OIT（Order Independent Transparency，顺序无关透明）

This file is a document to explain how to do the Order Independent Transparency.

Alpha 通道

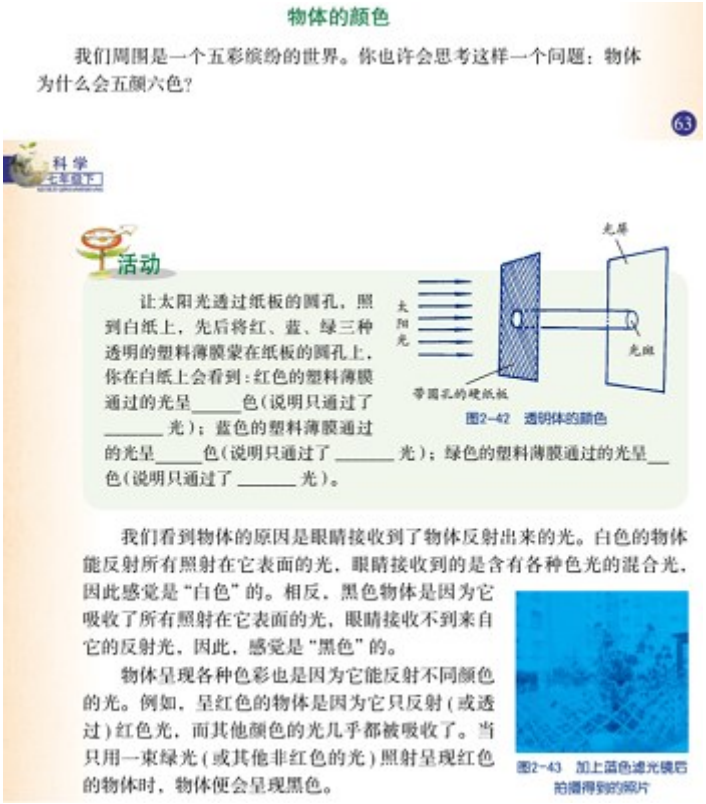
Porter 在 1984 年提出了 Alpha 通道（1.[Porter 1984]），目前在实时渲染中已被广泛地用于模拟物体的透明效果。

我们设对应到同一个像素（Pixel）的一系列片元（Fragment）的颜色、Alpha、深度为三元组$[C_i\ A_i\ Z_i]$，该像素最终的颜色$C_{Final}=\sum_i\left(\left(\prod_{Z_j\ Nearer\ Z_i}(1-A_j)\right)A_iC_i\right)$ //其中，$V(z_i)=\prod_{Z_j\ Nearer\ Z_i}(1-A_j)$又被称为可见性函数，即$C_{Final}=\sum_i(V(z_i)A_iC_i)$ 。

值得注意的是，在物理含义上， Alpha 模拟的是局部覆盖（Partial Coverage）而非透射率（Transmittance）。

Alpha 的含义是片元覆盖的面积占像素面积的比例（这也是我们用标量 float 而非向量 RGB 来表示 Alpha 的原因；这种情况在一些文献中被称作波长无关的（Wavelength-Independent））。比如，我们透过一条蓝色的真丝围巾观察一块红色的砖，我们看到砖的颜色大体为蓝色和红色“相加”； 真丝围巾的纤维本身是不透明的，只是真丝围巾的纤维之间存在着间隙，我们通过这些间隙看到了红色的砖，即真丝围巾“局部覆盖”了砖。

而透射率是波长相关的（Wavelengh-Dependent）；比如，我们透过一块蓝色的塑料薄膜观察一块红色的砖，我们看到的砖的颜色大体为黑色（即蓝色和红色“相乘”）；红色的砖只反射红色的光，而蓝色的塑料薄膜只允许蓝色的光通过，红色的砖的反射光全部会被蓝色的塑料薄膜吸收，即呈现出黑色（参考文献：[《科学七年级下册》（ISBN: 9787553603162）/第 2 章对环境的感觉/第 4 节光的颜色/物体的颜色]）；如果需要模拟透射率相关的效果，那么我们应当使用参与介质（Participating Media）（2.[Yusor 2013]、3.[Hoobler 2016]）相关的技术。



物体的颜色

我们周围是一个五彩缤纷的世界。你也许会思考这样一个问题：物体为什么会五颜六色？

63

科学 七年级下

活动

让太阳光透过纸板的圆孔，照到白纸上，先后将红、蓝、绿三种透明的塑料薄膜蒙在纸板的圆孔上，你在白纸上会看到：红色的塑料薄膜通过的光呈_____色（说明只通过了_____光）；蓝色的塑料薄膜通过的光呈_____色（说明只通过了_____光）；绿色的塑料薄膜通过的光呈_____色（说明只通过了_____光）。

图2-42 透明体的颜色

我们看到物体的原因是眼睛接收到了物体反射出来的光。白色的物体能反射所有照射在它表面的光，眼睛接收到的是含有各种色光的混合光，因此感觉是“白色”的。相反，黑色物体是因为它吸收了所有照射在它表面的光，眼睛接收不到来自它的反射光，因此，感觉是“黑色”的。

物体呈现各种色彩也是因为它能反射不同颜色的光。例如，呈红色的物体是因为它只反射（或透过）红色光，而其他颜色的光几乎都被吸收了。当只用一束绿光（或其他非红色的光）照射呈现红色的物体时，物体便会呈现黑色。

图2-43 加上蓝色滤光镜后拍摄得到的照片

图来自：[《科学七年级下册》（ISBN: 9787553603162）/第 2 章对环境的感觉/第 4 节光的颜色/物体的颜色]

根据 Alpha 的含义，不难理解可见性函数$V(z_i) = \prod_{z_j \text{ Nearer } z_i}(1 - A_j)$，即只有比当前片元“更近”（Nearer）的片元才会局部覆盖当前片元。（一些文献中将可见性函数 $V(Z_i)$称作透射率 $T(Z_i)$，在严格意义上是错误的。）

顺序性透明

在实时渲染中，比较常见的做法是将几何体排序后用 Over/Under 操作（1.[Porter 1984]、4. [Dunn 2014]）以递归的方式求解 C_{Final}：

1.OpaquePass 绘制不透明物体，得到 BackgroundColor 和 BackgroundDepth。

2.TransparencyPass 将 BackgroundDepth 用于深度测试（关闭深度写入）将透明物体从后往前/从前往后排序后用 Over/Under 操作以递归的方式求解 C_{Final}。

Over 操作

将几何体从后往前排序后用 Over 操作以递归的方式求解 C_{Final}

C_{Final_0} = BackgroundColor

$C_{Final\_n} = (A_nC_n) + (1-A_n)C_{Final\_n-1}$

Under 操作

将几何体从前往后排序后用 Under 操作以递归的方式求解 C_{Final}

$C_{Final\_0} = 0$

$A_{Total\_0} = 1$

$C_{Final\_n} = A_{Total\_n-1}(A_nC_n) + C_{Final\_n-1}$

$A_{Total\_n} = A_{Total\_n-1}(1-A_n)$ //A_{Total} 即可见性函数 $V(Z_i)$

OpaquePass 得到的图像将在最后以 A = 1, C = BackgroundColor 的形式合成上去

在严格意义上，只有将片元从后往前/从前往后排序，才能保证 Over/Under 操作的正确性。而在实时渲染中，排序的粒度是基于物体而非基于片元；如果物体内部存在穿插，那么片元的顺序将不符合从后往前/从前往后，从而导致 Over/Under 操作的结果存在错误。因此，人们不得不探索 OIT 算法来解决这个问题。

