PBRTv4-小专题-PBRT的坐标系统和变换

Dezeming Family

2023年9月18日

DezemingFamily系列文章和电子文档**全部都有免费公开的电子版**，可以很方便地进行修改和重新发布。如果您获得了DezemingFamily的系列文章，可以从我们的网站[https://dezeming.top/]找到最新的版本。对文章的内容建议和出现的错误也欢迎在网站留言。

2023-09-19：完成第一版。 2023-09-25：更新了一些内容，增加了“坐标系统和Film”以及“交互中的相机”这两个小节。修复了一些错误的地方。 2023-09-30：更新了一些内容，增加了“光栅空间与像素”小节。

目录

一　坐标系统

1 1　worldFromCamera

变换有很多种，比如worldFromCamera， worldFromRender， renderFromWorld， renderFromCamera。

在parse文件时，有可能会有两种定义相机到世界变换的形式：

- 第一种：在parse到Camera之前，parse到了Transform，那么该Transform就是cameraFromWorld变换矩阵。
- 第二种：定义了LookAt，那么计算LookAt矩阵作为worldFromCamera。求逆得到cameraFromWorld。

worldFromCamera的意思就是camera-to-world，将相机空间的内容变换到世界空间。

1 2　worldFromRender

命令行启动时的-render-coord-sys参数有“cameraworld”、“camera”以及“world”。

见CameraTransform::CameraTransform(.)函数。如果是“camera”参数，意味着worldFromRender就是worldFromCamera。如果是“cameraworld”参数，意味着worldFromRender是：

```
1 Translate(Vector3f(worldFromCamera(Point3f(0,0,0))))
```

如果是“world”参数，意味着worldFromRender是单位矩阵变换。

1 3　发射相机射线

在进行渲染发射采样射线时，相机会进行如下操作：

```
1 CameraRay{RenderFromCamera(ray)};
```

也就是把相机坐标系下的相机采样射线变换到渲染坐标系下。在静态场景下，进一步可以看出，如果是“camera”参数，意味着RenderFromCamera就是单位矩阵，那么相机坐标系下的相机采样射线就直接在当前空间采样。如果是“world”参数，就相当于将worldFromCamera作用于相机采样射线，把射线从相机空间变换到世界空间。如果是“cameraworld”参数，RenderFromCamera就是：

```
1 Translate(-Vector3f(worldFromCamera(Point3f(0,0,0))))*worldFromCamera
```

1 4　相机空间与屏幕空间之间的变换关系

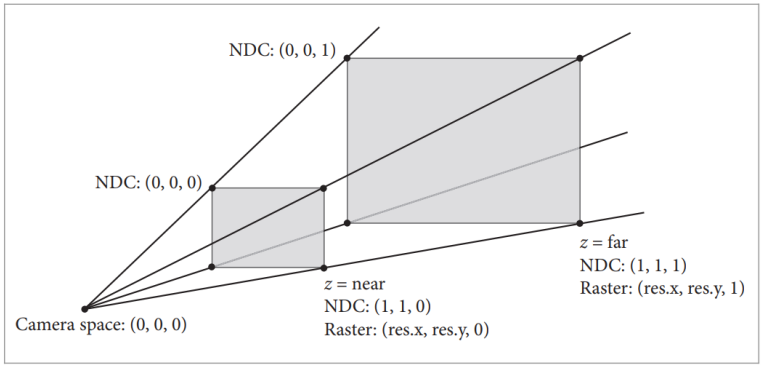
我们只关注透视投影相机，这是用得最多的相机类型。

在透视投影相机中，有四种相机变换关系：screenFromCamera，cameraFromRaster，rasterFromScreen，screenFromRaster。

screenFromCamera就是（后面再讲解）：

```
1 Perspective(fov, 1e-2f, 1000.f)
```

screenFromCamera负责将相机空间的内容变换到NDC坐标空间，在横截体内部的坐标值都会被映射到$(0,0,0)-(1,1,1)$内，横截体外部的区域按理来说都是不可见的（但深度值，也就是xyz值得z分量如果不在$(0,1)$范围内，也不会被直接剔除，需要代码判断是否在可见横截体内）：



