# 第一部分 并行编程

## 并行机制（Parallel Mechanism）

线程并行

CPU线程 Intel TBB //Work Stealing Task Scheduler

向量并行

CPU SIMD Instruction //X86 SSE 或 ARM NEON

GPU SIMT Architecture //Compute Shader HLSL/GLSL 或 GPGPU CUDAC/OpenCLC

分歧分支（Divergence Brunch）

执行屏障（Barrier）//区别于内存屏障

Vector parallelism using N lanes requires less hardware than thread parallelism using N threads

because in vector parallelism only the registers and the functional units have to be replicated N times.

In contrast, N-way thread parallelism requires replicating the instruction fetch and decode logic and perhaps enlarging the instruction cache.

Furthermore, because there is a single flow of control, vector parallelism avoids the need for complex synchronization mechanisms, thus enabling efficient fine-grained parallelism.

All these factors can also lead to greater power efficiency.

Thread parallelism can easily emulate vector parallelism—just apply one thread per lane.

However, this approach can be inefficient since thread synchronization overhead will often dominate.

SIMT processors may appear to have thousands of threads,

but in fact blocks of these share a control processor, and divergent control flow can significantly reduce efficiency within a block.

On the other hand, synchronization(Barrier In GLSL/HLSL) between fibers is basically free, because when control flow is emulated with masking the fibers are always running synchronously

全局内存->纹理缓存（Texture Cache）//反交错纹理 //计算着色器由像素着色器演化而来

局部/组共享内存->

//Bank Conflict //仅针对Local Memory

//Memory Interleave //超标量指令

//Warp 32 Thread

//Bank 32

//1 Word <-> 32 Bit

//内存中连续的Word属于不同的Bank //可以认为 //WordAddress = (Address/4Byte) //BankIndex = WordAddress%32

//Bank Conflict：同一次请求（即SIMT架构中同一条指令）中，同一Wrap中不同线程访问地址对应于同一Bank中的Word

//最佳Request Pattern/Memory Access Pattern：相邻ThreadID访问相邻的Word

比如 uint32 Addr[N]

//Addr[LocalID.x] -> 最佳

//Addr[LocalID.x\*2] -> Two-Way Bank Conflict //比如Double

//Addr[LocalID.x\*3] //不存在Bank Conflict //访问Double时可以加入An Extra Word

伪共享（False Sharing）

Threads also have different memory behavior than vector operations.

In particular, in vector parallelism we often want nearby vector lanes to access nearby memory locations,//空间局部性 反交错纹理

but if threads running on different cores access nearby memory locations it can have a negative impact on performance (due to false sharing in caches, which we discuss in Section 2.4).

CPU Map := A simple way around both problems is to break large vector operations into chunks and run each chunk on a thread, possibly also vectorizing within

each chunk

内存屏障（Memory Barrier）内存一致性（Memory Consistency）

https://www.threadingbuildingblocks.org/docs/help/tbb\_userguide/Design\_Patterns/Lazy\_Initialization.html

//Lazy Initialization本身而言 个人倾向于使用RefCount 每次调用原子+1 仅当RefCount为1时初始化 （其余忙式等待？）

volatile被弃用 //C++规范并没有关于内存屏障的定义

ReadBarrier/Acquire语义 which means that all of its subsequent reads will happen after the acquiring read.

WriteBarrier/Release语义 which means that all of its prior writes will be seen before the releasing write.

# Intel TBB

Yannis Minadakis. "Using Tasking to Scale Game Engine Systems". GDC 2011.

<http://software.intel.com/en-us/articles/using-tasking-to-scale-game-engine-systems>

Designing the Framework of a Parallel Game Engine

<https://software.intel.com/en-us/articles/designing-the-framework-of-a-parallel-game-engine>

重新排序（reorder）

//现代的 编译器 或 CPU 可能重新排序

//源代码上 串行 假定 并一定能够被保证

//条件语句 并不能保证 条件执行代码在测试语句之后执行

//编译器无法感知多线程

Acquire 所有（源代码上）后续的读取

Release 所有（源代码上）之前的写入

volatile 确保写入立即发生 //并不能解决reorder的问题

### 内存分配器

Richard L. Hudson, Bratin Saha, Ali-Reza Adl-Tabatabai, Benjamin C. Hertzberg. "McRT-Malloc: a scalable transactional memory allocator". Proceedings of the 5th international symposium on Memory management ACM 2006.

Alexey Kukanov, Michael J.Voss. "The Foundations for Scalable Multi-core Software in Intel Threading Building Blocks." Intel Technology Journal, Volume11, Issue 4 2007.

### 任务调度器

Alexey Kukanov, Michael J. Voss. "The Foundations for Scalable Multi-core Software in Intel Threading Building Blocks". Intel Technology Journal 2007.

Andrey Marochko. "TBB initialization, termination, and resource management details, juicy and gory". Intel Software Developer Zone 2011.

<https://software.intel.com/en-us/blogs/2011/04/09/tbb-initialization-termination-and-resource-management-details-juicy-and-gory>

《Structured Parallel Programmming》 / 8.5 Load Balancing

//ThreadSpecificData使WorkerThread在LocalTaskDeque中SpawnTask

//从VictimTaskQueue中Acquire的发生频率在实际中是较低的（即使在WorkerThread中）

#### 工作窃取（Work Stealing）

1.Thread <–> Ready Pool //Thread Own Deque (double-ended queue) //Spawned Task

2.The Shared Deque //Enqueued Task

//注意 Spawn和Enqueue是不同的

static tbb::task::spawn //Push到主调Thread 拥有的Deque的End处

static tbb::task::enqueue //Push到The Shared Deque的End处

//调度算法

在一个Task完成（可以理解为execute返回）后，Thread选择下一个Task

1. tbb::task::execute的返回值（不为NULL） //Schedule Bypass

2. Task的前驱（当前Task是最后一个后继（实现中体现为引用计数））

3. Thread的Deque的End处Pop得到的Task //当前（主调）Thread最近Spawn的Task //深度优先

4. Thread相干（Affinity）的Task

5. The Shared Deque的Approximately Beginning处Pop得到的Task //the Least recently Enqueue的Task /广度优先

6. 随机选择的其它某个Thread的Deque的Beginning处Pop得到的Task //其它Thread的the Least recently Spawn的Task //广度优先

//在嵌套并行中，广度优先可能导致空间需求爆炸

//语义

Spawn强调局部性（Cache命中）

Enqueue强调公平

//为避免没有Worker Thread 执行The Shared Deque中的Task，The Scheduler创建了一个额外的Worker Thread

根据以上调度算法，Spawned Task可能始终不被执行

而Enqueued Task一定会被执行（The 额外的Worker Thread）

//Recycle

custom\_scheduler<>::local\_wait\_for\_all //custom\_scheduler.h

//Row 527 switch(t->state()) ...

Task -> Work

Thread -> Worker

Worker 1-1 Deque (Task/Work)

1、Task Spawn -> Push Task/Work To Top Of Own Deque

//Space Guarantee

1-1、Cilk Plus -- Steal-Continuation (Continuation As A New Task) -- cilk\_spawn -> Worker(Current Thread) Execute Child Task

Execute Order Same As Serial

The Deque (And The Stack Space) 倍数 No More Than Number Of Workers

1-2、TBB -- Steal-Child -- task\_group::run –> Spawn All Child Before Steal

Execute Order Reverse Of Serial

The Deque (And Stack Stack Space) 倍数 Size Of Work Before Steal

2、Task/Work Stealing -> Pop Task From Top Of Own Deque ---- Unitl Empty（深度优先/串行） Then Choose Random Victim Worker’s Deque And Steal Work From Bottom Of Own Deque（在需要实际并行时 才 广度优先/并行）

//Thief Worker Steal From Begining Of The Call Tree

//Thief Worker Distant From Victim Worker Avoid Cache Conflict

//Time Guarantee

2-1、Cilk Plus -- Steal-Continuation – cilk\_sync

In Map Pattern, In Most Cases, Worker Rearch The clik\_sync Earlier Has Finshed The Work And Worker Will Not Wait At clik\_sync

Cilk Plus Scheduling May Be Greedy As Long As Plenty of Parallel Slack

2-2、TBB -- Steal-Child – tbb::task\_group::wait

Tbb Scheduling Not Greedy – Work/Span Model Not Applicable -> Worker That Call tbb::task\_group::wait May Steal Works From Other Pattern To Keep Busy(In Most Cases, Not All Other Works Finished But They Are All Being Stolen By Other Workers). Even Though Other Works From The Same Pattern Finished, The Whole Pattern May Wait Until The Work Stolen From Other Structure Finished.

//Cache-Oblivious Programming //缓存无视编程

Divide And Conquer //Problem Chopped Into Finer And Finer Pieces Until A Piece Fits Into Outer=Level Cache

Fork/Join –> QuickSort //分支/合并 -> 快速排序

if(Sequence Is Short) //Base Case

{

Use A Sort Optimized For Short Sequences

}

else

{

//Divide //Serial QuickSort\_Helper

Choose A Partition Key (枢轴) K From the Sequence //《数据结构》中 The Last One

Permute The Sequence That Confirms //std::partition

1.Keys to the Left of K are Less than K

2.Keys to the Right of K are greater than K

//Conquer

Recursively Sort the Subsequence to the Left of K //Parallel 1 //Smaller -> Child

Recursively Sort the Subsequence to the Right of K //Parallel 2 //Longer ->Continuation

}

//Cilk Plus

QuickSort\_Parallel T \*first T \*last

while( (last - first) > QUICKSORT\_CUTOFF ) // Sequence Not Short Enough

{

//Divide

T \*middle = Helper\_Divide(first,last); //std::partition

//Conquer

if( (middle - first) <= (last - (middle+1)) )

{

cilk\_spawn QuickSort\_Parallel(first, middle); //Smaller -> Child

first = middle + 1; //Longer ->Continuation

}

else

{

cilk\_spawn QuickSort\_Parallel(middle + 1, last); //Smaller -> Child

last = middle; //Longer ->Continuation

}

}

//Base Case

Helper\_Sort(first, last) //std::sort

//

Fork/Join Implement Map //分支/合并 实现 映射

Map\_Recursive lower upper

while( (upper - lower) > GRAIN\_SIZE ) //

{

//Divide

unsigned m = lower + (upper - lower)/2;

//Conquer

cilk\_spawn Map\_Recursive(lower); //Child

lower = m; //Continuation

}

//Base Case

for(size\_t i = lower ; i < upper; ++i)

{

functor(i);

}

//cilk\_sync

Fork/Join Implement Reduce //分支/合并 实现映射

Reduce\_Recursive lower upper

//Divide

unsigned m = lower + (upper – lower)/2;

//Conquer

Fork/Join Implement Scan //分支/合并 实现扫描

#### 延续传递风格（Continuation Passing Style）

//源于函数式编程

// Continuation 延续函数

//将ChildFunction的Result显式地传入ContinuationFunction

//区别于 直接风格 （Direct Style） //TBB称作阻塞风格（Blocking Style）

在tbb中

tbb::task::execute

Spawn Continuation //TaskCotinuation中往往存储ChildFunction的Result

Spawn Child (Of Continuation)//将TaskCotinuation中的变量的指针传入

当前函数返回 释放线程栈

//调度器绕过（Schedule Bypass）和回收（Recycling）

Spawn Continuation

Recycle As Child (Of Continuation) //调度器绕过

return this //回收

//

//Implement Steal-Continuation Semantics In TBB

tbb::task\_group -> Steal-Child

tbb::task -> Steal-Continuation

#### 并行映射（Parallel Map）

James Reinders,Arch Robison,Michael McCool. "Recursive Implementation Of Map". Structured Parallel Programming: Patterns for Efficient Computation, Chapter 8.3, 2012.

#### 并行归约（Parallel Reduce）

James Reinders,Arch Robison,Michael McCool. "Reductions And HypeObjects". Structured Parallel Programming: Patterns for Efficient Computation, Chapter 8.10, 2012.

#### Flow Graph

Wavefront

<https://software.intel.com/en-us/node/506116>

拓扑排序的并行变体

## //函数式编程

## GPGPU

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 平台\API | CUDA | OpenCL | Direct3D12 | Vulkan |
| WindowsDesktop | √（只支持NVIDIA GPU） | √ | √（官方） | √ |
| LinuxDesktop | √（只支持NVIDIA GPU） | √ | × | √ |
| OS X | √（只支持NVIDIA GPU） | ×（官方弃用） | × | ×（MoltenVK） |
| WindowsRuntime | × | × | √（官方） | ×（ICoreWindowInterop） |
| LinuxAndroid | × | × | × | √（官方） |
| IOS | × | ×（官方弃用） | × | ×（MoltenVK） |

|  |  |  |  |
| --- | --- | --- | --- |
| CUDA C | OpenCL C | HLSL | GLSL |
| threadIdx | get\_local\_id | SV\_GroupThreadID | gl\_LocalInvocationID |
| blockDim | get\_local\_size | N/A | gl\_WorkGroupSize |
| blockIdx | get\_group\_id | SV\_GroupID | gl\_WorkGroupID |
| gridDim | get\_num\_groups | N/A | gl\_NumWorkGroups |
| N/A | get\_global\_id | SV\_DispatchThreadID | gl\_GlobalInvocationID |
| N/A | get\_global\_offset | N/A | N/A |
| N/A | get\_work\_dim | N/A | N/A |
|  |  |  |  |
| \_\_shared\_\_ | \_\_local | groupshared | shared |
|  |  |  |  |
| \_\_syncthreads | barrier(CLK\_LOCAL\_MEM\_FENCE) | GroupMemoryBarrierWithGroupSync | memoryBarrierShared+barrier |
| N/A | N/A | GroupMemoryBarrier | memoryBarrierShared |
| N/A | barrier(CLK\_GLOCAL\_MEM\_FENCE) | DeviceMemoryBarrierWithGroupSync | memoryBarrier+barrier |
| N/A | N/A | DeviceMemoryBarrier | memoryBarrier |
| N/A | N/A | N/A | memoryBarrierAtomicCounter |
| N/A | N/A | N/A | memoryBarrierBuffer |
| N/A | N/A | N/A | memoryBarrierImage |
|  | N/A | AllMemoryBarrierWithGroupSync | groupMemoryBarrier+barrier |
|  | N/A | AllMemoryBarrier | groupMemoryBarrier |

### 扫描

Mark Harris, Shubhabrata Sengupta, John D. Owens. "Parallel Prefix Sum (Scan) with CUDA." GPU Gems 3 Chapter 39 2007

<https://developer.nvidia.com/gpugems/GPUGems3/gpugems3_ch39.html>

Duane Merrill, Andrew Grimshaw. "Parallel Scan for Stream Architectures." University of Virginia, Department of Computer Science, Technical Report 2009.

<https://libraopen.lib.virginia.edu/public_view/kd17cs85f>

Bank Conflict //Local

Permuted-Scan

//Advantage:Conflict-Free Padding-Free

//Drawback: Global -> UnCoalesced

//作者并没有找到一种时间复杂度为O(1)的计算内存索引到元素Rank的映射关系的方法

//"Unfortunately we are unaware of an O(1) function for performing such a mapping of element ranks to memory indices."——论文原文

我发现可以用倒位序（出自FFT）计算该映射关系

//以ThreadGroup维度为256为例

//HLSL MemoryIndex = reversebits(ElementRank)>>23 //23=(32-log(2,512)) /512为局部共享内存维度

//GLSL MemoryIndex = bitfieldReverse(ElementRank)>>23 //23=(32-log(2,512))

3.4.2 Reduce-then-scan

The intermediate values computed during the reduction kernels are not saved and must be recomputed later.

At the expense of performing some redundant calculations during the downsweep phase, the reduce-then-scan strategy moves 25% fewer bytes through global memory than scan-then-add.

3.4.3 Two-level streaming reduce-then-scan

dispatch a fixed number C of threadblocks in which threads are “re-used”.

We choose C large enough to saturate all SMs.

This illustration depicts a constant C=4 number of bottom-level threadblocks in which each threadblock serially processes cycles of b=4 values.

Reduction “upsweep” threadblocks are shown in light green, scan “downsweep” threadblocks are shown in dark green.

### 基数排序

Mark Harris, Shubhabrata Sengupta, John D. Owens. "Parallel Prefix Sum (Scan) with CUDA." GPU Gems 3 Chapter 39 2007

<https://developer.nvidia.com/gpugems/GPUGems3/gpugems3_ch39.html>

Nadathur Satish, Mark Harris, Michael Garland. "Designing Efficient Sorting Algorithms for Manycore GPUs." NVIDIA Technical Report 2008.

<https://www.nvidia.com/object/nvidia_research_pub_002.html>

Nadathur Satish, Mark Harris, Michael Garland. "Designing Efficient Sorting Algorithms for Manycore GPUs." IPDPS 2009.

<http://research.nvidia.com/content/designing-efficient-sorting-algorithms-manycore-gpus>

Duane Merrill, Andrew Grimshaw. "High performance and scalable radix sorting: A case study of implementing dynamic parallelism for GPU computing". Parallel Processing Letters 2011.

<https://code.google.com/archive/p/back40computing/wikis/RadixSorting.wiki>

//《Structured Parallel Programming》

Pack Pattern –> Scan Pattern + Conditional Scatter Pattern

Split Pattern

### SDF（Signed Distance Field）

Kenny Erleben, Henrik Dohlmann. "Signed Distance Fields Using Single-Pass GPU Scan Conversion Tetrahedra." GPU Gems 3 Chapter 34 2007

# 第二部分 计算器图形学

# 场景

## 场景图（SceneGraph）/场景树（SceneTree）

SceneGraph

//SceneGraph实际上可以认为是骨架的一种，只是该骨架的IK程度很高，会被逻辑开发工程师大量控制

//即rigid hierarchical animation 等同于hkaMeshBinding

NVIDIA SceniX

<https://developer.nvidia.com/scenix-download>

Markus Tavenrath, Christoph Kubisch. "Advanced Scenegraph Rendering Pipeline." GTC 2013.

<http://on-demand.gputechconf.com/gtc/2013/presentations/S3032-Advanced-Scenegraph-Rendering-Pipeline.pdf>

Markus Tavenrath. "NvPro-Pipeline A Research Rendering Pipeline." GTC 2015.

<http://on-demand.gputechconf.com/gtc/2015/presentation/S5148-Markus-Tavenrath.pdf>

<https://github.com/nvpro-pipeline/pipeline>

Scene Management Engine

NVSG(NVIDIA Scene Graph SDK) -> SceniX -> NvPro-Pipeline(dp::sg)

### SceneTree Construction

SceneGraph

一般是DAG（有向无环图）

对SceneGraph的操作会同步到SceneTree

以Event: Node Added为例

调用堆栈：

Group::AddChild -> (Group::postAddChild) -> (Subject::notify)

-> ObjectObserver::notify -> ObjectObserver:: onPostAddChild

-> SceneTree::addSubTree

SceneTree::addSubTree

SceneTreeGenerator::apply深度优先遍历SceneGraph

在SceneTree中addObject (

调用堆栈：SceneTreeGenerator::preTraverseGroup -> GeneratorState::pushObject -> GeneratorState::insertNode

)

结论：

在SceneGraph中的环起到Instantiation（实例化）的作用，但是并不能节省内存（内存中需要同时存在SceneGraph和SceneTree，并且还需要将SceneGraph转化为SceneTree）

图形引擎一般只提供SceneTree而不提供SceneGraph（可以在逻辑层实现（如果需要））

SceneTree的叶节点并不占用资源，不同叶节点内部的指针指向同一个结构（该结构占用资源），即可实现Instantiation（实例化），以达到节省内存的目的

使用DrawInstanced处理SceneTree中的实例化？？？

指向同一个“占用资源”的结构体的节点数量是不确定的

在着色器代码中，无法确定常量缓冲的大小？？？

着色器代码中的常量缓冲能否大于实际大小？？？

Transform Tree Update On GPU

//Level-Wise waits for previous results //Risk of little work per level

//Leaf-Wise runs to top, then concats path downwards per thread //Favors more total work over redundant calculations

个人不建议这么做！

### Shader Scatter Data

Direct3D12 VertexBufferView/IndexBufferView/ConstantBufferView

驱动需要分配大块内存以实现Discard语义

Holger Gruen. "Constant Buffers without Constant Pain." NVIDIA GameWorks Blog 2015.

<http://developer.nvidia.com/content/constant-buffers-without-constant-pain-0>

使用Default内存类型的Buffer 以Cache ModelTransformMatrix //引入ResourceBarrier后，同一资源可以同时作为UBV和UAV

CPU将发生改变的（使用Dirty标志）ModelTransformMatrix写入Upload内存类型的Buffer

由Shader将数据从Upload内存类型的Buffer复制到Default内存类型的Buffer

结论：

在UMA中，GPU访问Default和Upload内存类型的Buffer的效率相同，不建议采用

### Frustum Culling

使用XOR得到ChangedResult

作用于geometryInstanceVisibility

//没有使用层次性结构

### Occlusion Culling

Occlusion Culling + Indirect(VS Predict??? //InstanceCount为0不执行)

Christoph Kubisch. "OpenGL Scene-Rendering Techniques." GTC 2014.

<http://on-demand.gputechconf.com/siggraph/2014/presentation/SG4117-OpenGL-Scene-Rendering-Techniques.pdf>

Christoph Kubisch, Markus Tavenrath. "OpenGL 4.4 Scene Rendering Techniques." GTC 2014.

<http://on-demand.gputechconf.com/gtc/2014/presentations/S4379-opengl-44-scene-rendering-techniques.pdf>

<https://github.com/nvpro-samples/gl_cadscene_rendertechniques>

<https://github.com/nvpro-samples/gl_vk_threaded_cadscene>

Pierre Boudier, Christoph Kubisch. "GPU-Driven Large Scene Renderng". GTC 2015.

<http://on-demand.gputechconf.com/gtc/2015/presentation/S5135-Christoph-Kubisch-Pierre-Boudier.pdf>

<https://github.com/nvpro-samples/gl_occlusion_culling>

Wavefront

<https://software.intel.com/en-us/node/506116>

拓扑排序的并行变体

Flow Graph

//CAD场景

//MMORPG无缝大地图

无缝大地图并不是一个图形引擎层的概念，而是应当在逻辑层实现（准确的说应当是流（Stream）系统）

Dietrich, Andreas, Enrico Gobbetti, and Sung-Eui Yoon, “Massive-Model Rendering

Techniques,” IEEE Computer Graphics and Applications, vol. 27, no. 6, pp.

20–34, November/December 2007. Cited on p. 693, 695

## ­­包围体层次（Bounding Volume Hierarchy）

BVH（包围体层次）与SceneGraph（场景图）是两种完全不同的概念

### 视锥体剔除（View Frustum Culling）

//层次性加速

When you reach a node whose bounding volume does not intersect the query, you can (X) cull the entire subtree

An efficient parametric algorithm for octree traversal.

向量(a,b,c,d)表示平面ax + by + cz + d = 0

点P(px,py,pc)到平面上任意一点在平面法向量(a,b,c)上的投影为(a \* xp + b \* yp + c \* pc + d) / 根号(a^2 + b^2 + c^2)，特别地，当平面法向量为单位向量时，根号(a^2 + b^2 + c^2)为1，上式即为(px,py,pc,1)点积(a,b,c,d)

可根据投影与0的大小关系确定点P在平面的哪一侧

视锥体构造（Frustum Construct）

Gil Gribb, Klaus Hartmann. "Fast Extraction of Viewing Frustum Planes from the WorldView-Projection Matrix". GameDevs.org 2001.

根据VP矩阵推导出6个平面的一般式方程，转置之后向量相加/减即可

值得注意的是，当矩阵为右手系时，虽然z小于0，但w=-z大于0，不等式不需要变号

包围球构造（Bounding Sphere Construct）

Jack Ritter. "An efficient bounding sphere." Graphics Gems 1 1990.

轴对齐包围盒构造（Axis-Aligned Bounding Box Construct）

视锥体/包围球相交测试（Frustum/Bounding Sphere Intersection Test）

//参考文献 #include <DirectXCollide.h>

平面/球 检测

球心在平面法线上的投影

<=-r 背面

-r< <r 相交

>=r 正面

视锥体/轴对齐包围盒相交测试（Frustum/ Axis-Aligned Bounding Box Intersection Test）

平面/长方体 检测

AABB

比较 长方体中心在平面法向量上的投影 与 长方体8个半对角线在面法向量上的投影的最大值的 大小关系

由于长方体8个半对角线的x,y,z分量的绝对值都相同，求最大值时，只需要求绝对值即可

OBB

求长方体8个半对角线在面法向量上的投影的最大值时

将平面法向量变换到OBB的ModelSpace，求逆矩阵时，可以利用 “正交矩阵的逆矩阵为正交矩阵的转置”

其余过程与AABB相同

以上方式检测的结果具有误差，有8个顶点处会有部分的分离区域被误判为相交

### 射线相交测试（Ray Intersection）

Ray-Triangle Intersection

Tomas Moller, Ben Trumbore. "Fast, Minimum Storage Ray/Triangle Intersection." Journal of Graphics Tools, Volume 2, 1997.

设直线 = + t

设三角形 = + u( - ) + v( - )

构造线性方程组 + t = + u( - ) + v( - )

即 = -

根据克拉默法则

当系数行列式≠0时

线性方程组有唯一解 =

根据标量三重积

= ·(( - )×( - )) = ·(( - )×( - ))

反映了射线方向与△V0V1V2的法线方向之间夹角

在坐标系手性为右手系且三角形环绕顺序为逆时针时

( - )×( - )为△V0V1V2的法线方向的相反方向， ＞ 0即表示射线从三角形正面射入

由于浮点运算误差， == 0在实际中几乎不可能发生

一般会设置阈值Epsilon = 1×10-20，将 -Epsilon ＜ ＜ Epsilon都等同于 == 0处理，认为射线与三角形所在平面平行

### 构造（Construction）

参考文献：

Christian Lauterbach, Michael Garland, Shubhabrata Sengupta, David Luebke, Dinesh Manocha. "Fast BVH Construction on GPUs." Eurographics 2009.

<http://research.nvidia.com/publication/fast-bvh-construction-gpus>

Tero Karras. "Maximizing Parallelism in the Construction of BVHs, Octrees, and k-d Trees." HPC 2012.

<http://research.nvidia.com/publication/maximizing-parallelism-construction-bvhs-octrees-and-k-d-trees>

Tero Karras, Timo Aila. "Fast Parallel Construction of High-Quality Bounding Volume Hierarchies." HPC 2013.

<http://research.nvidia.com/publication/fast-parallel-construction-high-quality-bounding-volume-hierarchies>

Intel Embree

<https://embree.github.io/related.html> / Embree Related Papers

Fast Construction of SAH BVHs on the Intel Many Integrated Core (MIC) Architecture

LongestCommonPrefix

= 31-firstbithigh(Key1^Key2)

= 31-findMSB(Key1^Key2)

基数排序（Radix Sort）

Duane Merrill, Andrew Grimshaw. " High performance and scalable radix sorting: A case study of implementing dynamic parallelism for GPU computing." Parallel Processing Letters 2011.

