Real-Time Shadows

Mengzhu Wang

2024 年 5 月 16 日

1 Overview

实时渲染中有一个重要的约等式，只要满足在 $g(x)$ 的积分域很小或 $g(x)$ 足够光滑（变化不大）就近似准确，

$$\int_\Omega f(x)g(x)\mathrm{d}x \approx \frac{\int_\Omega f(x)\mathrm{d}x}{\int_\Omega \mathrm{d}x} \cdot \int_\Omega g(x)\mathrm{d}x$$

那么根据渲染方程的定义也可以近似得到

$$\begin{aligned} L_o(p,\omega_o) &= \int_{\Omega^+} L_i(p,\omega_i) f_r(p,\omega_i,\omega_o) V(p,\omega_i)\cos\theta_i \mathrm{d}\omega_i \\ &\approx \frac{\int_{\Omega^+} V(p,\omega_i)\mathrm{d}\omega_i}{\int_{\Omega^+}\mathrm{d}\omega_i} \cdot \int_{\Omega^+} L_i(p,\omega_i) f_r(p,\omega_i,\omega_o)\cos\theta_i \mathrm{d}\omega_i \end{aligned}$$

渲染方程后半部分就是初始代码框架中没有加阴影的 Blinn-Phong 模型，前半部分是需要实现的阴影方法。在实现了阴影算法，得到可见性信息后，将两部分相乘得到最终的着色结果。

对于场景中的每个物体都会默认附带一个 `ShadowMaterial` 材质，并会在调用 `loadOBJ` 时添加到 `WebGLRenderer` 的 `shadowMeshes[]` 中。当 `WebGLRenderer.js` 中的光源参数 `hasShadowMap` 为 true 时，开启 Shadow Map 功能，在正式渲染场景之前会以 `ShadowMaterial` 材质先渲染一遍 `shadowMeshes[]` 中的物体，从而生成所需的 Shadow Map。

2 Two Pass Shadow Map

Two pass shadow map 即对场景渲染两次

- 第一个 pass 从光源看向场景，记录深度图 shadow map。
- 第二个 pass 从相机看向场景，像素对应的世界坐标通过光源 MVP 变换，转换到光源空间中，它的深度和 shadow map 中对应深度比较。若 shadow map 的深度更小，说明该像素被遮挡产生阴影。

`ShadowMaterial` 使用 `Shader/shadowShader` 文件夹下的顶点和片元着色器。由于第一个 pass 从光源看向场景，在 `ShadowMaterial.js` 中需要向 `Shader` 传递 `uLightMVP` 矩阵。

光源 MVP 矩阵由 `DirectionalLight` 中的 `CalcLightMVP(translate, scale)` 函数实现。

```
CalcLightMVP(translate, scale) {
    let lightMVP = mat4.create();
    let modelMatrix = mat4.create();
    let viewMatrix = mat4.create();
    let projectionMatrix = mat4.create();

    // Model transform
    mat4.translate(modelMatrix, modelMatrix, translate);
    mat4.scale(modelMatrix, modelMatrix, scale);
    // View transform
    mat4.lookAt(viewMatrix, this.lightPos, this.focalPoint, this.lightUp);
    // Projection transform
    mat4.ortho(projectionMatrix, -150, 150, -150, 150, 0.01, 500);

    mat4.multiply(lightMVP, projectionMatrix, viewMatrix);
    mat4.multiply(lightMVP, lightMVP, modelMatrix);

    return lightMVP;
}
```

ShadowMaterial中加入了缓冲对象 light.fbo，那么后续在 WebGLRenderer 的 render() 调用 draw() 时，需要判断 frameBuffer 是否为空决定画在缓冲中还是屏幕上，可知 shadow map 画在 frameBuffer 中。PhongMaterial 也只需要从 light.fbo 中就能得到 shadow map。

第二个 pass 与不带阴影的渲染类似，只不过多了 visibility 项。该过程主要在 phongFragment.glsl 实现，其中的 useShadowMap 函数负责查询当前着色点在 shadow map 上记录的深度值，并与转换到 light space 的深度值比较后返回 visibility 项。这里的的查询坐标需要先转换到 NDC 标准空间 $[0, 1]$。

```
vec3 shadowCoord = (vPositionFromLight.xyz / vPositionFromLight.w + 1.0) / 2.0;
```

