**Final Report**

**Yangchen Zhao** and **Junming Chen**

Github Link

As mentioned in the milestone, our plan was to add three features to our current renderer: Normal Mapping, Mipmapping, and Environment Mapping. After days of work, we have perfectly finished implementing all three features and rendered amazing and realistic scenes.



# 1. Normal Mapping

For Normal Mapping, we initially follow the instructions on the PBRT Normal Mapping part. However, the code structure in the PBRT is quite different from our current code. This caused some troubles when we tried to add Normal Mapping to our renderer(we also tried to use the method in CSE272 hw0 before the checkpoint, but we found that Normal Mapping is more advanced and easier to implement. We then turned to use Normal Mapping with the methods in lecture 9.

We need to compute differentials. We maintain two numbers for a ray differential: the `radius` and the `spread` of the ray.

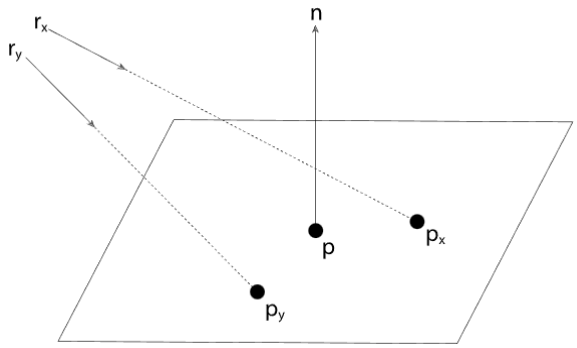
Here, the radius is approximately $\frac{length(dp/dx)+length(dp/dy)}{2}$ and the spread is approximately $\frac{length(dd/dx)+length(dd/dy)}{2}$. We now use the value of radius to approximate the value of dp/dy and the value of dp/dx.

## Partial derivatives

To decide how many texels a ray will cover, we need to calculate the $\begin{bmatrix} \frac{\partial u}{\partial x} & \frac{\partial u}{\partial y} \\ \frac{\partial v}{\partial x} & \frac{\partial u}{\partial y} \end{bmatrix}$. There are many version of differential methods to implement it.

### Path differentials

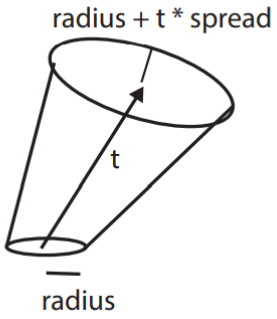
We can use *Path differentials* in PBR-Book, where we emit 2 extra auxiliary rays to estimate the Partial derivative.




But that is absolutely expensive to compute. The rays need bounce between object, and thus we will have rays at the number of $3^{bounce}$, which will increase the cost exponentially. We have tried it but it seems **not practical** for our torrey renderer based on CPU.

### A simple way inspired from RenderMan

Here is a conservative heuristic way mentioned in CSE 272, that we can approximate the partial derivatives based on the ray's distance and the spread coefficient of a surface.




Here the idea is approximate the ray as a cone. We use *radius* to approximate both $\frac{\partial p}{\partial y}$ and $\frac{\partial p}{\partial x}$, and use *spread* to approximate both $\frac{\partial d}{\partial y}$ and $\frac{\partial d}{\partial x}$. When a ray bounce between objects, we update the radius like this:

```
Vector3 hit_point_original = HitPointOriginal(ray, NearstObj);
Vector3 hit_point = ray.Origin + ray.distance * ray.Direction;
ray.radius += ray.spread * ray.distance;
```

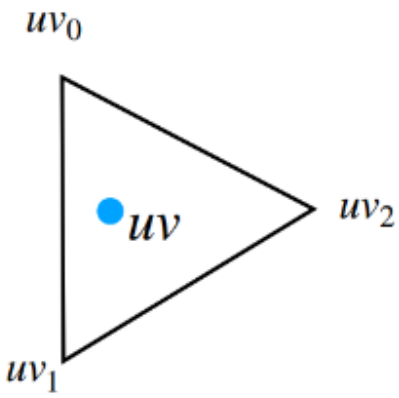
But what we actually want to get $\begin{bmatrix} \frac{\partial u}{\partial x} & \frac{\partial u}{\partial y} \\ \frac{\partial v}{\partial x} & \frac{\partial u}{\partial y} \end{bmatrix}$. Let's apply chain rules on it. P is actually the hit point on the object's surface.

$$\frac{\partial u}{\partial x} = \frac{\partial u}{\partial p}\frac{\partial p}{\partial x}$$

Since we now have $\frac{\partial p}{\partial x} \approx ray.radius$, we just need to figure out the Jacobin between $(u, v)$ and points. For the triangle primitive, we calculate it by barycentric coordinate.