//计算各个桶的大小

//以Radix8为例

输入序列 uin32[N]

DecodePass bit\_0s[N](1 For 0 And 0 For Others) bit\_1s[N] (1 For 1 And 0 For Others) ... bit\_7s[N]

ScanPass 求出各个Decoded Vector中1的个数，即桶的大小 //个人认为可以使用Reduce而非Scan

DistributionPass//ScatterPass 每个桶的(Base)Offset已经确定

## 细节等级（Level Of Detail）

### 生成（Generation）

可以由艺术家制作

可以由引擎根据网格简化（Mesh Simplification）/数据归约（Data Reduction）算法进行生成

边缘塌陷（Edge Collapse）

以D3DX库为例

子集定位（Subset Placement）策略

成本函数（Cost Function）

Garland-Herbert成本函数­­->顶点新位置到顶点在旧位置相邻的各个平面的距离的平方和

问题：基于体积变化，在锋利折叠处会有问题

解决：增加一个平面

//避免边缘交错，可以检测法向量是否翻转

可以考虑使用第三方库，比如Simplygon Studios的Simplygon SDK

### 选择（Selecion）

根据效益函数（Benefit Function）选择合适的LOD

基于范围（Range Baed）

基于投影面积（Projected Area Based）

屏幕覆盖（Screen-Space Coverage）——物体的包围体在投影面（Z=1）上的面积

估计屏幕覆盖

1包围球

求出球心在ClipSpace中的Z坐标

dot(SphereCenter - EyePosition, EyeDirection)

假定球心位于视野中央，估计投影面（Z=1）上的半径为

SphereRadius/(dot(SphereCenter - EyePosition, EyeDirection))\*1

根据PI\*（R^2）求出面积

2包围盒

磁滞（Hysteresis）

不同帧之间，效益函数的取值在相邻LOD的临界处不断变化，产生爆音（Popping）

可以引入磁滞解决

记录上一帧的效益函数的值

根据当前帧的有效函数的值相对于上一帧增加或减少，在将效益函数的值映射到LOD等级时采用不同的策略

增加 —————— —————— —————— ———

0 1 2 Detail Culling

减少 ———— —————— —————— —————

#### 细节剔除（Detail Culling）

可用于实现细节剔除（Detail Culling），当LOD达到某个阈值时，设置为绘制空对象（即不绘制）

### 切换（Switch）

离散LOD

会产生爆音（Popping）

融合LOD

1当前LOD

正常绘制

2下一等级LOD

基于Over操作进行Alpha融合DepthEnable-TRUE DepthWriteMask-ZERO

ColorOut = AlphaSource\*ColorSource + (1 − AlphaSource)\*ColorDestination

AlphaLOD

### 时间关键（Time-Critical）的LOD渲染

使用贪心（Greedy）算法，根据效益函数/代价函数（Cost Function 即消耗时间）选择LOD

### 层次性细节等级（Hierarchical Level Of Detail）

Carl Erikson, Dinesh Manocha, William V. Baxter III. "HLODs for faster display of large static and dynamic environments". SI3D 2001.

主要思想是将SceneGraph中的各个结点合并

## 遮挡剔除（Occlusion Culling）

与深度缓冲的区别

深度缓冲是 逆向画家算法的延伸 确定可见性 并不起到加速作用

HLSL EarlyDepthStencil

GLSL layout (early\_fragment\_tests) in;

Conservative Depth //To Allow EarlyZ

SV\_DepthGreaterEqual / SV\_DepthLessEqual //The Output Of Pixel Shader Will Be Clamped

只在完全从前往后绘制时才时Discard所有的像素着色器

评估 深度复杂性（Depth Complexity）

HZB 大小 长宽 1/4 相当于 四叉树

可以考虑用IndirectExecute减少不必要的GPU/CPU同步

IndirectExecute在功能上可以认为是Predication的超集

### 入口剔除（Portal Culling）

1.视锥体与入口（二维AABB）交错检测

2.根据摄像机位置和入口构造视锥体，重复以上过程

参考CryEngine 5.2.3 VisArea:: UpdatePortalCameraPlanes

FarPlane保持不变

NearPlane=XMPlaneFromPointNormal(

Portal[0]//任取一点,

NearPlaneNormal//已经归一化，不需要再次进行

)

以RightPlane为例，Left Top Bottom同理

TempPlane=XMPlaneFromPoints(

vCameraPos,

Portal[]//视口可以指定环绕顺序 XNAMath法线以在左手系以顺时针为正面

Portal[]

//通过比较点积比较FOV（Field Of View）：点积最大1时，FOV为180度、点积为0时，FOV为90度、点积最小—1时，FOV为0度；这种做法不适用于正交投影

//如果FOV增大，那么表明入口边界在视锥体之外，这种情况下入口和视锥体平面的交界即原视锥体平面，不应当更新

if(

XMVector3Less(

XMVector3Dot(LeftPlaneNormal, TempPlane),

XMVector3Dot(LeftPlaneNormal,RightNormal)

)

{

RightPlane=TempPlane

}

建议用HZB取代Portal Culling！！！

### 层次深度缓冲（Hierarchical Z Buffering）

在目前的硬件条件下，可以考虑用间接执行，以减少数据传输（通过设置Draw(Indexed)Instanced中的InstanceCount，0U表示剔除，1U表示正常绘制）

深度金字塔基于上一帧

利用层次性Early-Out会导致GPU和CPU无法并行

### MOC（Masked Occlusion Culling，掩码遮挡剔除）

Jon Hasselgren, Magnus Andersson, Tomas Akenine-Moller. "Masked Software Occlusion Culling". High Performance Graphics 2016.

<http://software.intel.com/en-us/articles/masked-software-occlusion-culling>

## 几何变换

### 四元数（Quaternion）/旋转变换

#### 复数（回顾）

数学 选修2-2 A版 ISBN 9787107202551

代数形式/平面直角坐标

Z = A + B **i**

三角形式/极坐标

Z=R(cosθ+sinθ**i**) //可以看作二维向量的数乘 R \* (cosθ, sinθ)

θ辐角主值

几何意义 X轴正向向Y轴正向的旋转

乘法

代数形式

Z1 = A1+B1 **i**

Z2 = A2+B2 **i**

Z1\*Z2 = (A1+B1 **i**)(A2+B2 **i**) = (A1A2 – B1B2) + (A1B2 + A2B1) **i**

三角形式

Z1=R1(cosθ1+sinθ1 **i**)

Z2=R2(cosθ2+sinθ2 **i**)

Z1\*Z2

= R1\*R2(cosθ1cosθ2 - sinθ1 sinθ2) + R1\*R2(cosθ1sinθ2 + sinθ1cosθ2) **i**

= R1\*R2(cos(θ1+θ2) + sin(θ1+θ2) **i**) //两角和与差公式

几何意义 模相乘 辐角相加

逆

代数形式

Z = A + B **i**

Z-1 = = = = A/(A^2+B^2) + (-B/(A^2+B^2))i

三角形式

Z = R(cosθ + sinθ**i**) = Rcosθ + Rsinθ**i**

代入得

Z = Rcosθ/((Rcosθ)^2+(Rsinθ)^2) + (-Rsinθ/((Rcosθ)^2+(Rsinθ)^2))i

= cosθ/R + (-sinθ/R)i

= (1/R)cos(-θ) + ((1/R)(sin(-θ)) i

= (1/r)(cos(-θ) + sin(-θ)i)

几何意义 模为倒数 辐角为相反数

旋转因子（Rotor）

模为1的复数，乘法不改变模，只改变辐角，可以认为充当转子

Rθ = cosθ + sinθ**i**

R-θ = cosθ - sinθ**i** 共轭 旋转-θ

矩阵形式

Z = A + B

由于

\* = ——> 1 \* 1 = 1

\* = ——> 1 \* i = i

\* = ——> 1 \* i = i

结合矩阵运算的性质 可以证明 矩阵形式满足代数形式的所有性质

= \*

取第1列即得到矩阵和向量相乘

= \*

可以认为是复数A1+B1**i**对应的变换矩阵

比如乘以Rθ （即cosθ + sinθ**i**）

= \*

比如乘以Rθ （即cosθ + sinθ**i**）

= \*

根据矩阵乘法，即

= \* （即二维空间的旋转变换）

且 = \* （被忽略）

#### 四元数代数

代数形式

Q = D + Ai + Bj + Ck

i\*i=-1 j\*j=-1 k\*k=-1 i\*j=k j\*k=i k\*i=j j\*i=-k k\*j=-i i\*k=-j

(q0\_x·i + q0\_y·j + q0\_z·k + q0\_w) \* (q1\_x·i + q1\_y·j + q1\_z·k + q1\_w)

= q0\_w·q1\_w - q0\_x·q1\_x - q0\_y·q1\_y - q0\_z·q1\_z

+ q0\_w·q1\_x·i + q0\_w·q1\_y·j + q0\_w·q1\_z·k

+ q1\_w·q0\_x·i + q1\_w·q0\_y·j + q1\_w·q0\_z·k

+ (q0\_y·q1\_z - q1\_y·q0\_z)i + (q0\_z·q1\_x –q1\_z·q0\_x)j + (q0\_x·q1\_y - q1\_x·q0\_y)k

实数—三维向量 有序对

q0 = [q0\_s, q0\_v]

q1 = [q1\_s, q1\_v]

q0×q1

= [q0\_s, q0\_v]×[q1\_s, q1\_v]

= [q0\_s·q1\_s - dot(q0\_v, q1\_v), q1\_v×q0\_s + q0\_v×q1\_s + cross(q0\_v, q1\_v)]

//R1[cosθ1+sinθ1**N1**] \* R2[cosθ2+sinθ2**N2**]

//= R1R2

//[cosθ1cosθ2 - sinθ1sinθ2\*dot(**N1**,**N2**), cosθ1sinθ2**N2** + cosθ2sinθ1**N1** + sinθ1sinθ2\*cross(**N1**,**N2**)]

纯四元数

D=0

//单位四元数

//D=0且A^2+B^2+C^2=1

单位模（Unit-Norm）四元数

D^2+A^2+B^2+C^2=1

#### 单位四元数->旋转变换

设

单位模四元数 Q = [cosθ, sinθ **N**] //N是单位向量

Q-1 = = = = = [cosθ, -sinθ **N**]

纯四元数P = [0, ] //三维空间内点

Q \* P \* Q-1 = [0, **N**\*dot(**N**,**P**) + (**P** - **N**\*dot(**N**,**P**))\*cos(2θ) + cross(**N**,**P**)\*sin(2θ) ] //恰好与上文公式中一致

//注意顺序

(q0×q1)表示的旋转变换是，先进行q1，再进行q0，从右往左

(q0×q1)×p×(q0×q1) -1

= (q0×q1)×p×(q1 -1×q0 -1)

= q0×(q1×p×q1 -1)×q0 -1

根据几何意义 Q和-Q 表示的是同一个旋转变换

几何意义

//DirectXMath.h -> DirectX::XMMatrixRotationAxis

//NormalAxis = Vector3Normalize(Axis)

设ON是OP在NormalAxia上的投影

由于P绕NormalAxia旋转得到P’ 显然ON也是OP’在NormalAxia上的投影

显然有

OP = ON + NP

OP’ = ON + NP’

根据向量投影

ON = **N**\*dot(**N**,**P**)

NP = OP – ON = **P** - **N**\*dot(**N**,**P**)

以下是垂直于NormalAxis的俯视图



根据叉乘的定义

//大小 由于 |**N**| = 1 **N**⊥NP 因此 |cross(**N**,**P**)| = △ONP的面积的2倍 = | **N** |×|NP| = |NP| = |NT|

//方向 由于NT⊥NP NT⊥**N** –> NT⊥平面ONP 因此cross(**N**,**P**)的方向与NT相同

NT = cross(**N**,**P**) //右手系

显然

NH = NP \* cos(2θ) = (**P** - **N**\*dot(**N**,**P**))\*cos2θ

HP’ = NT \* sin(2θ) = cross(**N**,**P**)\*sin(2θ)

OP’ = ON + NP’

= ON +NH + HP’

= **N**\*dot(**N**,**P**) + (**P** - **N**\*dot(**N**,**P**))\*cos(2θ) + cross(**N**,**P**)\*sin(2θ)

//不构造矩阵的快速算法

q×p×q-1

= (q×p)×q-1

= [q\_s·0 – dot(q\_v,), q\_v×0 + ×q\_s + cross(q\_v, )]×q-1

= [q\_s·0 – dot(q\_v,), q\_v×0 + ×q\_s + cross(q\_v, )]×[q\_s, -q\_v]

= [– dot(q\_v,), ×q\_s + cross(q\_v, )]×[q\_s, -q\_v]

= [0, -q\_v×– dot(q\_v,) + (×q\_s + cross(q\_v, ))×q\_s + cross(×q\_s + cross(q\_v, ), -q\_v)] //已经证明Scale分量为0

= [0, q\_v×dot(q\_v,) + ×(q\_s·q\_s) + cross(q\_v, )×q\_s + cross(q\_v,×q\_s + cross(q\_v, ))]

= [0, q\_v×dot(q\_v,) + ×(q\_s·q\_s) + cross(q\_v, )×q\_s + (cross(q\_v,×q\_s) + cross(q\_v,cross(q\_v, )))] //分配律

= [0, q\_v×dot(q\_v,) + ×(q\_s·q\_s) + (cross(q\_v, )×q\_s + cross(q\_v,)×q\_s) + cross(q\_v,cross(q\_v, ))] //数乘

= [0, q\_v×dot(q\_v,) + ×(q\_s·q\_s) + cross(q\_v, )×(q\_s·2) + cross(q\_v,cross(q\_v, ))]

= [0, (×dot(q\_v, q\_v) + cross(q\_v,cross(q\_v, )) + ×(q\_s·q\_s) + cross(q\_v, )×(q\_s·2) + cross(q\_v,cross(q\_v, ))] //拉格朗日公式

= [0, (×dot(q\_v, q\_v) + cross(q\_v,cross(q\_v, )) + ×(q\_s·q\_s) + cross(q\_v, )×(q\_s·2) + cross(q\_v,cross(q\_v, ))]

= [0, ×(dot(q\_v, q\_v) + q\_s·q\_s) + cross(q\_v, )×(q\_s·2) + 2cross(q\_v,cross(q\_v, ))]

= [0, + cross(q\_v, )×(q\_s·2) + cross(q\_v,cross(q\_v, ))×2] //单位模四元数dot(q\_v, q\_v) + q\_s·q\_s) = 1

= [0, + cross(q\_v, ×q\_s)×2 + cross(q\_v,cross(q\_v, ))×2] //数乘

= [0, + cross(q\_v, cross(q\_v, ) + ×q\_s)×2] //分配律

= [0, + cross((q\_x,q\_y,q\_z), cross((q\_x,q\_y,q\_z), ) + ×q\_w)×2]

#### 旋转变换->单位四元数

旋转相乘

兼容：欧拉角->四元数

RotationX(θ) -> [cos(θ/2),sin(θ/2)[1,0,0]]

RotationY(θ) -> [cos(θ/2),sin(θ/2)[0,1,0]]

RotationZ(θ) -> [cos(θ/2),sin(θ/2)[0,0,1]]

兼容：矩阵形式->单位四元数

1.列向量

Q = D + A + B + C

由于

\* = - —> i \* i = -1

\* = - —> j \* j = -1

\* = - —> k \* k = -1

\* = —> i \* j = k

\* = —> j \* k = i

\* = —> k \* i = j

结合矩阵运算的性质 可以证明 矩阵形式满足代数形式的所有性质

= \*

取第1列即得到矩阵和向量相乘

= \*

2.行向量

Q = D + A + B + C

由于

\* = - —> i \* i = -1

\* = - —> j \* j = -1

\* = - —> k \* k = -1

\* = —> i \* j = k

\* = —> j \* k = i

\* = —> k \* i = j

结合矩阵运算的性质 可以证明 矩阵形式满足代数形式的所有性质

= \*

取第1行

= \*

根据(AB)T = BT \* AT

= \*

设

单位模四元数 Q = [ QD QA QB QC ] //QD^2 + QA^2 + QB^2 + QC^2 = 1

Q-1 = = [ QD -QA -QB -QC ] //

三维空间内点 P = [ 0, PX, PY, PZ ]

Q \* P \* Q-1 = (Q \* P) \* Q-1

= ( \* ) \* Q-1

= \* \*

= \* \*

=

\*

根据运算结果可知

可以忽略矩阵的第1行和第1列 即

\*

矩阵->四元数

由于四元数的对映性（Antipodality），Q和-Q表示同一个旋转变换，同一旋转矩阵对应Q和-Q两个四元数

\_m00+\_m11+\_m22

=

=

=

-> QD =

\_m21 - \_m12

= 4QAQD

-> QA =

同理

-> QB =

-> QC =

正如上文中几何意义所述 Q和-Q表示的是同一个旋转变换

#### Slerp(Spherical Linear Interpolation)

//向量

**P** = **P1** +  **P2**

其中cosΩ =

//根据正弦定理可以证明，点P在圆上

角度相对于时间线性变化

//四元数

将四元数看作4维向量进行计算即可 根据cosΩ = 计算得到Ω

单位模四元数 Slerp后 仍是单位模四元数

### 对偶四元数（Dual Quaternion）/平移旋转变换

平移旋转变换又称为刚性变换（Rigid Transform）

为了保持简洁，尽量不引入不必要的名词，因此仍用平移旋转变换

#### 对偶数 （Dual Number）

代数形式

^a = a0 + a1ε // 1×ε = ε×1 = ε ε×ε = 0

矩阵形式

^a = a0 + a1

乘法

(a0 + a1ε)×(b0 + b1ε) = a0b0 + (a0b1 + b0a1)ε

平方根

设

a0 + a1ε = (r0 + r1ε)×(r0 + r1ε) = r0r0 + 2r0r1ε

得到

r0r0 = a0 且2r0r1 = a1

=> r0 = 且r1 = = //Valid Only When a0≠0

因此

= //Valid Only When a0≠0

共轭

^a = a0 + a1ε

= a0 – a1ε

逆

^a = a0 + a1ε

^a-1 = = = = //Valid Only When a0≠0

#### 对偶四元数代数

代数形式

^q = ^w + ^xi + ^yj + ^zk //^w = w0 + w1ε //^x = x0 + x1ε //^y = y0 + y1ε //^z = z0 + z1ε

由于//ε×i = i×ε = i //ε×j = j×ε = j //ε×k = k×ε = k

^q = q0 + q1ε = // q0 = w0 + x0i + y0j+ z0k //q1 = w1 + x1i +y1j +z1k

乘法

DualQuanternion\_Multiply(p0 + p1ε, q0 + q1ε)

= Quaternion\_Multiply(p0, q0) + (Quaternion\_Multiply(p0, q1) + Quaternion\_Multiply(p1, q0))ε

共轭

= + ε

模

模的平方

|q0 + q1ε|2 = (q0 + q1ε)×() = (q0 + q1ε)×( + ε) = |q0|2 + (q0× + q1×) ε

平方根

由于

q0× + q1×

= [q0\_s, q0\_v]×[q1\_s, -q1\_v] + [q1\_s, q1\_v]×[q0\_s, -q0\_v]

= [q0\_s·q1\_s - dot(q0\_v, -q1\_v), -q1\_v×q0\_s + q0\_v×q1\_s + cross(q0\_v, -q1\_v)] + [q1\_s·q0\_s - dot(q1\_v, -q0\_v), -q0\_v×q1\_s + q1\_v×q0\_s + cross(q1\_v, -q0\_v)]

= [2q0\_s·q1\_s + 2dot(q0\_v, q1\_v), ]

= 2q0\_s·q1\_s + 2dot(q0\_v, q1\_v)

因此

q0× + q1×是实数

=> |q0 + q1ε|2 = |q0|2 + (q0× + q1×) ε是对偶数（Dual Number）

=> |q0 + q1ε|2适用于对偶数（Dual Number）的平方根

=> |q0 + q1ε| = = = |q0| + ε = |q0| + ε

单位对偶四元数（Unit Dual Quaternion）

^q = q0 + q1ε是单位对偶四元数 <=> | q0 + q1ε | = 1 <=> |q0| = 1 且q0\_sq1\_s + dot(q0\_v, q1\_v) = 0

双共轭

=

逆

(**A** + **B**ε)-1 = 1/(**A** + **B**ε) = ((**A** + **B**ε)\*)/((**A** + **B**ε)\*((**A** + **B**ε)\*)) = ((**A** + **B**ε)\*)/|**A** + **B**ε|

存储->2个向量

HLSL -> float2x4

GLSL -> mat4x2

#### 平移旋转变换->单位对偶四元数

任意平移旋转变换都能用单位对偶四元数表示

//单位对偶四元数

//三维空间内任意一点

^p = [1,(0,0,0)] + [0, ]ε = 1 + [0, ]ε

//旋转

^r = r0 + [0, ]ε = r0 //其中r0 = [cos(θ/2), ×sin(θ/2)]且||=1即单位模四元数

= - [0, -]ε =

^r×^p×

= (^r×^p)×

=(r0×(1 + [0, ]ε))×

= (r0 + (r0×[0,])ε)×

= (r0 + (r0×[0,])ε)×

=|r0|2 + (r0×[0,]×) ε

= 1 + (r0×[0,]×) ε //根据四元数与旋转变换的关系，r0×[0,]×即表示绕旋转θ

//平移

^t = [1,(0,0,0)] + [0, (1/2)×]ε = 1 + [0, (1/2)×]ε

= 1 – [0, -(1/2)×]ε = 1 + [0, (1/2)×]ε //恰好=^t

^t×^p×

= (^t×^p)×

= (1 + [0, + (1/2)×]ε )×

= 1 + [0, + (1/2)× + -(1/2)×]ε

= 1 + [0, + ] ε

//任意平移旋转变换可以表示为以下对偶四元数

^t×^r

= r0 + ([0, (1/2)×]×r0)ε

首先证明：对于任意^t、^r，有 =

由于

=

=

=

=

=

=

//直接计算可以证明 = 且 =

=

=

= [t0\_s·r0\_s – dot(t0\_v, r0\_v), – r0\_v×t0\_s – t0\_v×r0\_s – cross(t0\_v, r0\_v)]

//----------------------------------------------------------------------------------------------------

= [r0\_s, -r0\_v]×[t0\_s, -t0\_v]

= [r0\_s·t0\_s – dot(-r0\_v, -t0\_v), (-t0\_v)×r0\_s + (-r0\_v)×t0\_s + cross(-r0\_v, -t0\_v)

= [t0\_s·r0\_s – dot(t0\_v, r0\_v), – r0\_v×t0\_s – t0\_v×r0\_s – cross(t0\_v, r0\_v)]

//----------------------------------------------------------------------------------------------------

=> =

因此

=

下面证明对偶四元数[0, (1/2)×]×r0是单位对偶四元数

|^t×^r|2

= (^t×^r)×()

= (^t×^r)×()

= ^t×(^r×)×

= ^t×|^r|2×

= ^t×1×

= |^t|2

= 1

=> |^t×^r| = 1

=> ^t×^r是单位对偶四元数

#### 单位对偶四元数->平移旋转变换

任意单位对偶四元数都表示一个平移旋转变换

设任意单位对偶四元数q0 + q1ε

证明等式r0 + ([0, (1/2)×]×r0)ε = q0 + q1ε对r0和有解 //其中r0 = [cos(θ/2), ×sin(θ/2)]且||=1即单位模四元数

设r0和，满足r0= q0且[0, (1/2)×]×r0 = q1 => 显然r0和满足以上等式

尝试证明满足r0= q0且[0, (1/2)×]×r0 = q1的r0和存在

下面证明r0存在

q0 + q1ε是单位对偶四元数

=> |q0| = 1

=> 单位模四元数r0 = q0存在

下面证明存在

等式[0, (1/2)×]×r0 = q1两边同时乘以

左边

= [0, (1/2)×]×r0×

= [0, (1/2)×]×(r0×)

= [0, (1/2)×]×|r0|2

= [0, (1/2)×]×1

= [0, (1/2)×]

右边

= q1×

= q1× // r0 = q0

设

q0 = [q0\_s, q0\_v]

q1 = [q1\_s, q1\_v]

q0 + q1ε是单位对偶四元数

=> q0× + q1× = 0

=> q0\_s·q1\_s + dot(q0\_v, q1\_v) = 0 //见前文

由于

q1×

= [q1\_s, q1\_v]×[q0\_s, -q0\_v]

= [q1\_sq0\_s - dot(q1\_v, -q0\_v), -q0\_v×q1\_s + q1\_v×q0\_s + cross(q1\_v, -q0\_v)]

q1×的Scale分量q1\_sq0\_s - dot(q1\_v, -q0\_v) 为0

=> = -q0\_v×q1\_s + q1\_v×q0\_s + cross(q1\_v, -q0\_v)存在

参考文献：

Ladislav Kavan, Steven Collins, Jiri Zara, Carol O'Sullivan. "Skinning with Dual Quaternions." I3D 2007.

<http://www.cs.utah.edu/~ladislav/kavan07skinning/kavan07skinning>.html

Ladislav Kavan, Steven Collins, Jiri Zara, Carol O'Sullivan. "Geometric Skinning with Approximate Dual Quaternion Blending." SIGGRAPH 2008.