//注：实际上，从前往后/从后往前的顺序还会导致相同材质的物体无法合批，从而导致状态切换过多，对性能造成不利影响。

深度剥离（Depth Peeling）

深度剥离（5.[Everitt 2001]）是一种比较古老的在实时渲染中被实际应用的 OIT 算法。

Render Pass

1.OpaquePas

绘制不透明物体，得到 BackgroundColor 和 BackgroundDepth。

2.NearestLayerPass

将 Depth 初始化为 BackgroundDepth 开启深度测试和深度写入 将透明物体按<材质,从前往后>排序后绘制得到 NearestLayerColor 和 NearestLayerDepth 并用 Under 操作将 NearestLayerColor 合成到 C_{Final}

3.SecondNearestLayerPass

将 Depth 初始化为 BackgroundDepth 开启深度测试和深度写入 将 NearestLayerDepth 绑定到片元着色器的纹理单元并在片元着色器中显式 Discard 掉 Depth NearerOrEqual NearestLayerDepth 的片元 将透明物体按<材质,从前往后>排序后绘制得到 SecondNearestLayerColor 和 SecondNearestLayerDepth 并用 Under 操作将 SecondNearestLayerColor 合成到 C_{Final}

4.ThirdNearestLayerPass

将 Depth 初始化为 BackgroundDepth 开启深度测试和深度写入 将 SecondNearestLayerDepth 绑定到片元着色器的纹理单元 以此类推...

... // N 个 Pass 可以剥离得到 N 个最近的层，应用程序可以根据自身的需求选择 Pass 的个数

N+2.CompositePass

最后用 Under 操作将 OpaquePass 得到的 BackgroundColor 合成到 C_{Final}

//注：深度剥离本身并不依赖于片元的顺序，之所以将透明物体从前往后排序是为了充分发挥硬件的 EarlyDepthTest 来提升性能

在理论上，深度剥离也可以从远到近剥离各层，并用 Over 操作合成到 C_{Final}。只不过，在 Under 操作中，如果应用程序选择的 Pass 个数过低，不能剥离得到所有的层，那么最远处的若干层会被忽略；由于 A_{Total}（即可见性函数 $V(Z_i)$）是单调递减的，最远处的若干层对 C_{Final} 的贡献是较低的，产生的误差也是较低的。这也是深度剥离采用 Under 操作而非 Over 操作的原因。

综合评价

显然，深度剥离有一个显著的缺陷——Pass 个数过多——在效率上存在着比较严重的问题；因此在被提出以后的数十年间并没有流行起来。

随机透明（Stochastic Transparency）

可见性函数$V(z_i) = \prod_{Z_j \text{ Nearer } Z_i}(1 - A_j)$的求解依赖于片元的顺序，导致了$C_{Final} = \sum_i(V(z_i)A_iC_i)$的求解依赖于片元的顺序。基于这个事实，Enderton 在 2010 年提出了随机透明：随机透明基于概率论的原理，利用硬件的 MSAA 特性进行随机抽样，给出了一种顺序无关的求解可见性函数 $V(Z_i)$的方式，以达到以顺序无关的方式求解 C_{Final} 的目的（6.[Enderton 2010]）。

随机深度（Stochastic Depth）

我们假设：通过设置 gl_SampleMask[]/SV_Coverage 的值，以确保片元[C_i A_i Z_i]在生成采样点[Z_i]时，每个采样点被覆盖的概率为 A_i；开启深度测试和深度写入，以确保较近的片元生成的采样点一定覆盖较远的片元生成的采样点；不同片元生成采样点时，每个采样点被覆盖的概率相互独立（Uncorrelated）；那么在最终生成的 Depth 图像中，对任意采样点[Z_s]，满足 Z_i NearerOrEqual Z_s 的概率即为可见性函数$V(z_i) = \prod_{Z_j \text{ Nearer } Z_i}(1 - A_j)$。

可以根据概率论的相关知识证明：满足 Z_i NearerOrEqual Z_s 即采样点[Z_s]被片元[C_i A_i Z_i]或被比片元[C_i A_i Z_i]更远的片元覆盖，即采样点[Z_s]不被比片元[C_i A_i Z_i]更近的片元[C_j A_j Z_j]覆盖，由于采样点[Z_s]不被比片元[C_i A_i Z_i]更近的片元[C_j A_j Z_j]覆盖的概率为 1 - A_j，且概率之间相互独立，最终的概率为各概率相乘，即$\prod_{Z_j \text{ Nearer } Z_i}(1 - A_j)$。

我们设，在 MSAA 中，1 个片元对应的采样点个数为 S，满足 Z_i NearerOrEqual Z_s 的采样点的个数为 Count(Z_i)，SV(Z_i) = Count(Z_i) / S；不难证明 SV(Z_i)的数学期望为 V(Z_i)，Count(Z_i)可以通过纹理采样得到，可以用 SV(Z_i)近似地表示可见性函数 V(Z_i)。

Alpha 校正（Alpha Correction）

不难证明：$\sum_i(V(Z_i)A_i) = 1 - \prod_i(1 - A_i)$

不妨设 A_0 = 0.4 A_1 = 0.7 A_2 = 0.6，我们有：

左边 = 0.4 + (1 - 0.4)×0.7 + (1 - 0.4)×(1 - 0.7)×0.6

= 0.4 + (1 - 0.4)×0.7 + (1 - 0.4)×(1 - 0.7)×0.6 + (1 - 0.4)×(1 - 0.7)×(1 - 0.6) - (1 - 0.4)×(1 - 0.7)×(1 - 0.6)

= 0.4 + (1 - 0.4)×0.7 + (1 - 0.4)×(1 - 0.7)×(0.6 + 1 - 0.6) - (1 - 0.4)×(1 - 0.7)×(1 - 0.6)

= 0.4 + (1 - 0.4)×0.7 + (1 - 0.4)×(1 - 0.7) - (1 - 0.4)×(1 - 0.7)×(1 - 0.6)

= 0.4 + (1 - 0.4)×(0.7 + 1 - 0.7) - (1 - 0.4)×(1 - 0.7)×(1 - 0.6)

= 0.4 + (1 - 0.4) - (1 - 0.4)×(1 - 0.7)×(1 - 0.6)

= 1 - (1 - 0.4)×(1 - 0.7)×(1 - 0.6) = 右边

不难证明：$\sum_i(SV(z_i)A_i)$的数学期望为$\sum_i(V(z_i)A_i)$即$1 - \prod_i(1 - A_i)$

Alpha 校正可以认为是对 SV(Z_i)进行归一化（Normalize），即假定$\frac{SV(Z_i)}{\sum_i(SV(Z_i)A_i)}=\frac{V(Z_i)}{\sum_i(V(Z_i)A_i)}$，从而得到$V(Z_i)=SV(Z_i)\frac{\sum_i(V(Z_i)A_i)}{\sum_i(SV(Z_i)A_i)}=SV(z_i)\frac{1-\prod_i(1-A_i)}{\sum_i(SV(Z_i)A_i)}$

