Lab-2

图形绘制技术

Advanced Texture Techniques

纹理的采样、映射以及生成

Lab2-高级纹理技术

2024 春季学期

1. 光线微分

1.1 简介

纹理过滤是渲染中常用的方法，其在光栅化渲染和光线追踪渲染中都有应用，用于减少使用低频信号对高频信息进行采样时出现的走样。

我们可以通过一个较为直观的方法理解为什么要做纹理过滤。在单向的光线追踪过程中，一个像素的值是通过在这个像素中均匀采样光线，获取 radiance 之后将其进行平均得到的。我们假设整个场景由一个恒定的环境光源照亮，并且物体表面为均匀反射材质。在这个情况下，由于光源或采样导致的噪点会消失，但是由纹理导致的噪点依然存在，这被称为纹理走样（Texture Aliasing）。

我们假设一个黑白交错的棋盘格摆放在相机正前方，棋盘格的颜色为完全的黑色或白色。棋盘格中棋盘的密度很高，以至于每个像素中可能包含多个格子。如果采样率足够高，那么理想的状态下，在相机中每一个像素的颜色都是黑白之间的某个灰色。在没有纹理采样的情况下，我们需要从每一个像素中发射多个光线，每次得到的材质信息都是黑色或者白色，进而对得到的黑色和白色进行平均，最终收敛至某种灰色；在有纹理过滤的情况下，每次得到的材质信息本身就为灰色，因此这一过程可以收敛得更快。

进而我们就可以理解最优的纹理过滤方法：将像素的边缘（一个正方形）投影至物体表面，由于物体表面的几何没有限制，因此投影之后正方形的直线可能变为曲线。考虑被这个投影后的正方形所覆盖的所有像素，将其平均即为最优的纹理滤波方式。

所以，一切的纹理过滤方法都是为了逼近上面提到的这一最优纹理过滤方法。注意以上论述在数学上并不严谨，仅仅为了方便理解。然而，在绝大多数情况下，由于几何本身的复杂性，这一最优的纹理过滤方法并不可能实现。因此，我们需要通过一些启发式的方法去逼近这一最优纹理过滤方法。

光线微分（Ray Differential）方法在离线渲染中很常用。我们通过光线微分方法产生一些数据，这些数据会决定纹理上滤波核的大小。其基本过程如下：

1. 在求交过程中，获取“当像素点坐标 (x, y) 偏移 1 时，空间中坐标 p 的偏移量”（$\partial p/\partial x$ 和 $\partial p/\partial y$），这一偏微分一般通过近似方法求解；
2. 对于特定的几何，获取“当纹理坐标 (u, v) 偏移 1 时，空间中坐标 p 的偏移量”（$\partial p/\partial u$ 和 $\partial p/\partial v$），这一偏微分对于不同的几何处理方式不同，**是本次实验需要完成的目标**；
3. 通过上述偏微分求解“当像素点坐标 (x, y) 偏移 1 时，纹理坐标 (u, v) 的偏移量”（$\partial u/\partial x$，$\partial u/\partial y$，$\partial v/\partial x$，$\partial v/\partial y$）；
4. 根据 $\partial u/\partial x$，$\partial u/\partial y$，$\partial v/\partial x$，$\partial v/\partial y$，计算最终的滤波核大小。滤波核可以是正方形的（双线性/三线性过滤），也可以是其他形状。在正方形滤波核的情况下，滤波核大小一般设置为上述四个偏微分中的最大值。