<http://www.cs.utah.edu/~ladislav/kavan08geometric/kavan08geometric>.html

NVIDIA Direct3D SDK 10.5 Code Samples / Skinning with Dual Quaternions

<https://developer.download.nvidia.com/SDK/10.5/direct3d/samples.html#QuaternionSkinning>

### 坐标变换 VS 法线变换

为了近似表示曲面，法向量并不一定垂直至于Mesh中的三角形

顶点位置/切线（Tangent）方向/双切线（Bitangent）方向

证明 在原切线/双切线上任取两点A B

变换前 D=B-A

变换后 D’=MB-MA=M(B-A)=MD //公式中不包含第4个分量，不应当平移

法线（Normal）方向

证明 在原平面上任取两点A B

变换前 D\*(B-A)= 0 //注意 D为行向量 (B-A)为列向量

变换后 D\*M-1\*(MB-MA)=D\*( M-1\*M)\*(B-A)=D\*(B-A)=0 //注意D为行向量

因此D’= D\*M-1 //注意D为行向量

在求方向时，由于一般在变换后会归一化（即Normalize），求逆矩阵时，可以只就出伴随矩阵而不再除以行列式

接下来推导的都是坐标变换，可以基于以上方式推导出相应的法线变换

### 手性（Chirality）

### 四元数切线空间（Quaternion TangentSpace）

参考文献

1. Ivo Zoltan Frey. "Spherical Skinning withDual-Quaternions and Qtangents". SIGGRAPH 2011.

<http://crytek.com/cryengine/presentations/spherical-skinning-with-dual-quaternions-and-qtangents>

使用Quaternion定义切线空间 取代Tangent BiTangent Normal

//Quaternion可以定义一个参考系

Tangent = Q(0,1,0,0)Q-1

Bitangent = Q(0,0,1,0)Q-1

Normal = Q(0,0,0,1)Q-1

NormalMap使用SNORM格式存储 //是否支持压缩？？？

//切线空间必须正交 Normal根据cross(Tangent,Bitangent)计算得出

// Asset

// if dot(cross(Tangent,Bitangent),normal) < 0 //Bitanget = - Bitangent

求出Quaternion

上文中给出了 已知 求出QD QA QB QC的方法

//Bitangent = cross(Normal,Tangent) //左手系/右手系都适用

由于

Tangent = Q(0,1,0,0)Q-1

Bitangent = Q(0,0,1,0)Q-1

Normal = Q(0,0,0,1)Q-1

即

//不是旋转变换？？？

//转化后assert(QA2 + QB2 + QC2 + QD2) == 1

//单位矩阵E -> 四元数(0,0,0,1)或(0,0,0,-1) 是合法的

文中的Reflection的含义未知？？？

//光栅化阶段之后，由于插值后的结果并不保证模仍为1，因此需要再次normalize

### 视野变换

根据Camera的Position ForwardDirection和UpDirection得到ViewSpace的原点和坐标轴

设

ViewSpace的原点在WorldSpace中的位置为P

ViewSpace的坐标轴（单位正交向量）在WorldSpace中为AX AY AZ

显然我们有

[ 1 0 0 Px ] [ AXx AYx AZx 0] [ Xview ] [ Xworld ]

[ 0 1 0 Py ] × [ AXy AYy AZy 0 ] × [ Yview ] = [ Yworld ]

[ 0 0 1 Pz ] [ AXz AYz AYy 0 ] [ Zview ] [ Zworld ]

[ 0 0 0 1 ] [ 0 0 0 1 ] [ 1 ] [ 1 ]

[ 1 0 0 Px ] [ AXx AYx AZx 0]

显然 [ 0 1 0 Py ] × [ AXy AYy AZy 0 ] 即视野变换的逆矩阵

[ 0 0 1 Pz ] [ AXz AYz AYz 0 ]

[ 0 0 0 1 ] [ 0 0 0 1 ]

根据 单位正交阵的逆矩阵即单位正交阵的转置 + 平移变换的几何意义

[ AXx AXy AXz 0] [ 1 0 0 - Px ]

得到 [ AYx AYy AYz 0 ] × [ 0 1 0 - Py ] 即视野变换矩阵

[ AZx AZy AZz 0 ] [ 0 0 1 - Pz ]

[ 0 0 0 1 ] [ 0 0 0 1 ]

Trick

求出ViewSpace原点到ViewSpace中任意一点Pview的向量在WorldSpace中的值

传统的

设ViewSpace原点在世界坐标中的位置OViewworld

根据 ViewInv×Pview - OViewworld 求出

但是 视野变换的逆矩阵 的第三行 即 平移了OViewworld

只要将第三行设置为[0 0 0 1] 即可直接求出向量在WorldSpace中的值

而不再需要减去OViewworld

### 投影变换

#### 正交投影

XXXXX

#### 透视投影

##### Reversed-Z

Nathan Reed. "Depth Precision Visualized." NVIDIA GameWorks Blog 2015.

<https://developer.nvidia.com/content/depth-precision-visualized>

最小正浮点数

FLT\_EPSILON //#include <float.h>

//Reversed Z

//Unity3D中

#include "HLSLSupport.cginc"

//...

#if defined(UNITY\_REVERSED\_Z)

//...

#else

//...

#endif

##### 透视投影的透视校正插值

Jim Blinn. "Hyperbolic Interpolation." IEEE 1992.

<https://doi.ieeecomputersociety.org/10.1109/MCG.1992.10028>

在Direct3D12、Vulkan中默认在插值时透视校正（由硬件完成）

HLSL/Reference for HLSL/Language Syntax/Variables/Data Types/InterpolationModifier

<http://msdn.microsoft.com/en-us/library/windows/desktop/bb509668>

GLSL Specification/Variables and Types/Interpolation Qualifiers

<http://www.khronos.org/registry/OpenGL/index_gl.php>

在Vulkan Specification中给出了透视校正插值的公式

直线图元

Rasterization/Line Segments/"perspective interpolation"

三角形图元

Rasterization/Polygons/"perspective interpolation"

比如我们有一个黑白相间的正方形，M恰好为正方形的边上的中点（可以认为是UV = 0.5的点）：



我们假定该正方形经过透视投影映射到二维空间中的梯形：



我们发现，根据“近大远小”的原理，黑白分界线会向远处偏移，在二维空间中，M（UV = 0.5的点）不再是梯形的边上的中点。因此，图形流水线的光栅化阶段在插值计算各个像素的UV时，不能简单地按照二维空间中的比例（即λ\_normdev）进行计算。

我们回忆透视投影中三维空间中的几何体顶点经过MVP变换映射到二维空间中的过程：

ModelSpace->WorldSpace->ViewSpace

//w\_clip = z\_view

ClipSpace -> NormalizedDeviceSpace

//由图形流水线中的固定功能阶段完成



我们设根据△ABC插值得到点P：

由于ModelSpace->WorldSpace->ViewSpace是线性变换，不难证明λ\_view = λ\_world = λ\_model。但是ViewSpace->ClipSpace->NormalizedDeviceSpace并不是线性变换，导致了二维空间中的λ\_normdev不再与三维空间中的λ\_view相等，从而产生了上述问题。

我们只要找到某种方式，基于λ\_normdev求出λ\_view，就可以做到按照三维空间中的比例λ\_model对顶点属性VTXATTR进行插值。

Blinn在1992年提出了用双曲（Hyperbolic）插值来解决这个问题，这也是目前图形流水线中的光栅化阶段所使用的方法，被称为透视校正（Perspective Correct）插值（由于双抛物面映射的透视校正插值由本人原创，在没有特殊说明的情况下，透视校正插值即指透视投影的透视校正插值）：

//w\_clip = z\_view //VTXATTR可以是任意顶点属性，包括但不限于UV

可以证明：

证明：

我们设三维空间中共面的四点P、A、B、C在ViewSpace中的坐标为、、、

根据λ\_normdev的定义，我们有：

上式也可以看作三维空间中的坐标：

即：

即：

由于P、A、B、C四点共面，根据共面向量定理，我们有：

存在唯一的实数λA\_view、λB\_view、λC\_view，使，且λA\_view + λB\_view + λC\_view = 1；

由于唯一性，我们有：

由于w\_clip = z\_view，我们有：

#### 双抛物面投影

##### 双抛物面映射

Wolfgang Heidrich, Hans-Peter Seidel. "View-independent Environment Maps." EUROGRAPHICS 1998.

<https://vccimaging.org/Publications/Heidrich1998VEM/Heidrich1998VEM.pdf>

Imagination Technologies Limited. "Dual Paraboloid Environment Mapping." Power SDK Whitepaper 2017.

<https://github.com/powervr-graphics/Native_SDK/tree/4.3/Documentation/Whitepapers>

双抛物面（Dual Paraboloid）映射在本质上是一种投影函数（Projector Function），它将球函数（Spherical Function）的定义域（三维空间中的任意方向，可以等价地表示为三维空间中的任意单位向量，一般用于表示从三维空间中的某个固定点出发的各个方向，可以形象地理解为从球心出发到球面上的任意点的方向）映射到二维空间中的坐标，以允许用图像（Image）来保存球函数。

我们设（椭圆）抛物面，不难理解它是由准线为z = 1、顶点为、焦点为原点的抛物线绕Z轴旋转一周得到的曲面，简单的说明如下：

令y = 0，我们得到抛物面 与平面ZOX的交线，它是由抛物线向Z轴正方向平移个单位得到的。

而抛物线又可以写成x2 = -2z，对照标准形式x2 = -2pz，可得p = 1，即抛物线的准线为、顶点为、焦点为。

将抛物线向Z轴正方向平移个单位后即得到抛物线，它的准线为、顶点为、焦点为。

设球函数的定义域中的某个自变量为，双抛物面映射的目标是将它映射到的二维空间中的坐标，下面会对该映射过程进行说明：



设从焦点F出发方向为的射线与抛物面交于点P，过点P作准线的垂线段PM，表示PM方向的单位向量为。

与向量FP和向量PM的夹角的角平分线PN的方向相同的向量为。

在四维齐次空间内，以上过程可以用线程变换表示为。

根据抛物线的性质，|PM| = |PF|，∆PMF是等腰三角形，∠NPM = ∠FMP，PN // FM。

又因为。

根据共线向量基本定理，存在唯一的实数λ，使，即，即。

在四维齐次空间内，以上过程可以用线程变换表示为。//矩阵中的第4行模仿透视投影变换，将z\_norm + 1写入w分量，在使用双抛物面映射生成图像时，数乘的操作由图形流水线中的固定功能阶段完成；矩阵中的a和b与透视投影变换中的相同，透视投影中的Reverse-Z等相关技术也可以应用到此处。

综上所述，在四维齐次空间内，整个双抛物面映射的过程可以用线性变换。 //其中，在使用双抛物面映射生成图像时，数乘的操作由图形流水线中的固定功能阶段完成

虽然可以根据以上过程可以将球函数的整个定义域映射到二维空间中，但是从焦点F出发方向接近Z轴负方向的射线，与抛物面的交点接近于无限远处，球函数的整个定义域在二维空间中对应的区域的面积并不是有限的。

因此，我们只将三维空间中z分量>0的方向用抛物面映射到二维空间中；对于三维空间中z分量<0的方向，我们用另一个抛物面进行映射，从而使球函数的整个定义域映射到二维空间中的面积有限的区域（不难证明该区域的边界是一个以原点为中心、半径为1的圆）。

但是这会使我们得到2个图像，分别对应于三维空间中z分量>0和z分量<0的方向。传统的实现（比如Heidrich在1998年给出的实现）用纹理地图集（Texture Atlas）来保存这2个图像。由于传统的纹理地图集具有一定的缺陷：比如在生成MipMap时，这2个图像的交界处的融合可能会导致走样（Aliasing），现代的实现更倾向于用纹理数组（Texture Array）来保存这2个图像。在接下来的讨论中，在没有特殊说明的情况下，我们会假定实现使用现代的纹理数组而非传统的纹理地图集。

用双抛物面映射读取图像

首先我们用双抛物面映射将三维空间中的方向（比如：环境映射中的反射方向（根据视线方向和表面法线计算得到）、泛光灯阴影映射中的光线方向）映射到二维空间中的。随后我们需要用一个线性变换将映射到纹理坐标UV，并根据三维空间中的方向的z分量>0或z分量<0确定纹理数组的索引。

用双抛物面映射生成图像

在某些情况下，我们需要用双抛物面映射生成图像来保存球函数（比如：环境映射用球函数表示光探头在三维空间中各个方向上受到的光照，泛光灯阴影映射用球函数表示泛光灯在三维空间中各个方向上到最近的几何体的距离）。

我们设三维空间中几何体的顶点在ModelSpace中的坐标为，与绘制普通的几何体时一样，我们用 M和V矩阵将它变换到ViewSpace，即= VM。

我们将从原点出发到的方向作为球函数的定义域中的自变量，即；用双抛物面映射将它映射到二维空间中的。由于的x和y分量的范围在之间，恰好与NormalizedDeviceSpace一致，不再需要进行其它变换。

##### 双抛物面投影的透视校正插值

本人原创

我们回忆用双抛物面映射生成图像的过程（为了行文简洁，我们只描述了其中一个抛物面，显然这并不会影响到我们的讨论）：

ModelSpace->WorldSpace->ViewSpace

ViewSpace->SphericalFunctionDomain

SphericalFunctionDomain->ClipSpace

//w\_clip = z\_viewnorm + 1

ClipSpace -> NormalizedDeviceSpace

//由图形流水线中的固定功能阶段完成

我们设图形流水线中的光栅化阶段按照λ\_hw进行插值。

在透视投影中：我们先用线性变换将向量从ModelSpace变换到ClipSpace：；随后用数乘（由图形流水线中的固定功能阶段完成）将向量从ClipSpace变换到NormalizedDeviceSpace：。

根据透视投影的透视校正插值中的相关讨论，我们有λ\_hw = λ\_model。

在双抛物面映射中：我们先用线性变换将向量从SphericalFunctionDomain变换到ClipSpace：；随后用数乘（由图形流水线中的固定功能阶段完成）将向量从ClipSpace变换到NormalizedDeviceSpace：。

对比透视投影的过程，我们不难理解有λ\_hw = λ\_viewnorm成立（详细的证明还是应当参考Blinn在1992年在论文双曲插值中的相关讨论）。

注：使用传统的纹理地图集的实现在ClipSpace -> NormalizedDeviceSpace中需要再乘上某个矩阵，将映射到NormDeviceSpace的一半（上下分或左右分，具体的计算方式取决于的实现），即。由于数乘满足交换律，我们有，整个过程仍可以看作是一个线性变换和一个数乘组成。



由于ModelSpace->WorldSpace->ViewSpace是线性变换，我们有λ\_view = λ\_world = λ\_model。但是ViewSpace->SphericalFunctionDomain并不是线性变换，因此λ\_viewnorm ≠ λ\_view，从而导致图形流水线中的光栅化阶段所使用的λ\_hw不再与三维空间中的λ\_view相等。

我们只要找到某种方式，基于λ\_viewnorm求出λ\_view，就可以做到按照三维空间中的比例λ\_model对顶点属性VTXATTR进行插值。

我们设三维空间中共面的四点P、A、B、C在ViewSpace中的坐标为、、、；设（k ≠ 0）为OPView与△AViewNormBViewNormCViewNorm的交点的坐标。

由于λ\_hw = λ\_viewnorm，我们有，等式两边同时除以k即。（等式Ⅰ）

由于P、A、B、C四点共面，根据共面向量定理，我们有：

存在唯一的实数λA\_view、λB\_view、λC\_view，使，且λA\_view + λB\_view + λC\_view = 1；

由于唯一性，我们有：

（等式Ⅱ）

（等式Ⅲ）

（等式Ⅳ）

，等式两边同时乘k即 （等式Ⅴ）

观察等式Ⅰ的形式，我们尝试将 作为VTXATTR输出到Rasterization，我们将在PixelShader中得到VTXATTR输入：。

观察得到的等式的形式，我们发现与等式Ⅰ恰好相差，我们尝试将作为VTXATTR输出到Rasterization，我们将在PixelShader中得到VTXATTR输入：（根据等式Ⅴ），恰好为k。

我们将以上两个结果相除即可得到按照三维空间中的比例λ\_model对顶点属性VTXATTRORIGIN进行插值的结果：（根据等式Ⅱ、Ⅲ、Ⅳ）

综上所述，在双抛物面映射中正确进行透视校正插值的方法为：

1.将正确的w\_clip值（比如z\_viewnorm + 1）写入到gl\_position (Vulkan)/ SV\_Position0(D3D12)的w分量。

2.对任意需要正确透视校正插值的VTXATTRORIGIN，用VertexShader/TessellationEvaluationShader/GeometryShade输出VTXATTRORIGIN×到Rasterization，Rasterization双曲插值后在PixelShader中对应的属性记作VtxAttrOrigin\_Div\_LengthPositonView。

3.用VertexShader/TessellationEvaluationShader/GeometryShader添加顶点属性并输出到Rasterization，Rasterization双曲插值后在PixelShader中对应的属性记作One\_Div\_LengthPositionView。

4.在PixelShader中将VtxAttrOrigin\_Div\_LengthPositonView和One\_Div\_LengthPositionView相除即可得到正确透视校正插值的VtxAttrOrigin。

# 几何体

### 网格（Mesh）

#### 蒙皮（Skin）

又被称为顶点融合（Vertex Blend）/封套（Envelope）/骨架子空间变形（Skeleton-Subspace Deformation）

##### 矩阵线性融合（Matrix Linear Blend）

//Blend -> Stitch（缝合）

float4x4 InverseJointBind[256]

float4x4 JointAnimate[256]

float4 BlendWeight

uint4 BlendIndex

计算顶点融合

BlendWeight.x\*JointAnimate[BlendIndex.x]\*InverseJointBind[BlendIndex.x]\*Position

+BlendWeight.y\*JointAnimate[BlendIndex.y]\*InverseJointBind[BlendIndex.y]\*Position

+BlendWeight.z\*JointAnimate[BlendIndex.z]\*InverseJointBind[BlendIndex.z]\*Position

+BlendWeight.w\*JointAnimate[BlendIndex.w]\*InverseJointBind[BlendIndex.w]\*Position

法线变换应当使用Normal\*JointBind\*InverseJointAnimate计算

糖果包装纸（Candy Wrapper） Skin Collapse To A Single Point

##### 对偶四元数线性融合（Dual Quaternion Linear Blend）

NVIDIA Direct3D SDK 10.5 Code Samples / Skinning with Dual Quaternions

<https://developer.download.nvidia.com/SDK/10.5/direct3d/samples.html#QuaternionSkinning>

DLB(Dual Quaternion Linear Blend)

设 ^b = w0·^q0 + w0·^q0 + w0·^q0 + w0·^q0 = b0 + b1ε

|w0·^q0 + w0·^q0 + w0·^q0 + w0·^q0| = |^b| = |b0| + ε

由于对偶四元数的模是对偶数（Dual Number） //见上一章中证明

适用于对偶数（Dual Number）的逆

= = - ε

设^bn =

^bn = = ^b× = (b0 + b1ε)( - ε) = + ( - )ε

设^bn = r0 + ([0, (1/2)×]×r0)ε //其中r0是单位模四元数

由于^bn是单位对偶四元数，以上等式一定有解，即r0和一定存在 //见前面章节

//旋转

r0 =

= //显然是单位模四元数

+ cross(r0\_v, cross(r0\_v, ) + ×r0\_s)×2 //不构造矩阵 //见前文

//平移

[0, (1/2)×]

= ([0, (1/2)×]×(r0×) = ([0, (1/2)×]×r0)×

= ( – ) ×

= ( – ) ×

= –

= – //注意[(b0\_s·b1\_s + dot(b0\_v, b1\_v)), ]是实数，因此满足交换律

= –

= –

=

=

=

=

=> =

=

顶点变换

四元数切线空间变换

理论上完美的方法应当是SCLERP（Screw Linear Interpolation，螺旋线性插值）//可以认为是SLERP的泛化

DLB在原理上与着色中融合法线的方式的相同

先在光栅化阶段线性融合，再在像素着色器阶段归一化

虽然与理论上完美的本质的融合（基于角度融合）不同，但是近似值往往是足够的（尤其是N1和N2“较近”（即夹角较小））

//糖果包装纸（Candy-Wrapper）/ 蒙皮塌陷（Skin Collapse）

沿最短轨迹融合 //根据螺旋运动（Screw Motion）//原文中是插值，但个人认为是融合

论文只涉及根据Weight融合，并不涉及到根据时间t进行插值

个人理解Translate -> Linear | Rotation -> Slerp

与融合时的方式相似 原理来自于Phone着色中的法线插值

对映性Antipodality

上文中 –Q和Q表示相同的旋转变换

但是融合结果不同

可以认为，相对于四元数而言，旋转矩阵维度丢失

//传统的工具链是矩阵

//但在Maya 2015或3ds Max 2016中支持已经支持对偶四元数

//传统的工具链在插值产生动画曲线时，使用四元数

//因此，直接使用该四元数即可

对于只提供矩阵的情形

论文给出以下方法

1.On-The-Fly

融合时，使d1、d2、d3与d0的点积大于0，以选择d1、d2、d3的符号

2.预计算

遍历Skeleton

根节点的符号任意（一般取与UpDirection不超过π）

每个子节点的四元数与父节点的四元数点积大于0，以选择子节点四元数的符号

#### 变形（Morph）

### 法线贴图（Normal Map）

归一化后的法线在光栅化插值并不一定归一化，需要再次归一化

### 地形（Terrain）

细分 相当 “小光栅化” 产生较密的顶点来对HeightMap进行“接近于像素级别的”采样

可以不使用细分，而是将这些运行时生成的顶点预处理成Mesh，细分的好处在于可以在着色器中进行LOD

为什么不在像素着色器中进行？

像素着色器写入到ColorBuffer和DepthBuffer的位置（即x和y坐标）在光栅化阶段就已经确定，对HeightMap进行采样后只能修改SV\_Depth（即Z坐标），即无法修改x和y

如何解决？

使用计算着色器

未解决？

使用四边形图元取代三角形图元，计算着色器Dispatch中的x和y作为四边形图元的Width和Height，z作为四边形图元的个数，用计算流水线取代现有的图形流水线？

倾向：光栅化阶段的分辨率不可确定

细分能算二次贝塞尔曲线，计算着色器也能算

但是透视投影带来的非一一映射

#### 地形映射（Relief Mapping）

与SSR相同，利用Ray-Tracing进行采样 解决了Depth的问题，这种情况下不需要Depth

Morgan McGuire, Michael Mara. “Efficient GPU Screen Space Ray Tracing”. JCGT 2014.

#### 置换贴图（Displacement Map）/高度场（HeightField）

DirectX SDK (June 2010) Samples / DetailTessellation11

NVIDIA SDK11 Samples / TerrainTessellation

<https://developer.nvidia.com/dx11-samples>

NVIDIA GameWorks Vulkan and OpenGL Samples / Terrain Tessellation Sample

<https://developer.nvidia.com/gameworks-vulkan-and-opengl-samples>

Displacement/HeightMap在TagentSpace中定义，适用于任何几何体，不仅适用于地形（平行水平面的长方形几何体）

##### 基于密度的细分（Density-based Tessellation）

[AMD 2010]

CreateDensityMapFromHeightMap

HeightMap->DensityMap

比较当前像素与周围像素之间的变化

LT T RT

L M R

LB B RB

DensityMap中的数据表示Variation

CreateEdgeDensityVertexStream

DensityMap->DensityBuffer

根据三角形面片中的UV，求出三角形三条边上的各个点在DensityMap中的采样结果（表示Variation）的最大值，分别作为影响三角形域的3个外部细分因子的依据

以上3个最大值的最大值作为影响三角形域的（唯一的）内部细分因子的依据

以上过程的不正确性在于忽略三角内部的点在DensityMap中的采样结果

##### Level Of Detail

面片丢弃

Vulkan Specification Tessellation/Tessellator Patch Discard

gl\_TessLevelOuter中任tiao意一个小于或等于0

面片剔除

Programming Guide for Direct3D 11 Graphics Pipeline/Tessellation Stages

//Hull Shader中的剔除

//由于Patch的Height不确定，应当将Frustum投影到二维平面（得到梯形，也可以取梯形的AABB）再与Patch进行Intersection Test

//以上过程可以用四叉树，但是由于地形的特殊性，四叉树的所以叶节点的大小都相同，因此可以使用二维数组（即只存储四叉树的叶节点）

//CPU端的剔除

//一般认为Tile的Intersecton Test在CPU端进行，因为Tile往往与Asset的加载/释放有关

//一个Tile往往由若干个Patch组成

//同一Patch中的LOD一定相同，显然整个Tile的LOD都相同是不切实际的

//Hull Shader中的Intersection Test可以认为是更细粒度的剔除（即Tile->CPU Cull Patch->Hull Shader Cull）

##### Normal Map

错误做法：

在Domain Shader中求出PositionWorld，在Pixel Shader中cross ddx ddy求出Normal

原因：

以上方式求得的Normal为Face Normal而非Vertex Normal，在三角形图元之间会出现明显的棱角

//对于平面构成的几何体（比如长方形），不会有任何负面影响

//对于曲面构成的几何体，顶点法线为相邻的若干个平面法线的加权混合

正确做法：

使用Gather指令得到4个Texel

//x (0,1) y(1,1) z(1,0) w(0,0)

//w同时用于Displacement

DDX=(WorldSizePerTexel,0,z-w)

DDY=(0,WorldSizePerTexel,x-w)

normalize(cross(DDX,DDY)) = normalize(

-(z-w)\*WorldSizePerTexel,

-(x-w)\*WorldSizePerTexel,

WorldSizePerTexel\*WorldSizePerTexel

)

= normalize(

WorldSizePerTexel\*(

-(z-w),

-(x-w),

WorldSizePerTexel)

)

= normalize(

w-z,

w-x,

WorldSizePerTexel

)

以上计算结果为Vertex Normal，光栅化阶段对Normal进行插值后，在Pixel Shader中再次Normalize，Shading的效果正常

//关于Face Normal和Vertex Normal

Jason Shankel,Maxis. "Fast Heightfield Normal Calculation". Chapter 3.4 Game Programming Gems 3.

在NVIDIA GameWorks Vulkan and OpenGL Samples中，计算Normal在Domain Shader中进行

因为Normal对Shading效果的影响远大于Position（详见Normal Map）

细分因子必须时刻维持在较高的水平，以保证较高的采样频率

如果LOD算法导致细分因子产生较大的变换，那么Normal的采样频率也会随之变换，在视觉上会产生明显的“闪烁”

在NVIDIA SDK11 Samples 中，计算Normal在Pixel Shader中进行

根据Normal Map的原理，这种方案允许较低水平的细分因子，三角形图元个数较低，效率较高，个人更倾向于这种方案

//这种方案对HeightField的分辨率有一定要求，否则容易产生“栅格化”

//实际实验中，textureGather会产生“栅格化”

//textureOffset结合Linear采样并不会产生”栅格化” //Point采样仍会“栅格化”

//Normal的采样频率恒定（Per Pixel），与LOD无关，不可能“闪烁”

//实践中，这种方案能大幅度提升帧率

##### 纹理映射

Tri-Planar Projection

Stochastic Cliff Shading //Terrain Rendering In Far Cry 5 GDC 2018

### 贴花（Decal）

DirectX SDK (June 2010) Samples / DecalTessellation11

Jan Krassnigg. "A Deferred Decal Rendering Technique". Game Engine Gems 1 Chapter 20 2010

Tomas Akenine-Moller. "Decal Rendering". Real-Time Rendering 4th Edition Chapter 20.2 2018

//

绘制 立方体

### 头发（Hair）/毛发（Fur）

参考文献

//NVIDIA Direct3D SDK 10.5 Code Samples / Fur - Shells and Fins

//<https://developer.download.nvidia.com/SDK/10.5/direct3d/samples.html#Fur>

[Tariq 2008] Sarah Tariq, Louis Bavoil. "Real Time Hair Simulation and Rendering on the GPU." SIGGRAPH 2008.

<http://www.nvidia.com/object/siggraph-2008-hair.html>

[Yuksel 2010] Cem Yuksel, Sarah Tariq. "Advanced Techniques in Real-time Hair Rendering and Simulation." SIGGRAPH 2010.

<http://www.cemyuksel.com/courses/conferences/siggraph2010-hair>

NVIDIA SDK11 Samples / Hair

<https://developer.nvidia.com/dx11-samples>

NVIDIA HairWorks

<https://developer.nvidia.com/hairworks>

[Larsen 2017] Eric Larsen ,Dongsoo Han. "Real-Time Finite Element Method (FEM) and TressFX 4.0." GDC 2017.

<https://gpuopen.com/gdc2017-real-time-finite-element-method-and-tressfx-4-0>

AMD TressFX

<https://gpuopen.com/gaming-product/tressfx>

本章只介绍渲染，第三部分介绍物理模拟

渲染

### 草地（Grass）

Klemens Jahrmann, Michael Wimmer. "Interactive Grass Rendering Using Real-Time Tessellation." WSCG 2013.

<https://www.cg.tuwien.ac.at/research/publications/2013/JAHRMANN-2013-IGR/JAHRMANN-2013-IGR-paper.pdf>

Evgeny Makarov. " NVIDIA Turf Effects: Massive Grass Rendering With Dynamic Simulation." GTC 2015.