查询时通过 unpack 将 RGBA 转换为 float 的深度值，对应生成 shadow map 时在 shadowFragment.glsl 中的 pack 将 float 的深度值转为 RGBA 的操作。该操作可以增加精度。因为深度信息是一个连续的浮点数，它的范围和精度可能超出了一个 8 位通道所能提供的，通过分配深度值的不同部分到 RGBA 四个通道，可以用更高的精度来存储深度值，需要使用时从这四个通道解码即可。

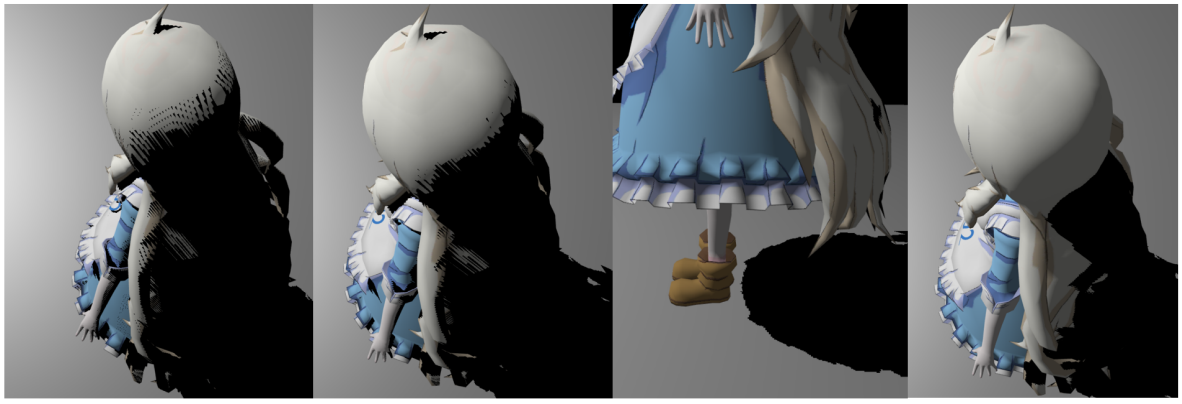
```
float useShadowMap(sampler2D shadowMap, vec4 shadowCoord){
  float shadowMapDepth = unpack(texture2D(shadowMap, shadowCoord.xy));
  return shadowCoord.z < shadowMapDepth + EPS? 1.0 : 0.0;
}
```

shadow map 存在自遮挡（self occlusion）问题。由于数值精度的限制且 shadow map 保存的是离散的深度值，也就是说 shadow map 上的每个采样点代表一块范围内图元的深度值，光线斜射到

平面时，实际相当于发射到多个垂直的小平面上，因此第二个 pass 时 shadow map 的深度可能会略小于物体表面的深度，从而误算为阴影，出现 shadow acne。解决该问题的一个方法是在比较深度时增加一个 bias，认为只有当前点的深度大 shadow map 一定阈值后才判断是阴影。

上述代码中的 EPS 就是给 shadow map 记录的深度加上 bias，能减少 shadow acne。bias 为常数时过小效果不佳，过大又会导致 detached shadow/peter panning。更好的一种方法是动态调整 bias，将 bias 与入射角关联，入射角越大，bias 越大，称为 normal bias。

```
float m = 300.0 / 2048.0 / 2.0; // frustumSize/shadowMapSize/2
float bias = max(m, (1.0 - dot(normalize(uLightPos), normalize(vNormal))) * m) * 0.15;
```



bias=0 bias=1e-3 perter panning normal bias

3 PCF (Percentage Close Filter)

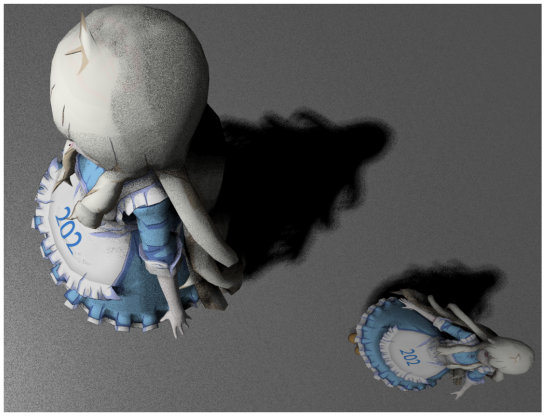
PCF 最初用于阴影边缘的抗锯齿反走样。PCF 不是直接对第一次 pass 得到的 shadow map 做滤波，因为对其滤波再比较，结果仍是二值化数据；也不是对两次 pass 后得到的存在锯齿的着色结果滤波，这会导致模糊。PCF 的滤波对象是 shading point 的深度和对应 shadow map 深度的比较得到的二值图。