为了叙述的方便，上述描述大量使用了“当……偏移 1 时”，而不是使用“向量……关于变

量……的变化率”，后者的表述较为模糊。我们假设所有的偏微分在局部为常数，因此使用了这一表述。

上述光线微分过程在 Moer-lite 中的实现如下：

```
inline void computeRayDifferentials(Intersection *intersection,
                                    const Ray &ray) {
  // 计算光线微分
  do {
    if (ray.hasDifferentials) {
      Point3f p = intersection->position;
      Vector3f n = intersection->normal;
      // ray.originX：从坐标为(x+1,y)的像素中发射的光线rx的起点
      Vector3f ox = Vector3f{ray.originX[0], ray.originX[1], ray.originX[2]};
      // ray.originY：从坐标为(x,y+1)的像素中发射的光线ry的起点
      Vector3f oy = Vector3f{ray.originY[0], ray.originY[1], ray.originY[2]};

      // 由于时间开销的原因，rx与ry并不实际与物体求交
      // n为求交获得的法向量，假设物体表面为平面，近似得到rx和ry与表面的交点
      float d = dot(n, Vector3f{p[0], p[1], p[2]});
      // ray.directionX：光线rx的方向
      float tx = -(dot(n, ox) - d) / dot(n, ray.directionX);
      if (std::isinf(tx) || std::isnan(tx))
        break;
      // ray.directionY：光线ry的方向
      float ty = -(dot(n, oy) - d) / dot(n, ray.directionY);
      if (std::isinf(ty) || std::isnan(ty))
        break;

      // 获取rx和ry与平面的近似交点
      Point3f px = ray.origin + tx * ray.directionX;
      Point3f py = ray.origin + ty * ray.directionY;

      // 由于dx和dy=1，因此dpdx = (px - p) / 1.0， dpdy = (py - p) / 1.0，1.0被省略
      intersection->dpdx = px - p;
      intersection->dpdy = py - p;

      // 通过上述偏微分求解dudx，dudy，dvdx，dvdy
      // 三个方程求解两个未知数存在冗余，从三个方程中挑选两个
      int dim[2];
      if (std::abs(n[0]) > std::abs(n[1]) && std::abs(n[0]) > std::abs(n[2])) {
        dim[0] = 1;
        dim[1] = 2;
      } else if (std::abs(n[1]) > std::abs(n[2])) {
        dim[0] = 0;
        dim[1] = 2;
      } else {
        dim[0] = 0;
        dim[1] = 1;
      }

      // 获取几何的dpdu和dpdv
      Vector3f dpdu = intersection->dpdu;
      Vector3f dpdv = intersection->dpdv;

      Vector3f dpdx = intersection->dpdx;
      Vector3f dpdy = intersection->dpdy;
      float A[2][2] = {{dpdu[dim[0]], dpdv[dim[0]]},
```

```
                        {dpdu[dim[1]], dpdv[dim[1]]};
      float Bx[2] = {dpdx[dim[0]], dpdx[dim[1]]};
      float By[2] = {dpdy[dim[0]], dpdy[dim[1]]};

      // B = A * x
      auto solveLinearSystem2x2 = [](const float A[2][2], const float B[2],
                                     float *x0, float *x1) {
        float det = A[0][0] * A[1][1] - A[0][1] * A[1][0];
        if (std::abs(det) < 1e-10f)
          return false;
        *x0 = (A[1][1] * B[0] - A[0][1] * B[1]) / det;
        *x1 = (A[0][0] * B[1] - A[1][0] * B[0]) / det;
        if (std::isnan(*x0) || std::isnan(*x1))
          return false;
        return true;
      };

      float dudx, dvdx, dudy, dvdy;
      if (!solveLinearSystem2x2(A, Bx, &dudx, &dvdx))
        dudx = dvdx = .0f;
      if (!solveLinearSystem2x2(A, By, &dudy, &dvdy))
        dudy = dvdy = .0f;

      intersection->dudx = dudx;
      intersection->dudy = dudy;
      intersection->dvdx = dvdx;
      intersection->dvdy = dvdy;

      return;
    }

  } while (0);

  // 处理特殊情况
  ...
}
```

实现的前半部分主要涉及光线与平面的求交，同学们可以自行推导。下面将对实现的后半部分，也就是线性方程组求解做出解释。同样地，我们假设所有的偏微分在局部为常数，因此考虑严谨性，推导中的部分等号应替换为约等号。同时，推导与代码中会混用微分符号和偏微分符号。

以 p_x 为例，我们想要得到 p_x 的纹理坐标 (u_x, v_x) 与交点 p 的纹理坐标 (u, v) 的差值 $(\Delta u_x, \Delta u_y)$。这一差值可以通过以下方程求解：

$$p_x = p + \Delta u_x \frac{\partial p}{\partial u} + \Delta v_x \frac{\partial p}{\partial v} \tag{1}$$

进行简单变形有：

$$p_x - p = \Delta u_x \frac{\partial p}{\partial u} + \Delta v_x \frac{\partial p}{\partial v} \tag{2}$$

因为 p_x 从 $(x+1, y)$ 像素中发射，我们认为有 $p_x - p = \frac{\partial p}{\partial x}$。

同时，我们假设 $\frac{\partial u}{\partial x}$ 和 $\frac{\partial v}{\partial x}$ 在局部几何上为常数，因此有 $\frac{\Delta u_x}{\Delta x} = \frac{\partial u}{\partial x}$ 和 $\frac{\Delta v_x}{\Delta x} = \frac{\partial v}{\partial x}$。因为 $\Delta x = 1$，因此有 $\Delta u_x = \frac{\partial u}{\partial x}$ 和 $\Delta v_x = \frac{\partial v}{\partial x}$。注意我们需要求解的最终目标在这里出现。