<http://on-demand.gputechconf.com/gtc/2015/presentation/S5748-Evgeny-Makarov.pdf>

<https://developer.nvidia.com/turfeffects>

Klemens Jahrmann, Michael Wimmer. "Responsive Real-Time Grass Rendering for General 3D Scenes." I3D 2017.

<https://www.cg.tuwien.ac.at/research/publications/2017/JAHRMANN-2017-RRTG/JAHRMANN-2017-RRTG-draft.pdf>

<https://github.com/klejah/ResponsiveGrassDemo>

3.GPU Pro

传统的——公告板（Billboard）

Kurt Pelzer, Piranha Bytes. "Rendering Countless Blades of Waving Grass". GPU Gems 2004.

在开启Alpha To Coverage的情况下，EarlyDepthStencil会与Depth Write冲突，但与Depth Test并不会冲突

当像素着色器写入SV\_Coverage时，AlphaToCoverage被忽略

AlphaToCoverage可以在着色器中以软件方式实现

将光栅化阶段产生的Coverage（像素着色器的读取SV\_Coverage / gl\_SampleMaskIn）与SV\_Target0.a转化得到的Coverage按位与的结果写入SV\_Coverage / gl\_SampleMask

### 布料（Cloth）

NVIDIA Direct3D SDK 10.5 Code Samples / Cloth Simulation

<https://developer.download.nvidia.com/SDK/10.5/direct3d/samples.html#Cloth>

Tae-Yong Kim, Nuttapong Chentanez, Matthias Müller-Fischer. "Long Range Attachments - A Method to Simulate Inextensible Clothing in Computer Games." Eurographics Association 2012.

<https://developer.nvidia.com/research>

PhysX Clothing

<https://docs.nvidia.com/gameworks/content/gameworkslibrary/physx/guide/Manual/Cloth.html>

NVIDIA Clothing

<https://docs.nvidia.com/gameworks/content/gameworkslibrary/physx/nvCloth/UsersGuide/Index.html>

### 公告板（Billboard）

公告板顶点坐标（局部坐标系中）

//以锚点为中心，位于XOY平面内的四边形

(-1.0,1.0,0.0f)

(1.0,1.0,0.0f)

(-1.0,-1.0,0.0f)

(1.0,-1.0,0.0f)

公告板局部坐标系

//概念上即ModelSpace

正上方向 —— UpDirection //比如Y轴正向 //以Maya Z Up为例

正右方向 //比如X轴正向

正前方向 —— FrontDirection/Normal //比如Z轴负向 //由于公告板在局部坐标中的形状，FrontDirection又叫Normal

锚点——AnchorLocation //原点O

//Unity3D

//Normal（控制光照）与FrontDirection（控制几何变换）解耦

//Editor: Renderer/NormalDirection -> Normal

//UE4

//

一般不会通过ModelTransform变换公告板

在确定了AnchorLocation，FrontDirection和UpDirection后，根据基变换将公告板变换到WorldSpace

//Right-LocalSpaceX //LocalSpace即ModelSpace

//Up-LocalSpaceY

//-Front-LocalSpace

//但由于P.z为0，对结果没有影响的FrontDirection可以省略

基变换（In WorldSpace）

[RightDirection.x UpDirection.x -FrontDirection.x] [P.x] [AnchorLocation.x]

[RightDirection.y UpDirection.y -FrontDirection.y] \* [P.y] + [AnchorLocation.y]

[RightDirection.z UpDirection.z -FrontDirection.z] [P.z] [AnchorLocation.z]

//由于P.z为0，可以不构造矩阵，从而优化如下

RightDirection \* P.x + UpDirection \* P.y + AnchorLocation

但是，在原理上

FrontDirection和UpDirection决定了ModelTransform中的Rotation

AnchorPosition决定了ModelTransform中的Translation

FrontDirection和UpDirection并不一定垂直

可以保留FrontDirection并重新推导UpDirection或保留UpDirection并重新推导FrontDirection

推导过程与推导ViewTransform的过程类似

//左手系

//确定UpDirection和FrontDirection的过程往往

RightDirection = normalize(Cross(UpDirection, FrontDirection)) //单位向量叉乘后的结果不一定是单位向量，除非垂直

//UpDirection = normalize(Cross(FrontDirection, RightDirection)) //保留FrontDirection并重新推导UpDirection为例

或//FrontDirection = Cross(RightDirection, UpDirection) //由于P.z为0，FrontDirection对结果没有影响

//右手系

...

// FrontDirection，UpDirection，AnchorLocation可以在任意空间中（包括但不限于WorldSpace、ViewSpace）

公告板全局坐标

1.球对称（Spherical Symmetry）/世界朝向（World-Oriented）

公告板的UpDirection为世界空间的UpDirection

//-----------------------------------------------------------------------------------

//视平面对齐（ViewSpace-Aligneded）

公告版的FrontDirection为传感器摄像机（用于推导ViewTransform）的EyeDirection

//保留FrontDirection并重新推导UpDirection

//世界空间的UpDirection和传感器摄像机（用于推导ViewTransform）的UpDirection相同

//即屏幕对齐（Screen-Aligned）

//可用于显示注释文本，即“公告板”的命名由来

//但在某些文献中被认为是2D Sprite而非公告板

//公告板的FrontDirection和UpDirection为传感器摄像机（用于推导ViewTransform）的EyeDirection和UpDirection

//保留FrontDirection并重新推导UpDirection

屏幕对齐公告板的局部坐标系在ViewSpace中是预定义的（与ViewSpace相同），不需要计算

可以AnchorLocation在ViewSpace中的值

使用向量加法（+ AnchorLocation）将公告板从ModelSpace（即局部坐标系）变换到ViewSpace

//--------------------------------------------------------------------------------------------------------------------------------------

1-2.视点朝向（ViewPoint-Oriented）

球对称性（Spherical Symmetry）

//公告版的FrontDirection为从公告板的AnchorLocation到传感器摄像机的EyePosition

//保留FrontDirection并重新推导UpDirection

//Unity3D

//Editor:

//Renderer/RenderMode:Billboard+Renderer/BillboardAlignment:View -> 屏幕对齐 / 世界朝向——视平面对齐

//Renderer/RenderMode:Billboard+Renderer/BillboardAlignment:Facing -> 世界朝向——视点朝向

//UE4

//

//--------------------------------------------------------------------------------------------------------------------------------------

2.圆柱对称（Circular Symmetry）/ 轴向（Axial）

限制为绕某个轴旋转并尽可能朝向观察者

//圆对称（三维）/圆柱对称（Circular Symmetry）

//公告版的UpDirection为旋转轴

//公告版的FrontDirection为从公告板的AnchorLocation到传感器摄像机的EyePosition

//保留UpDirection并重新推导FrontDirection(与1-2的区别)

//理论上以上为视平面对齐（ViewSpace-Aligneded）的版本

//应当存在视点朝向（ViewPoint-Oriented）的版本

//示例

//远处的树 可以用绕WorldSpace的UpDirection旋转的公告板来实现

//雨 可以用绕RainDirection

#### 纹理动画

现代的 纹理数组 采样两次 //MipMap只融合一帧（XOY平面） 帧之间由应用程序根据时间融合

//传统的 Atlas 二维 纹理地图集

//其它 三维纹理 Z坐标表示不同帧率（即时间） //缺点 Mip Map会导致在相邻帧之间融合（融合XYZ空间）

### 粒子系统（Particle）

PopcornFX

<https://www.popcornfx.com>

AMD GPUOpen GPUParticles11

<https://gpuopen.com/gaming-product/gpuparticles11-directx-11-sdk-sample/>

#### 速度场（Velocity Field）

Mark J. Harris. " Fast Fluid Dynamics Simulation on the GPU." Chapter 38 GPU Gems 2004.

<https://developer.nvidia.com/gpugems/GPUGems/gpugems_ch38.html>

纳维-斯托克斯（Navier-Stoke）方程 //粘性 //不可压缩 //流体

### 冒充者（Impostor）

### 树（Tree）

Alexander Kharlamov, Iain Cantlay, Yury Stepanenko. "Next-Generation SpeedTree Rendering". GPU Gems 3

<https://developer.nvidia.com/gpugems/GPUGems3/gpugems3_ch04.html>

### 水面（Water）

Matthias Müller-Fischer. "Fast Water Simulation for Games Using Height Fields". GDC 2008.

<http://nvidiagameworks.github.io/GraphicsSamples/ComputeWaterSurfaceSample.htm>

基于HeightMap

模拟Rain

Disturbance 生成一个具有顶峰的分布

Curvature->Force

Clamp近似Damp

#### 折射（Refraction）

Tiago Sousa. "Generic Refraction Simulation". Chapter 19 GPU Gems 2 2005.

<https://developer.nvidia.com/gpugems/GPUGems2/gpugems2_chapter19.html>

//CryEngine //利用法线贴图的XY扰动UV //与物理没有任何关系

//折射掩码 不透明物体在透明物体之前

//OIT中的折射不考虑距离

//OIT会导致水面不会写入深度 //从而导致SSR无法进行

//可以用前向方式绘制水面 为水面单独编写像素着色器 从SV\_POSITION中读取当前像素的深度 进行RayTrace 从而产生SSR效果

#### 反射（Reflection）

传统的平面反射基本上已被次世代的SSR取代

基于 关于平面对称变换的传感器 进行渲染 得到环境映射

//在OpenGL中

//由于UV的影响 texelFetch的坐标为ivec2(Width - gl\_FragCoord.x, l\_FragCoord.y)

//半透明材质不写入深度 不会产生SSR效果

//可以用前向方式绘制水面 为水面单独编写像素着色器 从SV\_POSITION中读取当前像素的深度 进行RayTrace 从而产生SSR效果

//UE4中Lighting Mode设置为Surface PerPixel 并且 控制台参数r.ForwardLightin设置为1

### 海洋（Ocean）

#### FFT（Fast Fourier Transform，快速傅里叶变换）

Julius O. Smith III. "Mathematics of the Discrete Fourier Transform (DFT): with Audio Applications - Second Edition." 2007.

<https://www.dsprelated.com/freebooks/mdft/>

AMD Developer Central / WhitePaper / OpenCL Optimization Case Study Fast Fourier Transform

<https://developer.amd.com/resources/articles-whitepapers/opencl-optimization-case-study-fast-fourier-transform-part-1/>

<https://developer.amd.com/resources/articles-whitepapers/opencl-optimization-case-study-fast-fourier-transform-part-ii/>

Dan Petre, Adam Lake, Allen Hux. "OpenCL™ FFT Optimizations for Intel® Processor Graphics." IWOCL 2016.

<https://dl.acm.org/citation.cfm?id=2909451>

Microsoft DirectXMath XDSP

<https://github.com/Microsoft/DirectXMath/wiki/XDSP>

AMD GPUOpen clFFT

<https://gpuopen.com/compute-product/clfft/>

NVIDIA CUDA cuFFT

<https://developer.nvidia.com/cufft>

##### 傅里叶级数（Fourier Series）

**傅里叶级数（Fourier Series）**：周期性连续（Periodic-Continuous）的傅里叶变换

//《高等数学第六版下册》（ISBN 978-7-04-021277-8） 第十二章无穷级数 第七节傅里叶级数 2007

**三角函数系的正交性**

基函数cos(kx)和sin(kx)在[0, 2π]上正交 //k=1,2,3...

即：

= 0 //k=1,2,3... //等式1

= 0 //k=1,2,3... //等式2

= 0 //k1=1,2,3... //k2=1,2,3... //等式3

= //k1=1,2,3... //k2=1,2,3... //等式4

= //k1=1,2,3... //k2=1,2,3... //等式5

证明：

等式1：

= = 0

等式2：

= = 0

等式3：

当k1≠k2时

= /\*积化和差公式\*/ = = 0

当k1=k2时

设k1=k2=k

= /\*二倍角公式\*/ = = 0

等式4：

当k1≠k2时

= /\*积化和差公式\*/ = = 0

当k1=k2时

设k1=k2=k

= = /\*二倍角公式\*/ = =

等式5：

当k1≠k2时

= /\*积化和差公式\*/ = = 0

当k1=k2时

设k1=k2=k

= = /\*二倍角公式\*/ = =

**函数展开成傅里叶级数**

n为时域

x(n)为时域曲线 //小写x

k为频域

Xk为频域序列 //大写X

设时域曲线x(n)能在[0, N]上展开成傅里叶级数，即：

x(n) =

//将n’=n代入，以上展开即可变换到[0, 2π]上，与上文中讨论的区间一致

等式两边同时乘以cos(i)，并在[0, N]上求定积分：

= = /\*根据**三角函数系的正交性**，其余项都为0\*/ =

等式两边同时除以，得到：

ai = 即ak =

等式两边同时乘以sin(i)/\*i∈0,1,2...N\*/，并在[0, N]上求定积分：

= = /\*根据**三角函数系的正交性**，其余项都为0\*/ =

等式两边同时除以，得到：

bi = 即bk =

频域序列Xk即为和组成的二维向量：

Xk = =

将ak和bk用Xk代入后，得到：

x(n) = //该等式即为**傅里叶级数逆变换**

##### DFT（Discrete Fourier Transform，离散傅里叶变换）

**DFT（Discrete Fourier Transform，离散傅里叶变换）**：周期性离散（Periodic-Discrete）的傅里叶变换

n为时域

xn为时域序列 //小写x

k为频域

Xk为频域序列 //大写X

DFT即：

定义频域序列Xk为：//k∈0,1,2...N-1

Xk =

当k≤时，DFT中的Xk可以看作是用矩形法近似计算相应的傅里叶级数中的Xk的定积分，即：//《高等数学第六版上册》（ISBN 978-7-04-020549-7） 第五章定积分 第一节定积分的概念与性质 三、定积分的近似计算 2007

Xk[0] = = ≈

Xk[1] = = ≈

当k>时，根据奈奎斯特（Nyquist）定理，采样频率过低导致被积函数失真，矩形法无法正确计算出近似值，此时有Xk[0]= XN-k[0]和Xk[1]=－XN-k[1]成立：

Xk[0] = = = = = = XN-k[0]

Xk[1] = = = = = = －XN-k[1]

**IDFT（Inverse Discrete Fourier Transform，逆离散傅里叶变换）**即：

定义时域序列xn为：//n∈0,1,2...N-1

xn =

IDFT可以认为是计算傅里叶级数逆变换中0到的低频部分，而忽略到N的高频部分，即：

x(n)/\*傅里叶级数逆变换\*/ =

= ( /\*计算低频部分\*/ + /\*忽略高频部分\*/)

≈

xn/\*IDFT\*/ =

= + /\*低于奈奎斯特频率而失真\*/

= +

= + /\*变换到0~\*/

= +

=

以上两式相等，证明结束

##### FFT（Fast Fourier Transform，快速傅里叶变换）

三个DFT操作 //以及对应的定理

1.拉伸（Stretch）定理

设0~N-1上的时域序列xn对应的DFT频域序列为Xk；拉伸得到0~2N-1上时域序列

我们有：x’n对应的DFT频域序列

证明：

根据DFT定义：

Xk =

X’k =

X’k[0] =

= /\*偶数\*/ + /\*奇数\*/

= +

= + 0

当k≤N-1时：

上式 = Xk[0]

当k≥N时：

上式 =

=

=

= Xk-N[0]

X’k[1] =

= /\*偶数\*/ + /\*奇数\*/

= +

= + 0

当k≤N-1时：

上式 = Xk[1]

当k≥N时：

上式 =

=

=

= Xk-N[1]

2.移位（Shift）定理

设0~N-1上的时域序列xn对应的DFT频域序列为Xk；移位得到0~N-1上时域序列

我们有：x’n对应的DFT频域序列

证明：

根据DFT定义：

Xk =

X’k =

X’k[0] =

= +

=

=

=

=

= +

= Xk[0] + Xk[1]

X’k[1] =

= +

=

=

=

=

= +

= +

= Xk[1] + Xk[0]

X’k =

=

=

= Xk

3.加法定理 //实际上DFT满足线性（Linearity）运算

设0~N-1上的时域序列x1n对应的DFT频域序列为X1k；0~N-1上的时域序列x2n对应的DFT频域序列为X2k；加法得到0~N-1上时域序列x’n = x1n + x2n

我们有：x’n对应的DFT频域序列X’k = X1k + X2k

证明从略

**DIT（Decimation In Time，时域抽取）的FFT**

//除了DIT，(Cooley-Tukey)FFT还有另外一个变体：DIF(Decimation In Frequency，频域抽取)

设0~N-1上的时域序列xn /\*N为偶数\*/；

0~((N-1)/2)上的时域序列xevenn = x2n的DFT频域序列为Xevenk；

0~((N-1)/2)上的时域序列xoddn = x2n+1的DFT频域序列为Xoddk

//即：xn的偶数项构成xevenn，奇数项构成xoddn

xn的DFT频率序列Xk =

//其中 又被称为调节因子（Twiddle Factor）

证明：

根据拉伸定理

设0~N-1上时域序列xeven’n = =

xeven’n的DFT频率序列Xeven’k =

设0~N-1上时域序列xodd’n = =

xodd’n的DFT频率序列Xodd’k =

根据移位定理

设0~N-1上时域序列xodd’’n = = =

xodd’’n的DFT频率序列Xodd’k = Xodd’k

由于xn = xeven’n + xodd’’n

根据加法定理

xn的频率序列Xk = Xeven’k + Xodd’‘k = Xeven’k + Xodd’k

=

**倒位序（Bit-Reversal Permutation）**

在计算FFT时，需要不断将**原序列**分割成**2个**分别由偶数项和奇数项构成的**子序列**；同时，需要利用**倒位序**使在**逻辑上**不连续/\*分别由偶数项和奇数项构成\*/的2个子序列在**物理上**在原序列中连续，从而使每次分割时不需要移动原序列中的项

下图以长度为16的原始序列为例，演示了不断将原序列分割成2个子序列的过程：

显然，将原始序列的倒位序作为FFT的输入，即可保证每次分割生成的2个在逻辑上不连续的子序列在物理上在原序列中连续

大多数GPGPU编程语言中都内置了计算倒位序的函数

HLSL：reversebits

GLSL：bitfieldReverse

CUDA C：\_\_brev

对于没有内置计算倒位序的函数的编程语言，可以用查表法实现自己的BitFieldReverse函数

<https://graphics.stanford.edu/~seander/bithacks.html#BitReverseTable>

C/C++：

uint32\_t BitFieldReverse(uint32\_t value)

{

constexpr static uint32\_t const \_\_LookUpTable16[] = { 0U, 8U, 4U, 12U, 2U, 10U, 6U, 14U, 1U, 9U, 5U, 13U, 3U, 11U, 7U, 15U };

return \_\_LookUpTable16[(value & 0XF0000000U) >> 28U]

| ((\_\_LookUpTable16[(value & 0X0F000000U) >> 24U]) << 4U)

| ((\_\_LookUpTable16[(value & 0X00F00000U) >> 20U]) << 8U)

| ((\_\_LookUpTable16[(value & 0X000F0000U) >> 16U]) << 12U)

| ((\_\_LookUpTable16[(value & 0X0000F000U) >> 12U]) << 16U)

| ((\_\_LookUpTable16[(value & 0X00000F00U) >> 8U]) << 20U)

| ((\_\_LookUpTable16[(value & 0X000000F0U) >> 4U]) << 24U)

| (\_\_LookUpTable16[(value & 0X0000000FU)] << 28U);

}

长度为N的原始序列的第k项的倒位序为：

BitFieldReverse(k) >> (32U/\*Bit Length Of uint32\_t\*/－log2N)

**串行实现：**

//时间复杂度与**快速排序**相似，为O(Nlog2N)/\*+O(N)倒位序\*/

下图以长度为16的原始序列为例，演示了该算法的运行过程：



void FFT\_DIT(

uin32\_t N/\* Input：频域/时域序列长度\*/,

float2 x[N]/\*Input：0~N-1上的时域序列\*/,

float2 X[N] /\* Output：0~N-1上的频域序列\*/

)

{

for(uint32\_t k=0U; k<N; ++k)

{

uin32\_t n = BitFieldReverse(k) >> (32U/\*Bit Length Of uint32\_t\*/－log2N) //倒位序

X[k] = x[n] //根据DFT定义，计算0~0上/\*长度为1\*/的时域序列的DFT频域序列

}

for(uint32\_t N\_Level=2U; N\_Level<= N; N\_Level\*=2U) //O(Nlog2N)中的log2N

{

for(uint32\_t k\_begin=0U; k\_begin<=(N－N\_Level); k\_begin+= N\_Level) //O(Nlog2N)中的N

{

Butterfly\_DIT(N\_Level, X+k\_begin) //即等式\_.\_\_\_\_

}

}

}

void Butterfly\_DIT (

uin32\_t N/\* Input：频域序列长度\*/,

float2 X[N]/\* Input/Output：0~N-1上的频域序列\*/

)

{

for(uin32\_t k=0U; k<N/2; ++k)

{

float2 Xevenk = X[k]

float2 Xoddk = X[k+N/2]

X[K] = Xevenk + Xoddk

X[K+N/2] = Xevenk + Xoddk

}

}

**并行实现**

#### 置换贴图（Displacement Map）/高度场（HeightField）

Jerry Tessendorf. "Simulating Ocean Water". SIGGRAPH 2004.

<https://people.cs.clemson.edu/~jtessen/reports.html>

NVIDIA CUDA Samples/ FFT Ocean Simulation

NVIDIA SDK11 Samples / OceanCS

<http://developer.nvidia.com/dx11-samples>

NVIDIA WaveWorks

<http://github.com/NVIDIAGameWorks/WaveWorks>

菲利普斯频谱（Phillips spectrum）

### 雨（Rain）

雨滴（RainDrop）

NVIDIA Direct3D SDK 10.5 Code Samples / Rain

<https://developer.download.nvidia.com/SDK/10.5/direct3d/samples.html#rain>

### 矢量图形（Vector Graphics）/路径渲染（Path Rendering）

Charles Loop, Jim Blinn. "Rendering Vector Art on the GPU." GPU Gems 3 Chapter 25 2007.

<https://developer.nvidia.com/gpugems/GPUGems3/gpugems3_ch25.html>

GPU Accelerated Path Rendering

<https://developer.nvidia.com/gpu-accelerated-path-rendering>

### Alpha Test

Chris Green. "Improved Alpha-Tested Magniﬁcation for Vector Textures and Special Effects." SIGGRAPH 2007.

<http://www.valvesoftware.com/publications/2007/SIGGRAPH2007_AlphaTestedMagnification.pdf>

使用Signed Distance Field取代Alpha

### 体素（Voxel）

OpenVDB

<https://www.openvdb.org/>

NVIDIA GVDB VOXELS

<https://developer.nvidia.com/gvdb-samples>

# 颜色

辐射通量（radiant flux） Φ W

辐射强度（radiant intensity） I W·sr-1

辐射照度（irradiance） E W·m-2

//辐射出射度（radiant exitance） M W·m-2

辐射亮度（radiance） L W·m-2·sr-1

光通量（luminous flux） lm（流明）

光强度（luminous intensity） cd（坎德拉）

光照度（illuminance） lx（勒克斯）

//光出射度（

光亮度（luminance） nit（尼特）

Maureen C Stone. "Representing colors as three numbers." IEEE Computer Graphics and Applications 25.4 (2005): 78-85.

<https://wrf.ecse.rpi.edu//wiki/ComputerGraphicsFall2014/stone_colors.pdf>

光谱分布（Spectral Distribution）

|

光谱感受曲线（Spectral Sensitivity Curve）/视锥响应函数（Cone Response Function） //视锥细胞 //不同人之间存在差异 //Input //感受角度

|

L M S //相乘后积分

三色视觉（Trichromacy/ Trichromatism） 所有光谱分布可以用三个值精确表示（相对于视觉系统而言）

同色异谱（Metamerism）三刺激值（Tristimulus Value）相同的光谱不可区分 //色度学（Colorimetry）的基础

颜色匹配函数（Color-Matching Function） //Output //创建角度

//原色（Primary）的可加性

//三刺激值（Tristimulus Value）

//条件等色（Metamer）

CIE 1931 标准观察者 //2°视野 //定义了XYZ颜色空间 //y(λ)和CIE标准光度响应函数（Luminous Response Function）相同 Y即为Brightness/ Luminance(光亮度)

格拉斯曼（Grassmann）定律 三刺激值遵循线性变换 //量化所有颜色空间的基础 //原色（Primary）定义了基变换

CIE 1931 标准观察者 根据 颜色匹配实验（Color-Matching Experience）的结果转换而来 //基于格拉斯曼定律

不同颜色匹配函数之间 可通过 线性变换转换

颜色匹配函数 和 视锥响应函数 之间 可通过 "近似" 线性变换转换 //考虑到眼睛的光学特性的补偿（比如角膜对波长的吸收），就是精确的线性关系 //即色度适应

//CIE 1931 观察环境 2°视野+中立(neutral)背景

色度适应（Chromatic Adaptation）

视觉系统适应亮度和环境光 重新定义"White"

//Von Kries Transform //基于LMS

//参考白

三刺激值 "While"受色度适应影响

感觉颜色空间（Perceptual Color Space） 比如CIELAB/CIELUV 将三刺激值与参考"White"相除 定义唯一的"White"

//更适合

可以将一个观察环境中的颜色空间转换到感觉颜色空间 再转换到另一个观察环境

LMS颜色空间

最新CIE CAM02建议的变换矩阵 //变换矩阵不断演化

=

非线性颜色

在计算机图形学中 三刺激值与光强度成线性关系

在显示技术/数字视频/图像编码中 三刺激值与光强度并不是线性关系 //比如 投影仪显示黑色(0,0,0)

pixel-to-intensity function //

调制操作 //非线性变换

显示器主导（Display-Centric） //原色与显示器相同

宽范围的介质

//材质的颜色 表面被相等能量的白光照亮的反射光归约到三刺激值

光谱（Spectrum）

在自然条件下，光往往是由各种不同波长的光子（photon）组成的

读者可以复习下《物理选修3-5》（ISBN:9787107186110）第十七章中的相关知识

光速=光的波长\*光的频率 其中光速为常量，光的波长和光的频率之间存在着一一对应的关系

光的每种颜色在辐射度学上对应于一个函数——光中各种不同波长的光子所贡献的辐射亮度的分布（波长 -> 该波长的光子的辐射亮度）——该函数被定义为光的波谱（Spectrum）

显然，在实时渲染中，用函数来定义光的每种颜色是不现实的

在实时渲染中，定义域为可见光波长的函数，全部可以用3个系数来近似表示

将3个系数存放在纹理中与将球谐函数的各个系数存放在纹理中有着类似的思想

三刺激值（tristimulus value）

在明视（photopic）条件下（一般认为是在光亮度大于3.4勒特的条件下，人与人之间存在着一定的差异），人类的视网膜中的视锥细胞（cone cell）起主要作用。

人类的视网膜中的视锥细胞有三种，每种视锥细胞只对光中特定波长范围的光子（一般认为是短（420nm-440nm）中（530nm-540nm）长（560nm-580nm）三种波长范围，人与人之间存在着一定差异）敏感。

对于任意光，人类的大脑只会接受到三种不同的信号，分别来自人类的视网膜中的三种视锥细胞。

可以用三种刺激值来表示人类对任意光的感觉。

颜色匹配实验（color match experiment）

人们试图用三种单色光（monochromatic light）（只有特定波长的光子组成的，分别是红（波长）分别对应于人类的视网膜中的三种视锥细胞）来模拟人类对任意一束光的感觉。

