Precomputed Radiance Transfer

Mengzhu Wang

2024 年 5 月 19 日

1 Overview

物体在不同光照下的表现不同，PRT（Precomputed Radiance Transfer）是一个计算物体在不同光照下表现的方法。光线在一个环境中，会经历反射，折射，散射，甚至还会物体的内部进行散射。为了模拟具有真实感的渲染结果，传统的 Path Tracing 方法需要考虑来自各个方向的光线、所有可能的传播形式并且收敛速度极慢。PRT 通过一种预计算方法，该方法在离线渲染的 Path Tracing 工具链中预计算 lighting 以及 light transport 并将它们用球谐函数拟合后储存，这样就将时间开销转移到了离线中。最后通过使用这些预计算好的数据，可以达到实时渲染严苛的时间要求，同时渲染结果可以呈现出全局光照的效果。

预计算部分是 PRT 算法的核心，也是其局限性的根源。因为在预计算 light transfer 时包含了 visibility 以及 cos 项，这代表着实时渲染使用的这些几何信息已经完全固定了下来（可以基于球谐函数的旋转性质让光源旋转起来）。所以 PRT 方法存在的限制包括：

- 不能计算随机动态场景的全局光照
- 场景中物体不可变动

2 Precompute SH Coefficient

SH 的展开式为一系列基函数 $B_i(x)$ 的线性组合 $f(x) = \sum_i c_i \cdot B_i(x)$，其中 $B_i(x)$ 已知，只需要求出每个基函数的系数。SH 系数的计算过程称为投影，也就是 product integral，其式为 $c_i = \int_\Omega f(\omega) B_i(\omega) d\omega$。

2.1 SH of Lighting

用球谐函数表示环境光，就需要将环境光投影到球谐函数上来得到对应的系数，即

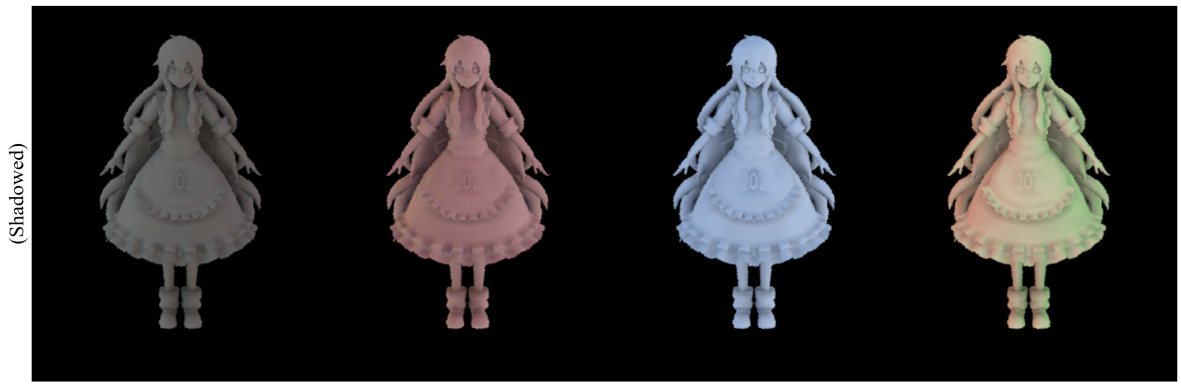
$$SH_{coeff} = \int_S L_{env}(\omega_i) SH(\omega_i) \mathrm{d}\omega_i$$

上式采用黎曼积分的方法计算，可得

$$\hat{SH}_{coeff} = \sum_i L_{env}(\omega_i) SH(\omega_o) \Delta\omega_i$$

ptr.cpp 的 ProjEnv::PrecomputeCubemapSH() 中，首先把六个 cubemap 的每个像素对应的单位方向向量保存在 cubemapDirs 中，从 cubemap 中获取 RGB 光照信息。然后使用CalcArea(u,v,width,height)函数计算 cubemap 上每个像素所代表的矩形区域投影到单位球面的面积。接着将单位方向向量传入 sh::EvalSH 求出每个球谐函数基函数的值。最后将 cubemap 上所有像素处的结果累加。

```
for (int i = 0; i < 6; i++)
{
    for (int y = 0; y < height; y++)
    {
        for (int x = 0; x < width; x++)
        {
            // TODO: here you need to compute light sh of each face of cubemap of each pixel
            // TODO: 此处你需要计算每个像素下 cubemap 某个面的球谐系数
            Eigen::Vector3f dir = cubemapDirs[i * width * height + y * width + x];
            int index = (y * width + x) * channel;
            Eigen::Array3f Le(images[i][index + 0], images[i][index + 1],
                              images[i][index + 2]);
            float delta = CalcArea(x, y, width, height);
            for (int l = 0; l <= SHOrder; l++)
                for (int m = -l; m <= l; m++) {
                    double basis_sh = sh::EvalSH(l, m, dir.cast<double>().normalized());
                    SHCoeffiecents[sh::GetIndex(l, m)] += Le * basis_sh * delta;
                }
        }
    }
}
```