Here we use the method in CSE272 and the idea in the CSE168 lecture:

From barycentric coordinates to UV goal: find $\begin{bmatrix} \frac{\partial u}{\partial x} & \frac{\partial u}{\partial y} \\ \frac{\partial v}{\partial x} & \frac{\partial v}{\partial y} \end{bmatrix}$




quiz: given $\frac{\partial b_i}{\partial x}$, what are $\frac{\partial u}{\partial x}$ and $\frac{\partial v}{\partial x}$?

$$u = (1 - b_1 - b_2)u_0 + b_1u_1 + b_2u_2$$
$$v = (1 - b_1 - b_2)v_0 + b_1v_1 + b_2v_2$$

Noticing that here the $s == b1$ and $t == b2$ .

```
Vector3 barycentric = Vector3(1 - b1 - b2, b1, b2);

if(triangle.mesh->uvs.size() != 0)
{
    Vector2 uv_0 = triangle.mesh->uvs[triangle.mesh->indices[triangle.index]
[0]];
    Vector2 uv_1 = triangle.mesh->uvs[triangle.mesh->indices[triangle.index]
[1]];
    Vector2 uv_2 = triangle.mesh->uvs[triangle.mesh->indices[triangle.index]
[2]];


    Vector2 uv_coor = { dot(Vector3(uv_0.x,uv_1.x, uv_2.x),
        barycentric),
     dot(Vector3(uv_0.y,uv_1.y,uv_2.y),
        barycentric)};
    //here s = b1, t = b2
    Vector2 duvds = uv_2 - uv_0;
    Vector2 duvdt = uv_2 - uv_1;
    Real det = duvds[0] * duvdt[1] - duvdt[0] * duvds[1];
    Real dsdu = duvdt[1] / det;
    Real dtdu = -duvds[1] / det;
    Real dsdv = duvdt[0] / det;
    Real dtdv = -duvds[0] / det;
    Vector3 dpds = p3 - p1;
    Vector3 dpdt = p3 - p2;
    dpdu = dpds * dsdu + dpdt * dtdu;
```

```
    dpdv = dpds * dsdv + dpdt * dtdv;

    u_coor = uv_coor.x;
    v_coor = uv_coor.y;
}
```

It is very easy to apply it to sphere, since $(\theta, \phi) = (\pi v, 2\pi u)$

```
object = std::make_shared<Shape>(Shape{ sphere });
distance = nearest_t;
Vector3 point_original = (origin + distance * direction) / fabs(sphere.radius);
Real theta = acos(point_original.y);
Real phi = atan2(-point_original.z, point_original.x) + c_PI;


u_coor = phi / (2 * c_PI);
v_coor = theta / c_PI;

dpdu = sphere.radius * Vector3(-sin(v_coor) * sin(u_coor),
    sin(v_coor) * cos(u_coor),
    0.) * 2. * c_PI;
dpdv = sphere.radius * Vector3(cos(v_coor) * cos(u_coor),
    cos(v_coor) * sin(u_coor),
    -sin(u_coor));
```

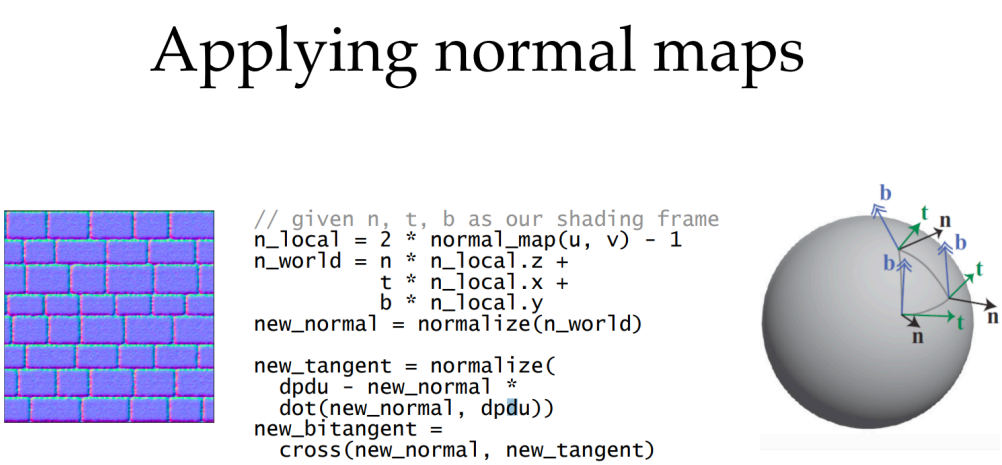
After getting the values of dpdu and dpdv, we now need to find the value of dudx, dudy, dvdx and dvdy. As mentioned above, we used the value of “radius” to approximate the value of dp/dy and the value of dp/dx. In this case, we can get the value of dudx, dudy, dvdx and dvdy by radius/dpdv and radius/dpdu. After getting these values, we can now calculate the value of du and dv and then get the shift point p’