人们试图通过颜色匹配实验，测得在模拟人类对可见光波段（一般认为是波长380nm-780nm，人与人之间存在着一定差异）的各种单色光的感觉时，所需的三种单色光R（波长650nm）G（波长526nm）B（波长444nm））的刺激值。

实验测得的结果被称作颜色匹配曲线（color match curve），如图X-X所示

颜色匹配曲线中的负刺激值的含义是，在模拟某些频率的单色光时，需要将用于模拟的红绿蓝三种单色光中的某种光添加到被模拟的单色光中，才能使观察者的感觉一致。

#### CIE XYZ颜色空间

CIE（Commission Internationale de L’Eclairage，国际照明委员会）

CIE 定义了三个假象色XYZ，并且规定XYZ的颜色匹配曲线如图X-X所示

X(λ)

Y(λ)

Z(λ)

同理，对于任意一束频谱为I(λ)的光，所需的三个假想色XYZ的刺激值为：

X= |380nm-780nm I(λ)\* X(λ) dλ

Y= |380nm-780nm I(λ)\* Y(λ) dλ

Z= |380nm-780nm I(λ)\* Z(λ) dλ

其中380nm-780nm即可见光波段

所有实际存在的颜色都在三维空间中的第一卦限（由于XYZ是假象色，实际上还包括了实际中并不存在的颜色）

XYZ颜色空间往往所为其它颜色空间的中心参考

颜色体（Color Volume） 颜色空间中的Space（一般是一个多面体）

Primaries（原色）

1. SRGB / Rec.709（HDTV）

色域 覆盖XY色度图33%

ITU-R Recommendation BT.709 <http://www.itu.int/rec/R-REC-BT.709>

ITU-R International Telecommunication Union Radiocommunication Sector国际电信联盟-无线电通信部门

BT Broadcasting Service Television 广播服务电视

Rec.709和SRGB的原色相同

可以认为Rec.709即SRGB

2.Rec.2020（UHDTV）

色域 覆盖XY色度图63%

转换 以XYZ作为中介 //WhitePoint LMS

//sRGB To XYZ

RGB = EOTF(inRGB)

XYZ = mult( RGB\_2\_XYZ\_mat, RGB)

//Chromatic Adaptation Transform

LMS = mult( XYZ\_2\_LMS\_mat, XYZ)

LMS = LMS \* (outWhite\_LMS / inWhite\_LMS)

XYZ = mult( LMS\_2\_XYZ\_mat, LMS)

//XYZ To Rec.2020

RGB = mult( XYZ\_2\_RGB\_mat, XYZ)

outRGB = OETF(RGB)

#### CIE xy色度图

色度（Chromaticity）可以认为是将Luminace从颜色空间中解耦

XYZ颜色空间 = Luminance + XY色度图

CIE XYZ颜色空间（即以上方式得到的三刺激值(X,Y,Z)的范围）是一个三维空间

可以通过线性变换

x=X/(X+Y+Z)

y=Y/(X+Y+Z)

z=Z/(X+Y+Z)

几何意义 点(x,y,z)即原点到(X,Y,Z)的直线与平面x+y+z=1 的交点

将XYZ颜色空间变换到三维空间中的平面x+y+z=1

由于z=1-x-y，因此z分量可以省略

几何意义 将平面x+y+z=1投影到XOY平面

将三维空间中平面x+y+z=1上的点变换到二维空间

该二维空间即CIE色度图

显然 XYZ颜色空间中 过原点的同一直线上的点具有相同的色度，只是亮度（Brightness，即Y，可以体现光亮度（Luminance）Y）不同 //可以看作是一个xyY坐标系

光谱轨迹（Spectrum Locus） 单色光所在的马蹄形路径

紫线（Purple Line） 连接光谱轨迹两端

所有可见光都在光谱轨迹和紫线围成的区域内

在光谱轨迹之外的颜色 -> 假想色（Imaginary Color）在物理上不存在

在紫线之外的颜色 -> 在物理上存在，但不可能被人类感觉到

#### CIE CAM02色貌模型

CAM -> Color Appearance Model（色貌模型）

02 -> 2002

色貌模型的作用

消除色度适应的影响 //即消除介质和观察环境不同的影响

XYZ三刺激值+参考白(Von) –色貌模型-> Brightness+Hue+Saturation

对色度图中的任意点(x,y)，作过(x,y)和白点(W的直线与光谱轨迹相交：

主波长（Dominant Wavelength） 在光谱轨迹上从红色一端到交点的曲线段 与光谱轨迹总长度 的比值

刺激纯度（Excitation Purity） 白点到(x,y)的距离 与白点到交点的距离 的比值

主波长和刺激纯度是色度学（Colorimetric）中的概念

//主波长 最具有代表性的单色光

//越接近光谱轨迹 刺激纯度/饱和度越高

//Hue和Saturation的概念是否依赖于色域？

在图形学中，使用色相（Hue）和饱和度（Saturation）来描述相应的概念，分别对应于光度学中的主波长和刺激纯度

综上，可以亮度、色调、饱和度（又称感觉颜色空间（Perceptual Color Space））来描述XYZ颜色空间中的任意一点

//Hue可以理解为角度

#### 色域（gamut）

色域（gamut）即颜色空间的中的平行六面体 //在色度图上的投影为三角形

三角形的三个顶点 即 原色（Primary） //一般可以认为是 色域所能接受的 饱和度最高的 ”红” ”绿“ ”蓝“

白点（White Point）即 三个原色的值相同时的点 //在 XYZ基 下并不一定是三角形中心

三原色值的范围（一般是0-1） //定义了最大亮度

色域映射（gamut mapping） 将色域外的颜色映射到色域内 //通常通过Scale和Clip

#### SRGB

#### Rec.2020

### 调制（modulation）操作

调制（modulation）操作依赖于原色

计算L=BRDF\*N 时

精确的做法是 频谱相乘 再基于原色（Primary）积分

实际的做法是 先基于原色积分 再逐分量相乘 导致调制操作依赖于原色

颜色空间：原色（Primary）、颜色分量转换函数（Color Component Transfer Function）、白点色度（White Point Chromaticity）

<http://www.color.org>

经过推导 调制操作与颜色空间的颜色分量转换函数也有关

//1.先调制（逐分量相乘），再变换

//2.先变换，再调制（逐分量相乘）

结果不同！！！

由于一般在Linear中计算光照，因此可以用scRGB取代sRGB？？？

第一代的HDR显示器：色域预期与DCI-P3相近，而非Rec.2020；最大亮度预期1000尼特，而非10000尼特（Rec.2020）

为兼容现有的资产，调试操作基于Rec.709的原色进行，但编码范围可以不再限制于0 – 1，即scRGB颜色空间；因此色域的变宽

实际上，在调制操作以后，可以在转换成Rec.2020

//可以区分不同的Asset

//在延迟渲染中，可以在GBuffer中用ID区分；类似与区分不同的Shading Model

//ColorBuffer基于Rec.2020

//传统的资产基于Rec.709原色调制，调制后转换到Rec.2020，输出到ColorBuffer

//次世代的资产基于Rec.2020原色调制，结果直接输出到ColorBuffer

NVIDIA White Paper中指出，随着时间的迁移，调制操作从基于Rec.709原色过渡到基于Rec.2020原色

### HDR输出流程

SDR显示器即HDTV

HDR显示器即UHDTV

场景参考的（Scene Referred）

PBR中传感器接收到的辐射亮度

场景参考的 -> DXGI\_FORMAT\_R16G16B16A16\_FLOAT

//颜色空间scRGB 原色与sRGB相同 但系数不再限制于[0,1] 因此色域增加

//可以在Rec.2020的空间中

OpenEXR Color Management

<http://www.openexr.com/OpenEXRColorManagement.pdf>

HDRI（High Dynamic Range Image，高动态范围图像）

Florian Kainz, Rod Bogart, Drew Hess. "The OpenEXR Image File Format". Chapter 26 GPU Gems 2004.

<http://developer.nvidia.com/gpugems/GPUGems/gpugems_ch26.html>

ILM（工业光魔）OpenEXR

<http://www.openexr.com/>

在D3D11中 BC6压缩格式已经解决了HDR的压缩问题

ACES//Tone Mapping

输出参考的（Output Referred）

显示器输出的Luminance

UI复合（Composite） //UI -> Blending //UI Asset In sRGB

OETF

R10G10B10A2

在HDR Display中很可能导致镜面反射（Specular）比自发光（Emissive）亮

在SDR Display，由于两者都超过了显示器的Saturation level，以上问题得不到体现

在编辑器层，将光源和Proxy Mesh耦合 //为光源添加 Show Proxy Mesh的选项？？？

自发光 –> 光源/粒子效果

#### ACES（Academy Color Encoding System，学院颜色编码系统）

Evan Hart. "UHD Color for Games". NVIDIA Whitepaper 2016.

<http://developer.nvidia.com/high-dynamic-range-display-development>

Thomas True. "HDR Programming". GTC 2016.

<http://on-demand.gputechconf.com/siggraph/2016/presentation/sig1611-thomas-true-high-dynamic-range-rendering-displays.pdf>

HD（High Definition） 高清

UHD（Ultra High Definition） 超高清

ACES（Academy Color Encoding System，学院颜色编码系统）

美国电影艺术与科学学院（The Academy of Motion Picture Arts and Sciences） 2012

外观修改变换（Look Modification Transform）

对应于传统的颜色分级（Color Grading）

RRT（Reference Rendering Transform，参考渲染变换）

对应于传统的色调映射（Tone Mapping）

HDRdisplay\_sample\shaders\ACES\ACES\_rrt.hlsl

ODT（Output Device Transform，输入设备转换）

根据ACES流程，OETF应当在ODT中完成；但是实际中，往往先进行UI复合，再进行OETF

#### OETF（Optical Electric Transfer Function，光电转换函数）

##### Rec.2020

ST 2084 （Perceptual Quantization，感知量化）

SMPTE（Society of Motion Picture and Television Engineers，电影电视工程师协会）

对RGB的每一个分量C

float m1 = 0.1593017578125f;// 2610 / 4096 \* 1 / 4

float m2 = 78.84375f; // 2523/4096 \* 128

float c1 = 0.8359375f; // 3424 / 4096

float c2 = 18.8515625f; // 2413 / 4096 \* 32;

float c3 = 18.6875f; // 2392 / 4096 \* 32;

C‘ =

JND（Just Notable Difference） //minance

编码值之间的步进（Step）应当不高于JND

HDTV //最大100-200尼特 -> Gamma 2.2 且 8位

UHDTV //最大10000尼特 -> ST 2084 且10位

##### SRGB

SRGB颜色空间 微软（Microsoft）和惠普（Hewlett-Packard）IEC 61966-2-1 （1993年）

<http://msdn.microsoft.com/en-us/library/dd372199>

颜色分量转换函数（Color Component Transfer Function）

<http://www.color.org/chardata/rgb/srgb.xalter>

对RGB的每一个分量C

C' = (C <= 0.0031308f) ? (12.92f \* C) : (1.055 \* pow(C, 1.0f/2.4f) - 0.055f)

//由于C的值一般在0到1之间，伽马校正后场景的亮度一般会变亮

进行颜色分量转换函数的FullScreenPass被称作伽马校正（Gamma Correction）

输入到颜色分量转换函数的RGB值被称作线性（Linear）RGB

Linear RGB

Linear RGB ->Luminance Y=0.212671R+ 0.715160G+ 0.072169B

Direct3D运行时自动伽马校正

Converting data for the color space

<http://msdn.microsoft.com/en-us/library/hh972627>

Format XXX\_SRGB

写入RenderTargetView 线性RGB —> SRGB 即伽马值1/2.2

读取RenderTargetView SRGB —> 线性RGB 即伽马值2.2 //一般发生在Alpha Blend中

读取ShaderResourceView SRGB —> 线性RGB 即伽马值2.2

### EOTF（Electrical Optical Transfer Function，电光转换函数）

由显示器完成 //与软件无关

一般情况下，EOTF和OETF互为反函数

以满足Output-Referred与显示器Luminance成正比的语义

Vulkan

VK\_EXT\_swapchain\_colorspace

HDTV/SDR Display -> VK\_COLOR\_SPACE\_SRGB\_NONLINEAR\_KHR

UHDTV/HDR Display -> VK\_COLOR\_SPACE\_HDR10\_ST2084\_EXT

Direct3D12

IDXGIswapChain3::SetColorSpace1

HDTV/SDR Display -> DXGI\_COLOR\_SPACE\_RGB\_FULL\_G22\_NONE\_P709

//DXGI\_FORMAT\_R8G8B8A8\_UNORM

UHDTV/HDR Display -> DXGI\_COLOR\_SPACE\_RGB\_FULL\_G2084\_NONE\_P2020

//DXGI\_FORMAT\_R10G10B10A2\_UNORM

# 直接光

镜面反射——表面散射 菲涅尔反射

漫反射——体反射

金属性（Metallic）

1. 金属和半导体（金属性强） 几乎不发生传输（Transmit），几乎只有表面散射
2. 绝缘体（Insulator）（金属性弱） 需要同时考虑体反射

均匀程度

1. 均匀（Homogeneous） 几乎没有光学不连续性，几乎不发生散射（Scatter）（散射改变光线的方向） 表现为透明（Transparent）
2. 不均匀（Heterogeneous） 光线在物体内传输，不断散射（改变方向）或吸收（改变能量）

光线射出的位置相对于入射点的偏移可以忽略（一般即小于一个像素）——局部子表面散射 表现为漫发射 可以用BRDF模拟

反照率（Albedo） 描述进入表面内部的光线（即折射光，能量为1-菲涅尔函数）中被散射出表面的能量比例（其余被吸收或散射）

以上条件无法满足时——全局子表面散射

透射=入射-吸收-外散射

吸收的比例可以根据贝尔朗伯定律（Beer-Lambert Law）计算得出，通常依赖于波长（即随着光子的波长的不同而不同，渲染时一般用RGB表示并进行相乘）

外散射

1.光学不连续性

1-1.米散射（Mie Scattering）

通常大于可见光波长---->不依赖于波长

1-2.瑞利散射(Rayleigh Scattering)

接近或小于可见光波长（表现为透明？？？）---->依赖于波长

2. 散射次数

2-1.单次

2-2.多次

## 镜面反射

### 微平面(Microfacet) BRDF

菲涅尔反射/即微平面 光学上平坦 粗糙度小于可见光波长

干涉？衍射？ 粗糙度和可见光波长相近

微几何体 效果以统计方式表现 粗糙度大于可见光波长

微平面理论

Robert L. Cook, Kenneth E. Torrance. "A Reflectance Model for Computer Graphics". SIGGRAPH 81.

适用于第一次反弹(first-bounce) 镜面反射

不适用于多次反射

不适用于体反射（即漫反射？？？）

基于微平面理论的BRDF

Bruce Walter, Stephen R. Marschner, Hongsong Li, Kenneth E. Torrance. "Microfacet Models for Refraction through Rough Surfaces". EGSR 2007.

设NormalMicro = l+v //历史上被称作HalfVector，即微观法线

BRDF(l,v)= D(<l+v>) \* F(<l,l+v>) \* G(l,v) / (4 \* max(0,cos<l,n>) \* max(0,cos<v,n>)) 其中n为宏观法线方向

在公式中 宏观法线normalmacro 粗糙度roughness 镜面颜色colorspecular 都被视为常量

由于4 \* max(0,cos<l,n>) \* max(0,cos<v,n>)描述宏观表面相对于l或v的foreshortening

公式中的G(l,v) / (4 \* max(0,cos<l,n>) \* max(0,cos<v,n>))可以被合并为可见性项（Visibility Term）

即BRDF(l,v)= D(<l+v>) \* F(<l,l+v>) \* V(l,v)

D(<n,l+v>) 法线分布(Normal Distribution)函数

像素着色器 n->宏观上的法向量 （微几何体的法向量）

法线分布函数 D(<n,l+v>) >微观上的法向量（微平面的法向量） 概率密度函数 在球面度0-4π上的积分是1

一个像素对应于多个微平面

只有（微观）法线方向与l和v的角平分线h（Half Vector）相同的微平面才会参与镜面反射

D(l+v)描述了符合上述条件的微平面

F(<l,l+v>) 菲涅尔（Fresnel）函数

菲涅尔函数根据l和(l+v)的夹角计算，注意微平面的法线方向是l+v而不是n（宏观法线）

G(l,v) 几何遮挡（Geometric Occlusion）函数

描述 阴影(Shadow入射光被遮挡)和遮蔽(Mask 视线被遮挡) 的效果

阴影和遮蔽被假定与微平面的方向无关

#### Normalized Blinn-Phong

//为避免NormalMicro和NormalMacro混淆

//建议记作HalfVector 而非NormalMicro

//D

NormalDistribution = ((Smoothness+2)/2π)\*(max(0,cos<HalfVector,Normalmacro>))Smoothness

//其中Smoothness与Glossiness正相关

//max(0,cos<HalfVector,NormalMacro>)

= max(0.0f,dot(normalize(HalfVector),normalize(NormalMacro)))

//F

Fresnel = ColorSpecular

//其中ColorSpecular即Fresnel (0)

V(l,v) = 1/4

即G(l,v)= max(0,cos<LightDirection,NormalMacro>) \* max(0,cos<ViewDirection, NormalMacro>)

物理意义：随着掠射角的增大，阴影和遮蔽减少

问题：当l和v偏离宏观法线n时，阴影和遮蔽增加过快，比实际中暗

BRDF(l,v)

= D(<l+v>) \* F(<l,l+v>) \* G(l,v) / (4 \* max(0,cos<l,n>) \* max(0,cos<v,n>))

=(Smoothness + 2)/8π)\*(max(0,cos<l+v,n>))Smoothness

#### GGX

Bruce Walter, Stephen R. Marschner, Hongsong Li, Kenneth E. Torrance. "Microfacet Models for Refraction through Rough Surfaces". EGSR 2007.

F(<l,l+v>)

Fresnel函数的Schlick近似

Christophe Schlick. "An Inexpensive BDRF Model for Physically based Rendering". Computer Graphics Forum 1994.

F(<l,l+v>) = ColorSpecular + (1 – ColorSpecular)((1 - cos<l,l+v>)^5) 其中ColorSpecular即F(0)

D(<l+v>)

GGX

float D\_GGX( float Roughness, float NoH )

{

//NoH: max(0,cos<NormalMacro,L+V>)

float a = Roughness \* Roughness;

float a2 = a \* a;

float d = ( NoH \* a2 - NoH ) \* NoH + 1; // 2 mad

return a2 / ( PI\*d\*d ); // 4 mul, 1 rcp

}

V(l,v)

基于GGX的Smith几何遮挡函数的Schlick近似

Christophe Schlick. "An Inexpensive BDRF Model for Physically based Rendering". Computer Graphics Forum 1994.

float V\_SmithSchlicKForGGX( float Roughness, float NoV, float NoL )

{

//NoV: max(0,cos<NormalMacro,V>)

//NoL: max(0,cos<NormalMacro,L>)

float a = Square( Roughness );

float a2 = a\*a;

float Vis\_SmithV = NoV + sqrt( NoV \* (NoV - NoV \* a2) + a2 ); //遮蔽

float Vis\_SmithL = NoL + sqrt( NoL \* (NoL - NoL \* a2) + a2 ); //阴影

return rcp( Vis\_SmithV \* Vis\_SmithL );

}

## 漫反射

### 局部子表面散射（Lobal Subsurface Scattering）

ColorDiffuse=(1-Fresnel(<>))\*Albedo //在这里使用宏观的法线方向N

BRDF\_Lambert(l,v)=ColorDiffuse/π

### 全局子表面散射（Global Subsurface Scattering）

参考文献

1.Simon Green. "Real-Time Approximations to Subsurface Scattering." Chapter 16 GPU Gems 2004.

<http://developer.nvidia.com/gpugems/GPUGems/gpugems_ch16.html>

2.Eugene d'Eon, David Luebke. "Advanced Techniques for Realistic Real-Time Skin Rendering". Chapter 14 GPU Gems 3 2007.

<http://developer.nvidia.com/gpugems/GPUGems3/gpugems3_ch14.html>

3.Eric Penner, George Borshukov. "Pre-Integrated Skin Shading." GPU Pro 2 Chapter II-1.

4.Eric Penner. "Pre-Integrated Skin Shading." SIGGRAPH 2011.

<http://advances.realtimerendering.com/s2011/>

表面曲率（Surface Curvature）

环绕光（Wrap Light）

//[1. Green 2004]

面光源（Area Light）



如果已知各个方向的光线的辐射亮度LL，那么计算视线方向辐射亮度Lv=|wL BRDF(l,v) LLcos<l,n> dw

向量辐射照度（Vector Irradiance）

平行光或点光源计算Lv的公式为EN=ELcos<l,n> Lv= BRDF(l,v) EN

但是，现实中不可能存在平行光或点光源，因为球面度为0会导致辐射亮度为正无穷大，平行光/点光源是对现实中的面光源的近似处理，接下来分析上述计算方式在物理上的合理性

对于朗伯表面，由于BRDF为常量，面光源计算Lv的公式可以被写为Lv= BRDF(l,v) |wL LLcos<l,n> dw，平行光/点光源实际上是用ELmax(cos<l,n>,0)表示|wL LL max(cos<l,n>,0) dw

根据向量辐射照度的相关理论可以证明，当WL内的所有光线的反方向与表面法线的夹角都不大于90度时，可以用EL max(cos<l,n>)正确地表示|wL LL max(cos<l,n>,0) dw，其中EL和l分别为向量辐射照度的大小和方向（向量的大小为标量，显然是对RGB通道分别计算）

环绕光（Wrap Light）

EN = EL \* max((cos<l,n>+Wrap)/(1+Warp),0)

Wrap描述WL的大小

WL为整个半球时为1

WL为0（即平行光/点光源）时为0

#### 光源——预积分（Pre-Integrated）漫反射剖面图（Diffusion Profile）

//[2. d'Eon 2007]

表示比例

对于高度散射的透明材质而言，假定光线接触表面后在各个方向散射都相同

漫反射剖面图在各个方向都相同，可以简化为只与距离有关

EN 不仅来自光源ELcos<l,n> 还来自表面其它位置的散射

DiffusionProfile作为权重对EN进行融合 原理上与卷积有关

根据

根据曲率和<L,N> 预计算积分DiffuseLightLookUpTexture

计算积分时，用中心点的曲率近似表示周围点的曲率，应当假定表面光滑，即不应当应用法线贴图

EN = EL \* DiffuseLightLookUpTexture (曲率,<L,N>)

计算积分

GFSDK\_FaceWorks\_CalculateCurvatureLUTSizeBytes

GFSDK\_FaceWorks\_GenerateCurvatureLUT

计算曲率

GFSDK\_FaceWorks\_CalculateCurvatureSizeBytes

GFSDK\_FaceWorks\_CalculateMeshCurvature

计算 DiffuseLightLookUpTexture (曲率,<L,N>)

GFSDK\_FaceWorks\_EvaluateSSSDirectLight

#### 材质——法线贴图

法线分布函数映射/镜面高光阈值

模拟光滑表面

内散射 相邻位置散射到当前计算的位置（隐含Map模式）

法线模糊

BSSRDF？？？

以具有空间上的模糊效果漫反射模拟多次散射

镜面反射---->应用法线贴图扰动

漫反射--->不应用法线贴图（相当于应用法线贴图扰动后再模糊） （并不会对BRDF\_Lambert产生影响，只是影响通过El计算En的过程）

#### 阴影映射（Shadow Map）

查询深度图进行光线跟踪估计光线在物体内传输的距离，从而根据贝尔朗伯定律计算吸收

计算时假定深度图和计算光照的位置之间的空间全在物体内部，但是可能包含其它物体

## 材质/明暗模型（Shading Model）

NVIDIA MDL SDK

<https://developer.nvidia.com/mdl-sdk>

Andreas Mank. "MDL Materials to GLSL Shaders: Theory and Practice." GTC 2016.

<https://on-demand-gtc.gputechconf.com/gtcnew/sessionview.php?sessionName=sig1631-mdl+materials+to+glsl+shaders%3a+theory+and+practice>

Google Filament

<https://google.github.io/filament/>

Intel Embree

<https://embree.github.io/related.html> / Embree Related Papers

在目前的主流游戏引擎中

明暗模型SSS（SubSurface Scattering 子表面散射）往往特指基于NVIDIA FaceWorks中的相关技术实现的全局子表面散射

局部子表面散射往往被认为是明暗模型PBR（Physically Based Rendering 基于物理渲染）中的漫反射部分

大气散射在物理意义上属于全局子表面散射，但是大气散射基于体积渲染实现，与NVIDIA FaceWorks中的相关技术没有任何关系，往往被认为与明暗模型SSS无关

为了避免读者日后在使用目前的主流游戏引擎时产生不必要的困惑，本书的目录按照目前的主流游戏引擎中的相关概念进行组织，尽管这些概念存在着明显的错误

### PBR/微平面+局部子表面散射

Saurabh Bhatia. " glTF 2.0: PBR Materials." GTC 2017

<https://www.khronos.org/assets/uploads/developers/library/2017-gtc/glTF-2.0-and-PBR-GTC_May17.pdf>

#### Specular-Glossiness

Albedo

Specular

Glossiness

ColorSpecular = Specular

Roughness = 1 – Glossiness

Diffuse

ColorDiffuse = Albedo\*(1 - Fresnel\_ Schlick(ColorSpecular)) //ColorSpecular即Fresnel(0)

BRDF(l,v) = ColorDiffuse/π //Lambert

Specular

BRDF(l,v)= … //GGX

#### Metallic-Roughness

BaseColor

Metallic //用于保证能量守恒 即ColorSpecular + ColorDiffuse == BaseColor

Roughness

//BaseColor+Metallic –>Albedo+ColorSpecular

//非金属的ColorSpecular（显然为经验值）

//确保接下来的计算中ColorSpecular不会低于该值

ColorSpecularForDielectric = float(0.04f,0.04f,0.04f)

ColorSpecular = lerp(ColorSpecularForDielectric, BaseColor, Metallic)

Albedo = BaseColor - ColorSpecular

//接下来计算同上

#### 介于微观和宏观之间，不属于微平面理论范畴

本书归类到了几何体中

//NormalMap

法线Normal/凹凸Bump贴图

在切线空间中定义

可以用SNORM格式，不再需要[0 1]和[-1 1]之间的转换

在延迟渲染中，一般将法线从切线空间变换到世界空间

//Height Map （TargetSpace内，不仅适用于地形）

高度场HeightField/置换Displacement贴图

一般在Mesh中并不使用

一般在细分Tessellation中采样，用于表示地形或海洋

//Occlusion Map （SSAO的离线版本） //是否包含PRT？？？ //AO理论上不应当影响Specular

//Emissive MAP //自发光？？？ //一般忽略

### Wet 湿润

Sébastien Lagarde. "Water drop 3b – Physically based wet surfaces." WorldPress 2013.