Render Pass

1.OpaquePass

绘制不透明物体，得到 BackgroundColor 和 BackgroundDepth

2.StochasticDepthPass

将 Depth 初始化为 BackgroundDepth 开启 MSAA 开启深度测试（NearerOrEqual）和深度写入 用伪随机函数基于片元[C_i A_i Z_i]的 A_i 生成 gl_SampleMask[]/SV_Coverage 的值 将透明物体按<材质,从前往后>排序后绘制得到 StochasticDepth //注：随机透明本身并不依赖于片元的顺序，之所以从前往后排序是为了充分发挥硬件的 EarlyDepthTest 来提升性能

值得注意的是：

1.StochasticDepthPass 开启的 MSAA 是用于随机抽样的，随机透明本身并不要求除 StochasticDepthPass 以外的 Pass 开启 MSAA；如果应用程序有空间性反走样（Spatial AntiAliasing）的需求，那么可以在除 StochasticDepthPass 以外的 Pass 开启 MSAA（当然，也可以使用其它的空间性反走样算法（比如：FXAA））；除 StochasticDepthPass 以外的 Pass 用于空间反走样的 MSAA 和 StochasticDepthPass 用于随机抽样的 MSAA 并没有任何关系，比如：允许在用于空间反走样的 MSAA 是 4X 的同时，用于随机抽样的 MSAA 为 8X。

2.为了确保采样点被片元覆盖的概率相互独立，必须用伪随机函数基于片元[C_i A_i Z_i]的 A_i 生成 gl_SampleMask[]/SV_Coverage 的值，而不得使用硬件的 AlphaToCoverage 特性。

3.在 AccumulatePass 中计算 SV(Z_i)时，Z_i 为着色点的深度；为了保持一致，应当在片元着色器中将着色点的深度写入到 gl_FragDepth/SV_Depth（在默认情况下，采样点而非着色点的深度会被写入到最终生成的 Depth 图像中）。

4.由于硬件的限制，MSAA 最多为 8X，即 1 个片元对应的采样点的个数最多为 8；在论文原文中，作者提出可以使用多个 Pass 来模拟更多的采样点（6.[Enderton 2010]），但是出于效率的原因，实际应用中往往只使用 1 个 Pass。

3.AccumulateAndTotalAlphaPass

将 Depth 初始化为 BackgroundDepth 开启深度测试关闭深度写入 将 StochasticDepth 绑定到片元着色器的纹理单元并在片元着色器采样纹理得到 SV(Z_i) 开启 MRT 和 SeparateBlend/IndependentBlend 将透明物体按材质排序后绘制得到 StochasticColor = $\sum_i(SV(Z_i)A_iC_i)$、CorrectAlphaTotal = $\prod_i(1-A_i)$、StochasticTotalAlpha = $\sum_i(SV(Z_i)A_i)$ //注：由于关闭深度写入，透明物体的前后顺序不再对绘制的性能产生影响，只按材质排序；AlphaTotal 和 TotalAlpha 之间的关系为：TotalAlpha = 1 – AlphaTotal，术语"TotalAlpha"来自随机透明（6.[Enderton 2010]），术语"AlphaTotal"来自 Under 操作（1.[Porter 1984]、4. [Dunn 2014]）；StochasticTotalAlpha 只有在启用 Alpha 校正时才会被用到（理论上，在没有启用 Alpha 校正时，可以省 1 个 RT）

注：在论文原文中，AccumulatePass 和 TotalAlphaPass 是 2 个分离的 Pass（6.[Enderton 2010]）；但实际上，完全可以将它们合并到同一个 Pass；这种情况的出现可能是由于 SeparateBlend/IndependentBlend 在论文发表时并没有被硬件广泛支持。

4.CompositePass

在没有启用 Alpha 校正时，透明物体对 C_{Final} 的总贡献为：TransparentColor = $\sum_i(SV(Z_i)A_iC_i)$ = StochasticColor

在启用 Alpha 校正时，透明物体对 C_{Final} 的总贡献为：TransparentColor = $\begin{cases}\left(\sum_i\left(\left(SV(Z_i)\frac{1-\prod_i(1-A_i)}{\sum_i(SV(Z_i)A_i)}\right)A_iC_i\right)\right) & //\sum_i(SV(Z_i)A_i)>0\\ 0 & //\sum_i(SV(Z_i)A_i)=0\end{cases}=\begin{cases}(\sum_i(SV(Z_i)A_iC_i))\frac{1-\prod_i(1-A_i)}{\sum_i(SV(Z_i)A_i)} & //\sum_i(SV(Z_i)A_i)>0\\ 0 & //\sum_i(SV(Z_i)A_i)=0\end{cases}=$

$$\begin{cases} \text{StochasticColor}\frac{1-\text{CorrectAlphaTotal}}{\text{StochasticTotalAlpha}} & //\text{StochasticTotalAlpha} > 0 \\ 0 & //\text{StochasticTotalAlpha} = 0 \end{cases}$$ //注：显然，不透明物体的 StochasticTotalAlpha 为 0；但是，由于采样点是随机生成的，透明物体的 StochasticTotalAlpha 也可能为 0

随后，基于 CorrectAlphaTotal 用 Under 操作将 OpaquePass 得到的 BackgroundColor 合成到 C_{Final}

Tile/On-Chip Memory

随机透明在本质上是比较适合移动 GPU 的。

在传统的桌面 GPU 上，随机透明的性能瓶颈在于 MSAA，1 个片元对应于 S 个采样点的 MSAA 会使带宽的开销增加 S 倍。

然而在移动 GPU 上，这个问题得到了有效的解决，可以将开启 MSAA 的图像保存在 Tile/On-Chip Memory 中，并在 RenderPass 结束后丢弃，并不会与主存进行通信，从而将带宽开销降低到几乎为零。次世代的 API 允许应用程序显式地对此进行设置：使用 VK_IMAGE_USAGE_TRANSIENT_ATTACHMENT_BIT(Vulkan) / MTLStorageModeMemoryless(Metal)可以将图像的存储模式显式地设置为 Tile/On-Chip Memory（在 RenderPass 结束后丢弃，并不会写回主存）；不过在片元着色器（Fragment Shader）中，使用该存储模式的图像不再被允许用传统的 TextureUnit 来读取，而必须用 Subpass Input(Vulkan) / [color(m)]Attribute(Metal)来读取。传统的 API 并不允许将图像的存储模式显式地设置为 Tile/On-Chip Memory，但是可以用 FrameBufferFetch(OpenGL) / PixelLocalStorage(OpenGL)进行暗示（16.[Bjorge 2014] ）。

Vulkan

在 Vulkan 中，1 个 RenderPass 由若干个 SubPass 组成，RenderPass 中的不同 Attachment 的 MSAA 设置并不要求相同，但是同一 SubPass 引用的所有 ColorAttachment 和 DepthStencilAttachment 的 MSAA 设置应当相同（即与调用 DrawCall 时所绑定的 PipelineState 中的 MultisampleState 相同）。

随机透明可以在 1 个 RenderPass 中实现，具体如下： //假设应用程序并没有开启 MSAA 用于空间反走样

RenderPass:
  Attachment:
    0.BackGroupColor
    1.BackGroupDepth
    2.StochasticDepth (MSAA)
    3.StochasticColor
    4.CorrectAlphaTotal
    5.StochasticTotalAlpha
    6.FinalColor
  SubPass:
    0.OpaquePass:
      ColorAttachment: 0.BackGroupColor
      DepthStencilAttachment: 1.BackGroupDepth
    1.__CopyPass__Mentioned_In_StochasticDepthPass_Above_: //Rasterization MSAA
      InputAttachment: 1.BackGroupDepth
      ColorAttachment: 2.StochasticDepth (MSAA)
    2.StochasticDepthPass: //Rasterization MSAA
      DepthStencilAttachment: 2.StochasticDepth (MSAA)