Now, we can calculate $\begin{bmatrix} \frac{\partial u}{\partial x} & \frac{\partial u}{\partial y} \\ \frac{\partial v}{\partial x} & \frac{\partial u}{\partial y} \end{bmatrix}$ this way:

$$\frac{\partial u}{\partial x} = \frac{\partial p}{\partial x} / \frac{\partial p}{\partial u}$$

These are all we have done to implement the Normal Mapping. We have gotten all the values we need to calculate the shift point p.

### Shading normal




We first build a new struct called `ParsedNormalMap`.

```
using ParsedColor = std::variant<Vector3 /* RGB */, ParsedImageTexture>;
using ParsedNormalMap = ParsedImageTexture;
```

After that, for each `material`, we added a new parameter `normal_map`

```
using Color = std::variant<Vector3, Texture>;
using NormalMap = Texture;
struct Diffuse {
    Color reflectance;
    NormalMap normal_map;
};
```

We also changed the `Parse_scene.cpp` file to read the Normal Map Texture

```
std::tuple<std::string /* ID */, ParsedMaterial> parse_bsdf(
    std::map<std::string /* name id */, ParsedColor>& texture_map,
    ......
    ParsedNormalMap normal_map = { fs::path("none") };
    ......
    else if (type == "diffuse") {
        ParsedColor reflectance(Vector3{ 0.5, 0.5, 0.5 });

        for (auto child : node.children()) {
            std::string name = child.attribute("name").value();
            if (name == "reflectance") {
                reflectance = parse_color(
                    child, texture_map, default_map);
            }
            else if (name == "normal_map")
            {
                normal_map = std::get<ParsedImageTexture>(parse_texture(child,
default_map));
            }
        }
        return std::make_tuple(id, ParsedDiffuse{ reflectance ,normal_map });
```

For the Normal Mapping function, we first constructed the tangent space for meshes. Here n is just the shading normal. In order to construct the T frame, we first calculate the derivative of the point P with respect to the u to get the direction, which is the UV coordinate. After that, we remove the part that is not orthogonal to n to get t, in which we subtract the n * the projection of the n vector on dpdu from dpdu. Then we use the cross product to compute a vector b that is orthogonal to both n and t.

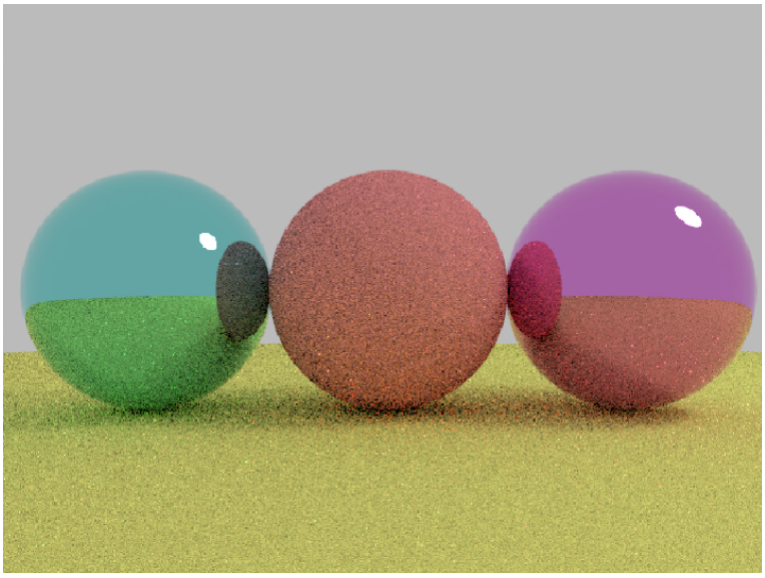
After we constructed the tangent space for meshes, we now need to actually get the normal which we should use in Normal Mapping. We first get the normal_map(u,v) and scale it then minus by one to get the local normal. After that, we convert the local normal to the space defined by the n,t, and b coordinate basis by multiplying the vectors n,t, and b.

```
Vector3 shading_normal = normal_p;
Vector3 t = normalize(ray.dpdu - shading_normal * dot(shading_normal,
ray.dpdu));
Vector3 b = cross(shading_normal, t);
NormalMap normal_map = std::visit([&](auto& map) {return map.normal_map; },
material);
if (normal_map.width == 1 && normal_map.height == 1)
    return Vector4{ shading_normal, 0. };
Vector3 local_coor = 2. * normal_map.GetColor(ray.u_coor, ray.v_coor, 0) - 1.;
//std::cout << local_coor.x << "," << local_coor.y << "," << local_coor.z <<
std::endl;
Vector3 normal_original = local_coor.z * shading_normal +
    local_coor.x * t +
    local_coor.y * b;
```