进行替换得：

$$\frac{\partial p}{\partial x} = \frac{\partial u}{\partial x}\frac{\partial p}{\partial u} + \frac{\partial v}{\partial x}\frac{\partial p}{\partial v} \tag{3}$$

变为矩阵形式：

$$\begin{bmatrix} \partial p/\partial x_x \\ \partial p/\partial x_y \\ \partial p/\partial x_z \end{bmatrix} = \begin{bmatrix} \partial p/\partial u_x & \partial p/\partial v_x \\ \partial p/\partial u_y & \partial p/\partial v_y \\ \partial p/\partial u_z & \partial p/\partial v_z \end{bmatrix} \begin{bmatrix} \partial u/\partial x \\ \partial v/\partial x \end{bmatrix} \tag{4}$$

注意 ∂x 中 x 为屏幕坐标，而 $\partial p/\partial x_x$ 中的下标 x 指代三维空间坐标。使用三个方程求解两个未知数存在冗余，因此我们通过一些启发式的方法从三个方程中选择两个，进而通过标准的线性方程组求解方法得到 $\partial u/\partial x$ 和 $\partial v/\partial x$。对于 $\partial u/\partial y$ 和 $\partial v/\partial y$，有类似的做法。

本次实验需要完成的部分与上述推导类似。在三角形上，为了求解 $\partial p/\partial u$ 和 $\partial p/\partial v$（注意求解目标的变化），有类似的等式：

$$p_i = p_0 + \Delta u\frac{\partial p}{\partial u} + \Delta v\frac{\partial p}{\partial v} \tag{5}$$

其中三角形的三个顶点为 p_0, p_1, p_2，$i = 1, 2$，p_0 为三角形上随机选取的一个顶点。你需要通过类似的推导方式在三角网格的求交过程中获取 $\partial p/\partial u$ 和 $\partial p/\partial v$。注意这里只有两个方程存在，因此你**不需要**从方程组中剔除冗余方程。

提示：上述代码中的 p_x 和 p_y 均为“伪坐标”，换句话说，它们通过一些近似方法得到，因此无法保证它们在物体的表面，进而无法查询它们的纹理坐标。因此，我们需要使用上述“曲折”的方法去估计纹理坐标 (u, v) 的变化率。然而，对于某个三角网格，我们可以确定任意三角形的三个点都在物体表面，同时这些顶点在原始的模型文件中就拥有纹理坐标。solveLinearSystem2x2 方法已经给出，你可以将其复用。

1.2 需要完成的部分

光线微分的绝大部分内容已经实现，你只需要在三角网格上求解 $\partial p/\partial u$ 和 $\partial p/\partial v$ 即可。

首先，你需要在 Ray.h 的代码中手动开启光线微分功能，在当前的代码仓库中，这一功能默认关闭。

```
struct Ray {
  ...
  //* 光线微分
  bool hasDifferentials = true;
  Point3f originX, originY;
  Vector3f directionX, directionY;
};
```

下一步，你需要在 Triangle.cpp 中填写 intersection 中的 dpdu 和 dpdv：

```
void TriangleMesh::fillIntersection(float distance, int primID, float u,
                                    float v, Intersection *intersection) const {
  intersection->distance = distance;
  intersection->shape = this;
  //* 在三角形内部用插值计算交点、法线以及纹理坐标
  ...

  //* 计算交点
  ...

  //* 计算法线
  ...

  //* 计算纹理坐标
  ...

  // TODO 计算交点的切线和副切线
  ...

  // 计算偏微分dpdu和dpdv
  // your code here ...

  intersection->dpdu=...
  intersection->dpdv=...
}
```

为了观察实验的效果，在测试时，你需要在 Mipmap.cpp 中强制开启双线性滤波：

```
Vector3f MipMap::lookUp(Vector2f uv, Vector2f duv0, Vector2f duv1) const {
  ...

  // return texel(0, x, y); // force no filter
  return bilinear(0, uv); // force bilinear

  ...
}
```