3.AccumulateAndTotalAlphaPass:
InputAttachment: 2.StochasticDepth (MSAA)
ColorAttachment: 3.StochasticColor 4.CorrectAlphaTotal 5.StochasticTotalAlpha
DepthStencilAttachment: 1.BackGroupDepth
4.CompositePass:
InputAttachment: 0.BackGroupColor 3.StochasticColor 4.CorrectAlphaTotal 5.StochasticTotalAlpha
ColorAttachment: 6.FinalColor
Dependency:
0.SrcSubPass:0 -> DstSubPass:1
//DepthStencilAttachment->InputAttachment: 1.BackGroupDepth
1.SrcSubPass:1 -> DstSubPass:2
//ColorAttachment->DepthStencilAttachment: 2.StochasticDepth (MSAA)
2.SrcSubPass:2 -> DstSubPass:3
//DepthStencilAttachment->InputAttachment: 2.StochasticDepth (MSAA)
3.SrcSubPass:3 -> DstSubPass:4
//ColorAttachment->InputAttachment: 0.BackGroupColor //SubPassDependencyChain: 0->1->2->3->4
//ColorAttachment->InputAttachment: 3.StochasticColor
//ColorAttachment->InputAttachment: 4.CorrectAlphaTotal
//ColorAttachment->InputAttachment: 5.StochasticTotalAlpha

Metal

Metal 在 API 层面并没有 InputAttachment 的概念，而是通过[color(m)]Attribute 允许在片元着色器中读取 ColorAttachment；但是，这样的设计存在着缺陷：[color(m)]Attribute 只允许读取 ColorAttachment，而不允许读取 DepthAttachment，需要增加 1 个额外的 ColorAttachment 并将 Depth 写入到该 ColorAttachment 中（17.[Apple]）。

并且，在 Metal 中开启 MSAA 时，通过[color(m)]Attribute 读取 ColorAttachment 会导致片元着色器对每个采样点执行一次，得到 ColorAttachment 在该采样点处的值，从而导致无法求解 SV(Z_i)（因为一次在片元着色器的执行中，我们无法得到所有采样点的数据，而只能得到某一个采样点的数据）；因此，硬件的 MSAA 不可用，我们只能尝试用多个 ColorAttachment 来模拟 MSAA，考虑到 ColorAttachment 的存在着个数上限（A7->4 个 A8,A9,A10,A11->8 个）和大小上限（A7->128 位 A8,A9,A10->256 位 A11->512 位），最多可以模拟 20X MSAA；由于没有开启硬件的 MSAA，硬件的深度测试不可用（DepthAttachment 中只有 1 个采样点），只能在片元着色器中基于可编程融合以软件的方式模拟 MSAA 的深度测试和深度写入，硬件会保证该 RMW 操作的原子性（下文在介绍 K-Buffer 时会对可编程融合的具体细节进行介绍）。

//注：Metal 中不存在 SubPass 的概念，因此缺少某种将 DepthAttachment 转换成 InputAttachment 的屏障（Barrier）机制。

随机透明在 Metal 中也可以在 1 个 RenderPass 中实现，具体如下： //假设应用程序并没有开启 MSAA 用于空间反走样

RenderPassDescriptor:
ColorAttachment:
0.FinalColor //Load:Clear //Store:Store //Format:R10G10B10A2_UNORM //HDR10
1.StochasticDepth0123 //Load:Clear //Store:DontCare //Format:R16G16B16A16_FLOAT
2.StochasticDepth4567 //Load:Clear //Store:DontCare //Format:R16G16B16A16_FLOAT
3.StochasticDepth89AB //Load:Clear //Store:DontCare //Format:R16G16B16A16_FLOAT
4.StochasticDepthCDEF //Load:Clear //Store:DontCare //Format:R16G16B16A16_FLOAT

5.StochasticColor //Load:Clear //Store:DontCare //Format:R10G10B10A2_UNORM //HDR10

6.CorrectAlphaTotal //Load:Clear //Store:DontCare //Format:R8_UNORM

7.StochasticTotalAlpha //Load:Clear //Store:DontCare //Format:R16_FLOAT //R8 的精度不够

//注：可以将[color(6)]和[color(7)]合并到同一个 ColorAttachment 后，再增加一个 StochasticDepthGHIJ，即可模拟 20X MSAA。

DepthAttachment:

Depth //Load:Clear //Store:DontCare

RenderCommandEncoder:

0.OpaquePass:

BackgroundColor->Color[0]

BackgroundDepth->Depth

1.StochasticDepthPass:

//复用 Depth 中的 BackgroundDepth，开启深度测试，关闭深度写入；同时开启[[early_fragment_tests]]，使用硬件的深度测试剔除掉比“Background”更远的片元

...->SampleMask

Read: color[1]/color[2]/color[3]/color[4]->...

Modify: ...

Write: ...->color[1]/color[2]/color[3]/color[4]

2.AccumulateAndTotalAlphaPass:

//复用 Depth 中的 BackgroundDepth

color[1]/color[2]/color[3]/color[4]->SV(Z_i)

...->Color[5]

...->Color[6]

...->Color[7]

3.CompositePass:

Color[5]->...

Color[6]->...

Color[7]->...

...->TransparentColor

...->CorrectAlphaTotal

Color[0]->BackgroundColor

...-> Color[0]

综合评价

由于移动 GPU 上的 MSAA 是高效的，随机透明在本质上是比较适合移动 GPU 的。我们可以使用次世代 API 充分挖掘移动 GPU 的相关优势。但是，由于 Metal 在设计上的缺陷，导致我们不得不在片元着色器中基于可编程融合以软件的方式模拟 MSAA 的深度测试和深度写入；不过，Metal 却又允许我们在一个几何体 Pass 中模拟最多 20X 的 MSAA（在桌面 GPU 上，需要用多个 Pass 才能模拟 8X 以上 MSAA）。（当然，在 Metal 上像桌面 GPU 那样使用多个 RenderPass 绘制并不会产生这些问题；但是，很有可能会导致开启 MSAA 的图像被从 Tile/On-Chip Memory 中写回主存，产生大量的带宽开销）

移动 GPU 擅长片元处理而不擅长几何处理（10.[Harris 2019]），随机透明的一个缺陷在于：随机透明需要 2 个几何体 Pass（StochasticDepthPass 和 AccumulateAndTotalAlphaPass），这可能会使几何处理成为性能的瓶颈。

Demo

Demo 地址：https://gitee.com/YuqiaoZhang/StochasticTransparency。该 Demo 改编自 NVIDIA SDK11 Samples 中的

StochasticTransparency（9.[Bavoil 2011]），在 NVIDIA 提供的原始代码中，存在着 3 个比较严重的问题：

1.我在前文中指出：“随机透明本身并不要求除 StochasticDepthPass 以外的 Pass 开启 MSAA”； 在 NVIDIA 提供的原始代码中，所有 Pass 都使用了相同的 MSAA 设置，导致随机透明的帧率反而低于深度剥离（个人测试的结果是：修正该问题后，帧率从 670 提升至 1170（用于对比的深度剥离为 1070））。