After we get the normal, we use this normal when we need to compute the color(we use this normal only when we need to calculate the color)

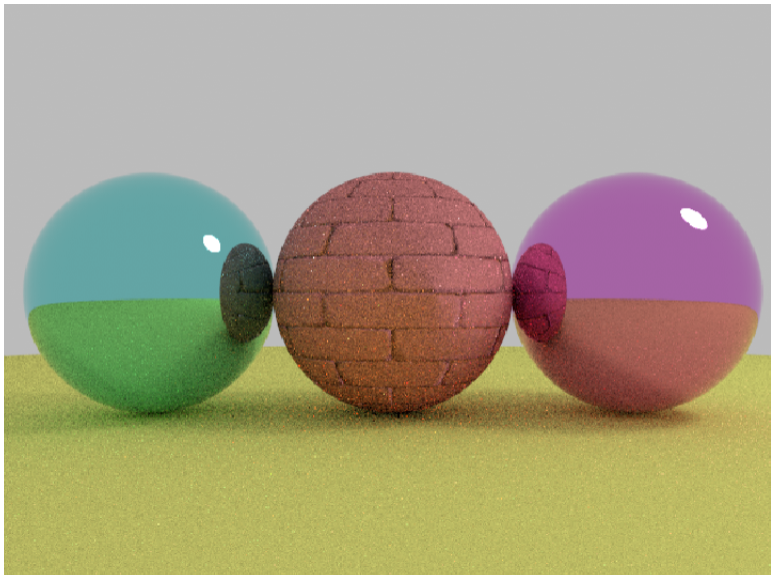
```
Vector3 scatter_direction = m_Sampler->SampleDirection(material, ray,
normal_transformed, m_Vars, scene, m_BVH, rng, &is_reflect, &is_refract);
//std::cout << "scatter:(" << scatter_direction.x << "," << scatter_direction.y
<< "," << scatter_direction.z << ")" << std::endl;
Real pdf = m_Sampler->GetPDF(material, ray, normal_transformed, m_Vars,
scatter_direction, scene, m_BVH, rng, is_reflect, is_refract);
//std::cout << pdf << std::endl;
Vector3 brdf = m_Sampler->GetBRDF(material, ray, normal_shading, m_Vars,
scatter_direction, scene, m_BVH, rng, is_reflect, is_refract);
```

## Experiments & Result



Without normal map 🔼

with normal map 🔽



## 2. Mipmapping

### Idea

Mipmapping is a kind of prefiltering, which is used to antialiaing on texture. The core of the mimapping is the relation between texture coordinate $(u, v)$ and film coordinate $(x, y)$. When a ray hit a object far away, it should cover a larger area of texels, since there the value of $\frac{\partial uv}{\partial xy}$ is higher, which means $(u, v)$ is more susceptible to the change of $(x, y)$.

### Partial derivatives

The differential method in **Mipmap** is the same with what we have talked in the **Normal Map** section.

Now, we can calculate $\begin{bmatrix} \frac{\partial u}{\partial x} & \frac{\partial u}{\partial y} \\ \frac{\partial v}{\partial x} & \frac{\partial u}{\partial y} \end{bmatrix}$ this way:

$$\frac{\partial u}{\partial x} = \frac{\partial p}{\partial x} / \frac{\partial p}{\partial u}$$

So here we can infer the footprint of a ray by:

$$\begin{aligned} footprint &= average(||\frac{\partial u}{\partial x}||, ||\frac{\partial v}{\partial y}||, ||\frac{\partial u}{\partial y}||, ||\frac{\partial v}{\partial x}||) \\ &= average(||\frac{\partial p}{\partial x}||, ||\frac{\partial p}{\partial y}||) / average(||\frac{\partial p}{\partial u}||, ||\frac{\partial p}{\partial v}||) \end{aligned}$$

```
Texture& texture = std::get<Texture>(m_color);
// Approximate du/dx = (dp/dx) / (dp/du) = radius / dpdu
Real dp_duv = (length(ray.dpdu), length(ray.dpdv)) / 2.;
Real footprint = ray.radius / dp_duv;
Real level = texture.GetLevel(footprint);
surface_color = texture.GetColor(ray.u_coor, ray.v_coor, level);
```