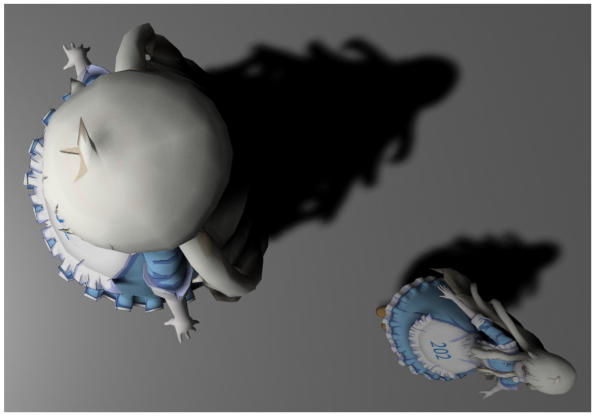
PCF 考虑采样一定范围的内的点而非单个点，对其周围一定范围内的 shading point 和对应点的 shadow map 深度值进行比较，得到二值结果，然后求平均作为当前点的 visibility，实现软化阴影的效果。filter size 越大，阴影越软。

PCF 在 `phongFragment.glsl` 中实现，它采用一个圆盘滤波核进行随机采样，将坐标偏移值存储在数组 `poissonDisk` 中。随机采样函数的质量将与最终渲染效果的好坏息息相关，根据结果可知泊松圆盘采样优于均匀圆盘采样。

```
float PCF(sampler2D shadowMap, vec4 coords, float filterSize) {
  float block = 0.0;
  //uniformDiskSamples(coords.xy);
  poissonDiskSamples(coords.xy);
  for (int i = 0; i < NUM_SAMPLES; i++) {
```

```
    float shadowMapDepth = unpack(texture2D(shadowMap, coords.xy +
                                  poissonDisk[i] * filterSize / 2048.0));
    if (shadowMapDepth + EPS < coords.z) block++;
  }
  return 1.0 - block / float(NUM_SAMPLES);
}
```





uniformDiskSamples poissonDiskSamples

4 PCSS (Percentage Closer Soft Shadow)

PCSS 的目标是使得得到的阴影在和遮挡物距离近的地方偏向于硬阴影，在和遮挡物距离远的地方偏向于更为模糊的软阴影。PCSS 的步骤可以总结为

- Blocker search (getting the average blocker depth in a certain region)
- Penumbra estimation (use the average blocker depth to determine filter size)
- PCF

`phongFragment.glsl` 中 `findBlocker` 函数计算遮挡物的平均深度。对 shadow map 的一定范围做局部深度测试，找到范围内的 blocker 并计算平均深度，作为当前 blocker 的深度。

```
float findBlocker(sampler2D shadowMap, vec2 uv, float zReceiver ) {
  poissonDiskSamples(uv);
  float depth = 0.0;
```

```
  int cnt = 0;
  for (int i = 0; i < NUM_SAMPLES_BLOCKER_SEARCH; i++) {
    float zBlocker = unpack(texture2D(shadowMap, uv + poissonDisk[i]
                    * 10.0 / 2048.0));
    if (zBlocker + EPS < zReceiver) {
      depth += zBlocker;
      cnt++;
    }
  }
  return cnt == 0? 1.0 : depth / float(cnt);
}
```

接着在 PCSS 函数中调用 findBlocker，利用相似三角形计算伴影直径并传递给 PCF 函数以调整其滤波核大小。伴影计算公式如下：

$$w_{Penumbra} = (d_{Receiver} - d_{Blocker}) \cdot w_{Light} / d_{Blocker}$$

```
float PCSS(sampler2D shadowMap, vec4 coords){
  // STEP 1: avgblocker depth
  float zBlocker = findBlocker(shadowMap, coords.xy, coords.z);
  // STEP 2: penumbra size
  float penumbraSize = (coords.z - zBlocker) * LIGHT_WIDTH / zBlocker;
  // STEP 3: filtering
  return PCF(shadowMap, coords, penumbraSize);
}
```