(Shadowed)

Indoor GraceCathedral Skybox CornellBox

2.2 SH of Lighting Transport

光路传输方程为

$$L(x,\omega_o)=\int_S f_r(x,\omega_i,\omega_o)L_i(x,\omega_i)H(x,\omega_i)\mathrm{d}\omega_i$$

对于表面处处相等的漫反射表面，可以简化得到 Unshadowed 光照方程

$$L_{DU}=\frac{\rho}{\pi}\int_S L_i(x,\omega_i)\max(N_x\cdot\omega_i,0)\mathrm{d}\omega_i$$

由球谐函数性质,可以将积分项内的光照辐射度和传输函数项分开,各自计算球谐系数。`preprocess` 实现预计算过程,划分为环境光投影和传输投影两部分。调用 `ProjEnv::PrecomputeCubemapSH()` 预计算得到的环境光系数保存在 `m_LightCoeffs` 中，这是一个 $3\times(SHOrder+1)^2$ 的矩阵，每一列存储了环境光球谐系数的 RGB 值。预计算得到的传输项系数保存在 `m_TransportSHCoeffs` 中，这是一个 $(SHOrder+1)^2\times VertexCount$ 的矩阵，每一列 j 存储了第 j 个顶点的所有球谐系数。

预计算传输项的主要函数是 `sh::ProjectFunction(SHOrder, shFunc, m_SampleCount)`，该函数会取 lambda 函数返回的结果 `shFunc` 投影在基函数上得到系数。

对于 Unshadowed 情况，只需将 $M^{DU}=\max(N_x\cdot\omega_i,0)$ 投影到球谐系数里。对于有自阴影的 Shadowed 漫反射传输，预计算方程中增加可见性项变为

$$L_{DS}=\frac{\rho}{\pi}\int_S L_i(x,\omega_i)V(\omega_i)\max(N_x\cdot\omega_i,0)\mathrm{d}\omega_i$$

因此投影的传输项变为 $M^{DU}=V(\omega_i)\max(N_x\cdot\omega_i,0)$。

```
auto shFunc = [&](double phi, double theta) -> double {
    Eigen::Array3d d = sh::ToVector(phi, theta);
    const auto wi = Vector3f(d.x(), d.y(), d.z());
    double H = wi.normalized().dot(n.normalized()) / M_PI;
    if (m_Type == Type::Unshadowed)
    {
        // TODO: here you need to calculate unshadowed transport term of a given direction
        // TODO: 此处你需要计算给定方向下的 unshadowed 传输项球谐函数值
        if (H > 0.0)
            return H;
        return 0;
    }
    else
    {
        // TODO: here you need to calculate shadowed transport term of a given direction
        // TODO: 此处你需要计算给定方向下的 shadowed 传输项球谐函数值
        Ray3f ray(v, wi.normalized());
        if (H > 0.0 && !scene->rayIntersect(ray))
```

```
            return H;
        return 0;
    }
};
```

2.3 Run

在 Linux 中运行

```
mkdir build
cd build
cmake ..
make -j8
./nori ../scenes/ptr.xml
```

正确计算的环境光球谐函数系数保存在 cubemap 目录下的 `light.txt` 里，传输项球谐函数系数保存在 cubemap 目录下的 `transport.txt` 里。

3 Real-Time SH Lighting

预计算 lighting 以及 light transport 部分按照 txt 格式保存，在 `engin.js` 中的 88-114 行实现读取分割并保存为 Array。解析后的数据储存在了两个全局变量 `precomputeL`、`precomputeLT` 中，其中 `precomputeL` 将预计算环境光的 $SH\ vector$ 保存为一个长度为 SH 系数数量的 Array，该 Array 中每个元素都是代表着 RGB 的三维数组；`precomputeL` 将预计算 $light\ transfer$ 的 $SH\ vector$ 保存为一个长度为 $VertexCount * SHCoefficientCount$ 的 Array。