### Image pyramid

Mipmapping contains a image pyramid, where we prefilter the texture with different levles. It si helpful to reduce the cost, since we actually implement a more time efficient way on searching. Here we just use box filter to build a Mipmap.

```
struct MipMap {
std::vector<std::shared_ptr<Image<T>>> pyramid;
MipMap() = default;
MipMap(Image<T>& img)
{
        pyramid.push_back(std::make_shared<Image<T>>(img));
        int max_lavel = min((int)log2(Real(max(img.width, img.height))) + 1,
MAX_MIP_DEPTH);
        for (int i = 0; i < max_lavel; i++)
        {
            Image<T>& last_img = *pyramid[pyramid.size()-1];
            int new_w = max(last_img.width / 2, 1);
            int new_h = max(last_img.height / 2, 1);
            Image<T> new_img(new_w, new_h);
            for (int y = 0; y < new_img.height; y++)
            {
                for (int x = 0; x < new_img.width; x++)
                {
                    new_img(x, y) = (last_img(2 * x, 2 * y) +
                                     last_img(2 * x + 1, 2 * y) +
                                     last_img(2 * x + 1, 2 * y + 1) +
                                     last_img(2 * x, 2 * y + 1)) / Real(4);
                }
            }
            pyramid.push_back(std::make_shared<Image<T>>(new_img));
        }
    }
}
```

#### Bilinear lookup between texels

Given an integer `level`, we will look up corresponding texture in the pyramid. And we adopt bilinear interpolation between the four adjacent texels.

```
T Lookup(Real u, Real v, Real uscale, Real vscale, Real uoffset, Real voffset,
int level)
{
    Image<T>& current_img = *pyramid[level];

    Real x = current_img.width * modulo(uscale * u + uoffset, 1.);
    Real y = current_img.height * modulo(vscale * v + voffset, 1.);
    auto i = static_cast<int>(x);
    auto j = static_cast<int>(y);


    // Clamp integer mapping, since actual coordinates should be less than 1.0
    if (i >= current_img.width)  i = current_img.width - 1;
    if (j >= current_img.height) j = current_img.height - 1;

    auto pixel_00 = current_img(i, j);
    auto pixel_01 = current_img(i, modulo(j + 1, current_img.height));
    auto pixel_10 = current_img(modulo(i + 1, current_img.width), j);
    auto pixel_11 = current_img(modulo(i + 1, current_img.width), modulo(j + 1,
current_img.height));
```

```
    auto pixel = pixel_00 * ((Real)i + 1. - x) * ((Real)j + 1. - y)
        + pixel_10 * (x - (Real)i) * ((Real)j + 1. - y)
        + pixel_01 * ((Real)i + 1. - x) * (y - (Real)j)
        + pixel_11 * (x - (Real)i) * (y - (Real)j);
    return pixel;
}
```

#### Trilinear lookup between level

Here we just interpolate the `level`, where we get a weighted average of two level's textures bilinear interpolation result.

```
T Lookup(Real u, Real v, Real uscale, Real vscale, Real uoffset, Real voffset,
Real level)
{
    if (level <= 0)
    {
        return Lookup(u, v, uscale, vscale, uoffset, voffset, 0);
    }
    else if (level < Real(pyramid.size() - 1))
    {
        int low_level = max((int)level, 0);
        int high_level = min(low_level + 1, (int)pyramid.size() - 1);
        Real portion = level - low_level;
        return Lookup(u, v, uscale, vscale, uoffset, voffset, low_level) * (1 -
portion) +
            Lookup(u, v, uscale, vscale, uoffset, voffset, high_level) *
portion;
    }
    else
    {
        return Lookup(u, v, uscale, vscale, uoffset, voffset,
(int)pyramid.size() - 1);
    }
}
```

