more vivid and similar to real ones.

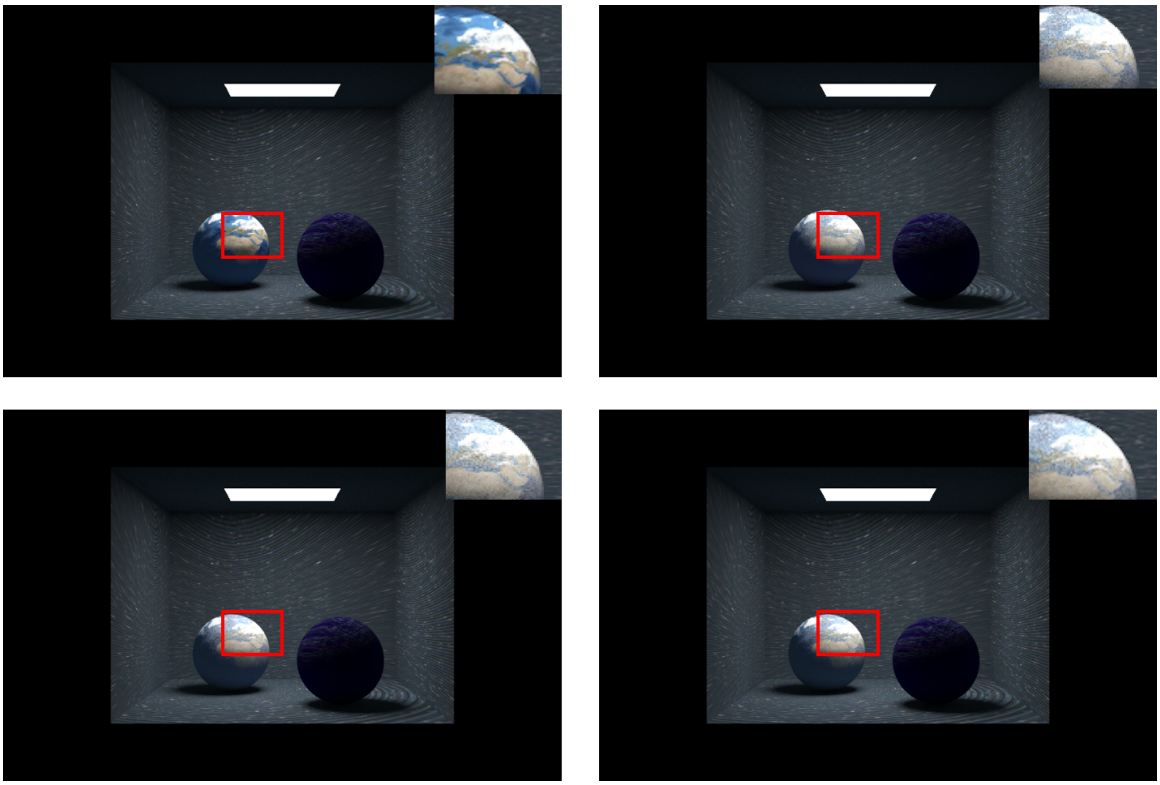

Figure 4: Glint rendering results

References

[1] Beibei Wang, Miloš Hašan, Nicolas Holzschuch, and Ling-Qi Yan. Example-based microstructure rendering with constant storage. *ACM Transactions on Graphics*, 39(5):1–12, 2020.

[2] Ling-Qi Yan, Miloš Hašan, Wenzel Jakob, Jason Lawrence, Steve Marschner, and Ravi Ramamoorthi. Rendering glints on high-resolution normal-mapped specular surfaces. *ACM Transactions on Graphics*, 33(4):1–9, 2014.