2.我在前文中指出：“论文原文中的 2 个分离的 Pass（AccumulatePass 和 TotalAlphaPass）应当合并到同一个 Pass”；NVIDIA 提供的原始代码并没有这么做（个人测试的结果是：修正该问题后，帧率从 1170 提升至 1370）。

3.我在前文中指出：“在 AccumulatePass 中计算 $SV(Z_i)$时，Z_i 为着色点的深度；为了保持一致，在 StochasticDepthPass 中应当在片元着色器中将着色点的深度写入到 gl_FragDepth/SV_Depth”；NVIDIA 提供的原始代码并没有这么做，导致在求解 $SV(Z_i)$时，Z_i Equal Z_s 几乎不可能成立，产生较大的误差；不过 Alpha 校正可以很好地修正这个误差，在效果上并没有产生太大的影响。

K-Buffer

在 Porter 提出 Alpha 通道的同一年，Carpenter 提出了 A-Buffer：在 A-Buffer 中，每个像素对应于一个链表，存放对应到该像素的所有片元；基于深度对链表中的片元排序后，用 Over/Under 操作即可得到 C_{Final}（11.[Carpenter 1984]）。虽然，目前的硬件在理论上已经可以通过 UAV(Direct3D)和原子操作实现 A-Buffer，但是，由于实现的过程极其繁琐（编程是一门艺术，A-Buffer 的实现极不优雅）且效率低下（主要是链表的地址不连续导致缓存命中率下降），几乎不存在 A-Buffer 的实际应用。

在 2007 年，Bavoil 在 A-Buffer 的基础上进行了改进，将每个像素对应的片元个数限定为 K 个，提出了更具有实用价值的 K-Buffer（12.[Bavoil 2007]）。

RMW 操作

在生成 K-Buffer 的 Pass 中，在每个片元生成时会进行以下 RMW（Read Modify Write，读取-修改-写入）操作：

1.Read：读取当前片元所对应的像素所对应的 K 个片元。

2.Modify：结合当前片元，对读取得到的 K 个片元进行修改。 //在 OIT 算法中，一般是将当前片元插入到这 K 个片元中得到 K+1 个片元，并找出两个“最接近”的片元进行融合，再次得到 K 个片元

3.Write：将修改后的 K 个片元写入当前片元所对应的像素。

对目前的硬件而言，只要在片元着色器中访问 StorageImage(OpenGL/Vulkan) / UAV(Direct3D)，就可以做到每个像素对应于 K 个片元并且在每个片元生成时对这 K 个片元进行 RMW 操作。但是，事情远远没有这么简单，一般而言，对应于同一像素的不同片元的 RMW 操作必须“互斥”才能保证最终结果是的正确性。

在 API 层面，对应于同一像素的不同片元之间的同步发生在 Alpha 融合阶段，也就是说，这些片元在片元着色器阶段是并行执行的。

在桌面 GPU 上，对应于同一像素的不同片元的 RMW 操作的“竞态条件（Race Condition）”显得尤其显著。

不妨回忆，在顺序性透明中，我们将物体从前往后/从前往后排序，并依次调用 Draw Call，这意味应用程序调用 Draw Call 的顺序隐含着某种依赖关系。但是，GPU 的设计者往往希望尽可能地提升并行度以充分挖掘 GPU 的性能；在实际中，GPU 仍并行地处理这些 Draw Call，只不过会在某个同步点进行同步，使最终结果满足应用程序所期望的依赖关系。

在桌面 GPU（Sort Last Fragment）上，这个同步点发生在片元着色器之后 Alpha 融合之前的 Reorder Buffer 中（13.[Ragan-Kelley 2011]），也就是说，即使在 Draw Call 顺序上存在着依赖关系的片元，在 GPU 中也是并行处理的（即存在竞态条件），更不用说在同一个 Draw Call 中（即在应用层不存在任何期望的依赖关系）的片元。

在移动 GPU（Sort Middle）上，由于没有 Reorder Buffer 的存在，对应于同一像素的不同片元是串行执行的

（13.[Ragan-Kelley 2011]）。这也是移动 GPU 并不需要 Depth PrePass（10.[Harris 2019]）的原因，GPU 会在 EarlyDepthTest 阶段确定最终覆盖像素的片元，并只为该片元执行片元着色器；由于在大多数情况下（比如绘制不透明物体），最终覆盖像素的片元只有一个，串行执行并不会对性能有太大的影响。这也解释了移动 GPU 排斥 Discard 的原因，Discard 会导致 GPU 无法确定最终覆盖像素的片元，可能会导致对应于同一像素的多个片元被执行，然而在移动 GPU 上，这些片元是串行执行的，在效率上低于桌面 GPU（并行执行）。

但是，由于片元着色器阶段在 API 层面被看作是并行执行的，而 GPU 硬件的内部实现是不公开的，应用程序还是不应当作出“对应于同一像素的不同片元在访问 StorageImage(OpenGL/Vulkan) / UAV(Direct3D)时的 RMW 操作不存在竞态条件”的假定。还是有很多其它潜在的因素可能会对此造成影响，比如：执行依赖并不代表内存依赖，由于 StorageImage(OpenGL/Vulkan) / UAV(Direct3D)并不保存在 Tile/On-Chip Memory 中，考虑到缓存机制的存在，虽然，对应于同一像素的不同片元是串行执行的，但是，之前的片元对 StorageImage(OpenGL/Vulkan) / UAV(Direct3D)的写入并不一定会对之后的片元可见（即之前的片元写入到了缓存，而之后的片元从主存中读取；至少在 API 层面完全允许 GPU 的设计者这么做）。 //个人认为还是有必要通过实验来确定

Bavoil 在 2007 年提出 K-Buffer 时，同时提出了两种硬件上的设计——片元调度（Fragment Schedule）和可编程融合（Programmable Blending）——来解决对应于同一像素的不同片元的 RMW 操作存在竞态条件的问题（12.[Bavoil 2007]）。目前，这两种设计都已经在实际中被硬件广泛支持。

片元调度（Fragment Scheduling）

片元调度对应于目前的 RasterOrderView(Direct3D) / FragmentShaderInterlock(OpenGL/Vulkan) / RasterOrderGroup(Metal)（14.[D 2015]、15.[D 2017]），往往适用于桌面 GPU。

使用片元调度实现 K-Buffer 的片元着色器的代码大致如下：

```
    Do Shade //这部分代码并不需要互斥

//Enter Critical Section //进入临界区
#if RasterOrderView(Direct3D)
Read From ROV
#elif FragmentShaderInterlock(OpenGL/Vulkan)
beginInvocationInterlockARB
#elif RasterOrderGroup(Metal)
Read From ROG
#endif

    Do K-Buffer RMW //这部分代码处于临界区保护内

//Leave Critical Section //离开临界区
#if RasterOrderView(Direct3D)
Write To ROV
#elif FragmentShaderInterlock(OpenGL/Vulkan)
endInvocationInterlockARB
#elif RasterOrderGroup(Metal)
Write To ROG
#endif
```

//注：在理论上，对 ROV/ROG 读写的内容并不重要，读写 ROV/ROG 只是为了进入/离开临界区（从这一点上，

OpenGL/Vulkan 的设计更为优雅）；“Do K-Buffer RMW”已经处于临界区的保护之中，不再有读写 ROV/ROG 的必要，K-Buffer 的存储只需要使用常规的 UAV(Direct3D) / StorageImage(OpenGL/Vulkan)即可（14.[D 2017]）。