### Texture

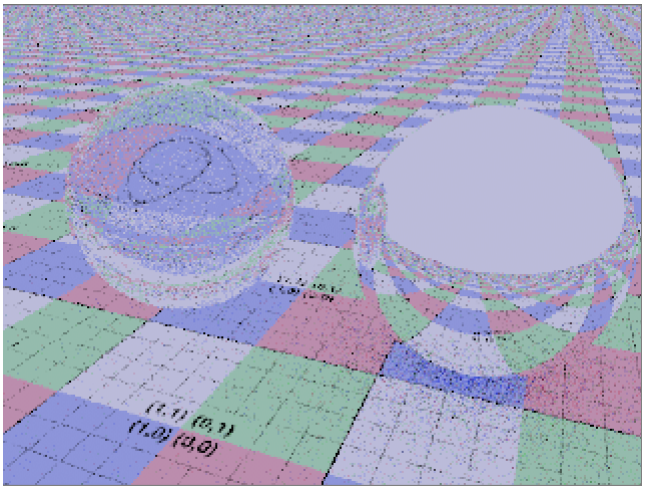
We include the `Mipmap` within `Texture`. So actually every `Texture` will have a unique Mipmap, which is initiated at the begin of construction function. When we hit a object with texture, we look up it's mipmap to get the surface color. And calculate the level by footprint.

```
struct Texture {
    ParsedImageTexture parsedImageTexture;
    MipMap<Vector3> mipMap;
    int width = 0, height = 0;

    Vector3 GetColor(Real u, Real v, Real level)
    {
        return mipMap.Lookup(u, v, parsedImageTexture.uscale,
parsedImageTexture.vscale,
            parsedImageTexture.uoffset, parsedImageTexture.voffset, level);
    }

    Real GetLevel(Real footprint)
    {
        Real scaled_footprint = footprint * max(width, height) *
            max(parsedImageTexture.uscale, parsedImageTexture.vscale);
        Real level = log2(max(scaled_footprint, (Real)1e-9f));
        return level;
```

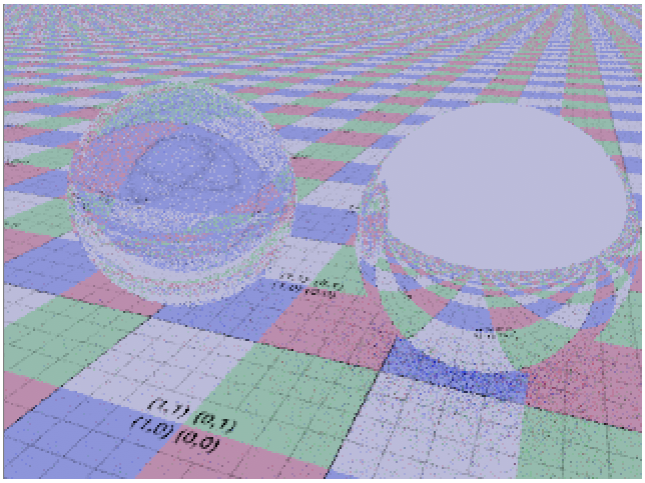
```
    }
}
```

## Experiments & Result

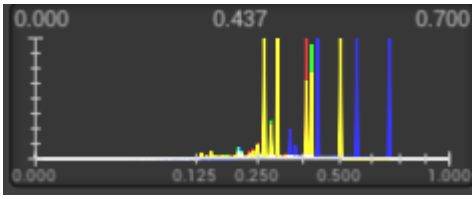
Here we use choose `spp=2` to make the difference clear. As we can see, there are fewer noisy and alias when we use mipmap.



Without Mipmap 🔼, with Mipmap 🔽

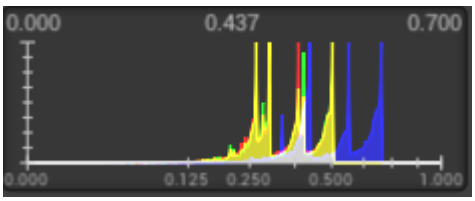


And we can also check the histogram of the results.




w/o Mipmap 🔼

with Mipmap 🔽



# 3. Environment Map

## Idea

If we warp our scene in a huge sphere, which emit lights from every directions, we can use it to represent the lighting environment. So basically, we need to stick the environment texture in a infinite light sphere. And we do importance sampling on the 2D distribution of the environment map's luminance. It makes sense, since the sun in a environment map will have the major contribution to a shading point then other light source.

