TAA 时序反混淆原理和实践

Dezeming Family

2021 年 1 月 10 日

DezemingFamily 系列书和小册子因为是电子书，所以可以很方便地进行修改和重新发布。如果您获得了 DezemingFamily 的系列书，可以从我们的网站 [https://dezeming.top/] 找到最新版。对书的内容建议和出现的错误欢迎在网站留言。

偶然翻到以前记的笔记，想到目前网上也没有什么比较通俗和详细的 TAA 算法描述，因此我就把笔记重新整理了一遍，加了点自己新的理解和体会，形成了这本小册子。

TAA 又称为时序反混淆技术，依赖的一个事实是，渲染时一秒钟会产生好几十帧的图像，因此可以快速积累原始样本，将这些样本用于做抗锯齿计算。

一　重投影技术的由来

计算光照着色的消耗很大，但是人们发现，帧与帧之间有很多信息都可以复用，根据摄像机位置，照明条件和可见表面区域等相关信息，我们可以提取出可复用的信息。在两帧之间的相机、灯光、可见表面点及表面点属性（例如表面纹理，BRDF 等）改变非常小，因此很多信息都是可以重复使用的。

重复使用可以降低生成单个帧的平均成本，缓存机制允许在连续帧之间的像素着色器中存储、跟踪和检索前面计算的结果。（像素着色器是图形函数，它计算每个像素的最终成像效果。根据分辨率，可能需要以每秒几十帧的速度为每帧渲染、照亮、着色和为超过几百万像素上色）。

二　技术原理

论文 [1] 提出了一种基于反向重投影的实时应用缓存策略，但是所幸我们当前不需要了解缓存机制，因为我们只需要借鉴其中的重投影方法和策略。

在生成每个帧时，将所需的数据存储在视口大小的缓冲区中（比如与 viewRay 相交的第一个表面的位置、表面法向量等）。由于每个像素的值都是在下一帧中生成的，因此我们将其表面位置重新投影到上一帧中，以确定它以前是否可见。如果可见，我们可以重用它的缓存值，这样就不用执行冗余且可能很昂贵的计算来抗锯齿了（比如每像素多采样抗锯齿）。否则我们将当前渲染结果直接用来成像（或者多采样抗锯齿以后获得多个样本来成像），并在缓存中为下一帧提供它。

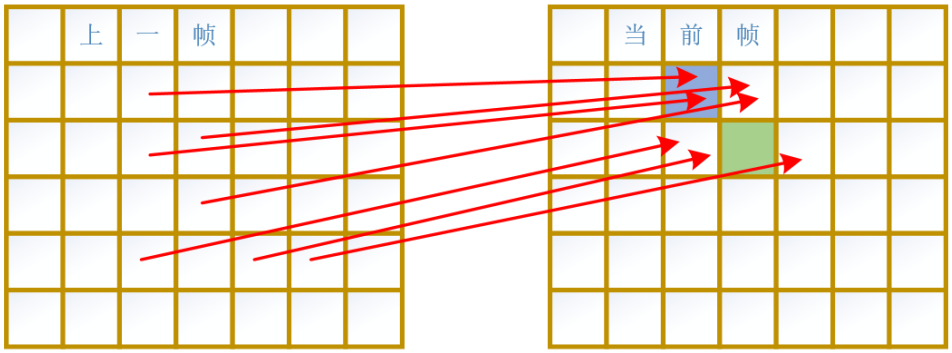
2 1　重投影

我们投影表面到先前帧，测试它的深度与先前帧深度相同，如果相同表示没有被遮挡，就可以复用。

我们定义一些符号表示：$f_t[p]$ 表示 t 时刻像素属性，$d_t[p]$ 表示像素深度,t 表示 t 时刻，p 表示像素 p，$f_t(\cdot)$，$d_t(\cdot)$ 表示对位置采样得到像素值（注意一般使用双线性插值采样）。对于像素 $p(x,y)$，我们设其在 3D 裁剪空间 t-1 时刻的位置为 $\pi_{t-1}(p)=(x',y',z')$。

对于重投影的深度值一样，则判断 $f_{t-1}[p(x',y')]$ 可用到 $f_t[p(x,y)]$ 上。

我们为什么要重投影，而不是把上一帧的交点位置投影到当前帧呢？这是因为，很有可能上一帧的好几个像素在投影后都投影在了当前的某个像素上，或者有可能因为精度问题，当前帧里相邻很近的几个像素中，有的拥有一个甚至多个上一帧像素信息而有的没有：



上图中，蓝框里可以获得多个前一帧信息，而绿框里没有接收到任何前一帧信息，但是绿框相邻像素都接收到了前一帧的信息。而通过重投影，我们就能根据插值来避免这些情况（在 GPU 多线程并行中，多个像素投影到当前帧是一个很难解决的问题）。

2 2 指数平滑滤波器

我们把新值和旧值使用指数平滑滤波器进行组合，$s_t[p]$ 表示像素 p 在当前帧渲染得到的结果：

$$f_t[p] \leftarrow (\alpha)s_t[p] + (1-\alpha)f_{t-1}(\pi_{t-1}(p)) \tag{二.1}$$

如果先前的样本不可用，则 α 设置为 1。

在静态场景中，设 α 为 $1/t$ 即可，随着渲染帧数增加，相当于对时域样本求平均。

对于动态场景，我们需要记录有效数量的样本，更精确地说，即存储每个像素的方差衰减因子 $N_t[p]$ 来设置每个像素的指数平滑因子 $\alpha_t[p]$。表面第一次可见时，$\alpha_t[p] \leftarrow 1$，$N_t[p] \leftarrow 1$，之后我们应用如下公式：