可编程融合（Programmable Blending）

可编程融合对应于目前的 FrameBufferFetch(OpenGL) / [color(m)]Attribute(Metal)（16.[Bjorge 2014]、17.[Apple]），往往适用于移动 GPU。

可编程融合允许在片元着色器中读取 ColorAttachment，对 ColorAttachment 进行 RMW 操作，硬件会保证对应于同一像素的不同片元对同一 ColorAttachment 的 RMW 操作的互斥性。我们只需要开启 MRT，就可以基于可编程融合实现 K-Buffer。比如，在 OIT 算法中，我们需要实现 1 个像素对应于 4 个片元[C A Z]构成的 K-Buffer，相关的片元着色器代码（基于 Metal）大致如下：

```
struct KBuffer_ColorAttachment
{
   //一般[[color(0)]]是用于存放 CFinal 的
   half4 C0A0[[color(1)]]; //R8G8B8A8_UNORM
   half4 C1A1[[color(2)]]; //R8G8B8A8_UNORM
   half4 C2A2[[color(3)]]; //R8G8B8A8_UNORM
   half4 C3A3[[color(4)]]; //R8G8B8A8_UNORM
   half4 Z0123[[color(5)]]; //R16G16B16A16_FLOAT
};

struct KBuffer_Local
{
   half4 CA[4]
   half Z[4]
};

 fragment KBuffer_ColorAttachment KBufferPass_FragmentMain(..., KBuffer_ColorAttachment kbuffer_in)
 {
      CA = Shade(...) //这部分代码并不需要互斥
      Z = ... //一般即 position.z

      KBuffer_Local kbuffer_local;

      //KBuffer Read 操作
      kbuffer_local.Z[0] = kbuffer_in.Z0123.r; //对 ColorAttachment 的 Read 操作会进入临界区
      kbuffer_local.Z[1] = kbuffer_in.Z0123.g;
      kbuffer_local.Z[2] = kbuffer_in.Z0123.b;
      kbuffer_local.Z[3] = kbuffer_in.Z0123.a;
      kbuffer_local.CA[0] = kbuffer_in.C0A0;
      kbuffer_local.CA[1] = kbuffer_in.C1A1;
      kbuffer_local.CA[2] = kbuffer_in.C2A2;
      kbuffer_local.CA[3] = kbuffer_in.C3A3;

      //KBuffer Modify 操作
```

```
        … //根据应用程序的具体需求//这部分代码处于临界区保护内

        //KBuffer Write 操作
        KBuffer_ColorAttachment kbuffer_out;
        kbuffer_out.C0A0 = kbuffer_local.CA[0];
        kbuffer_out.C1A1 = kbuffer_local.CA[1];
        kbuffer_out.C2A2 = kbuffer_local.CA[2];
        kbuffer_out.C3A3 = kbuffer_local.CA[3];
        kbuffer_out.Z0123 = half4(kbuffer_local.Z[0], kbuffer_local.Z[1], kbuffer_local.Z[2], kbuffer_local.Z[3]); // 对 ColorAttachment 的 Write 操作会离开临界区
        return kbuffer_out;
    }
```

MLAB（Mult Layer Alpha Blending，多层 Alpha 融合）

Salvi 分别在 2010 年、2011 年、2014 年提出的 OIT 算法全都是基于 K-Buffer 实现的（18.[Salvi 2010]、19.[Salvi 2011]、20.[Salvi 2014]），我们选取最新的（即 2014 年）的 MLAB 进行介绍。

K-Buffer

MLAB 将 K-Buffer 中片元的格式定义为[A_iC_i 1 - A_i Z_i]。

首先将 K-Buffer 中的片元全部初始化为 $C_i = 0, A_i = 0, Z_i =$ 无限远（即[0 1 无限远]）的“空片元”。 //在实际实现中，由于 Z_i 的取值在 0 到 1 之间，只需要保证比 Z_i 所有可能的取值都远即可。

在片元生成时，K-Buffer 的 Modify 操作会根据 Z_i 从近到远排序，将当前片元插入到合适的位置，得到 K+1 个片元；再基于 Under 操作的规则，将最远的 2 个片元融合（[A_iC_i 1 - A_i Z_i]和[$A_{i+1}C_{i+1}$ 1 - A_{i+1} Z_{i+1}]融合后得到[$A_iC_i + (1 - A_i)A_{i+1}C_{i+1}$ $(1 - A_i)(1 - A_{i+1})Z_i$] //注：这个融合规则兼容初始化产生的“空片元”），再次得到 K 个片元。 //当之后插入的片元在两个被融合的片元之间时，会产生误差；根据 Salvi 的说法，融合最远的 2 个片元的误差是最小的，这可能与较远的片元对 C_{Final} 的贡献较低有一定关系（可见性函数 $V(Z_i)$是单调递减的，较远的片元的 $V(Z_i)$的值较低）。

最后，使用 Under 操作，基于 K-Buffer 中的 K 个片元，求出透明物体对 C_{Final} 的总贡献。

Render Pass

MLAB 涉及到的 RenderPass 如下：

1.OpaquePass

绘制不透明物体，得到 BackgroundColor 和 BackgroundDepth

2.KBufferPass

复用 OpaquePass 得到的 BackgroundDepth 开启深度测试关闭深度写入 用 Clear 操作将 K-Buffer 中的片元全部初始化[0 1 无限远] 将透明物体按材质排序后绘制得到 K-Buffer //注：由于关闭深度写入，透明物体的前后顺序不再对绘制的性能产生影响，只按材质排序

3.CompositePass

用 Under 操作，基于 K-Buffer 中的 K 个片元，求出透明物体对 C_{Final} 的总贡献 TransparentColor 和 AlphaTotal

随后，基于 AlphaTotal 用 Under 操作将 OpaquePass 得到的 BackgroundColor 合成到 C_{Final}

Tile/On-Chip Memory

K-Buffer 在本质上是比较适合移动 GPU 的。

在传统的桌面 GPU 上，K-Buffer 会使带宽的开销增加 K 倍。

同样地，在移动 GPU 上，我们也可以用次世代 API 将 K-Buffer 保存在 Tile/On-Chip Memory 中，不与主存进行通信，从而将带宽开销降低到几乎为零。

Vulkan

由于 Vulkan 尚未支持 FrameBufferFetch，目前无法用 Vulkan 基于可编程融合实现 K-Buffer（当然，可以用 Vulkan 基于片元调度实现 K-Buffer，只是片元调度并不适用于移动 GPU；不过，OpenGL 支持 FrameBufferFetch，可以考虑用 OpenGL 基于可编程融合实现 K-Buffer，只是 OpenGL 无法做到将 K-Buffer 保存在 Tile/On-Chip Memory 中）。

Metal

MLAB 在 Metal 中可以在 1 个 RenderPass 中实现，具体如下： //假设 K-Buffer 中的 K 值为 4

RenderPassDescriptor:

ColorAttachment:

0.FinalColor //Load:Clear //Store:Store //Format:R10G10B10A2_UNORM //HDR10

1.AC0_1SubA0 //Load:Clear //ClearValue:[0 0 0 1] //Store:DontCare //Format:R16G16B16A16_FLOAT

2.AC1_1SubA1 //Load:Clear //ClearValue:[0 0 0 1] //Store:DontCare //Format:R8G8B8A8_UNORM //PixelStorage 的大小存在着上限（A7->128 位 A8,A9,A10->256 位 A11->512 位）；由于可见性函数单调递减，较远的片元贡献较低，优先考虑降低精度

3.AC2_1SubA2 //Load:Clear //ClearValue:[0 0 0 1] //Store:DontCare //Format:R8G8B8A8_UNORM

4.AC3_1SubA3 //Load:Clear //ClearValue:[0 0 0 1] //Store:DontCare //Format:R8G8B8A8_UNORM

5.Z0123 //Load:Clear //ClearValue:[无限远 无限远 无限远 无限远] //Store:DontCare //Format:R16G16B16A16_FLOAT

DepthAttachment:

Depth //Load:Clear //Store:DontCare

RenderCommandEncoder:

0.OpaquePass:

BackgroundColor->Color[0]

BackgroundDepth->Depth

1.KBufferPass:

Read: Color[1]/Color[2]/Color[3]/Color[4]/Color[5]->4 个 Fragment

Modify: ...