## Infinite area light

We add some code to parse infinite area light as `envmap`. It include a `Texture`, a scale of light intensity `intensity_scale`, a transform matrix, and a 2D distribution table `Distribution2D`.

```
struct ParsedEnvMap
{
    Texture texture;
    Matrix4x4 light_to_world, world_to_light;
    Distribution2D dist_2d;
    Real intensity_scale = 1.;
    ParsedEnvMap(ParsedImageTexture parsed_ImageTexture, Matrix4x4
light_to_world, Real                  intensity_scale){};
    Vector2 GetUV(Vector3 direction){};
};
```

Here is a xml format example

```
<emitter type="envmap">
    <float name="intensity_scale" value="1"/>
    <texture type="bitmap" name="reflectance">
        <string name="filename" value="graveyard_pathways_1k.exr"/>
        <float name="uscale" value="1"/>
        <float name="vscale" value="1"/>
    </texture>
    <transform name="toWorld">
        <rotate angle="0" x="1"/>
    </transform>
</emitter>
```

## Distribution

Here we regard the environment map as a 2D table of luminance, where we want to apply importance sample based on the luminanceof different $(u, v)$. Here we firstly sample the row (v axis), and then sample the corresponding  value within the row (u axis).

### Distribution1D

In the `Distribution` structure, there are two vector: `pdf` and `cdf`. Firstly, the construction function get a list of probablity `func`, which is the PDF of that variable. Then we accumulate the PDF to figure out the CDF.

To inverse transform sample a point given a random variable generator `rng`, we return the lowwer bound of CDF. Notice that in `GetPDF`, we do not return the $p \in [0, 1]$. Instead, we return the originally corresponding value in `func`. That will make the importance sampling easier.

```
struct Distribution1D
{
    std::vector<Real> pdf, cdf;
    Real funcIntegral;
    Distribution1D() = default;
    Distribution1D(std::vector<Real>& func)
    {
        pdf = func;
        cdf.resize(func.size() + 1);
        cdf.push_back(0.);
        for (int i = 0; i < (int)func.size(); i++)
        {
            cdf[i + 1] = cdf[i] + pdf[i];
        }
        funcIntegral = cdf[func.size()];
        for (int i = 0; i < (int)func.size(); i++)
        {
            cdf[i] /= funcIntegral;
        }
        cdf[func.size()] = 1.;
    }
    int sample(pcg32_state& rng)
    {
        Real random_p = next_pcg32_real<Real>(rng);
        int index = 0;
        index = (int)(std::lower_bound(cdf.begin(), cdf.end(), random_p) -
cdf.begin())-1;
        return index;
    }
    Real GetPDF(int index)
    {
        return pdf[index];
    }
};
```

### Distribution2D

In the `Distribution2D` structure, we use a `Distribution1D`: `marginal_rows` to store the marginal probability of every row(v axis). And we form a 2D table by a list of `Distribution1D`: `conditional_v`.

First of all, given a 1D vector `func` and it `width` and `height` we push them back to `conditional_v` row by row. After that, we get the sum of each row `funcIntegral`, and put them into `marginal_rows`. There the `sinTheta` is just used to correct the distortion, and actually will have no effect on the final probability.

To inverse transform sample a 2D point given a random variable generator `rng`, we will sample `marginal_rows` first, to choose the index of a row(v axis), and then sample `conditional_v[row]` to sample the proper column(u axis).

```
struct Distribution2D
{
    // marginal_rows: marginal p(v)  each row.
    std::shared_ptr<Distribution1D> marginal_rows;
    // conditional_v: conditional p(u|v) for each(u,v)
```

```
    std::vector<std::shared_ptr<Distribution1D>> conditional_v;
    int width, height;
    Distribution2D(const std::vector<Real>& func, int width, int height)
        :width(width), height(htight)
    {
        for(int v = 0; v < height; v++)
        {
            std::vector<Real> row = { func.begin() + v*width, func.begin() +
(v+1)*width-1 };
            conditional_v.emplace_back(new Distribution1D(row))
        }
        std::vector<Real> marginal_func;
        for(int v = 0; v < height; v++)
        {
            marginal_func.push_back(conditional_v[v]->funcIntegral);
        }
        marginal_rows = std::make_shared<Distribution1D>(marginal_func);
    }
    Vector2i sample(pcg32_state& rng)
    {
        int row_index = marginal_rows.sample(rng);
        return { conditional_v[row_index].sample(rng), row_index};
    }
    Real GetPDF(Vector2i uv)
    {
        int u = uv[0];
        int v = uv[1];
        return conditional_v[v]->pdf[u] / marginal_rows->funcIntegral;
    }
};
```

## PDF, light Direction, and Emission

### PDF

We want to get PDF to appl MC integral in Rendering Equation. So acutally we need to get the distribution of solid angle. Since $\theta = 2\pi u, \phi = \pi v$ on sphere coordinate, we get

$$p(w) = \frac{p(\theta, \phi)}{sin(\theta)} = \frac{p(u, v)}{2\pi^2 sin(\theta)}$$