<https://seblagarde.wordpress.com/2013/04/14/water-drop-3b-physically-based-wet-surfaces>

// WetLevel和AccumulatedWater在实际中未必真正需要

//Dry – WetLevel-> Thin Layer Of Water

Thin Layer Of Water

Darken Diffuse + Boost Specular

//Albedo-Specular-Glossiness To BaseColor-Metaliic-Roughness

Roughness = 1.0 –Glossiness

//个人倾向

//Metalic = smoothstep (0.4, Albedo+Specular, Specular) //ColorSpecularForDielectric:=0.4f BaseColor:=Albedo+Specular

//Blog原文

// Metalic = saturate((dot(Specular, 0.33) \* 1000 - 500) )

//根据Matalic和Roughness猜测Porosity //一般认为非金属(Dielectric)更倾向于透气(Porosity)

//Start和End的取值取决于引擎的Roughness/Glossiness范围 //Start: 0.5f 足够光滑，雨水不能渗入，临界点 //End: 0.9f 足够粗糙，被认为完全渗透，临界点

float porosity = (1.0f – Metalic)\*smoothstep(0.5f, 0.9f, Roughness) /\* \*porositymask（可以存储在Specular的Alpha通道）\*/

float factor = lerp(1.0f, 0.2f, prosity); //1.0f无衰减 0.2f完全衰减（基于STAF Database的范围选取）

//Darken Diffuse

ColorDiffuse \*= lerp(1.0f, factor, WetLevel); //WetLevel允许在Dry和Wet之间平滑过渡

//Boost Specular

Roughness \*= lerp(1.0f, factor, WetLevel\*0.5f);

//Glossiness = lerp(1.0f, Glossiness, lerp(1.0f, factor, WetLevel\*0.5f)); //

//Thin Layer Of Water –AccumulatedWater-> Margin Of Puddle -AccumulatedWater-> Puddle

Puddle

Specular(Glossiness/Normal) Of Water + Darken Diffuse Of Surface

Specular = lerp(Specular,(0.02f,0.02f,0.02f), AccumulatedWater) //水潭边缘区域的AccumulatedWater介于0到1之间 //水的Specular:=(0.02f,0.02f,0.02f)

Glossiness = lerp(Glossiness, 1.0f, AccumulatedWater) //谁的Glossiness=1.0

Normal = lerp(Normal, (0.0f,0.0f,1.0f)) //假定(0.0f,0.0f,1.0f)竖直向上

Ripple（涟漪）

粒子系统？

### SSS/微平面+全局子表面散射

NPR

<https://github.com/unity3d-jp/UnityChanToonShaderVer2_Project>

## 面光源（Area Light）

离线渲染中 向量辐射亮度（Vector Irradiance）

照明——LTC(Linearly Transformed Cosines) 次世代//Real-Time Polygonal-Light Shading with Linearly Transformed Cosines

阴影——Soft Shadow

# 直接光/阴影

## 阴影映射（Shadow Mapping）

本质上也是一种RayCast

NormalizedDevice -> UV

Direct3D Viewport ->XXXX || UV (0,0) -> LeftTop

Vulkan Viewport ->glClipControl || UV (0,0) -> LeftBottom

平行光

限制光源的视锥体恰好包含传感器的视锥体，提高ShadowMap的分辨率

NearZ接近于0

FarZ取摄像机视锥体在光源的ViewSpace中的最远距离

可以考虑用视锥体剔除后得到的BV合并产生的AABB重新确定NearZ

聚光灯/点光源

视锥体并不一定是正四棱锥，可以是斜四棱锥（在VR中，摄像机的视锥体也是斜视锥体）

ShadowMap的视锥体的Z轴方向应当和摄像机的视锥体的Z轴方向平行，并不是聚光灯的聚焦方向

但是以上方式并不能解决问题，几何体并不总是垂直于Z轴

当几何体平行于Z轴时，ShadowMap的视锥体的Z轴方向应当和摄像机的视锥体的Z轴方向垂直才是最佳的

聚光灯ShadowMap的斜视锥体的Z轴与摄像机的视锥体的Z轴方向垂直，避免视锥体重叠

泛光灯

立方体映射 //双抛物面映射

A limitation of shadow map methods using a single projection is that

the light is assumed to view the scene from somewhere outside of it, something

like a spotlight. Positional lights inside a scene can be represented

by any structure that captures the parts of the scene needed. The usual

approach is to use a six-view cube, similar to cubic environment mapping.

These are called omnidirectional shadow maps. The main challenge for

omnidirectional maps is avoiding artifacts along the seams where two separate

maps meet. King and Newhall [659] analyze the problems in depth

and provide solutions, and Gerasimov [390] provides some implementation

details. Forsyth [355, 357] presents a general multi-frustum partitioning

scheme for omnidirectional lights that also provides more shadow map resolution

where needed.

Using omni shadows with dual paraboloid mapping creates fewer seams,

but normally does not work well, due to non-linear interpolation problems

[1174], i.e., straight edges become curves. Osman et al. [976] attack

the problem by tessellating objects based on distance from the light.

自阴影走样

误差

Z方向的精度

一、浮点运算精确度

深度反转

ZNear和ZFar之间的距离

### 透视混叠（Perspective Aliasing）

仅会在透视投影的情况下发生，正交投影不会发生

为了简化讨论，假设光源为平行光（正交投影）

采样频率过高???

比如球在地面上的阴影，边缘的曲线可能出现阶梯状伪影（Stairstepping Artifacts）

采样频率过低???

小物体（比如电话线）不产生阴影

在近处，传感器图像中的多个像素对应于ShadowMap中的同一个像素（即采样频率过高）

在远处，传感器图像中相邻像素在ShadowMap中对于的像素存在着较大的间隔（即采样频率过低）

无论如何调节ShadowMap的分辨率都无法使近处和远处都具有合适的采样频率

解决：

平行光：

级联阴影映射

理想状态下，阴影映射中的单位纹素所对应的世界空间大小随着传感器的距离连续地线性降低，但目前的硬件无法实现

级联阴影映射可以实现离散地降低

Nvidia的VRWorks中的Lens-Matched Rendering提供了类似的思想

聚光灯/点光源：

ShadowMap的视锥体的Z轴方向应当和摄像机的视锥体的Z轴方向平行，并不是聚光灯的聚焦方向

### 投影混叠（Projective Aliasing）

针对光源的投影矩阵而言，在透视投影或正交投影的情况下都会发生

当表面法线和光源方向垂直时，En=El\*cosln为0，传感器图像中的多个像素对应于ShadowMap中的同一个像素（即采样频率过高）

阴影膨胀？？？

解决：深度斜率偏移

### 自阴影（Self Shadowing）/表面痤疮（Surface Acne）

精度问题（解决：深度常量偏移）或投影混叠

### 彼得潘化（Peter Panning）

深度偏移过大导致物体与阴影分离

### 阴影边缘闪烁（Shadow Edge Shimmering）

抖动（Jitter）

Flickering of shadow quality

Swimming

主要是光源摄像机的信息不断变化导致，阴影缓存时忘记将LastPosition更新为CurrentPosition导致刷新频率过高时，就会出现这种情形

//传感器图像的采样率阴影映射图像，部分像素被跳过

仅在平行光阴影中存在

为提高阴影贴图的有效分辨率，将阴影贴图聚焦在视锥体的包围盒上

由于视锥体包围盒的不稳定性，导致闪烁

1.以纹素大小的增量移动光源（Moving the Light in Texel-Sized Increments）

http://msdn.microsoft.com/en-us/library/ee416324

The vWorldUnitsPerTexel value is calculated by taking a bound of the view frustum, and dividing by the buffer size.

//在WorldSpace或Light的ViewSpace中计算Max-Min是等效的

//因为ViewTransform并不改变物体的形状

float fWorldUnitPerTexel = (Max-Min)/TextureSize;

//仅修改MinX（Left）MinY（Bottom）MaxX（Right）MaxY（Top） 并不修改NearZ和FarZ

Max = floor（Max /= fWorldUnitPerTexel）\* fWorldUnitPerTexel //Max是否可以考虑用Ceil???

Min = floor（Min /= fWorldUnitPerTexel）\* fWorldUnitPerTexel

2.量子化缩放和偏移

Flickering of shadow quality/Swimming

Fan Zhang,Alexander Zaprjagaev,Allan Bentham. Practical Cascaded Shadow Maps. ShaderX7.

Approximate Solution

## 深度偏移（Depth Bias）

二、ShadowMap和摄像机得到的图像中的像素在三维空间中所对应的位置并非恰好完全一致



上图改为两个斜率不同的直线更直观

A B是摄像机得到的图像中的像素

D E是ShadowMap中的像素

A 正常阴影 A较D离光源近

B 自阴影走样 B较D离光源远

显然将ShadowMap中的值进行一定程度的偏移可以避免自阴影走样

以上图为例，最理想的效果是将E偏移至F，D偏移至G

常数偏移也是需要的，因为表面切线垂直于光源的位置理论上也是需要进行偏移的

可以用Vulkan中的深度偏移（Depth Bias）近似地实现该效果

Vulkan规范（ http://www.khronos.org/registry/vulkan/specs/1.0-extensions/pdf/vkspec.pdf）中的/Rasterization/Polygons/Depth Bias章节

深度斜率指三角形图元在XOZ或YOZ上的投影所在的直线的斜率

VkPipelineRasterizationStateCreateInfo结构体中的depthBiasSlopeFactor成员

在深度反转的情况下，深度偏移斜率因子应当设置为负值

聚光灯 透视投影 ShadowMap 越远精度越低，

## 地形之类的非封闭体不允许剔除正面

自适应阴影映射 层次性

## 延迟阴影（Deferred Shadowing）

缓存友好（Cache-Friendly）

阴影预处理（Shadow Prepass）/延迟阴影（Deferred Shadowing）/Z-Prepass

PSSM涉及到较多的ShadowMap，增加一个预处理（Prepass）过程仅进行ShadowMap的采样比较，并将结果（称为ShadowFactor，在柔和阴影中可以是0到1之间的某个值）保存到某个纹理中

在逻辑上ShadowFactor与场景中的光源相对应

Gbuffer

Prepass

Light

PostProcess

柔和阴影在物理上对应于半影

显然平行光并不会产生半影

只有点光源/聚光灯才有可能产生半影

如果只是为了抗锯齿（空间反走样），那么从物理正确的角度不应当使用PCF而是应当在ShadowPrepass之后进行一次FXAA

## 级联阴影映射（Cascaded Shadow Mapping）

MSDN http://msdn.microsoft.com/en-us/library/ee416307

又被称为 平行分割阴影映射（Parallel Split Shadow Mapping）

最早提出在1999年

Tadamura, Katsumi, Xueying Qin, Guofang Jiao, and Eihachiro Nakamae. “Rendering Optimal Solar Shadows Using Plural Sunlight Depth Buffers”. Computer Graphics International 1999, pp. 166, 1999. Cited on p. 359

仅适用于平行光(Directional Light)

### 视锥体分区（Partitioning the Frustum）

适用于透视传感器

透视传感器满足 传感器图像中的像素在NearZ处的子视锥体对应的三维空间较小，在FarZ处的子视锥体对应的三维空间较大

当不同的子视锥体使用相同的阴影映射时，Z较远的子视锥体应当较大

#### 场景朝向

当场景在传感器的ViewSpace中具有高动态范围的Z时，CSM的作用明显

在极端情况下，如果整个场景都在某个子视锥体中，那么其它子视锥体的阴影映射实际上并没有被用到，CSM的作用不明显

一般情况下，视线平视（平行于地面）时，场景在传感器的ViewSpace中的Z的范围较高；视线俯视（垂直地面向下）时，场景在传感器的ViewSpace中的Z的范围较低

#### 光源朝向/视锥体竞争（Dueling Frusta）

Light的Postion和Rotation（决定EyeDirection和UpDirection）由用户设定，从而计算出LightViewMatrix

LightProjectionMatrix由OrthographicOffCenter算出

求出各个子视锥体在光源的ViewSpace的AABB，从而确定OrthographicOffCenter的Left，Right，Top，Bottom，Far

OrthographicOffCenter的Near可以有两种方法确定

1．根据场景AABB，ZNear为0和场景AABB的最大值 （有可能场景AABB超出范围）

2．根据子视锥体AABB，ZNear为0

Left，Right，Top，Bottom，Far不建议修改，否则GBuffer投影到Light空间中的坐标可能超出范围 （比如超过FarZ为负值（反转情况下），但是写入后为0，导致比较错误）

实际测试中，尤其是Left，Right，Top，Bottom的值**较大**时，由于浮点数运算的误差（IEEE Float 在0处精度最大 正负无穷大处精度最小 这也是深度反转能提高精度的原因），可能导致AABB过小，边界处的像素在投影后超出AABB的范围

为了避免超出边界，甚至可以按比例放大

或者使用包围球（大约相当于每个边界扩大了(根号2-1)/2

参考#include<DirectXCollision.h> BoundingSphere::CreateFromPoints

这与以离散级联选择的方式也有一定关系

不同的级联可以使用不同的深度常数偏移（和深度斜率偏移？？？）

//ZNear可以为负值？？？

投影使用Z反转是有利的，当深度缓冲为浮点类型时，可以使Far平面处（即靠近子视锥体的空间）有较高的精确度

在确定了各个分区的间隔后，有两种方式构造子视锥体

适配场景（Fit To Scene）

子视锥体只使用分区间隔的FarZ，NearZ与传感器视锥体相同

适配级联（Fit To Cascade）

子视锥体使用分区间隔的NearZ和FarZ

子视锥体在光源的ViewSpace的AABB之间存在着重叠

适配级联

随着光源方向和传感器的视线方向的夹角变小，重叠不断变大，当光源方向和传感器的视线方向相同时，退化为适配场景

适配场景

传感器视锥体的NearZ到分区间隔的NearZ之间的空间被浪费，但在光源方向和传感器的视线方向的夹角变化时，重叠较稳定

### 几何着色器多播

NVIDIA http://nvidiagameworks.github.io/GraphicsSamples/CascadedShadowMapping.htm

//Draw Call合并

//几何着色器并不是像素级的

//但可以用上述方式对三角形进行测试？？？

//gl\_Layer

//SV\_RenderTargetArrayIndex

X和Y方向的精确度

不同级联的深度常数偏移和深度斜率偏移不同，可以在几何着色器中（而非光栅化阶段）进行深度偏移方便进行实时调整

比较快的方式是使用Draw（Indexed）Instanced，在几何着色器中根据SV\_InstanceID写入SV\_RenderTargetArrayIndex

### 级联选择（Cascade Selection）

基于间隔（Interval Based）

使用向量指令进行比较从而避免了分支的使用

float3 fComparison = vCurrentPixelDepth.xxx > g\_fCascadeFrustumsEyeSpaceDepths.xyz;//映射

float fIndex = dot(

float3(

CASCADE\_COUNT\_FLAG >= 1,

CASCADE\_COUNT\_FLAG > =2,

CASCADE\_COUNT\_FLAG > =3)

, fComparison

);//归约 一般认为效率高于fComparison.x+ fComparison.y+ fComparison.z

fIndex = min( fIndex, CASCADE\_COUNT\_FLAG );

iCurrentCascadeIndex = (int)fIndex;

//GLSL

bvec3 vGreaterThan =greaterThan(gl\_FragCoord.zzz,NormalizedFarPlanes.xyz);//映射

float fIndex = dot(vec3(vGreaterThan.xyz),vec3(1.0f,1.0f,1.0f));//归约

int iIndex = int(fIndex);

基于纹理坐标



连续的方式，较上述离散方式效果更好

不断迭代，乘以VPT（T即TextureUV）矩阵，选择精度最高的满足UV在0-1内且Z近于ZFar的分区

由于分区间存在重叠，这种方式保证了使用精度较高的分区

比远平面更远时Farther Than KG3D\_FarZ，但是ShadowMap中被Clamp到 KG3D\_FarZ

根据NVIDIA

设置SV\_ViewportArrayIndex，LightProjectionMatrix可以共用同一个

Viewport仅处理x和y，不处理z

By using texture atlases or arrays, the different shadow maps can be treated as one large texture, thus minimizing texture cache access delays [312].

## 柔和阴影（Soft Shadow）

物理上由面光源（Area Light）产生

BRDF对面光源应当是不适用的，但是实际中仍使用BRDF

显然面光源不应当是平行光，但是实际中平行光仍可以使用PCSS

平行光并不会产生半影，如果只是为了空间反走样，那么应当用FXAA而非PCSS

从人类感觉（视网膜上得到的是二维图像）上，硬阴影（Hard Shadow）可能与实际几何体混淆，比如被误认为是表面上的褶皱（En = El \* cos l n 中的cos l n接近于0）

As can be seen in Figure 9.4, a correct soft shadow is sharper the closer the shadow casting geometry is

to the receiver.

PCF（Percentage Closer Filtering，百分比接近过滤）

类似于图像处理中的放大 能在某种程度上能解决阶梯状伪影（见上文CSM）的问题

可以使用GLSL中的TexelGather加速

PCSS（Percentage Closer Soft Shadows，百分比近似柔和阴影）

[Fernando 2005] Randima Fernando. “Percentage-Closer Soft Shadows”. SIGGRAPH 2005

http://download.nvidia.com/developer/presentations/2005/SIGGRAPH/Percentage\_Closer\_Soft\_Shadows.pdf

Blocker Search

### PCF（Percentage Closer Filter，百分比靠近过滤）

#### 自阴影（Self Shadowing）

偏导数深度偏移

Calculating a Per-Texel Depth Bias with DDX and DDY for Large PCFs

MSDN http://msdn.microsoft.com/en-us/library/ee416307

中心周围的采样点所对应的几何体在光源的裁剪坐标中的深度与中心并不一定相等

锥采样

用ddx,ddy求出传感器图像中的像素在深度贴图中的纹理坐标偏导数

由于

[ GBufferOffsetU GbufferOffsetV ]

\*

[ ddx(ShadowMapU) ddx(ShadowMapV) ]

[ ddy(ShadowMapU) ddy(ShadowMapV) ]

=

[ ShadowMapOffsetU ShadowMapOffsetV]

求出

[ ddx(LightU) ddx(LightV) ]

[ ddy(LightU) ddy(LightV) ]

的逆矩阵

即可根据中心周围采样点的UV偏移[ShadowMapOffsetU ShadowMapOffsetV]

求出[GBufferOffsetU GBufferOffsetV]

根据

[ GBufferOffsetU GbufferOffsetV ] \*[ ddx(ShadowMapDepth) ] = [ ShadowMapOffsetDepth ]

[ ddy(ShadowMapDepth) ]

即可根据求得的[ SensorOffsetX SensorOffsetY ]求出[ ShadowMapOffsetDepth ]

即中心周围的深度偏移

### PCSS（Percentage Closer Soft Shadows，百分比接近柔和阴影）

### CHS（Contact Hardening Shadows，接触硬化阴影）

Holger Gruen, Jon Story. "Taking Advantage of Direct3D 10.1 to Accelerate Performance and Enhance Quality." EuroGraphics 2009.

<https://developer.amd.com/wordpress/media/2012/10/TakingAdvantageofDirect3D10.pps>

Holger Gruen. "Fast Conventional Shadow Filtering." Chapter VII-1 GPU Pro 2010.

DirectX SDK (June 2010) Samples / ContactHardeningShadows11

NVIDIA GameWorks DirectX Samples / D3D Soft Shadows Sample

<https://developer.nvidia.com/gameworks-directx-samples>

AMD GPUOpen / ShadowFX

<https://gpuopen.com/gaming-product/shadowfx/>

### VSM（Variance Shadow Map，方差阴影映射）

MSDN

<http://msdn.microsoft.com/en-us/library/ee416307>

NVIDIA Direct3D SDK 10.5 Code Samples / Variance Shadow Mapping

<https://developer.download.nvidia.com/SDK/10.5/direct3d/samples.html#VarianceShadowMapping>

## Frustum-Traced Shadows（锥跟踪阴影）

Chris Wyman, Rama Hoetzlein, Aaron Lefohn. "Frustum-Traced Raster Shadows: Revisiting Irregular Z-Buffers." I3D 2015.

<https://research.nvidia.com/publication/frustum-traced-raster-shadows-revisiting-irregular-z-buffers>

Jon Story. "Hybrid Ray Traced Shadows". GDC 2015.

<https://developer.nvidia.com/content/hybrid-ray-traced-shadows>

<http://developer.download.nvidia.com/assets/events/GDC15/hybrid_ray_traced_GDC_2015.pdf>

Jon Story. "Advanced Geometrically Correct Shadows for Modern Game Engines". GDC 2016.

<https://developer.nvidia.com/hybrid-frustum-traced-shadows-0>

<http://developer.download.nvidia.com/gameworks/events/GDC2016/jstory_hfts.pdf>

NVIDIA ShadowWorks

<https://developer.nvidia.com/shadowworks>

Real Time Rendering //

"With a tile/edge intersection test in place, it is possible to hierarchically traverse a triangle."

"Both OCR and UCR can be implemented using tiled traversal by shrinking the tile size to one pixel."

OCR(Overestimated Conservative Rasterization) //Outer-Conservative Rasterization //API中的CR

UCR(Underestimated Conservative Rasterization) //Inner-Conservative Rasterization

Implement 32 MSAA By Software —— "Since samples can lie anywhere within a texel, conservative rasterization is required"

To Solve 透视混叠 //One Pixel In Eye-Space Match Large Volume In View Space In the Distance

//To Do List:

//Q: How Shoud We Do If One Pixel In Eye-Space Match More Than One Pixel In Light-Space?

//A: Calculate A Threshould Distance?

# Depth PrePass

Tile架构不应当进行Depth PrePass

FPK(Forward Pixel Kill)

"Some sort-middle architectures perform a z-prepass per tile for opaque geometry ..." Real-Time Rendering Fourth Edition

"Avoid depth prepasses" Mali GPU Best Practices / Application Logic / Avoid Depth Prepasses

<https://developer.arm.com/graphics/developer-guides/mali-gpu-best-practices>

# 延迟着色

//Vulkan与Direct3D12的UV的V坐标相反 //加载的贴图资产也会相反，Mesh中的UV相同，最终着色器代码相同

//Vulkan与Direct3D12 关于Y的ViewportTransform不同 FrameBuffer坐标Y相反

## G-buffer

VK\_FORMAT\_D32\_SFLOAT

关闭深度剪辑

depthClampEnable为VK\_TRUE 允许z/w不在[0,1]内

//Normal In WorldSpace

//GL\_RGB8\_SNORM 有符号 //错误！GL\_RGB8

//OpenGL ES 不支持GL\_RGB8\_SNORM //非Color Renderable

//只能使用GL\_RGB8 //应当Encode:(N+1.0f)\*0.5f且Decode:N\*2.0f-1.0f

右手系

投影矩阵 对角阵 1/Width 1/Height -1 1

顶点着色器中 x/=z y/=z 即可

深度缓冲即摄像机坐标 D3D12中 纹理格式D32\_FLOAT 视图格式R32\_FLOAT

像素着色器

SV\_RenderTargetArrayIndex

SV\_Target[n]

片元着色器

gl\_Layer

//以下变量显式声明

layout(location=0) out vec4 FragColor0;

layout(location=1) out vec4 FragColor1;

...

layout(location=7) out vec4 FragColor7;

反走样

光栅化阶段输出Coverage值

Direct3D12:

uint SV\_Coverage

Vulkan(GL\_KHR\_vulkan\_glsl):

in int gl\_SampleMaskIn[ ]

光栅化阶段得到的Coverage值与用作渲染目标的纹理是否启用多重采样无关

Direct3D12：

D3D12\_RASTERIZER\_DESC/MultisampleEnable

D3D12\_GRAPHICS\_PIPELINE\_STATE\_DESC/SampleDesc SampleMask

Vulkan:

VkPipelineMultisampleStateCreateInfo

如果光栅化输出的该值不为全1，则为边缘

线程相干性

设计并行程序时，各着色器的执行速度应当尽可能相同

用模板测试将普通像素和复合像素的渲染分成2个不同的Pass（也可以用深度测试）

DxCaps

调试/图形/启动图形调试

工具/选项 图形诊断/捕获 取消勾选 在兼容模式下捕获

HLSL调试器

必须启用调试信息

1.fxc.exe HLSL编译器/启用调试信息

2.调用D3Dcompile D3DCOMPILE\_DEBUG| D3DCOMPILE\_SKIP\_OPTIMIZATION

3.Unity3D #pragma enable\_d3d11\_debug\_symbols（非HLSL标准语法，Unity3D特有，见Unity3D文档）

不要使用SV\_VertexID和SV\_InstanceID 否则 像素历史记录会无法使用

GL\_KHR\_vulkan\_glsl

http://www.khronos.org/registry/vulkan/specs/misc/GL\_KHR\_vulkan\_glsl.txt

//计算着色器 硬件标准

A compute shader on Direct3D 11 is also known as DirectCompute 5.0.

When you use DirectCompute with cs\_5\_0 profiles, keep the following items in mind:

The maximum number of threads is limited to D3D11\_CS\_THREAD\_GROUP\_MAX\_THREADS\_PER\_GROUP (1024) per group.

The X and Y dimension of numthreads is limited to D3D11\_CS\_THREAD\_GROUP\_MAX\_X (1024) and D3D11\_CS\_THREAD\_GROUP\_MAX\_Y (1024).

The Z dimension of numthreads is limited to D3D11\_CS\_THREAD\_GROUP\_MAX\_Z (64).

The maximum dimension of dispatch is limited to D3D11\_CS\_DISPATCH\_MAX\_THREAD\_GROUPS\_PER\_DIMENSION (65535).

The maximum number of unordered-access views that can be bound to a shader is D3D11\_PS\_CS\_UAV\_REGISTER\_COUNT (8).

Supports RWStructuredBuffers, RWByteAddressBuffers, and typed unordered-access views (RWTexture1D, RWTexture2D, RWTexture3D, and so on).

Atomic instructions are available.

Double-precision support might be available. For information about how to determine whether double-precision is available, see D3D11\_FEATURE\_DOUBLES.