创建使用预计算数据的材质 `PRTMaterial`

```
class PRTMaterial extends Material {
    constructor(vertexShader, fragmentShader) {
        super({
            'uPrecomputeLR': { type: 'PrecomputeL', value: null },
            'uPrecomputeLG': { type: 'PrecomputeL', value: null },
            'uPrecomputeLB': { type: 'PrecomputeL', value: null },
        }, ['aPrecomputeLT'], vertexShader, fragmentShader, null);
    }
}

async function buildPRTMaterial(vertexPath, fragmentPath) {
    let vertexShader = await getShaderString(vertexPath);
    let fragmentShader = await getShaderString(fragmentPath);
```

```
    return new PRTMaterial(vertexShader, fragmentShader);
}
```

在 index.html 中引入 PRTMaterial 材质

```
<script src="src/materials/PRTMaterial.js" defer></script>
```

在 loadOBJ.js 中也加入 PRTMaterial 材质

```
switch (objMaterial) {
    // TODO: Add your PRTmaterial here
    case 'PRTMaterial':
        material = buildPRTMaterial("./src/shaders/prtShader/prtVertex.glsl",
                   "./src/shaders/prtShader/prtFragment.glsl");
    break;
}
```

在 engine.js 中加入使用 PRTMaterial 材质的模型

```
let maryTransform = setTransform(0, -35, 0, 20, 20, 20);
loadOBJ(renderer, 'assets/mary/', 'mary', 'PRTMaterial', maryTransform);
```

PRTMaterial 里定义了三个 uniform 变量 uPrecomputeLR 、uPrecomputeLG、uPrecomputeLB，分别对应 precomputeL 的 RGB 三个通道，每个有 9 个元素，即前三阶球谐函数工 9 个。预计算数据在 WebGLRenderer.js 中存储到 uniform 变量中。

```
let precomputeL_RGBMat3 = getMat3ValueFromRGB(precomputeL[guiParams.envmapId]);
if (k == 'uPrecomputeLR') {
    gl.uniformMatrix3fv(
        this.meshes[i].shader.program.uniforms[k],
        false,
        precomputeL_RGBMat3[0]);
}
if (k == 'uPrecomputeLG') {
    gl.uniformMatrix3fv(
        this.meshes[i].shader.program.uniforms[k],
        false,
        precomputeL_RGBMat3[1]);
}
if (k == 'uPrecomputeLB') {
    gl.uniformMatrix3fv(
```

```
        this.meshes[i].shader.program.uniforms[k],
        false,
        precomputeL_RGBMat3[2]);
}
```

编写材质对应的 shader 时，在 vertexShader 中通过对应 SH 系数之间点积并累加计算 vColor，接着把 vColor 传递到 fragmentShder 中插值后着色。

```
// prtVertex.glsl
attribute vec3 aVertexPosition;
attribute mat3 aPrecomputeLT;

uniform mat4 uModelMatrix;
uniform mat4 uViewMatrix;
uniform mat4 uProjectionMatrix;
uniform mat3 uPrecomputeLR;
uniform mat3 uPrecomputeLG;
uniform mat3 uPrecomputeLB;

varying highp vec3 vColor;

vec3 PRT() {
    vec3 res;
    for (int i = 0; i < 3; i++)
        for (int j = 0; j < 3; j++)
            res += vec3(uPrecomputeLR[i][j], uPrecomputeLG[i][j], uPrecomputeLB[i][j])
            * aPrecomputeLT[i][j];

    return res;
}

void main(void) {
    gl_Position = uProjectionMatrix * uViewMatrix * uModelMatrix *
                vec4(aVertexPosition, 1.0);
    vColor = PRT();
}
```

直接将 vColor 赋值给 gl_FragColor 的结果会偏暗，可以做 tone mapping 进行颜色校正。

```
// prtFragment.glsl
#ifdef GL_ES
precision mediump float;
```

```
#endif

varying highp vec3 vColor;

vec3 toneMapping(vec3 color){
    vec3 result;

    for (int i=0; i<3; ++i) {
        if (color[i] <= 0.0031308)
            result[i] = 12.92 * color[i];
        else
            result[i] = (1.0 + 0.055) * pow(color[i], 1.0/2.4) - 0.055;
    }

    return result;
}

void main(void) {
    gl_FragColor = vec4(toneMapping(vColor), 1.0);
}
```