由于屏幕空间是二维的，而NDC空间是三维，因此在转换时，就默认屏幕板在NDC空间的$(0,0,0)-(1,1,0)$屏幕上。NDCFromScreen是将屏幕空间的内容投射到NDC空间的近相机平面：

```
Transform NDCFromScreen =
    Scale(1 / (screenWindow.pMax.x - screenWindow.pMin.x),
        1 / (screenWindow.pMax.y - screenWindow.pMin.y), 1) *
    Translate(Vector3f(-screenWindow.pMin.x, -screenWindow.pMax.y, 0));
Transform rasterFromNDC =
    Scale(film.FullResolution().x, -film.FullResolution().y, 1);
rasterFromScreen = rasterFromNDC * NDCFromScreen;
```

screen是根据屏幕长宽比计算的，它把屏幕的短边设置为了1.0，长边为1.0的长宽比倍数：

```
Float frame = parameters.GetOneFloat("frameaspectratio", Float(film.
    FullResolution().x) / Float(film.FullResolution().y));
Bounds2f screen;
if (frame > 1.f) {
    screen.pMin.x = -frame;
    screen.pMax.x = frame;
    screen.pMin.y = -1.f;
    screen.pMax.y = 1.f;
} else {
    screen.pMin.x = -1.f;
    screen.pMax.x = 1.f;
    screen.pMin.y = -1.f / frame;
    screen.pMax.y = 1.f / frame;
}
```

1 5 Perspective(...)函数

用户指定的角度视场(fov)是通过缩放投影平面上的(x,y)值来计算的，以便视场内的点投影到视图平面上$[-1,1]$之间的坐标。对于正方形图像，x和y在屏幕空间中都位于$[-1,1]$之间。否则，图像较窄的方向将映射到$[-1,1]$，较宽的方向映射到按比例较大的屏幕空间值范围。因此Perspective(...)函数的计算和screen的计算是一一对应的。切线等于直角三角形的对边与邻边的比率。这里，邻边具有长度1，因此对边具有长度$tan(fov/2)$。按此长度的倒数缩放可将视野映射到范围$[-1,1]$。

```
Float invTanAng = 1 / std::tan(Radians(fov) / 2);
return Scale(invTanAng, invTanAng, 1) * Transform(persp);
```

缩放以后，近平面区域内点的y坐标值都会映射到$[-1,1]$，x坐标的值被映射到$[-w/h,+w/h]$之间。但是注意，NDCFromScreen函数可能会将更大的区域映射到NDC空间，而不是原来的fov对应的图像范围。

在屏幕的高¿宽时（$h > w$）就会发生这样的现象，比如$h = 1200$，$w = 1000$，那么screenFromCamera函数就会把视横截体内的更大的区域三维坐标的x和y值都映射到$[-5/6, -1] - [5/6, 1]$之间。但是，整个屏幕的fov是会变化的，这是我觉得PBRT的一个很怪异的地方：PBRT会将更大的区域，即扩展到$[-1, -6/5] - [1, 6/5]$，所以说此时fov算是改变了。此时的NDCFromScreen就是：

```
Scale(1 / (1 - (-1)), 1 / (1.2 - (-1.2)), 1) * Translate(Vector3f(1, 1.2, 0)
    );
```

如果宽¿高（$w > h$），那么screenFromCamera函数就会把视横截体内的区域三维坐标的x和y值都映射到$[-6/5, -1] - [6/5, 1]$之间。此时整个相机视野的fov并不会改变，还是之前的fov。这点之前在《三维视角投影变换与代码实战》文章中介绍过。我个人觉得这样的实现有些奇怪，不明白为什么PBRT要这么处理。

二　交互中的相机设置

交互中的结果如下：

```
Transform cameraMotion;
if (gui)
    cameraMotion = renderFromCamera * gui->GetCameraTransform() *
        cameraFromRender;
GenerateCameraRays(y0, cameraMotion, sampleIndex);
```