平行光 正交投影

泛光灯 透视投影 六面

聚光灯 透视投影

视锥提剔除

遮挡剔除

BRDF

经验 Lambert

几何光学 BRDF

//

## L-Buffer——光照预过程（Light PrePass）——传统的

又被称为 延迟光照（Deferred Lighting）/预光照（Pre-Lighting）

设BRDF能使Lv写成如下形式

单个光源：Lv = ColorDiffuse\*f1(EL,V,Normal) + ColorSpecular\*f2(EL,L,V,Normal,Roughness)

所有光源叠加：∑Lv = ColorDiffuse\*∑f1(EL,V,Normal) + ColorSpecular\*∑f2(EL,L,V,Normal,Roughness)

现代的图形引擎中BRDF\_Specular一般采用GGX，ColorSpecular并不能分离

//一说为兼容不支持MRT的硬件

//基本被淘汰

基本思路是将Deferred Shading的第二阶段分成多个步骤

1.对每个几何体，绘制到MRT中，并产生Full Screen的ColorDiffuse、ColorSpecular、Normal和Roughness——即G-Buffer

2.对每个光源，累加在上文中的f1和f2，并产生Full Screen的∑f1和∑f2——即L-Buffer //每个光源只访问G-Buffer中的Normal和Roughness，从而节省带宽

3.绘制一个Full Screen Triangle，读取L-Buffer中的∑f1和∑f2、G-Buffer中的ColorDiffuse和ColorSpecular，计算ColorDiffuse\*∑f1+ColorSpecular\*∑f2——即∑Lv

//在现代的Defered Shading中，可以在一个Pass同时计算所有光源，G-Buffer只需要读取一次，L-Buffer被弃用

//第2、3步据说能提高纹理缓存的命中率

//在Shadow PrePass中有类似的说法

//但根据HBAO+中对纹理缓存的相关描述，Light PrePass并没有能够提高纹理缓存的命中率的理由；相关的硬件可能年代久远，与现代的硬件存在着较大的差异

## 材质系统

明暗模型的个数一般是极其有限的：

1.Local SubSurface Scattering

2.Global SubSurface Scattering

3.透明？？？

虽然材质的个数可能有无数个，但是相同明暗模型的不同材质产生的GBuffer相同，计算光照的着色器代码也是相同的

一般使用一个纹理存放明暗模型ID，在计算光照时，使用Switch语句根据明暗模型ID选择相应的着色器代码计算光照

虽然存在分歧分支（Branch Divergence）的可能性，但是相邻区域的明暗模型ID相同的概率较大，并且明暗模型ID的上限一般不超过5个，并不会对性能产生致命的影响

## MSAA

NVIDIA GameWorks DirectX Samples / Antialiased Deferred Rendering

<https://developer.nvidia.com/gameworks-directx-samples>

//实用性分析

在移动平台的Tile架构中，MSAA的开销可能低于FXAA //见后文

//插值修饰符是sample

在计算光照时，先根据各个Sample分别计算光照，再Blend即可（4X MSAA 即相加后除以4）

texelFetch的坐标可以根据ivec2(gl\_FragCoord.xy)得到

//个人认为这种简单的方式的实用性较高，能同时兼容sample和centroid，并且不存在分歧分支（见下文中的讨论），在性能上与下文中的方法差异并不明显

//考虑用计算着色器加速？？？ Z分量为4 计算4X MSAA？？？

原文中的一些技术要点

//针对插值修饰符是centroid时的优化 //即同一几何体中不同采样点在GBuffer中的值是相同的

//以4X MSAA为例

1.Complex Pixel Detection

生成的GBuffer时，将SV\_Coverage写入到Buffer中

SV\_Coverage表示最后一次写入的像素的覆盖情况

SV\_Coverage不为1111一般为边缘，但是SV\_Coverage中为0的二进制位并不代表没有GBuffer数据，相关数据可能来自之前写入的几何体

//SV\_Coverage用于Adaptive Shading

//2根据连续性 对4个采样点之间的差异进行分析 这种方式不需要SV\_Coverage Buffer的存在

像素被分为NormalPixel和Complex Pixel

NormalPixel //SV\_Coverage为1111（4X MSAA） 一般为非边缘

根据任意一个采样点进行计算

Complex Pixel//SV\_Coverage不为1111（4X MSAA） 一般为边缘

理想情况是，根据任意一个被覆盖的采样点进行计算（因为插值修饰符时centriod），再乘以根据SV\_Coverage得到的系数即可

但是，并不能确定哪个采样点被覆盖，因此对每个采样点都进行计算，再Blend（即相加后除以4）

2. Adaptive Shading of Complex Pixels

由于存放SV\_Coverge的Buffer表示最后一次写入的信息，显然SV\_Coverge中为1的二进制位在GBuffer中对应的数据时相同的，在计算光照时只需进行一次

3.Seperate Complex Pixel Pass

！！！在实际测试中，开启Seperate Complex Pixel Pass会使帧率下降

//原始Sample中甚至未开启[earlydepthstenil]

原文中的说法是 同一Warp中存在分支时的Thread Coherency会受到影响 //实际上即分歧分支（Divergence Brunch）

使用Stencil分成两个Pass

//来自实践的质疑

在实践中，Deferred Shading普遍采样ID Buffer区分不同的ShadingModel

分歧分支的含义是

在SIMT架构中

在同一Wrap中的像素（一般认为 屏幕被分成若干个16×16的Tile 同一Tile中的像素在同一个Wrap中）共用同一个指令寄存器

从而导致不同分支的语句无法并行执行，必须在指令寄存器中分别存放各个分支的指令，分别执行一次；消耗时间相当于串行执行各个分支的语句的和

由于不同Wrap中的像素可以并发执行不同的分支

Complex Pixel又仅限于屏幕上极少的区域（一般即几何体边缘）

Single Pass的GPU执行时间一般会小于Seperate Complex Pixel Pass（即串行执行各个Pass的时间），只是在最坏情况下会与Seperate Complex Pixel Pass相同

并且Seperate Pass还会引入额外的开销（比如读写Stencil Buffer），导致性能反而低于Single Pass的情形

# 基于区块（Tile-Based）

## 前向+（Forward Plus）

AMD GPUOpen ForwardPlus11

<http://gpuopen.com/gaming-product/forwardplus11-directx-11-sdk-sample>

由于GPU计算的速度远大于GPU与显存通信的速度

延迟渲染中GPU写入和读取Gbuffer的开销较大

传统的前向

绘制平行光得到DepthMap

以只比较不写入的方式进行深度测试（由于浮点误差可以考虑加一个偏移值）

开启像素着色器的[earlydepthstencil]，避免计算会被覆盖的像素的光照

在像素着色器中，遍历所有点光源/聚光灯并计算光照 问题：GPU计算的光源个数过多

次世代的前向+

1.Depth Prepass->得到Depth Map

//平行光没有位置，并不会受益于接下来的Tiled Lighting Culling

//但是从带宽的角度，不写入颜色缓冲可能会有性能提升

即用传统的前向渲染计算场景中唯一的平行光的光照，得到传统的Depth Map

2.Tiled Light Culling -> 生成Per-Tile Light List

对每个Tile使用并行归约求出最小和最大Depth并构造新的视锥体

并与点光源的包围体进行相交测试（算法细节->视锥体剔除）

在AMD的ForwardPlus11实现中

并没有用并行归约，而是用asuint转换成uint后用InterlockedMax/ InterlockedMin求出最值再用asfloat转换成float

但从理论上而言

原子操作（Interlocked）的本质是互斥，会降低并行性，可能在GPU中的实现存在差异

并且asuint或asfloat的转换效率是极低的

但是实验结果表明，原子操作更快

在AMD的ForwardPlus11实现中

可以用USE\_DEPTH\_BOUNDS控制

忽略计算视锥体的Near和Far平面，而是只用Left、Right、Top、Bottom 4个平面进行测试

每个Tile得到一个索引数组（标识相交的点光源）

显然，暂时存储在组共享内存中，用InterlockedAdd维护索引数组的大小

不同Tile的光源个数可以不同

只要不同Tile在不同的WaveFront（AMD）/Warp（NVIDIA）中，即不会产生分歧分支（可以认为指令寄存器相互独立）//在Tile Forward中，for语句中的条件测试即分支

经验表明 Tile大小16x16

线程组的个数应当能被WaveFront整除

目前对AMD或NVIDIA的GPU而言，可以认为WaveFront大小为64

如果过大，那么会导致光源剔除时的粒度过大，无法做到有效的剔除光源

组共享内存->桶冲突（Bank Conflict）

在AMD GCN(Graphics Core Next)架构中，有32个桶，%32后相同的数组索引会引起桶冲突

经验表明 组共享内存 数组大小256较合适

提高缓存命中率

在光源剔除时，只存放Position和Radius，不存放Color

3.Forward Shading

与传统前向区别不大，只是在遍历点光源时，从每个Tile的索引数组中获取

根据SV\_Position确定像素在FrameBuffer中的坐标并确定Tile

一般用终止符（类似于字符串中的\0）表示索引个数

在AMD的ForwardPlus11实现中

定义终止符LIGHT\_INDEX\_BUFFER\_SENTINEL

在循环次数较高的情况下不建议unroll，因为Cache命中率会降低

可以考虑用3D纹理(TileX \* TileY \* 256 R16\_UINT )存放索引数组，但是需要测试会不会导致Cache命中率降低从而产生性能问题

在AMD的ForwardPlus11实现中

256对应于MaxNumLightsPerTile

格式R16\_UINT应该能保证NumActivePointLights（上限2048）和NumActiveSpotLights（上限2048）不溢出

一般情况下终止符LIGHT\_INDEX\_BUFFER\_SENTINEL被定义为最大的R16\_UINT值，即0xFFFF（HLSL编译器不支持将x大写，即0XFFFF会发生编译错误）

## 延迟

可用于夜景渲染，在平行光

在夜景中，可能不满足明视条件，即三刺激值理论不适用

AMD GPUOpen TileLighting11

<http://gpuopen.com/gaming-product/tiledlighting11-directx-11-sdk-sample>

Andrew Lauritzen. "Deferred Rendering for Current and Future Rendering Pipelines". SIGGRAPH 2010.

Gareth Thomas. "Advancements in Tiled-Based Compute Rendering". GDC 2015

惯用的延迟渲染

1.将光照输入存放在内存中（即G-buffer）

2.对每个光源

使用光栅化来scatter光源体并且剔除 //使用完全的光源网格（球或圆锥）往往是过度的，使用OBB（方向包围盒）

从G-buffer中读取光照输入 //光源重合时额外负担 GPU访问Global内存是最慢的

计算光照

使用Blend累加光照 //光源重合时额外负担 GPU访问Global内存是最慢的

重排序计算以提取相关性

可以将平行光、点光源、聚光灯存放在3个不同LightList中（即SRV Buffer），在一个Draw Call中完成计算

在AMD的TileLighting11实现中，使用地图集（Atlas）存放所有的阴影映射

个人认为可以使用细分着色器，从传感器的深度映射中重构三维场景再变换到各个点光源/聚光灯的ClipSpace中

在这个过程中，可以借助几何着色器多播

# 基于集群（Cluster-Based）

Ola Olsson, Markus Billeter, Ulf Assarsson. "Clustered Deferred and Forward Shading." High Performance Graphics 2012.

<http://www.cse.chalmers.se/~uffe/clustered_shading_preprint.pdf>

Intel Developer Zone / Forward Clustered Shading

<https://software.intel.com/en-us/articles/forward-clustered-shading>

Intel Developer Zone / Clustered Shading Android Sample

<https://software.intel.com/en-us/blogs/2014/07/30/clustered-shading-android-sample>

Light / Decal / Light Probe(The Image Based Lighting)

Ola Olsson, Erik Sintorn, Viktor Kampe, Markus Billeter, Ulf Assarsson. "Efficient Virtual Shadow Maps for Many Lights."

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

## 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[Ci Ai Zi]，该像素最终的颜色 //其中，又被称为可见性函数，即 。

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。



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，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(Zi)称作透射率T(Zi)，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CFinal：

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CFinal。

### 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CFinal

CFinal\_0 = BackgroundColor

CFinal\_n = (AnCn) + (1-An)CFinal\_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CFinal

CFinal\_0 = 0

ATotal\_0 = 1

CFinal\_n = ATotal\_n-1(AnCn) + CFinal\_n-1

ATotal\_n = ATotal\_n-1(1-An) //ATotal即可见性函数V(Zi)

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CFinal

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CFinal

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CFinal

//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CFinal。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ATotal（即可见性函数V(Zi)）是单调递减的，最远处的若干层对CFinal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）

可见性函数的求解依赖于片元的顺序，导致了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(Zi)的方式，以达到以顺序无关的方式求解CFinal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[Ci Ai Zi]在生成采样点[Zi]时，每个采样点被覆盖的概率为Ai；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[Zs]，满足Zi NearerOrEqual Zs的概率即为可见性函数。

可以根据概率论的相关知识证明：满足Zi NearerOrEqual Zs即采样点[Zs]被片元[Ci Ai Zi]或被比片元[Ci Ai Zi]更远的片元覆盖，即采样点[Zs]不被比片元[Ci Ai Zi]更近的片元[Cj Aj Zj]覆盖，由于采样点[Zs]不被比片元[Ci Ai Zi]更近的片元[Cj Aj Zj]覆盖的概率为1 - Aj，且概率之间相互独立，最终的概率为各概率相乘，即。

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

### Alpha校正（Alpha Correction）

不难证明：

不妨设A0 = 0.4 A1 = 0.7 A2 = 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 - 0.4)×(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：的数学期望为即

Alpha校正可以认为是对SV(Zi)进行归一化（Normalize），即假定，从而得到

### 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[Ci Ai Zi]的Ai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[Ci Ai Zi]的Ai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(Zi)时，Zi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(Zi) 开启MRT和SeparateBlend/IndependentBlend 将透明物体按材质排序后绘制得到StochasticColor = 、CorrectAlphaTotal = 、StochasticTotalAlpha = //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CFinal 的总贡献为：TransparentColor = = StochasticColor

在启用Alpha校正时，透明物体对CFinal 的总贡献为：TransparentColor = //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CFinal

### 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(Zi)（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(Zi)

...->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(Zi)时，Zi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(Zi)时，Zi Equal Zs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CFinal（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[ AiCi 1 - Ai Zi ]。

首先将K-Buffer中的片元全部初始化为Ci = 0, Ai = 0, Zi = 无限远（即[0 1 无限远]）的“空片元”。 //在实际实现中，由于Zi的取值在0到1之间，只需要保证比Zi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Zi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（[ AiCi 1 - Ai Zi ]和[ Ai+1Ci+1 1 - Ai+1 Zi+1 ]融合后得到[ AiCi + (1 - Ai) Ai+1Ci+1 (1 - Ai)(1 - Ai+1) Zi] //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CFinal的贡献较低有一定关系（可见性函数V(Zi)是单调递减的，较远的片元的V(Zi)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CFinal 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CFinal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CFinal

### 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）

可见性函数的求解依赖于片元的顺序，导致了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(EyeSpaceZi Ai)作为可见性函数V(Zi)的估计值，从而达到以顺序无关的方式求解CFinal的目的（22.[McGuire 2013]、4.[Dunn 2014]）。

### 权重函数

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

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

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

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

//注：根据定义，可见性函数V(Zi)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Ai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：

归一化即假定，从而得到

### 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 = 、CorrectAlphaTotal = 、WeightedTotalAlpha = //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CFinal 的总贡献为：TransparentColor =

随后，基于CorrectAlphaTotal用Under操作将OpaquePass得到的BackgroundColor合成到CFinal

### 综合评价

权重融合用预定义的权重函数W(EyeSpaceZi Ai)近似地表示可见性函数V(Zi)，省去了求解可见性函数V(Zi)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(Zi)估计值的权重函数W(EyeSpaceZi Ai)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>

Chris Wyman. "Exploring and Expanding the Continuum of OIT Algorithms." HPG 2016.

<https://research.nvidia.com/publication/2016-06_Exploring-and-Expanding>

## Over操作和Under操作

Alex Dunn. "Transparency (or Translucency) Rendering." NVIDIA GameWorks Blog 2014.

<https://developer.nvidia.com/content/transparency-or-translucency-rendering>

Transparent //Alpha -> Cover //Visible –> //相加

Translucent //Transmittance –> //相乘

//ROP refers “Rasterization OPerator”?

For rendering algorithms, these can be roughly divided into light-based and view-based effects. Light-based effects are those in which the object causes light to be attenuated or diverted, causing other objects in the scene to be lit and rendered differently. View-based effects are those in which the semitransparent object itself is being rendered.

Light-based //Participating Media

Our point here is that alpha simulates how much the material covers the pixel.

The over operator is less convincing simulating other transparent effects, most notably viewing through colored glass or plastic. A red filter held in front of a blue object in the real world usually makes the blue object look dark, as this object reflects little light that can pass through the red flter. See Figure 5.32. When over is used for blending, the result is a portion of the red and the blue added together. It would be better to multiply the two colors together, as well as adding in any reflection the transparent object itself. This type of physical transmittance is discussed in Sections 14.5.1 and 14.5.2.

Objects that interpenetrate are impossible to resolve on a per-mesh basis for all view angles, short of breaking each mesh into separate pieces.

As noted earlier, all the transparency algorithms discussed in this section blend various colors instead of filtering them, mimicking pixel coverage.

To give a color filter effect, the opaque scene is read by the pixel shader and each transparent surface multiplies the pixels it covers in this scene by its color, saving the result to a third buffer. This buffer, in which the opaque objects are now tinted by the transparent ones, is then used in place of the opaque scene when resolving the transparency buffers.

This method works because, unlike transparency due to coverage, colored transmission is order-independent.

Weighted, Blended Order-Independent Transparency

<http://casual-effects.blogspot.com/2014/03/weighted-blended-order-independent.html>

Implementing Weighted, Blended Order-Independent Transparency

<http://casual-effects.blogspot.com/2015/03/implemented-weighted-blended-order.html>

Fast Colored Transparency

<http://casual-effects.blogspot.com/2015/03/colored-blended-order-independent.html>

1. Over操作：

从后往前绘制透明几何体

初项：

ColorDST = ColorBackGround/\*不透明背景\*/

递推公式：

ColorDst = Alphasrc\*Colorsrc + (1-Alphasrc)\*Colordst //与Alphadst无关，并不更新Alphadst

//SRC：FragmentShader输出

//DST：RenderTarget读取

不难证明，通项公式：

ColorDST = //背景可以看作ai = 1且ci = ColorBackGround

2. Under操作

2-1：从前往后绘制透明几何体

初项：

ColorDST = 0 且AlphaDST = 1 //物理含义：透射率初始值为1

递推公式 ：

ColorDST = ColorDST + AlphaDST \* (AlphaSRC\*ColorSRC) //Pre-Multiplied（预乘）Alpha = "AlphaSRC\*ColorSRC"作为FragmentShader输出

AlphaDST = AlphaDST\*(1 - AlphaSRC) //物理含义：从（离摄像机）最近层到“当前”层的透射率（Transmittance）

2-2：绘制不透明背景Alphasrc = 1且Colorsrc = ColorBackGround

Under操作与Over操作是等效的

不难证明Alpha的通项为：AlphaDST = ，代入ColorDST的递推公式中，即得到ColorDST通项，与Over操作相同！

Under操作的用例：

Case 1：透射率阈值（Transmittance Thresholding）

Daniel Wexler, Larry Gritz, Eric Enderton, Jonathan Rice. "GPU-Accelerated High Quality Hidden Surface Removal." Graphics Hardware 2005.

<https://research.nvidia.com/publication/gpu-accelerated-high-quality-hidden-surface-removal>

Alpha通项的物理含义：从（离摄像机）最近层到“当前”层的透射率（Transmittance）

在Under操作中，Alpha会不断趋向于0；Alpha降低到某个阈值时，可以EarlyOut

根据ViewSpace-Z将场景中的不透明物体分成若干个Bucket //可以理解为Cascade

在渲染时，读取上一个Bucket的Alpha，Discard掉Alpha小于某个阈值的像素 //不写入深度时，Discard不影响EarlyZ

Case 2：深度剥离（Depth Peeling）

NVIDIA OpenGL SDK 10 Code Samples / Dual Depth Peeling

<https://developer.download.nvidia.com/SDK/10.5/opengl/samples.html#dual_depth_peeling>

Chris Wyman. "Exploring and Expanding the Continuum of OIT Algorithms." HPG 2016.

<https://research.nvidia.com/publication/2016-06_Exploring-and-Expanding>

## 深度剥离（Depth Peeling）

Cass Everitt. "Interactive Order-Independent Transparency." NVIDIA WhitePaper 2001.

<https://www.nvidia.com/object/Interactive_Order_Transparency.html>

## 随机透明（Stochastic Transparency）

### 随机透明（Stochastic Transparency）

Eric Enderton, Erik Sintorn, Peter Shirley, David Luebke. "Stochastic Transparency." I3D 2010.

<https://research.nvidia.com/publication/stochastic-transparency>

Samuli Laine, Tero Karras. "Stratified Sampling for Stochastic Transparency." EGSR 2011.

<https://research.nvidia.com/publication/stratified-sampling-stochastic-transparency>

NVIDIA SDK11 Samples / StochasticTransparency

<http://developer.nvidia.com/dx11-samples>

//Alex Dunn. "Transparency (or Translucency) Rendering." NVIDIA GameWorks Blog 2014.

//<https://developer.nvidia.com/content/transparency-or-translucency-rendering>

//Sort-Middle Tiled下的MSAA是高效的，Stochastic Transparency有很大的发挥空间

概率为

深度偏移（Depth Bias）

在用VisualStudio图形分析器调试时，我们发现对于部分Sample，当z==zi时，并不成立

可能是浮点数误差导致

我们将浮点数转换为定点数(补码)后，可以进一步明确这个事实



//1.简单随机抽样

//对每个片元，每个采样点被覆盖的概率为Alpha

//对每个片元，被覆盖的采样点的个数满足二项分布

//2.分层抽样

//分别统计被覆盖的采样点个数为0 1 2 ... S-1 的片元的频率

//Eric Enderton等人用 概率论和数理统计的相关知识 分析得出 分层抽样的标准误差更低

### 彩色随机阴影映射（Colored Stochastic Shadow Map）

Morgan McGuire, Eric Enderton. "Colored Stochastic Shadow Maps". I3D 2011.

<http://research.nvidia.com/publication/colored-stochastic-shadow-maps>

彩色的（Colored）——依赖于波长的（Wavelength-Dependent）

## 现象学（Phenomenological）

### 权重融合OIT（Weighted Blended Order-Independent Transparency）

Morgan McGuire, Louis Bavoil. "Weighted Blended Order-Independent Transparency. " JCGT 2013.

<http://jcgt.org/published/0002/02/09/>

Alex Dunn. "Transparency (or Translucency) Rendering." NVIDIA GameWorks Blog 2014.

<https://developer.nvidia.com/content/transparency-or-translucency-rendering>

设同一SamplePoint上不同透明几何体的ShadingPoint的输出为Ai Ci Zi

通项公式：

令 //T(Zi)即Under操作中的Aacc //表示从（离摄像机）最近层到“当前”层的透射率（Transmittance），我们有： 。

Weighted Blended尝试将T(Zi)近似表示成Order Independent的等式： //其中V(Z)是一个随着（离摄像机的）距离衰减的自定义的可见性函数，从而有：

RT1 RGB通道：

RT1 A通道：

RT2 A通道：

Step1：RT1的RGB通道与A通道相除，得到透明物体的Cacc

Step2：Under操作，在Step1的基础上，使用RT2的A通道（T(Zi)的精确值）叠加不透明物体的背景颜色Cbackground //RT2并不是必须的，不透明物体的背景颜色可以看作Ai=1 Ci=Cbackground复用RT1中T(Zi)的近似值

### 现象学散射模型（Phenomenological Scattering Model）

Morgan McGuire, Michael Mara. "Phenomenological Transparency". TVCG 2017.

<http://research.nvidia.com/publication/phenomenological-transparency>

Morgan McGuire, Michael Mara. "A Phenomenological Scattering Model for Order-Independent Transparency". I3D 2016.

<http://research.nvidia.com/publication/phenomenological-scattering-model-order-independent-transparency>

//结合**参与介质**的相关概念

**材质**

传统的

BRDF（微平面/镜面反射 + 朗伯/漫反射）

局部覆盖（Partial Coverage） //即Alpha //可以认为是“单色的”/独立于波长的透射率

次世代

BTDF

Index Of Refractive 折射率（标量）

Transmittance 透射率（RGB）//消光系数

Collimation 准直系数 //相函数（Phase Function）//首选散射方向

0 -> 稠密的 各向异性的 参与介质 比如浑浊的水、雾

1 -> 光学上均匀/各向同性的 参与介质 不存在多次散射 比如玻璃

Transmittance和Collimation用于近似表示贝尔朗伯（Beer Lambert）定律/相函数（Phase Function），从而避免实时计算

对于一些厚度随视线变换比较大的物体（比如冰雕），可能会有比较明显的误差

**透射**

对任意半透明表面有

Lout

= Alpha \* (Lreflect + T \* Lin) + (1-Alpha)\* Lin //Alpha Blend中Over操作

= Alpha \* Lreflect //反射 BRDF

+ (Alpha \* T + 1 - Alpha) \* Lin //透射 BTDF

假设在同一个像素中，有2个半透明表面和1个（最远的）不透明表面

L2

= Alpha2\*Lreflect2 //反射

+ (Alpha2\*T2 + 1 – Alpha2)\*L1 //透射

= Alpha2\*Lreflect2

+ (Alpha2\*T2 + 1 – Alpha2)\*(Alpha1\*Lreflect1 + (Alpha1\*T1 + 1 – Alpha1)\*L0)

= Alpha2\*Lreflect2 + (Alpha2\*T2 + 1 – Alpha2)\*Alpha1\*Lreflect1 //反射

+ (Alpha2\*T2 + 1 – Alpha2)\* (Alpha1\*T1 + 1 – Alpha1)\*L0 //透射 顺序无关！！！

显然，透射是顺序无关的，我们将∏Alphai\*Ti + 1 – Alphai记作B (即Background)

但是，由于第i个半透明表面的Alpha\*Lreflect只受到第i – N半透明表面的透射影响，反射并不是顺序无关的

注：个人认为，可以忽略半透明表面的反射，T即可认为传统的半透明表面的Color.RGB

因为我们要用Alpha Blend中的Over操作将2个半透明表面合成到传统的颜色缓冲（即1个（最远的）不透明表面）中

我们希望L2可以近似表示为

(1 - B) \* Lreflect待定 //反射

+ B \* L0 //透射

其中1 – B项即透射率

显然，反射Lreflect的系数w与顺序相关，应当采取某种方式近似表示

//发展自权重融合（Weight Blended）

Lreflect待定 = (∑wi\*Alphai\* Lreflecti)/(∑wi\* Alphai\*(1-Ti))

w在物理含义上为Lreflect之前的Layer的透射率

显然随着深度的增加而增加

wi = clamp([10 \* (1 – 0.99 \* Depth) \* Alphai \* (1 - Ti)]^3, 0.01, 30) 其中Depth即 gl\_FragCoord.z,

//McGuire Bavoil公式 (1 – 0.99 \* Depth) \*Alphai \* (1 - Ti)]^3

//10 scaled for an infinite far plane and float16 precision

在深度反转中，应当作相应的调整

wi = clamp([10 \*(0.01 + 0.99 \* Depth) \*Alphai \* (1 - Ti)]^3, 0.01, 30)

分母∑wi\* Alphai\*(1-Ti)即归一化

#### 透射（Transmittance）

局部覆盖（Partial Coverage） Over操作（传统的）

透射（Transmittance） 贝尔朗伯定律（次世代）

#### 折射（Refraction）

折射定律/斯涅耳（Snell）定律

入射角与折射角的正弦值的比值等于折射率（Index Of Refraction）比值

折射率与光的波长有关，即色散（Dispersion）

但是，在实时渲染中忽略，用标量而非RGB表示折射率

#### 焦散（Caustic）

又称漫折射（Diffusion）

## K-Buffer

### K-Buffer

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>

### 自适应透明（Adaptive Transparency）

Marco Salvi, Jefferson Montgomery, Aaron Lefohn. "Adaptive Transparency." High Performance Graphics 2011.

<https://software.intel.com/en-us/articles/adaptive-transparency-hpg-2011>

Leigh D. "Order-Independent Transparency Approximation with Raster Order Views (Update 2017)." Intel Software Developer Zone 2017.

https://software.intel.com/en-us/articles/oit-approximation-with-pixel-synchronization-update-2014

PC基于RasterizerOrderedView(Direct3D12) / RasterOrderGroup(Metal)

Mobile基于SubpassInput(Vulkan)/ProgrammableBlending(Metal)

### 多层Alpha融合（Multi-layer Alpha Blending）

Marco Salvi, Karthik Vaidyanathan. "Multi-layer Alpha Blending." I3D 2014.