Write: 4 个 Fragment->Color[1]/Color[2]/Color[3]/Color[4]/Color[5]

//复用 Depth 中的 BackgroundDepth

3.CompositePass:

Color[1]/Color[2]/Color[3]/Color[4]/Color[5]->...

...->TransparentColor

...->AlphaTotal

Color[0]->BackgroundColor

...->Color[0]

综合评价

由于移动 GPU 上的 K-Buffer 是高效的，MLAB 在本质上是比较适合移动 GPU 的。我们可以使用次世代 API 充分挖掘移动 GPU 的相关优势。但是，由于 Vulkan 尚未支持 FrameBufferFetch，目前无法用 Vulkan 基于可编程融合实现 K-Buffer（不过，OpenGL 支持 FrameBufferFetch，可以考虑用 OpenGL 基于可编程融合实现 K-Buffer，只是 OpenGL 无法做到将 K-Buffer 保存在 Tile/On-Chip Memory 中）。

相对于随机透明而言，MLAB 有一个明显的优势：只需要一个几何体 Pass（KBufferPass），这对不擅长几何处理（10.[Harris 2019]）的移动 GPU 而言是一个不错的福音。但是，K-Buffer 的 RWM 操作必须“互斥”，对桌面 GPU 而言，会导致 GPU 无法并行处理对应于同一像素的片元，从而引入额外的开销，开销的大小取决于场景的深度复杂度（Depth Complexity）。（由于移动 GPU 本身就是串行处理的（13.[Ragan-Kelley 2011]），并不引入额外的开销）

Demo

Demo 地址：https://gitee.com/YuqiaoZhang/MultiLayerAlphaBlending。该 Demo 改编自 Metal Sample Code 中的 Order Independent Transparency with Imageblocks（21.[Imbrogno 2017]）。在 Apple 提供的原始代码中，用到了 A11 GPU（iPhone 8 以后）中才具有的特性 Imageblock；但是，PixelLocalStorage(OpenGL) / Imageblock(Metal)的本质是允许自定义 FrameBuffer 中像素的格式（16.[Bjorge 2014]、21.[Imbrogno 2017]），这与 K-Buffer 的本质（为 RMW 操作构造临界区）并没有任何关系；于是我对 Demo 进行了修改，使用[color(m)]Attribute 实现了相关的功能，在 iPhone 6（A8 GPU）上顺利运行，粉碎了 Apple 企图忽悠我换 iPhone 新机型的阴谋！

//注：Metal 特性和 OpenGL 特性之间的对应关系为：[Color(m)]Attribute 对应于 FrameBufferFetch，本质是用于实现可编程融合；而 ImageBlock 对应于 PixelLocalStorage，本质是用于自定义像素格式。

权重融合（Weighted Blended）

可见性函数$V(z_i) = \prod_{z_j \text{ Nearer } z_i}(1 - A_j)$的求解依赖于片元的顺序，导致了$C_{Final} = \sum_i(V(z_i)A_iC_i)$的求解依赖于片元的顺序。同样是基于这个事实，McGuire 在 2013 年提出了权重融合：用一个预定义的权重函数 $W(EyeSpaceZ_i\ A_i)$作为可见性函数 $V(Z_i)$的估计值，从而达到以顺序无关的方式求解 C_{Final} 的目的（22.[McGuire 2013]、4.[Dunn 2014]）。

权重函数

McGuire 认为，只依赖于 $EyeSpaceZ_i$ 的权重函数可能会导致 A_i 较低的“极近”的片元对 C_{Final} 产生过大的影响，权重函数应当同时依赖于 $EyeSpaceZ_i$ 和 A_i。同时，作者给出了三个建议的权重函数（经作者验证，当 $EyeSpaceZ_i$ 的范围在 0.1 到 500 之间且 $EyeSpaceZ_i$ 为 16 位浮点数时，效果良好）：

1.$W(EyeSpaceZ_i\ A_i)$ = clamp(10.0f / (0.00001f + pow($EyeSpaceZ_i$ / 5.0f, 2.0f) + pow($EyeSpaceZ_i$ / 200.0f, 6.0f)), 0.01f, 3000.0f) * A_i

2.$W(EyeSpaceZ_i\ A_i)$ = clamp(10.0f / (0.00001f + pow($EyeSpaceZ_i$ / 10.0f, 3.0f) + pow($EyeSpaceZ_i$ / 200.0f, 6.0f)), 0.01f, 3000.0f) * A_i

3.$W(EyeSpaceZ_i\ A_i)$ = clamp(0.03f / (0.00001f + pow($EyeSpaceZ_i$ / 200.0f, 4.0f)), 0.01f, 3000.0f) * A_i

//注：根据定义，可见性函数 $V(Z_i)$不可能超过 1；但是，当片元“极近”时，权重函数的值却可能高达 3000，这可能是 McGuire 认为权重函数需要依赖于 A_i 的原因。

归一化

在讨论随机透明的 Alpha 校正时，我们已经证明过：$\sum_i(V(Z_i)A_i) = 1 - \prod_i(1 - A_i)$

归一化即假定 $\frac{W(EyeSpaceZ_i\ A_i)}{\sum_i(W(EyeSpaceZ_i\ A_i)A_i)} = \frac{V(Z_i)}{\sum_i(V(Z_i)A_i)}$，从而得到 $V(Z_i) = W(EyeSpaceZ_i\ A_i)\frac{\sum_i(V(Z_i)A_i)}{\sum_i(W(EyeSpaceZ_i\ A_i)A_i)} = W(EyeSpaceZ_i\ A_i)\frac{1-\prod_i(1-A_i)}{\sum_i(W(EyeSpaceZ_i\ A_i)A_i)}$

Render Pass

1.OpaquePass

绘制不透明物体，得到 BackgroundColor 和 BackgroundDepth

2.AccumulateAndTotalAlphaPass

将 Depth 初始化为 BackgroundDepth 开启深度测试关闭深度写入 将透明物体按材质排序后绘制得到 WeightedColor = $\sum_i(W(EyeSpaceZ_i\ A_i)A_iC_i)$ 、 CorrectAlphaTotal = $\prod_i(1 - A_i)$ 、 WeightedTotalAlpha = $\sum_i(W(EyeSpaceZ_i\ A_i)A_i)$ //注：由于关闭深度写入，透明物体的前后顺序不再对绘制的性能产生影响，只按材质排序；AlphaTotal 和 TotalAlpha 之间的关系为：TotalAlpha = 1 − AlphaTotal，术语”TotalAlpha”来自随机透明（6.[Enderton 2010]），术语”AlphaTotal”来自 Under 操作（1.[Porter 1984]、4. [Dunn 2014]）

