全景图生成算法详解

最近整理以前的老代码，看到了2018年写的全景图生成插件，有感而发，写了这篇文章。

当初我自己实现这个插件的时候，主要是用于VR的全景视频生成，后来，正如大家看到的，VR死了。而现在，套了个元宇宙的马甲，死灰复燃，也没有掀起多少浪花。

这个技术现在有没有烂大街了我不知道。我当初写这个插件的时候，确定肯定一定是没有的，反正我没有找到多少资料，几乎都是自己理解，自己想明白，自己写的。因此，是不是有更好的方案？是不是我这个是很low的实现方式？按道理不会，但是我不确定。所以，如果发现我这个实现方式不好什么的，不要介意，毕竟是我自己闭门造车想出来的:)

言归正传，我们先来看看全景图是什么样的：



这是随手百度到的一张图，侵删。

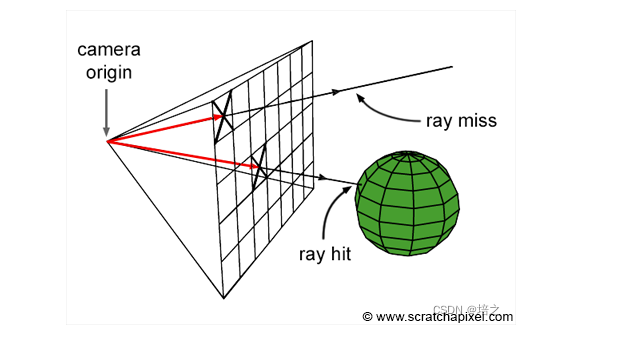
我们仔细看，全景图跟一般的图是不同的，不同在于：一般的图，可以理解为一个普通相机拍到的图片，四四方方，内容就是一个矩形，也就是一个相机镜头拍到的样子，专业的术语说，就是一张透视/正交生成的图像。而全景图，其实本质上是一个球面映射。什么意思呢？理解一下游戏里最常见的天空盒，天空球。天空盒的意思，是一个正方体，人在正方体的最中点，周边六个面，六张图，也就是所谓的cubemap，贴上去，你以为你看到的是一整个世界，其实你看到的只是几张图片包围着的你。而全景图，其实就是所谓的天空球，把一个正方体，换成一个球，你就在球心，然后球面上贴上一张图片，这张图片就是全景图，然后你以为你看到了世界，其实你看到的只是一张图片:(

图片如何贴上去？这个，就需要一点点图形学基础了，你必须要明白简单的Mesh，UV坐标。如果这个基础都没有，建议回去看看基础，理解一下即可，也不是很艰难的是，我有专门的文章介绍。

好，这里，假设你已经完全理解了上面的内容，那么，全景图本质上就是一张贴在一个球上的图片，如何生成？好问题，我们分开两个思路来说明。

第一个思路：其实就是光线跟踪的思路。这个思路我没有做过全景图生成，只做过一般的渲染，但是原理是一样的。首先，假设你现在有一张全景图，4096 \* 2048的。你先简单建立一个坐标系，建立一个球。假设你已经把这个全景图贴到了球上，那么：**任意全景图上的一个像素，都可以转换为一条空间射线！**这个空间射线，其实就是做光线跟踪的射线。

这个看不懂？那应该你不熟悉光线跟踪的计算逻辑。这里简单解释一下：假设你渲染一个1920 \* 1080的图片，光线跟踪，其实就是从人眼方向，往这个图片发送1920 \* 1080条射线，每一条射线，经过一系列的复杂的采样、反射等等，得到一个颜色值。最终每一条射线，都得到一个颜色，最终得到渲染图片。



图片来自网络，侵删。

有兴趣的，可以自己找资料看看。老早之前我就有想过写相关文章，但是发现已经太多了，自己又没有能力写得出彩，写得不一样，就一直没写。

所以，看到这里，应该很容易看懂了。光线跟踪渲染全景图，跟渲染一般的图片，本质上没有区别，区别仅仅在于计算的射线不同。

既然如此，直接上光线跟踪就完事了，为什么这个方案并没有大行其道？原因在于，这个仅仅适用于3D渲染。但是，全景图的绝大部分来源，都不是3D渲染，而是相机拍摄。典型的，例如全景摄像机，你无法用这个方案。因此，我们还是着重讲解另外一个方案。

第二个思路：既然相机是先得到图片，那么我先得到图片，再转换成全景图不就完事了吗？完全正确。

这里第一个容易被想到的思路，是先生成cubemap，得到一个正方体，再把正方体的像素，算成一个球面的像素即可。这个方案，基本上是一些引擎上购买的一些插件，都是这个方案。例如之前我看过有人用一些引擎上的插件，都是差不多的方案。