$$\alpha_t[p] = \frac{1}{N_{t-1}[p]+1} \tag{二.2}$$

$$N_t[p] \leftarrow N_{t-1}[p]+1 \tag{二.3}$$

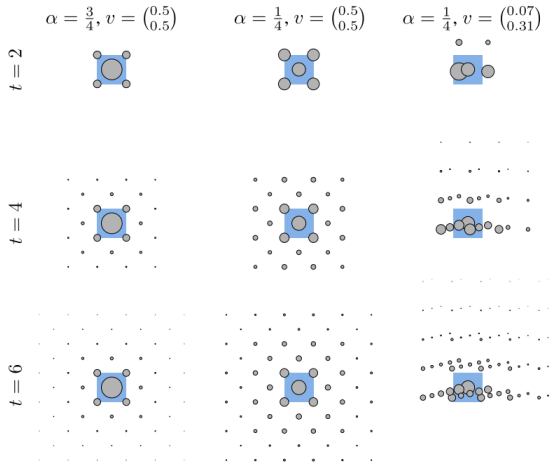
但哪怕某样本一直可见，我们也不能让这个均匀权重无限的积累下去，因此有新的方法：

$$N_t[p] \leftarrow \left(\alpha_t[p]^2 + \frac{(1-\alpha_t[p])^2}{N_{t-1}[\pi_{t-1}(p)]}\right) \tag{二.4}$$

对于新出现的未被遮挡的样本来说，其实就是 $N_t[p] \leftarrow 1$。

2 3 时间积累效果

我们分析一下参与构成当前显示图像的每个像素的所有样本的权重：



注意从上到下看，随着时间积累，影响图像显示值的样本范围会扩大，先前样本的权重也会减小。对比第 1 列和第 2 列可知，当 α 比较小的时候，当前样本值对最终显示值的影响就会更小。由第 3 列可知，当相机在 x 和 y 方向平移不一致时，样本聚集也会不一致。

对于静止的画面，我们需要在不同的时间 t 中随机抖动相机，来获得一个像素内不同位置的采样，将时序样本组合起来得到作为最终显示的结果。为了防止模糊，需要根据样本光栅位置与像素中心的距离生成权重，距离越远，权重值越低。

对于运动的画面，自然需要选择最接近像素中心的样本来限制模糊量，同样也要为抑制模糊来施加权重。

在论文 [2] 中，为了更好抑制模糊，会渲染原屏幕长宽都乘以 2 的大小的缓冲区，即原来的每个像素现在变为四个子像素，类似于 SSAA（超采样抗锯齿），然后针对子像素，进行一些样本的选择和优化过程，这里不再多说了。

三　代码实践

为了更简洁，我们不考虑把以前所有的样本都存到一个缓冲里，而是只保留先前帧的结果和先前帧的几何信息（空间位置等），也不考虑对每个像素的样本使用的图像空间滤波器。我们以 [3] 中的代码为例进行讲解，我们只关注时序滤波，不关注里面的方差引导空间滤波。

获得相机投影矩阵的函数如下，该函数其实就是根据相机参数来实现的透视投影矩阵的反投影。

```
glm::mat4 GetViewMatrix(const Camera& cam) {
    return glm::inverse(glm::mat4(glm::vec4(cam.right,     0.f),
                                  glm::vec4(cam.up,        0.f),
                                  glm::vec4(cam.view,      0.f),
                                  glm::vec4(cam.position, 1.f)));
}
```

BackProjection() 函数进行反投影计算：

```
    //把当前点变换到上一帧的屏幕空间中
    glm::vec4 viewspace_position = prev_viewmat * glm::vec4(current_gbuffer[
        p].position, 1.0f);
    float clipx = viewspace_position.x / viewspace_position.z /** tanf(PI /
        4)*/;
    float clipy = viewspace_position.y / viewspace_position.z /** tanf(PI /
        4)*/;
    //转化到NDC空间
    float ndcx = -clipx * 0.5f + 0.5f;
    float ndcy = -clipy * 0.5f + 0.5f;
    //转化到实际光栅屏幕空间
    float prevx = ndcx * res.x - 0.5f;
    float prevy = ndcy * res.y - 0.5f;
```

isReprjValid() 函数判断反投影以后的值是否合理（是不是当前点在上一帧时并不在屏幕里），判断反投影以后是否属于不同的几何体（比如同一个三角面片），判断点的向量差距是不是太大。（其实比较了是否是同一个几何体就没必要比较反投影以后的屏幕空间深度了，因为是同一个几何体的话深度肯定是相同的，如果前一帧某物体被挡住，当前帧它出现了，则与该物体相交的 Ray 得到的世界坐标位置反投影以后肯定和像素索引的几何体不同。）

然后计算有意义的历史样本长度 prevHistoryLength 并计算 α 即可：

```
    float color_alpha = max(1.0f / (float)(prevHistoryLength + 1),
        color_alpha_min);
    //当前颜色
```

```
3     color_acc[p] = current_color[p] * color_alpha + prevColor * (1.0f -
          color_alpha);
```

四　总结

当然我描述的只是比较简单的情况，论文 [2] 和 Siggraph 的课程 [4] 里也有一些更深入的描述，大家可以进行参考。总之，TAA 就是利用多帧之间的可复用信息来反混淆罢了。

参考文献

[1] Nehab D , Sander P V , Lawrence J , et al. Accelerating Real-Time Shading with Reverse Reprojection Caching[J]. 2007.

[2] Yang L , Nehab D , Sander P V , et al. Amortized Supersampling[J]. ACM Transactions on Graphics (TOG), 2009, 28(5):1-12.

[3] https://github.com/ZheyuanXie/CUDA-Path-Tracer-Denoising

[4] Siggraph 2014. Brian Karis (Epic Games, Inc.) High-Quality Temporal Supersampling