1.3 测试

实验提供了场景 texture。在强制使用双线性滤波时，你得到的结果如下。

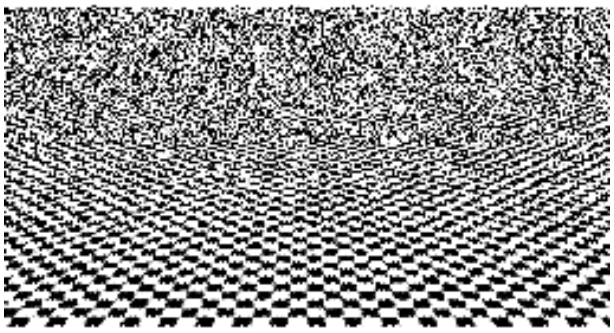

图 1: 双线性滤波结果

在你完成了光线微分之后，得到的结果如下。

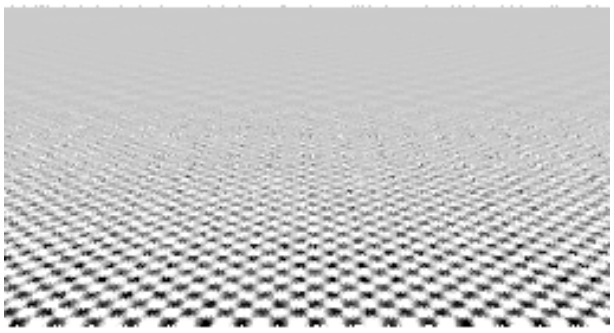

图 2: 光线微分结果

请注意，修改场景文件中的 spp 会直接影响渲染效果，因此在此次实验的过程中，spp 不应该被修改。

2. 纹理映射

在这一部分的实验中，你需要实现多种类型的纹理映射方法。

这一部分内容较为简单，只涉及坐标系变换。在完成这一部分实验的过程中，你不需要考虑光线微分和 mipmap 的相关内容，将光线微分功能关闭即可。当然，你也可以通过链式法则自行推导相关内容，并更新相关的偏微分值。如果你完成了这一部分内容，请在文档中体现。

2.1 球面纹理映射

球面纹理映射忽略模型文件自带的纹理坐标，转而使用着色点所在的位置获取纹理坐标。假设球面投影所使用的球面中心为 $(0,0,0)$，当前着色点的世界坐标为 p，球面投影首先计算归一化的方向向量 $\vec{p}$:

$$\vec{p} = \frac{p-(0,0,0)}{|p-(0,0,0)|} \tag{6}$$

接着计算其俯仰角 ϕ 和方位角 θ:

$$\phi = \arccos(\vec{p}_z), \theta = \arctan\frac{\vec{p}_y}{\vec{p}_x} \tag{7}$$

注意这里隐式地规定了球面的坐标系。最后将 θ 和 ϕ 映射至纹理坐标 (u,v):

$$u = \frac{\theta}{2\pi}, v = \frac{\phi}{\pi} \tag{8}$$

在此之后，你需要额外处理反三角函数的周期性，确保获取的纹理坐标在 $[0,1]$ 之间。

2.2 圆柱纹理映射

与球面纹理映射相似，圆柱纹理映射也使用着色点所在位置获取纹理坐标。其过程与球面纹理映射极为相似，仅有最后的映射不同:

$$u = \frac{\theta}{2\pi}, v = \begin{cases} \vec{p}_z & \vec{p}_z \geq 0 \\ \vec{p}_z + 1 & \vec{p}_z < 0 \end{cases} \tag{9}$$

2.3 平面纹理映射

平面纹理映射将物体表面上的某一点映射至任意的二维线性空间，在这个空间内的坐标作为纹理坐标。我们使用一组线性无关的基向量定义这个二维线性空间:

$$\vec{v}_\alpha, \vec{v}_\beta \tag{10}$$

计算三维点投影到定义的二维线性空间的坐标：

$$u' = \vec{v}_\alpha \cdot p, v' = \vec{v}_\beta \cdot p \tag{11}$$

这里使用了向量点乘的几何意义。为了得到真正的纹理坐标 (u, v)，你需要将这两个数字映射至 $[0, 1]$ 区间。你可以通过减去其整数部分以及加上其负数部分的方法进行映射。举例来说，1.7 和-0.3 都应该被映射至 0.7。

在本次实验中，我们定义的基向量为：

$$\begin{cases} \vec{v}_\alpha = (1, 0, 0) \\ \vec{v}_\beta = (0, 1, 0) \end{cases} \tag{12}$$

你也可以定义任意的基向量，并且测试纹理映射的效果。

2.4 需要完成的事情

你需要根据以上公式修改 Texture.cpp 中的方法：

```
TextureCoord UVMapping::map(const Intersection &intersection) const {
  // your code here ...
}
```

intersection 中保存了你需要的全部信息。你需要返回一个 TextureCoord 对象，如果你对链式法则推导没有兴趣，则只需要填写其中的 coord 即可。

此次实验的代码在完成实验后即删除，并恢复至原始代码。如果你想要将其长久保存在你的 Moer-lite 中，可以考虑继承 TextureMapping 类，通过 json 初始化这些新的类。