先从渲染空间变换到相机空间，然后作用于相机的变换，然后再变换到世界空间。 GetCameraTransform()函数为：

```
Transform GetCameraTransform() const { return movingFromCamera; }
```

movingFromCamera变换可以在鼠标事件和按键事件中修改。

看向的方向主要在鼠标事件处理中修改：

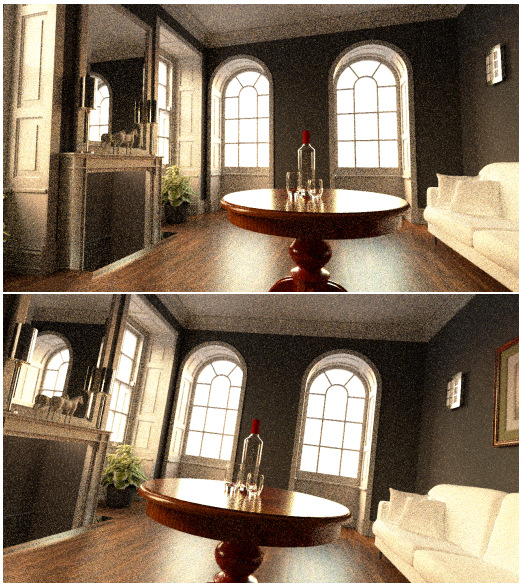
```
bool GUI::processMouse() {
    bool needsReset = false;
    double amount = 1.f;
    if (!pressed)
        return false;
    if (xoffset < 0) {
        movingFromCamera = movingFromCamera * Rotate(-amount, Vector3f(0, 1,
            0));
        needsReset = true;
        xoffset = 0;
    }
    if (xoffset > 0) {...}
    if (yoffset > 0) {...}
    if (yoffset < 0) {...}
    return needsReset;
}
```

前进和后退等功能主要在按键处理GUI::processKeys()中修改：

```
handleNeedsReset(
```

```
    'a', [&](Transform t) { return t * Translate(Vector3f(-moveScale, 0, 0))
        ; });
handleNeedsReset(
    'd', [&](Transform t) { return t * Translate(Vector3f(moveScale, 0, 0));
        });
```

看向的方向是根据绕(0,1,0)和(0,0,1)来旋转的，移动和旋转的功能可以实现类似第一人称射击游戏的移动交互方式。但是如果你旋转了窗口，就会偏离主方向。下图第一张没有旋转视角，第二张是旋转过视角的结果：



这是因为旋转变换中，绕X轴和绕Y轴的变换不是按照有序的（先统一Y轴旋转，再统一都X轴旋转），这样就会导致绕Y轴和绕X轴旋转交杂在一起，造成主轴偏离。

修改的方法也不难，就是把变换分离开，变成前进的变换、绕X轴的变换和绕Y轴的变换。但是前进的变换如果被分离出来，就需要前进时根据当前相机变换过以后的方向来前进。这是因为在渲染中，WavefrontPathIntegrator::Render()函数：

```
cameraMotion = renderFromCamera * gui->GetCameraTransform() *
    cameraFromRender;
```

相机射线经过cameraFromRender变换以后，就会变到相机空间。然后当相机前进时，就需要根据相机此时变换后的位置来前进。

修改方法，在GUI类中，增加几个变量：

```
Vector3f cameraForward;
Vector3f cameraRight;
Transform TranslateFromCamera;
Transform RotateFromCamera;
Transform RotateFromCamera_X;
Transform RotateFromCamera_Y;
```

然后在GUI的构造函数中初始化：

```
cameraForward = Vector3f(0.0f, 0.0f, 1.0f);
cameraRight = Vector3f(1.0f, 0.0f, 0.0f);
```

在GUI::processMouse()中，以鼠标左移为例：

```
if (xoffset < 0) {
    Transform rotate_temp = Rotate(-amount, Vector3f(0, 1, 0));
```

```
    RotateFromCamera_X = RotateFromCamera_X * rotate_temp;
    RotateFromCamera = RotateFromCamera_Y * RotateFromCamera_X;
    cameraForward = rotate_temp(cameraForward);
    cameraRight = rotate_temp(cameraRight);
    movingFromCamera = TranslateFromCamera * RotateFromCamera;

    needsReset = true;
    xoffset = 0;
}
```

GUI::processKeys()也需要修改，以前进为例：

```
auto handleNeedsReset = [&](char key, std::function<Transform(Transform)>
    update) {
    if (keysDown.find(key) != keysDown.end()) {
        TranslateFromCamera = update(TranslateFromCamera);
        needsReset = true;
    }
};
handleNeedsReset(
    'a', [&](Transform t) { return t * Translate(-moveScale * cameraRight);
        });
movingFromCamera = TranslateFromCamera * RotateFromCamera;
```