3.CompositePass

透明物体对 C_{Final} 的总贡献为：TransparentColor = $\sum_i\left(\left(W(EyeSpaceZ_i\ A_i)\frac{1-\prod_i(1-A_i)}{\sum_i(W(EyeSpaceZ_i\ A_i)A_i)}\right)A_iC_i\right) = (\sum_i(W(EyeSpaceZ_i\ A_i)A_iC_i))\frac{1-\prod_i(1-A_i)}{\sum_i(W(EyeSpaceZ_i\ A_i)A_i)} = WeightedColor\ \frac{1 - CorrectAlphaTotal}{WeightedTotalAlpha}$

随后，基于 CorrectAlphaTotal 用 Under 操作将 OpaquePass 得到的 BackgroundColor 合成到 C_{Final}

综合评价

权重融合用预定义的权重函数 $W(EyeSpaceZ_i\ A_i)$近似地表示可见性函数 $V(Z_i)$，省去了求解可见性函数 $V(Z_i)$的过程，在某种程度上可以认为是随机透明的简化版（省去了 StochasticDepthPass）。当然，权重融合的误差也是最大的，因为作为可见性函数 $V(Z_i)$估计值的权重函数 $W(EyeSpaceZ_i\ A_i)$与场景中的实际情况不存在任何关系。

Demo

Demo 地址：https://gitee.com/YuqiaoZhang/WeightedBlendedOIT。该 Demo 改编自 NVIDIA GameWorks Vulkan and OpenGL Samples 中的 Weighted Blended Order-independent Transparency（23.[NVIDIA]）。权重融合是所有 OIT 算法中

最简单的，我也并没有对 Demo 作任何实质性的修改。

参考文献

1.[Porter 1984] Thomas Porter, Tom Duff. "Compositing Digital Images." SIGGRAPH 1984.
https://keithp.com/~keithp/porterduff/p253-porter.pdf

2.[Yusor 2013] Egor Yusor. "Practical Implementation of Light Scattering Effects Using Epipolar Sampling and 1D Min/Max Binary Trees." GDC 2013.
https://software.intel.com/en-us/blogs/2013/03/18/gtd-light-scattering-sample-updated
https://software.intel.com/en-us/blogs/2013/06/26/outdoor-light-scattering-sample
https://software.intel.com/en-us/blogs/2013/09/19/otdoor-light-scattering-sample-update

3.[Hoobler 2016] Nathan Hoobler. "Fast, Flexible, Physically-Based Volumetric Light Scattering." GDC 2016.
http://developer.nvidia.com/VolumetricLighting

4.[Dunn 2014] Alex Dunn. "Transparency (or Translucency) Rendering." NVIDIA GameWorks Blog 2014.
https://developer.nvidia.com/content/transparency-or-translucency-rendering

5.[Everitt 2001] Cass Everitt. "Interactive Order-Independent Transparency." NVIDIA WhitePaper 2001.
https://www.nvidia.com/object/Interactive_Order_Transparency.html

6.[Enderton 2010] Eric Enderton, Erik Sintorn, Peter Shirley, David Luebke. "Stochastic Transparency." I3D 2010.
https://research.nvidia.com/publication/stochastic-transparency

7.[Laine 2011] Samuli Laine, Tero Karras. "Stratified Sampling for Stochastic Transparency." EGSR 2011.
https://research.nvidia.com/publication/stratified-sampling-stochastic-transparency

8.[McGuire 2011] Morgan McGuire, Eric Enderton. "Colored Stochastic Shadow Maps". I3D 2011.
http://research.nvidia.com/publication/colored-stochastic-shadow-maps

9.[Bavoil 2011] Louis Bavoil, Eric Enderton. "Constant-Memory Order-Independent Transparency Techniques." NVIDIA SDK11 Samples / StochasticTransparency 2011.
https://developer.nvidia.com/dx11-samples

10.[Harris 2019] Pete Harris. "Arm Mali GPUs Best Practices Developer Guide." ARM Developer 2019.
https://developer.arm.com/solutions/graphics/developer-guides/mali-gpu-best-practices

11.[Carpenter 1984] Loren Carpenter. "The A-buffer, an Antialiased Hidden Surface Method." SIGGRAPH 1984.
https://dl.acm.org/citation.cfm?id=80858

12.[Bavoil 2007] Louis Bavoil, Steven Callahan, Aaron Lefohn, Joao Comba, Claudio Silva. "Multi-Fragment Effects on the GPU using the k-Buffer." I3D 2007.
https://i3dsymposium.github.io/2007/papers.html

13.[Ragan-Kelley 2011] Jonathan Ragan-Kelley, Jaakko Lehtinen, Jiawen Chen, Michael Doggett, Frédo Durand. "Decoupled Sampling for Graphics Pipelines." ACM TOG 2011.

http://people.csail.mit.edu/jrk/decoupledsampling/ds.pdf

14.[D 2015] Leigh D. "Rasterizer Order Views 101: a Primer." Intel Developer Zone 2015.
https://software.intel.com/en-us/gamedev/articles/rasterizer-order-views-101-a-primer

15.[D 2017] Leigh D. "Order-Independent Transparency Approximation with Raster Order Views (Update 2017)." Intel Developer Zone 2017.
https://software.intel.com/en-us/articles/oit-approximation-with-pixel-synchronization-update-2014

16.[Bjorge 2014] Marius Bjorge, Sam Martin, Sandeep Kakarlapudi, Jan-Harald Fredriksen. "Efficient Rendering with Tile Local Storage." SIGGRAPH 2014.
https://community.arm.com/developer/tools-software/graphics/b/blog/posts/efficient-rendering-with-tile-local-storage

17.[Apple] Metal Sample Code / Deferred Lighting
https://developer.apple.com/documentation/metal/deferred_lighting

18.[Salvi 2010] Marco Salvi,Kiril Vidimce, Andrew Lauritzen, Aaron Lefohn. "Adaptive Volumetric Shadow Maps." EGSR 2010.
https://software.intel.com/en-us/articles/adaptive-volumetric-shadow-maps

19.[Salvi 2011] Marco Salvi, Jefferson Montgomery, Aaron Lefohn. "Adaptive Transparency." High Performance Graphics 2011.
https://software.intel.com/en-us/articles/adaptive-transparency-hpg-2011

20.[Salvi 2014] Marco Salvi, Karthik Vaidyanathan. "Multi-layer Alpha Blending." I3D 2014.
https://software.intel.com/en-us/articles/multi-layer-alpha-blending

21.[Imbrogno 2017] Michael Imbrogno. "Metal 2 on A11 – Imageblocks." Apple Developer 2017.
https://developer.apple.com/videos/play/tech-talks/603

22.[McGuire 2013] Morgan McGuire, Louis Bavoil. "Weighted Blended Order-Independent Transparency. " JCGT 2013.
http://jcgt.org/published/0002/02/09/

23.[NVIDIA] NVIDIA GameWorks Vulkan and OpenGL Samples / Weighted Blended Order-independent Transparency
https://github.com/NVIDIAGameWorks/GraphicsSamples/tree/master/samples/gl4-kepler/WeightedBlendedOIT