2.5 测试

实验提供了场景 bunny-texture。三种纹理映射的结果应如图所示。

图 3: 球面纹理映射结果

图 4: 圆柱纹理映射结果

图 5: 平面纹理映射结果

3. 过程式纹理：木纹

3.1 Perlin 噪声与木纹

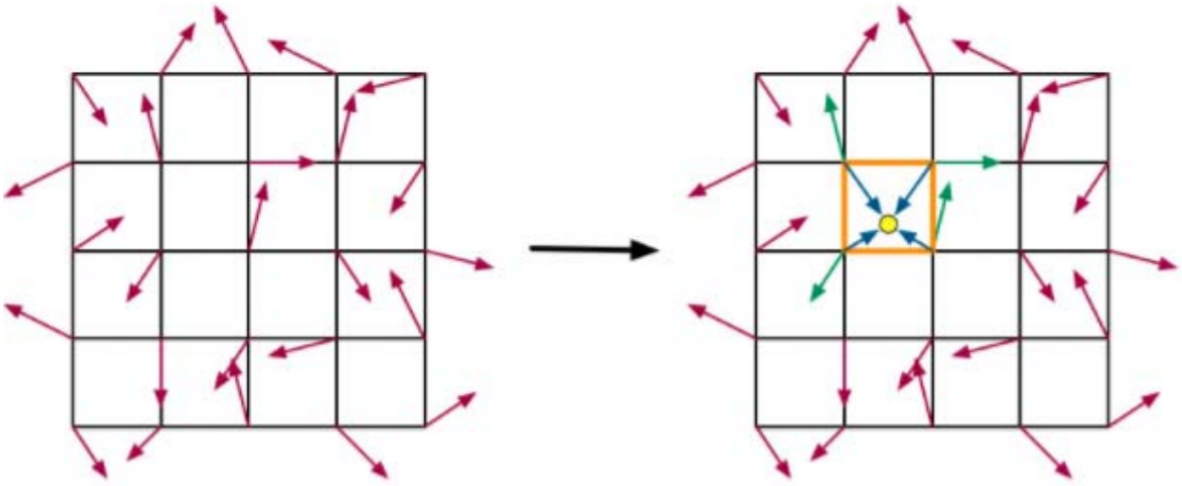

图 6: 梯度噪声

与课程中的过程类似，我们需要先生成 Perlin 噪声，如图3.1所示；然后在 Perlin 噪声的基础上通过一些简单的处理方法生成木纹。

Perlin 噪声本身就是启发性的，你可以改变其中任意环节来提升或者改变视觉效果。这里只介绍其中一种方法。你可以在这一方法的基础上进行改进，从而达成更换好的视觉效果。**以下讨论的坐标均为图片坐标**，对于采样的纹理坐标 (u, v)，你需要将其乘以图片分辨率从而得到浮点数的图片坐标。

3.1.1 预处理

在生成 Perlin 噪声之前，如图3.1左图所示，我们首先要在整个 UV 空间均匀地设置格点。**图片空间的大小与设置的格点数量有关：若你设置了 5×5 的格点，则图片空间的大小为 4×4。**每一个格点都有一个单位向量，这些单位向量都是随机生成的。显然，在计算机中，格点的单位向量是通过某种伪随机方法生成的；换句话说，我们可以根据格点的坐标**计算**这一单位向量而无需**保存**。当然，你需要的确保每次计算得到的的格点单位向量都是一致的。格点的分布可以均匀也可以非均匀：均匀的格点可以方便后续的查找最近点的插值操作；非均匀的格点可以使得梯度值的分布更加随机，只是你可能需要一些辅助数据结构（如 KNN）去查询距离某一点最近的四个格点。

在这一部分，你需要完成的事情有：

- 创建一个新的二维数据结构，其中保存着对应格点的单位向量；
- 或者保存一个随机种子，每次将格点坐标以及随机种子进行某种运算，进而通过伪随机数生成方法得到格点的单位向量。

3.1.2 纹理采样

Perlin 噪声以及木纹的生成需要在运行时进行计算。对于任意纹理采样点 $p = (u, v)$，其四周可以查询到四个格点 l_0, l_1, l_2, l_3。每个格点持有一个单位向量 $\vec{v}_0, \vec{v}_1, \vec{v}_2, \vec{v}_3$。此时，我们为每一个格点计算一个额外的、指向 p 的向量 $\vec{d}_i$：

$$\vec{d}_i = p - \vec{v}_i \quad i = 0, 1, 2, 3 \tag{13}$$

注意这个向量不用归一化。紧接着计算四个点乘 f_i：

$$f_i = \vec{d}_i \cdot \vec{v}_i \quad i = 0, 1, 2, 3 \tag{14}$$

最后对 f_i 进行插值即可。插值的方法有很多，本次实验的参考实现采用双线性插值。假设

$$\begin{aligned} l_0.x = l_3.x < l_1.x = l_2.x \\ l_0.y = l_1.y > l_2.y = l_3.y \end{aligned} \tag{15}$$