经过一些修改，就可以使得相机变成真正的FPS相机了。代码我放在了网站的github上。

三　坐标系统和Film

在生成GBufferFilm时，会有一个cameraTransform参数，根据我们的场景文件的设置，该参数与coordinatesystem这一项有关：

```
Film "gbuffer"
"string filename" [ "living-room.exr" ]
"bool savefp16" [ false ]
"string coordinatesystem" [ "world" ]
"integer yresolution" [ 720 ]
"integer xresolution" [ 1280 ]
```

而且对于GBufferFilm，coordinatesystem只能是"camera"或者"world"，不能是"cameraworld"。见GBufferFilm::Create(...)函数：

```
std::string coordinateSystem = parameters.GetOneString("coordinatesystem", "
    camera");
AnimatedTransform outputFromRender;
bool applyInverse = false;
if (coordinateSystem == "camera") {
    outputFromRender = cameraTransform.RenderFromCamera();
    applyInverse = true;
} else if (coordinateSystem == "world")
    outputFromRender = AnimatedTransform(cameraTransform.WorldFromRender());
```

如果之前定义的渲染坐标系（命令行参数）是"camera"，而且这里也是"camera"，那么相机射线就是在世界空间采样，RenderFromCamera()是一个单位矩阵。注意这里有一个applyInverse变换，表示在计算时使用逆变换（比如生成GBuffer时，将顶点或法向量作用于outputFromRender的逆变换）。也就是把渲染空间的值变换到相机空间。

如果之前定义的渲染坐标系（命令行参数）是"world"，而且这里也是"world"，那么相机射线就是在世界空间采样，WorldFromRender()是一个单位矩阵。

所以，如果之前定义的渲染坐标系（命令行参数）和Film输出坐标系不同时，outputFromRender才不会是单位矩阵。

四　交互中的相机

PerspectiveCamera::GenerateRay(..)中会对采样射线进行变换：

```
Point3f pFilm = Point3f(sample.pFilm.x, sample.pFilm.y, 0);
Point3f pCamera = cameraFromRaster(pFilm);
......
return CameraRay{RenderFromCamera(ray)};
```

WavefrontPathIntegrator::GenerateCameraRays(..)函数中，将movingFromCamera作用到相机射线：

```
if (cameraRay) cameraRay->ray = movingFromCamera(cameraRay->ray);
```

movingFromCamera就是cameraMotion：

```
if (gui)
    cameraMotion =
        renderFromCamera * gui->GetCameraTransform() * cameraFromRender;
```

交互中会实时更新，这些矩阵都可以打印出来，并用于其他用途，比如时序重投影。

五　光栅空间与像素

GetCameraSample(...)函数中可以看到，像素会抖动：

```
FilterSample fs = filter.Sample(sampler.GetPixel2D());
CameraSample cs;
// Initialize _CameraSample_ member variables
cs.pFilm = pPixel + fs.p + Vector2f(0.5f, 0.5f);
cs.time = sampler.Get1D();
cs.pLens = sampler.Get2D();
cs.filterWeight = fs.weight;

if (GetOptions().disablePixelJitter) {
    cs.pFilm = pPixel + Vector2f(0.5f, 0.5f);
    cs.time = 0.5f;
    cs.pLens = Point2f(0.5f, 0.5f);
    cs.filterWeight = 1;
}
```

也就是说，像素$(0,0)$的中心是光栅空间的$(0.5,0.5)$，像素$(150,120)$的中心是$(150.5,120.5)$。

在命令行参数中加入"–disable-pixel-jitter"就可以避免像素抖动。

使用像素抖动和不使用抖动的渲染结果如下（都是渲染了5128spp）：

参考文献

[1] https://github.com/mmp/pbrt-v4

[2] https://pbrt.org/

[3] https://pbrt.org/resources

[4] Pharr, Matt, Wenzel Jakob, and Greg Humphreys. Physically based rendering: From theory to implementation. MIT Press, 2023.

[5] https://www.cnblogs.com/pointer-smq/p/7522416.html