Actually, here $(u, v) \in [0, 1]^2$, but actually our `Distribution2D` get $(u', v') \in [width, height]$, so $p(u, v) = \frac{p(u', v')}{width * height}$

```
ParsedEnvMap& envmap = std::get<ParsedEnvMap>(light);
Vector4 direc_light_4 = envmap.world_to_light * Vector4(normalize(direction),
0.);
Vector3 direc_light = normalize(Vector3(direc_light_4.x, direc_light_4.y,
direc_light_4.z));
Vector2 uv_real = envmap.GetUV(direc_light);
Vector2i uv_int = { (int)(uv_real[0] * (Real)envmap.texture.width),
(int)(uv_real[1] * (Real)envmap.texture.height) };
Real pdf = envmap.dist_2d.GetPDF(uv_int) / envmap.dist_2d.marginal_rows-
>funcIntegral;
pdf *= (Real)envmap.texture.width * (Real)envmap.texture.height;
Real sin_theta = sin(uv_real[1] * c_PI);
if (sin_theta < 0.)
return 0.;
Real pdf_wo = pdf / (2. * c_PI * c_PI * sin_theta);
PDF_value = pdf_wo;
```

**Direction**

Sample on the `Distribution2D` of the luminance, we get the $(u, v)$ and get the direction in the light coordinate. Transform it to the world coordinate is necessary.

```
ParsedEnvMap& envmap = std::get<ParsedEnvMap>(light_random);
Vector2i uv_int = envmap.dist_2d.sample(rng);
Vector2 uv_real = { (Real)uv_int[0] / (Real)envmap.texture.width,
                    (Real)uv_int[1] / (Real)envmap.texture.height };
Real theta = uv_real[1] * c_PI;
Real phi = uv_real[0] * 2. * c_PI;

Vector3 direc_light = { -cos(phi) * sin(theta),
                        cos(theta),
                        sin(phi) * sin(theta) };
Vector4 direc_world_4 = envmap.light_to_world * Vector4(direc_light, 0.);
Vector3 direc_world = normalize(Vector3(direc_world_4.x, direc_world_4.y,
direc_world_4.z));

return direc_world;
```

**Emission(based on Mipmap)**

`Intensity_scale` is used to scale the light intensity. Since we are sampling on a infinite sphere, we use $\frac{\partial dir}{\partial x}$ instead of $\frac{\partial point}{\partial x}$ to calculate the partial derivative between texture coordinate and film coordinate.

$$\frac{\partial u}{\partial x} = \frac{\partial dir}{\partial x} * \frac{\partial u}{\partial dir}$$

Since the direction $d = [-cos(\phi)sin(\theta), -cos(\theta), sin(\theta)sin(\phi)]^T$, we get

$$||\frac{\partial dir}{\partial u}|| = ||\frac{\partial u}{\partial dir}||^{-1} = 2\pi sin(\theta)$$

$$||\frac{\partial dir}{\partial v}|| = ||\frac{\partial v}{\partial dir}||^{-1} = \pi * 1$$

We then use it to calculate the footprint and to pick color from corresponding mipmap level.

```
Real footprint = (abs(1. / max(sin(theta), 1e-9)) / ((Real)envmap.texture.width
* 2. * c_PI) + 1. / ((Real)envmap.texture.height * c_PI)) / 2.;
Real level = envmap.texture.GetLevel(footprint);
Vector3 color = envmap.texture.GetColor(uv_real[0], uv_real[1], level);
return color * envmap.intensity_scale;
```

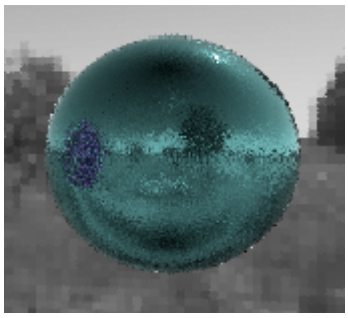
## Experiments & Result

Here is a beautiful rendering result. From the left to right: `Mirror`, purple `Plastic`, `glass` with `eta=1.5`, and cyan `blinn_microfacet` with `roughness=800`. This image is only lighted by environment map. We can see every sphere with different material get reasonable lighting, as the sun from the right up corner make more contribution than other light source, and the light is highlight result instead of scattered random light points sampled by BRDF sampling.

And also we take a experiment to find compare the sampling schemes. It is obviously that we successfully  done importance sampling based on the environment radiance.

Importance Sampling 🔼

Random Sapling 🔽



## 4.Final Result

Here is the reference image rendered by **Blender**. There might be little transformation between camera and object in our Torrey renderer.

Below is the gallery of our rendering results.

graveyard pathways 🔼

autumn 🔼

country road