计算线性插值比例 r_x 和 r_y：

$$
\begin{aligned}
r_x &= \frac{p.x - l_0.x}{l_1.x - l_0.x} \\
r_y &= \frac{p.y - l_2.y}{l_1.y - l_2.y}
\end{aligned} \tag{16}
$$

双线性插值的结果 f 如下：

$$
\begin{aligned}
x_{\text{low}} &= (1 - r_x) f_3 + r_x f_2 \\
x_{\text{high}} &= (1 - r_x) f_0 + r_x f_1 \\
f &= (1 - r_y) x_{\text{low}} + r_y x_{\text{high}}
\end{aligned} \tag{17}
$$

f 即为 Perlin 噪声的采样值。为了模拟木纹，需要对 Perlin 噪声做出简单的后处理：

$$
f' = (f \cdot 5 - \lfloor f \cdot 5 \rfloor) \cdot c \tag{18}
$$

f' 即为最后返回的木纹纹理颜色。其中常数 5 可以替换为任意正数，你可以随意地改变这个数字观察实验结果。c 为某一种棕色，你也可以自行设置成喜欢的颜色。在此次实验中，最终生成的纹理颜色不会影响实验评阅。

3.2 需要完成的事情

为了在代码里实现 WoodTexture，你需要在 Moer-lite 中加入额外的头文件：

```
// (newly created) WoodTexture.h
#pragma once

#include "Mipmap.h"
#include "Texture.h"
#include <CoreLayer/ColorSpace/Spectrum.h>
class WoodTexture : public Texture<Spectrum> {
public:
  WoodTexture() = delete;

  WoodTexture(const Json &json);

  virtual Spectrum evaluate(const Intersection &intersection) const override;

  virtual Spectrum evaluate(const TextureCoord &texCoord) const override;

private:
  // necessary data
  // your code here ...
};
```

以及添加额外的实现：

```
// (newly created) WoodTexture.cpp
#include "WoodTexture.h"
#include <ResourceLayer/Factory.h>

```

```
WoodTexture::WoodTexture(const Json &json) : Texture<Spectrum>() {
  // 给出的示例场景中，json文件并不包含额外的域；你可以自行添加
  // your code here ...
}

Spectrum WoodTexture::evaluate(const Intersection &intersection) const {
  // your code here ...
}

Spectrum WoodTexture::evaluate(const TextureCoord &texCoord) const {
  // your code here ...
}

// 为了让json文件可以索引到WoodTexture，这一语句不可以删除
REGISTER_CLASS(WoodTexture, "woodTex")
```

记得在根目录的 CMakeLists.txt 中添加对应的.cpp 文件：

```
add_executable(Moer
    ...
    src/FunctionLayer/Texture/WoodTexture.cpp
    ...
  )
```

在此之后，由于添加了新文件，你需要重新使用 cmake 生成项目文件。

3.3 测试

由于过程式纹理本身的随机性，此次实验仅要求视觉效果近似。换句话说，只要你的实验结果看起来像木纹，那么你的这次实验就可以通过。本次实验对于最终结果的评价完全依赖于视觉效果，因此你完全可以不遵循上面所说的实现。如果你使用了其他的实现方法，请在你最终的报告中描述你的实现方法。