这个方案具体实现是怎么样的呢？其实核心代码比较简单。首先，我们知道，cubemap是一个六面体，可以理解为六张图片。六张图片，是怎么来的呢，其实就是六个正方形的镜头，渲染得到的。因此，当你知道这个镜头的参数（也就是透视投影矩阵相关参数）的时候，你是可以算出来任意一个像素的射线的。大概代码是这样的：

void Camera::getCameraToViewportRay(Real screenX, Real screenY, Ray\* outRay) const

{

Matrix4 inverseVP = (getProjectionMatrix() \* getViewMatrix(true)).inverse();

Real nx = (2.0f \* screenX) - 1.0f;

Real ny = 1.0f - (2.0f \* screenY);

Vector3 nearPoint(nx, ny, -1.f);

// Use midPoint rather than far point to avoid issues with infinite projection

Vector3 midPoint (nx, ny, 0.0f);

// Get ray origin and ray target on near plane in world space

Vector3 rayOrigin, rayTarget;

rayOrigin = inverseVP \* nearPoint;

rayTarget = inverseVP \* midPoint;

Vector3 rayDirection = rayTarget - rayOrigin;

rayDirection.normalise();

outRay->setOrigin(rayOrigin);

outRay->setDirection(rayDirection);

}

以上代码摘自OGRE。绝大部分引擎，都有差不多的代码，原理也差不多的。以下代码，是我自己的全景图计算射线的代码。我这里的射线，假设了坐标点就是坐标原点，所以只需要计算方向即可。因为实质计算的时候，已经根据镜头的参数，生成了cubemap，现在仅仅是cubemap到全景图的映射，而不是去3d世界去采样颜色，所以不需要position了。

以下为代码，其实非常简单：

Vector3 PanoramaScene::BuildDirection(float xDegree, float yDegree)

{

xDegree += mBeginDegree;

float xRadian = Math::DegreesToRadians(xDegree);

float yRadian = Math::DegreesToRadians(yDegree);

float y = *cos*(yRadian);

float r = *sin*(yRadian);

float x = r \* *cos*(xRadian);

float z = r \* *sin*(xRadian);

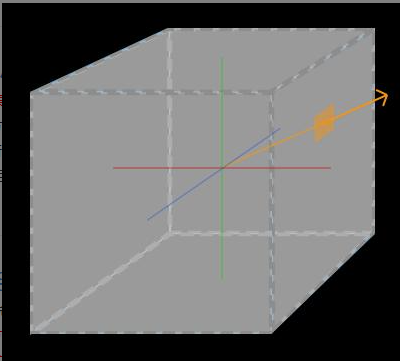
Vector3 v(x, y, z);

return v.normalisedCopy();

}

那么，这么做就一劳永逸了吗？其实不是的，这个方案，有非常多的问题。我这里大概说一下。

第一个问题，是采样。如果你从头开局，自己做过天空盒，就会知道，一不小心，天空盒中间就会出现一条线。如图：



图片来自网络，侵删。

这个是为什么呢？很简单，拼接的时候，你不能用线性采样，只能用最近点采样。原因在于，线性采样的时候，边缘的像素采样，会出现黑边。这个产生的原因，如果你自己动手写个图片采样，就很容易懂。这里，我不打算赘述了，有兴趣的可以去看我另外的关于材质采样的文章。

第二个问题，是一些跟引擎相关的问题。最典型的，例如用UE4生成窗口图片的时候，边缘部分是黑边。这个做法，主要是UE4考虑到整体效果而做的设定。也就是说，当你使用一些引擎/镜头生成cubemap图的时候，图的边缘一旦有问题，整个全景图转换就出了大问题。而很多时候，图的来源，你是无法确定或者选择的。所以，这个方案比较low的原因就在这里了。

第三个问题，这个是通用问题，无解问题。这个问题产生的根源，在于游戏渲染的时候，很多时候会采用billboard渲染。例如粒子系统，就会大量采用。Billboard的意思，是渲染一些细小物体的时候，会用一些小图来替代。例如火焰、烟花……。其实很多时候是一些小图片。为什么看不出来问题？因为这些图片，永远面向镜头。当你镜头渲染的时候，这些小图片也是跟着旋转，所以你看不出来问题。除了billboard，一切跟镜头参数相关的一些渲染，都会出问题。例如镜头曝光。因为你渲染六个镜头的图片的时候，每个镜头的方向都是不同的，跟光源/平行光等等的一些计算，结果也是不同的，这会导致生成的六张cubemap拼接出现一些无法控制的变化。

这个问题不仅仅出现在全景图生成，在很多领域都存在，属于无解问题。例如UE4自己推出了一个大屏的拼接融合方案，球幕渲染方案等等，需要多台电脑一起渲染一个超大屏的时候，明确列出了一大堆不能正常使用的效果。大部分就是镜头相关的效果。