实验提供了场景 bunny-texture-wood。

图 7: 木纹过程纹理的示例

图中所使用的网格大小为 4，木材的颜色为 $[164, 116, 73]$。

4. 过程式纹理：细胞 *

这一部分为上述木纹纹理的拓展，因此作为可选内容。如果你的精力和时间仅允许你实现一部分的细胞过程纹理，请在实验文档中写出你的思路。

4.1 Voronoi 噪声

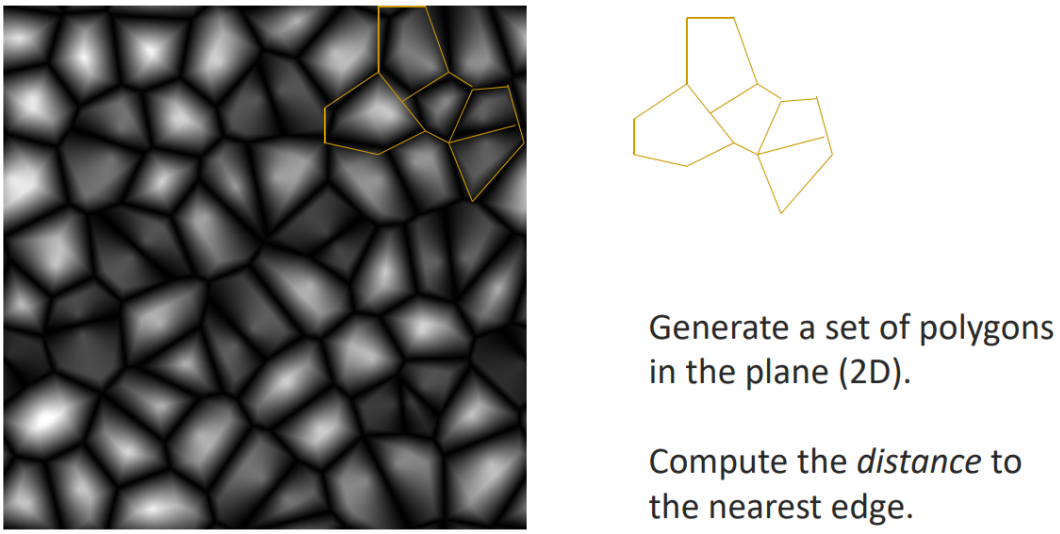

图 8: 细胞纹理

在课堂中，我们展示了图4.1。其生成方式看起来很简单：

1. 在二维平面上随机撒点 b_i；
2. 二维平面上的任意点都存在一个距离它最近的 b_i。对于任意的 b_i，距离它最近的所有点构成了一个多边形，因此整个二维平面被划分成了许多多边形，我们将这些多边形称为 P_i；
3. 平面上每一个点的纹理值即为其距离最近的多边形边界的距离。

然而，如果你顺着这一思路去实现细胞过程式纹理，其过程将会极其复杂。我们在此顺着先前的 Perlin 噪声，介绍 Voronoi 噪声，进而通过 Voronoi 噪声来生成类似的细胞过程式纹理。

在 Perlin 噪声的生成过程中，我们在二维平面上均匀地设置了格点 l，并且每一个格点都有对应的随机单位向量 $\vec{v}$。显然，在生成细胞纹理的过程中，平面上的点并没有按照均匀的方式排列。为了在格点分布上引入随机性，我们引入偏移格点 b，每一个格点 l_i 都有任意的 b_i：

$$b_i = l_i + \vec{v}_i \tag{19}$$

对于二维平面上的任意点 p，由于格点排列的均匀性，我们可以轻易地查询到距离它最近的格点 l_n，其对应的偏移格点为 b_n。l_n 周围的八联通区域内存在八个格点 l_i，其中 $i = 0, 1, ..., 7$，它们对应的偏移格点为 b_i，其中 $i = 0, 1, ..., 7$。

我们现在考虑点 p 到最近偏移格点的距离。由于每一个格点偏移向量 $\vec{v}$ 都为单位向量，**所以我们只需计算点 p 与点集 $\{b_n, b_0, ..., b_7\}$ 之间的最近距离 d_{point}** 而无需考虑其他的偏移格点。感兴趣的同学可以证明这一点，对于其他任意的 b，其与 p 的距离不可能小于 d_{point}。让每一个点 p 在纹理采样时都返回 d_{point}，则我们可以看到近似的细胞纹理。

课堂中展示的细胞纹理上，任意采样点 p 返回的并非为 d_{point}，而是为 d_{edge}，也就是某一点到最近多边形边界的距离。在这里你需要自行进行一些几何推导，从而得到 d_{edge}。

提示：有这样的一些几何事实可以利用：

- 如果任意两个偏移格点之间存在多边形边界，则多边形边界垂直平分两个偏移格点之间的线段；
- 我们已经查找到了与点 p 距离最近的偏移格点 b_*，所以点 p 必然位于对应的多边形 P_* 中。P_* 可能与其周围的八个偏移格点构成的多边形之间形成边界。

4.2 需要完成的事情

这一部分的操作与 WoodTexture 完全相同。为了在代码里实现 CellTexture，你需要在 Moerlite 中加入额外的头文件：

```
// (newly created) CellTexture.h
#pragma once

#include "Mipmap.h"
#include "Texture.h"
#include <CoreLayer/ColorSpace/Spectrum.h>
class CellTexture : public Texture<Spectrum> {
public:
  CellTexture() = delete;

  CellTexture(const Json &json);

  virtual Spectrum evaluate(const Intersection &intersection) const override;

  virtual Spectrum evaluate(const TextureCoord &texCoord) const override;

private:
  // necessary data
  // your code here ...
};
```