第四个问题，无法确保帧完全同步。这个问题主要出现在全景相机，而不是3D渲染。全景相机/摄像机，一般是像一个球一样的，例如这样的：



图片来自网络，侵删。

这个原理是什么呢，其实差不太多，是N个相机，拍摄N张图片，然后拼接成一张全景图。

相机越多，越容易出现不同步。拍摄静态场景的时候，是没问题的，一旦出现高速运动的物体，这个一点点的不同步，都会被放大，导致拼接出问题。

这里，我仅仅知道这个会出问题，至于如何解决，有什么更好的方案，我没有具体做过，没什么发言权，就不打算详细往下讲了。

回到正题，上面讲到cubemap转换成全景图，这个方案其实问题很多的。为了规避这些问题，我们把这个方案做得更好，不用cubemap转换成全景图，而是任意N张图片，转换成全景图。这个跟cubemap原理类似，区别在于：图片是正常渲染的图片，而不是cubemap。Cubemap镜头是正方形的，这里是传统的矩形的，例如六张1920 \* 1080的图片，生成全景图。

这个方案，能够规避cubemap生成的全景图的1、2个问题，但是无法规避3、4个问题。这个方案，首先生成N张图片，这个N可以是很多张。实际情况，根据自己的需要。我一般用6张1920 \* 1080或者更多一些。下面，以6张1920 \* 1080为例讲一下如何转换成全景图。

1. 先渲染6张1920 \* 1080的图片，镜头参数除了宽高比，其他跟cubemap渲染差不多。
2. 得到底图后，计算需要转换的全景图的任意像素的射线。
3. 计算这个射线在六张图片的哪张图、哪个UV坐标。
4. 做一个线性采样即可得到这个颜色。

我这里的代码大概是这样的：

void PanoramaRendererCPU::SampleOnePixel(VertexSet& VS)

{

// calculate pixel position

int x = *floor*((VS.Pos.x + 1.0f) / 2.0f \* float(mPanoramaWidth) + 0.5f);

int y = mPanoramaHeight - *floor*((VS.Pos.y + 1.0f) / 2.0f \* float(mPanoramaHeight) + 0.5f);//floor((1.0f - (VS.Pos.y + 1.0f) / 2.0f) \* float(mPanoramaHeight) + 0.5f);

// 1 to width

// not from 0, so we need to -1

x--; y--;

int pos = y \* mPanoramaWidth + x;

float u = VS.UV.x \* float(mWidth);

float v = VS.UV.y \* float(mHeight);

int XL = *floor*(u);

int XH = XL + 1;

int YL = *floor*(v);

int YH = YL + 1;

float xGradient = u - XL;

float yGradient = v - YL;

unsigned long Col[4];

int BasePos = VS.Index \* mWidth \* mHeight;

Col[0] = mTextureData[YL \* mWidth + XL + BasePos];

Col[1] = mTextureData[YL \* mWidth + XH + BasePos];

Col[2] = mTextureData[YH \* mWidth + XL + BasePos];

Col[3] = mTextureData[YH \* mWidth + XH + BasePos];

unsigned long ColTop = ImageBlock::BlendPixel(Col[0], Col[1], xGradient);

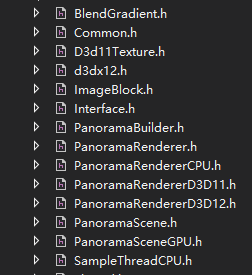
unsigned long ColBottom = ImageBlock::BlendPixel(Col[2], Col[3], xGradient);

unsigned long ColFinal = ImageBlock::BlendPixel(ColTop, ColBottom, 1 - yGradient, true);

mBackTexture[pos] = ColFinal;

}

这部分代码，我并没有放github上，但是放了个demo在github上，感兴趣的，可以自己去看。我当初实现了三种采样，CPU的，D3D11的，D3D12的。为什么实现三种？无非就是效率问题，CPU效率较低，但是GPU实现两种，当初有考虑到UE4的Renderer可以是d3d11，也可以是d3d12.我都写了就完事了。核心代码大概如图：



所以说，这东西说穿了，核心代码没多少的。

后来我放了demo网上，而没有放源码，反而收获了较大的恶意，这是我没想到的。我估计白嫖党会觉得，我不放源码是原罪？我自己倒是不觉得。首先，我写这个源码的时候，是卖给了其他公司的。其次，现在我不知道，当时这个技术还是比较少见的，至少网上找不到，否则别人也不用出钱让我来做是不。当初英伟达出了个差不多的东东，也不见人家开源，也没见人喷。看来我还是太弱势了：）。再一个，考虑到大环境，当初的考量是：说不定我刚开源，就成了别人自主研发了？反正这东西，真的随缘。现在这里讲到这地步了，想必大家都能自己写一个了，开源不开源的，无所谓了。