以及添加额外的实现：

```
// (newly created) CellTexture.cpp
#include "CellTexture.h"
#include <ResourceLayer/Factory.h>

CellTexture::CellTexture(const Json &json) : Texture<Spectrum>() {
  // 给出的示例场景中，json文件并不包含额外的域；你可以自行添加
  // your code here ...
}

Spectrum CellTexture::evaluate(const Intersection &intersection) const {
  // your code here ...
}

Spectrum CellTexture::evaluate(const TextureCoord &texCoord) const {
  // your code here ...
}

```

```
// 为了让json文件可以索引到CellTexture，这一语句不可以删除
REGISTER_CLASS(CellTexture, "cellTex")
```

记得在根目录的 CMakeLists.txt 中添加对应的.cpp 文件：

```
add_executable(Moer
    ...
    src/FunctionLayer/Texture/CellTexture.cpp
    ...
  )
```

在此之后，由于添加了新文件，你需要重新使用 cmake 生成项目文件。

4.3 测试

由于过程式纹理本身的随机性，此次实验仅要求视觉效果近似。换句话说，只要你的实验结果看起来像细胞，那么你的这次实验就可以通过。本次实验对于最终结果的评价完全依赖于视觉效果，因此你完全可以不遵循上面所说的实现。返回 d_{point} 即可完成实验目标，如果你成功推导出了 d_{edge} 的求解方法，请在文档中给出效果图和简要推导过程。

实验提供了场景 bunny-texture-cell。

图 9: 细胞过程纹理：点距离

图 10: 细胞过程纹理：边界距离

图中所使用的网格大小为 16。

5. 杂项与错误修正

5.1 错误修正

在 DirectIntegrator.cpp 中，对于 DirectIntegratorSampleBSDF 积分器，你需要在求交之后调用 computeRayDifferentials 方法。注意积分器的名称并**不**是 DirectIntegratorSampleLight。

```
Spectrum DirectIntegratorSampleBSDF ::li(Ray &ray, const Scene &scene,
                                std::shared_ptr<Sampler> sampler) const {
  ...
  auto intersection = intersectionOpt.value();
  computeRayDifferentials(&intersection, ray); // add this line
  ...
}
```

在 Intersection.h 中，有一处笔误：

```

inline void computeRayDifferentials(Intersection *intersection,
                                    const Ray &ray) {
    float dudx, dvdx, dudy, dvdy;
    if (!solveLinearSystem2x2(A, Bx, &dudx, &dvdx))
        dudx = dvdx = .0f;
    if (!solveLinearSystem2x2(A, By, &dudy, &dvdy)) // modify: Bx -> By
        dudy = dvdy = .0f;

}
```

6. 作业

6.1 你需要做的事情

- 根据上述“错误修正”章节，修改你本地的 Moer-lite 源码；
- 按照要求完成“光线微分”；
- 按照要求完成“纹理映射”；
- 按照要求完成“过程式纹理：木纹”；
- 如果你有足够的精力，你可以试着完成“过程式纹理：细胞”。这一部分是可选的；
- 如果你依然有足够的精力，你可以试着实现某种**各向异性过滤**方法。在之后，课程将会给出一份各向异性过滤方法的简介。这一部分也是可选的；
- 撰写实验报告，提供实验结果，提交作业。

6.2 提交的文件

你需要提交一份压缩包，在这个压缩包中包含了你的 **pdf 格式的实验报告**。实验报告的撰写形式不限，可以使用 LaTeX，也可以使用更为简单的 markdown 或者 word。**实验报告的形式不会影响分数**，所以我们不推荐你在工具上花费很多精力。我们不鼓励在实验报告的长度上“卷”，过于长的实验报告**不会**获得更高的分数。

对于每个小实验，你需要在实验报告中包含以下内容：

- 你对于这个问题的理解；
- 你的推导过程以及实现思路；
- 你的实现代码，仅需简单表示即可，可以使用 C++ 或者伪代码；
- 你的实验结果；
- 你为这个实验做出的额外工作（新的场景，新的方法，你觉得没有人做过的东西）。

如果你觉得你不小心对实验报告中的图片进行了修改，压缩包中也可以包含你的渲染图片的原图。**此次实验中压缩包中不必包含任何代码文件，你应该在实验报告中将你的实现描述清楚**。你也可以在压缩包中包含你自己搭建的新场景。

6.3 提交截止时间

本次作业放出的时间即为开始时间。

本次作业的提交截止日期为 **2024 年 5 月 20 日晚 23:59**，以教学立方时间为准。逾期提交的作业将会按照逾期的时间将分数乘以对应比例。

6.4 提交方式

请将实验报告以及可能的图片打包为.zip 格式提交至教学立方，压缩包的名称应遵循以下格式：

姓名 + 学号 + 图形绘制技术 Lab1

注意此次作业仅接受教学立方平台的提交。