<https://software.intel.com/en-us/articles/multi-layer-alpha-blending>

### 自适应体积阴影映射（Adaptive Volumetric Shadow Maps）

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>

# 参与介质（Participating Media）

在目前的图形引擎中

局部子表面散射往往被定义为标准的PBR明暗模型

而SSS（SubsurfaceScattering）明暗模型往往特指基于FaceWorks相关技术实现的全局子表面散射

大气散射在物理意义上也属于全局子表面散射，但是在实时渲染中采用的是与FaceWorks完全不同的另一套技术

深度线索（Depth Cue）

空气透视（Aerial Perspective）

天空颜色可以被认为是空气透视的一种极限情况

水下 介质对光线传播的影响 较强的一种情形

## 传统的

### 深度雾（Depth Fog）

PositionZ = MP[2][3]/(Depth - MP[2][2]) //ViewSpace位置

ColorFinal=FogFactor\*ColorSrc+(1- FogFactor)\*ColorFog

ColorSrc输入纹理

ColorFog输入常量

FogFactor = e^(FogScale\*PositionZ) 比尔朗伯定律（Beer-Lambert Law）

### 天空盒（SkyBox）

天空

远景地形

物理含义：视差（Parallax） 比如观察月亮随着人移动

天空盒分辨率 满足一定大小 与屏幕分辨率和视野（FovAngleY（垂直角度））

### 公告板云（Billboard Cloud）

## 次世代

### 参与介质（Participating Media）

Sebastien Hillaire. "Physically Based Sky, Atmosphere and Cloud Rendering in Frostbite". SIGGRAPH 2016.

<http://www.ea.com/news/physically-based-sky-atmosphere-and-cloud-rendering>

Nathan Hoobler. "Fast, Flexible, Physically-Based Volumetric Light Scattering". GDC 2016.

<http://developer.nvidia.com/VolumetricLighting>

Iain Cantlay, Andrei Tatarinov. "From Terrain To Godrays - Better Use of DX11 Tessllation". GDC 2014.

<http://developer.nvidia.com/content/terrain-godrays-better-use-dx11-tessellation>

Bartlomiej Wronshki. "Volumetric Fog: Unified compute shader based solution to atmospheric scattering". SIGGRAPH 2014.

体积云

Patapom Bomb. "Real-Time Volumetric Rendering". Revision 2013.

Andrew Schneider. "The Real-time Volumetric Cloudscapes of Horizon: Zero Dawn". SIGGRAPH 2015.

Jean-Philippe Grenier. "Volumetric Clouds”. Area by Autodesk 2016.

文档 http://area.autodesk.com/blogs/game-dev-blog/volumetric-clouds

源代码 http://github.com/greje656/clouds

Egor Yusov. "Real-Time Rendering of Physically Based Clouds Using Precomputed Scattering". Chapter II-4 GPU PRO 6 2016.



真空中

LSensor\_Vaccum = LLightSource \* Visible(LightSource\_To\_OpaqueSurface) \* BRDF

Reflect 一般即不透明表面的Fresnel

参与介质中

LSensor\_ParticipatingMedia =

//透射

LLightSource

\* Visible(LightSource\_To\_OpaqueSurface) //阴影映射

\* BeerLambertLaw(LightSource\_To\_OpaqueSurface) //表面的雾效果

\* BRDF

\* BeerLambertLaw(OpaqueSurface\_To\_Sensor) //表面的雾效果

+

//内散射

|0 | Sensor\_To\_ OpaqueSurface |

LLightSource

\*BeerLambertLaw(|LightSource\_To\_ Sensor + Sensor\_To\_OpaqueSurface \* X|) //参与介质的体积阴影效果

\* PhaseFunction(<LightSource\_To\_ Sensor + Sensor\_To\_OpaqueSurface \* X ,

OpaqueSurface\_To\_Sensor>) //相函数

\* BeerLambertLaw(|Sensor\_To\_OpaqueSurface \* X|) //参与介质的自阴影效果

dX

其中X是无量纲

根据向量数乘的定义

|LightSource\_To\_ Sensor + Sensor\_To\_OpaqueSurface \* X

X ----> 0 - | Sensor\_To\_OpaqueSurface |

定义了线段Sensor\_To\_OpaqueSurface上各个点的位置

#### 消光（ExTinction）/ 贝尔朗伯定律（Beer-Lambert Law）

吸收（Absorption）吸收系数σa （单位 m-1）度量

外散射（Out-Scattering） 散射系数σs （单位 m-1）度量

消光 -> 吸收+外散射

定义 消光（ExTinction）系数 σt = σa + σs （单位 m-1） 度量吸收和外散射的叠加效果

消光系数是依赖于波长（wavelength Dependent）的

在实时渲染中，可以用(σtR, σtG, σtB)/即(σaR+σsR, σaG+σsG, σaB+σsB)表示

假定参与介质中(σtR, σtG, σtB)保持不变

BeerLambertLaw(Thickness)

= ( e^(-σtR\* Thickness) , e^(-σtG\* Thickness), e^(-σtB\* Thickness) ) //无量纲

= ( e^(-(σaR+σsR)\* Thickness) , e^(-(σaG+σsG)\* Thickness), e^(-(σaG+σsG)\* Thickness) )

= ( e^(-σaR\* Thickness) \* e^(-σsR\* Thickness), e^(-σaG\* Thickness) \* e^(-σsG\* Thickness),

e^(-σaB\* Thickness) \* e^(-σsB\* Thickness) )

其中，σt\* Thickness被定义为光学（Optical）深度

不考虑乘以PhaseFunction(0)???

#### 相函数（Phase Function）

PhaseFunction可以认为是BRDF的Volume Analog —— Nathan Hoobler（NVIDIA）

在光学上以统计的方式——相函数——描述介质粒子在各个方向上对入射光线的反射

定义域 相角（反射光线和入射光线夹角）

值域 反射光线能量与假定各向同性（且无吸收）时的透射光线的能量的比值 //无量纲

根据能量守恒，相函数在整个球面（球面度4π）上的积分一定小于或等于1，小于1表明部分能量被吸收

显然，相函数只能表示一个粒子对入射光线的散射效果

在上文中，我们假定OpaqueSurface\_To\_Sensor上的所有粒子的相函数相同

//可以将该相函数看作是各个粒子的相函数加权平均后得到的近似结果

相函数是依赖于波长的

在实时渲染中，往往用某个RGB向量乘以一个表示强度的标量

可以用在数值上与上文中用于度量外散射的散射系数(σsR, σsG, σsB)相同的RGB向量（但单位不同，此处的RGB向量为无量纲）

粒子散射 根据粒子的半径和可见光波长的关系 可以分为两类

瑞利散射（Rayleigh Scattering）粒子的半径远小于可见光波长（比如大气）

米散射（Mie Scattering）粒子的半径与可见光波长相近（比如雾、云）

XXXX不讨论 粒子的半径远大于可见光波长

瑞利散射

强烈依赖于波长（比如大气对蓝色波长散射强烈，吸收可以忽略）

几乎各向同性

PhaseFunction\_Rayleigh(PhaseAngle) = (σsR, σsG, σsB) \*//波长

(3/(16π))\*(1+cos(PhaseAngle)^2) //强度

米散射

带强烈前向叶的各向异性

米散射的相函数非常复杂

一般用亨利格林斯(Henyey-Greenstein)函数近似表示

对任意相函数

定义 首选散射方向g = |4π cos(PhaseAngle) \* PhaseFunction(PhaseAngle) dw

可以认为是对cos(PhaseAngle)按照PhaseFunction(PhaseAngle)加权平均

取值在 1到-1之间­

1 -> cos(0) 前向散射

0 -> cos(π/2) 各向同性散射

-1 -> cos(π) 后向散射

首选散射方向 / 离心率（Eccentricity）

PhaseFunction\_HenyeyGreenstein(PhaseAngle) = (σsR, σsG, σsB) \*//波长

(1 - g^2) / ( 4π \* ( ( 1 + g^2 - 2\*g\*cos(PhaseAngle) )^(3/2) ) ) //强度

其中g即亨利格林斯函数所要近似表示的米散射相函数的首选散射方向

一般在0到1之间 （不会小于0，米散射不会向后散射）

HenyeyGreenStein的Schlick近似

k = 1.55\*g – 0.55\*(g^3)

PhaseFunction\_Schlick\_HenyeyGreenStein =

(1 - k^2) / ( 4π \* ( ( 1 + k \* cos(PhaseAngle) )^2 ) )

但是在g接近-1或1时，误差较大，个人认为不建议使用

#### 透射（Transmittance）

即上文公式中的

LSensor\_Transmittance

= LLightSource

\* Visible(LightSource\_To\_OpaqueSurface) //阴影映射

\* BeerLambertLaw(LightSource\_To\_OpaqueSurface) //表面的雾效果

\* BRDF

\* BeerLambertLaw(OpaqueSurface\_To\_Sensor) //表面的雾效果

根据乘法交换律

= LLightSource

\* Visible(LightSource\_To\_OpaqueSurface)

\* BRDF

\* BeerLambertLaw(LightSource\_To\_OpaqueSurface) //表面的雾效果

\* BeerLambertLaw(OpaqueSurface\_To\_Sensor) //表面的雾效果

= LSensor\_Vaccum

\*BeerLambertLaw(LightSource\_To\_OpaqueSurface)

\* BeerLambertLaw(OpaqueSurface\_To\_Sensor)

#### 内散射（In-Scattering）

即上文公式中的

LSensor\_In-Scattering

= |0 |OpaqueSurface\_To\_Sensor|

LLightSource

\*BeerLambertLaw(|LightSource\_To\_ Sensor + Sensor\_To\_OpaqueSurface \* X|) //参与介质的体积阴影效果

\* PhaseFunction(<LightSource\_To\_ Sensor + Sensor\_To\_OpaqueSurface \* X ,

OpaqueSurface\_To\_Sensor>) //相函数

\* BeerLambertLaw(|Sensor\_To\_OpaqueSurface \* X|) //参与介质的自阴影效果

dX



#### 直观理解

正如上文所述，内散射中用数值上与用于度量外散射的散射系数(σsR, σsG, σsB)相同的RGB向量表示粒子散射对波长的依赖

可以定义 反照率（Albedo） ρ = σs / σt = σs / (σa + σs)

反照率表示散射（外散射+内散射）与吸收之间的关系

0->较暗 1->较亮 主要体现在内散射中

RGB向量(σsR, σsG, σsB)同时影响到外散射和内散射

在参与介质的深度较低时，内散射占主导地位，参与介质的颜色与(σsR, σsG, σsB)相近

在参与介质的深度较高时，外散射占主导地位，参与介质的颜色与(σsR, σsG, σsB)的逆（可以近似理解为1 - (σsR, σsG, σsB)）相近

### 体积雾（Volumetric Fog）

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>

### 体积光（Volumetric Light / GodRay）

Nathan Hoobler. "Fast, Flexible, Physically-Based Volumetric Light Scattering." GDC 2016.

<http://developer.nvidia.com/VolumetricLighting>

Iain Cantlay, Andrei Tatarinov. "From Terrain To Godrays - Better Use of DX11 Tessllation". GDC 2014.

<http://developer.nvidia.com/content/terrain-godrays-better-use-dx11-tessellation>

假定Visible(LightSource\_To\_ParticipatingMedia(X))只有0或1两种取值 （不考虑柔和阴影）

根据定积分的可加性

=

|0 |OpaqueSurface\_To\_Sensor|

LLightSource

\*BeerLambertLaw(|LightSource\_To\_ Sensor + Sensor\_To\_OpaqueSurface \* X|) //参与介质的体积阴影效果

\* PhaseFunction(<LightSource\_To\_ Sensor + Sensor\_To\_OpaqueSurface \* X ,

OpaqueSurface\_To\_Sensor>) //相函数

\* BeerLambertLaw(|Sensor\_To\_OpaqueSurface \* X|) //参与介质的自阴影效果

dX

-

|X1 X2

LLightSource

\*BeerLambertLaw(|LightSource\_To\_ Sensor + Sensor\_To\_OpaqueSurface \* X|) //参与介质的体积阴影效果

\* PhaseFunction(<LightSource\_To\_ Sensor + Sensor\_To\_OpaqueSurface \* X ,

OpaqueSurface\_To\_Sensor>) //相函数

\* BeerLambertLaw(|Sensor\_To\_OpaqueSurface \* X|) //参与介质的自阴影效果

dX



体积渲染

用细分取代光线跟踪

根据ShadowMap细分生成一个Light Volume，并根据Sensor的VP矩阵投影

Light Volume同一视线方向（即Sensor \_To\_OpaqueSurface）上交点会位于同一个像素，可以根据正面或背面计算上述积分

[Iain Cantlay, Andrei Tatarinov. 2014]

计算不定积分

L(X)

=|0 X

LLightSource

BeerLambertLaw(|LightSource\_To\_ Sensor + Sensor\_To\_OpaqueSurface \* t|)

\* PhaseFunction(<LightSource\_To\_ Sensor + Sensor\_To\_OpaqueSurface \* t ,

OpaqueSurface\_To\_Sensor>)

\* BeerLambertLaw(|Sensor\_To\_OpaqueSurface \* t|)

dt

平行光(Directional Light)

平行光方向为常量

LightSource\_To\_OpaqueSurface + Sensor\_To\_OpaqueSurface \* t ≈ LightDirection

相函数为常量

PhaseFunction(<LightSource\_To\_ Sensor + Sensor\_To\_OpaqueSurface \* t ,

OpaqueSurface\_To\_Sensor>)

≈ PhaseFunction(<LightDirection, -ViewDirection>) //OpaqueSurface\_To\_Sensor即-ViewDirection

平行光距离远大于介质深度

LightSource\_To\_OpaqueSurface >> Sensor\_To\_OpaqueSurface \* t

贝尔朗伯定律为常量

BeerLambertLaw(|LightSource\_To\_ Sensor + Sensor\_To\_OpaqueSurface \* t|)

\*BeerLambertLaw(|Sensor\_To\_OpaqueSurface \* t|)

= e ^ ( -σt \* ( |LightSource\_To\_ Sensor + Sensor\_To\_OpaqueSurface \* t|

+ |Sensor\_To\_OpaqueSurface \* t|) )

≈ e ^ ( -σt \* ( |LightSource\_To\_ Sensor + 0 | + |0| ) )

= e ^ ( -σt \* |LightSource\_To\_ Sensor| ) //实际中对RGB三个分量各计算一次

= BeerLambertLaw(|LightSource\_To\_ Sensor| )

不定积分为初等函数

L(X)

=|0 X

LLightSource

\* PhaseFunction(<LightDirection, -ViewDirection>)

\* BeerLambertLaw(|LightSource\_To\_ Sensor| )

dt

= LLightSource

\* PhaseFunction(<LightDirection, -ViewDirection>)

\* BeerLambertLaw(|LightSource\_To\_ Sensor| )

\*(X - 0)

点光源（Point Light）/泛光灯（Omni Light）

聚光灯（Spot Light）

预计算不定积分（离散方式模拟）并存储到LUT（Look-Up Table）

二维 1.<LightSource\_To\_ Sensor, Sensor\_To\_OpaqueSurface> 2.不定积分的自变量X

个人认为还需要确定|LightSource\_To\_ Sensor|

### 体积云（Volumetric Cloud）

“糖粉（Powdered Sugar）”效果

效果 暗边

在统计意义上 表面内越深 内散射的可能性越强

贝尔定律（吸收+外散射） E = e^(-Depth)

糖粉效果（内散射）E = 1 – e^(-Depth\*2)

贝尔糖粉效果（以上两式相加） e^(-Depth) + 1 – e^(-Depth\*2)

球形大气求交

确保云正确地下降到地平线

//球形大气求交



#define EARTH\_RADIUS 150000.0

#define CLOUDS\_START 1500.0

#define CLOUDS\_END 5000.0

#define MAX\_CLOUD\_DISTANCE 12000.0

//命名规范

//Offset 未归一化

//Direction 归一化

用视野变换中介绍的Trick将ViewOffset从ViewSpace变换到WorldSpace

find\_atmosphere\_intersections

//求出 EARTH\_RADIUS + CLOUDS\_START或CLOUDS\_END 在ViewDirection上的交点 到摄像机位置的距离

//解三角形

c = EARTH\_RADIUS + CLOUDS\_START或CLOUDS\_END

EarthCenterToSurfaceOffset = ( 0, 0, EARTH\_RADIUS ) //假定Z轴为WorldSpace的UpDirection

d = dot( EarthCenterToSurfaceOffset, ViewDirection\_WorldSpace )

if(d>0) //视线在地平线之上

{



b^2 = dot( EarthCenterToSurfaceOffset, EarthCenterToSurfaceOffset)

h^2 = b^2 – d^2

(d+a)^2 = c^2 – h^2 = c^2 – (b^2 – d^2) = d^2 + c^2 – b^2

a = sqrt(d^2 + c^2 – b^2) – d //由于c大于b 因此sqrt(d^2 + c^2 – b^2)一定大于d

}

else if(d == 0) //视线与地平线平行

{

EarlyOut？

}

else if(d < 0) //视线在地平线之下

{

EaryOut？

}

ViewDirection \* 上述a 求出CloudStartOffset或CloudEndOffset

用投影变换中介绍的Trick将CloudStartOffset或CloudEndOffset变换到ViewSpace

# 间接光（Indirect Light）

## 镜面反射

### 辐射亮度环境映射（Radiance Environment Map）

Reflection Mapping //模拟镜面 //平面反射（Planar Reflection）的预计算版本

Glossy Reflection Mapping //游戏引擎中又叫作间接高光（Indirect Specular）

光探头（Light Probe）

环境映射又被叫做IBL（Image Based Lighting，基于图像的光照）

一个与IBL相近的名称IBR（Image Based Rendering）是指Post Process（后处理）

### IBL（Image-Based Lighting，基于图像的光照）

环境映射（Environment Mapping）：保存球函数（Spherical Function）的图像 //任意高频（通过增加分辨率）//任意范围环境辐射亮度（HDR，访问变慢）

光探头（Light Probe）——即环境映射（Environment Map）

高光光探头（Specular Light Probe）：光探头用于高光映射

Image-Based Lighting：用纹理数据照亮场景

投影函数（Projector Function）：将r（reflected view vector）转换为纹理uv

环境映射根据投影函数分：

1.立方体映射（Cube Mapping）

#if 0

1. 球体映射（Sphere Mapping） //圆形图像

对球形探头（Spherical Probe）正交投影得到的图像（Image）

Refleced View Vector –> View Space (rx, ry, rz) //或者认为是球形映射得参考系

h = n = r + (0, 0, 1) = (rx, ry, rz+1)

//归一化后相除 角度不变

u = 0.5\*(rx/|h|)+0.5

v = 0.5\*(ry/|h|)+0.5

//当Refleced View Vector与 View Vector (0,0,1) //正交投影方向 夹角大于90度 //在球体映射中对映区域很小

#endif

#### 双抛物面映射（Dual Paraboloid Mapping）

Wolfgang Heidrich, Hans-Peter Seidel. "View-independent Environment Maps." EUROGRAPHICS 1998.

<https://vccimaging.org/Publications/Heidrich1998VEM/Heidrich1998VEM.pdf>

Imagination Technologies Limited. "Dual Paraboloid Environment Mapping." Power SDK Whitepaper 2017.

[https://github.com/powervr-graphics/Native\_SDK/tree/4.3/Documentation/Whitepapers](https://github.com/powervr-graphics/Native_SDK/tree/4.3/Documentation/Whitepapers/Dual%20Paraboloid%20Environment%20Mapping.Whitepaper.pdf)

用（椭圆）抛物面代替球面 //r和(0,0,1)的交角控制在90度以内

得到2个圆形图像 分别表示2个环境半球 根据rz的符号选择访问哪一个圆形图像

//纹理采样比球形映射和立方体映射更均匀 //但在rz=0附近时 需要处理2个圆形图像之间的缝隙

投影函数

u = 0.5\*(rx/(rz+1))+0.5 //该函数形状为抛物线??? //查高中数学//圆锥曲线

v = 0.5\*(ry/(rz+1))+0.5

与材质集成

1.反射映射（Reflection Mapping）

理想状态下的镜面/光学上平坦的表面： r = 2dot(n,v)n - v //L = F(n,r)\*L(r) //F即Fresnel（h与n相同） L即Environment Mapping

被反射的物体应当较远 //整个反射器（Reflector）表面与球形探头的中心的距离都假定为0

2.高光反射（Specular Mapping）

#### 重要性采样（Importance Sample）

Ravi Ramamoorthi. "Topic, Monte Carlo Integration." Berkeley University Of California, CS 294-13, Advanced Computer Graphics, Fall 2009.

<http://inst.eecs.berkeley.edu/~cs294-13/fa09/lectures/scribe-lecture4.pdf>

Mark Colbert, Jaroslav Kivánek. "GPU-Based Importance Sampling." GPU Gems 3 Chapter 20 2007.

<https://developer.nvidia.com/gpugems/GPUGems3/gpugems3_ch20.html>

矩形法（Quadrature Method）

在矩形法 [ 《高等数学第六版上册》（ISBN 978-7-04-020549-7） 第五章定积分 第一节定积分的概念与性质 三、定积分的近似计算 2007 ] 中，我们将积分区间等分成若干个小区间， 并用均匀的频率对被积函数进行采样，分别（近似地）代表被积函数在各个小区间上的平均值，形成若干个Δx相同的矩形，将这些矩形的面积的和作为定积分的近似值（当然，实际中还需要考虑到正负号的问题），如下图：

矩阵法的不足

矩形法（使用均匀的频率进行采样）并不是“最科学的”方法；显然，在被积函数的函数值较小的区间上，可以用较低的频率进行采样，每个采样点（近似地）代表被积函数在较大范围的区间（即Δx较大）上的平均值，从而减少了采样次数且提升了效率（虽然相对于该区间的局部而言，可能会产生较大的误差，但是，由于该区间内的函数值较小，相对于整个积分区间而言，影响是极低的，并不会产生明显的误差）；而在被积函数的函数值较大的区间上，可以用较高的频率进行采样，每个采样点（近似地）代表被积函数在较小范围的区间（即Δx较小）上的平均值，由于该区间内的函数值较大，可以明显地降低整个积分区间的误差，如下图：

蒙特卡洛积分（Monte Carlo Integration）

蒙特卡洛积分可以认为是对矩形法的泛化：按照某个PDF（Probability Density Function，概率密度函数）的频率对被积函数进行采样，每个采样点（近似地）代表被积函数在长为1/PDF的区间上的平均值；可以证明，这些矩形的面积的和的数学期望（Expected Value）即为定积分的值。

重要性采样（Importance Sampling）

重要性采样是蒙特卡洛积分中所使用的一种方差缩减（Variance Reduction）技术（另一种常用的方差缩减技术是分层抽样（Stratified Sampling））。

重要性采样利用数学工具对我们在上文中定性分析得出的结论给出了一个精确的定量描述；可以证明，当PDF与被积函数相等时，蒙特卡洛积分的方差（Variance）达到最小值零（可以认为方差越低，误差越小）；重要性采样即使用一个与被积函数尽可能接近的PDF进行蒙特卡洛积分：在被积函数的函数值较小的区间上，PDF较小，即采样频率较低，每个采样点代表的区间范围（1/PDF）较大；而在被积函数的函数值较大的区间上，PDF较大，即采样频率较高，每个采样点代表的区间范围（1/PDF）较小。

采样点生成

现有的大多数随机数生成算法，往往只能按照均匀分布生成采样点；要按照某个PDF生成采样点，往往需要先求出PDF的定积分，得到CDF（Cumulative Distribution Function，（累积）分布函数），再求出CDF的反函数CDF-1，将均匀分布的采样点输入到CDF-1，即可得到按照某个PDF生成的采样点；虽然，当PDF与被积函数相等时，蒙特卡洛积分的误差最小，但是，被积函数的定积分往往并不容易求得（否则，直接求得被积函数的定积分即可，根本没有使用蒙特卡洛积分的必要），实际中往往使用一个与被积函数接近但是又比较容易求得定积分的函数作为PDF。

Josiah Manson, Peter-Pike Sloan. "Fast Filtering of Reflection Probes." EGSR 2016.

<https://www.ppsloan.org/publications/ggx_filtering.pdf>

CDF（Cumulative Distribution Function，（累积）分布函数）

PDF（Probability Density Function，概率密度函数）

//矩形法 –> 均匀采样

//采样频率过高

//蒙特卡洛方法 -> 可变频率采样

//方差减小技术 PDF = f(x)

//采样点并不是任意的 //PDF同时代表了采样点的频率 //可以根据 CDF的逆生成采样点 //大数定理/中心极限定理？？？

//可以理解成是一种归约从而使矩形等距的技术

However, when we skew sample directions, not all estimates of the integral are equal, and thus we must weight them accordingly when averaging all the samples. For instance, one sample in a low-value region of the PDF is representative of what would be many samples if uniform sampling were used.

//variance reduction

To define the optimal PDF for the illumination integral, we need an accurate approximation of the product between the material BRDF and the incident lighting. However, generating samples according to this product is quite involved (Clarberg et al. 2005). Fortunately, creating samples according to the BRDF alone is straightforward, and when we use it in combination with sample filtering—described in Section 20.4—it provides good results.

For even more efficient sampling schemes, it is also possible to consider the

distribution of radiance in the environment map jointly with the shape of the specular

lobe [270, 819].

//Mipmap Filtered Samples

//GPU Gems 3

The use of such deterministic number sequences, in combination with a relatively low sample count, produces an aliasing effect, as shown in Figure 20-5a, that we suppress by filtering the samples, as detailed in the next section.

As shown in Figure 20-5a, deterministic importance sampling causes sharp aliasing artifacts that look like duplicate specular reflections. In standard Monte Carlo quadrature, this problem is combated by introducing randomness, which changes aliasing into more visually acceptable noise, as is evident in Figure 20-5b. However, keeping the noise level low requires using hundreds or even thousands of samples for each pixel—which is clearly not suitable for real-time rendering on the GPU. We use a different approach—filtering with mipmaps—to reduce aliasing caused by the low sample count, as shown in Figure 20-5c.

//Real Time Rendering

To further reduce sampling variance (i.e., noise), we could also estimate the distance

between samples and integrate using a sum of cones, instead of single directions.

Sampling an environment map using cones can be approximated by point sampling

one of its mip levels, choosing the level whose texel size spans a solid angle similar to

that of the cone [280].

//HardWare Mipmap

Vulkan Specification / 15.6.7. Scale Factor Operation, Level-of-Detail Operation and Image Level(s) Selection

<https://www.khronos.org/registry/vulkan/specs/1.0/pdf/vkspec.pdf>

//MipMap

//理论上不是必要的，因为贡献不大，一次采样即可，并没有必要Average？？？

//实际中存在着近似 //PDF只取BRDF并没有考虑EnvironmentMap的影响

球面度|积分

//Art Owen. "Monte Carlo theory, methods and examples." Stanford University.

//<https://statweb.stanford.edu/~owen/mc/>

### SSR（Screen Space Reflection，屏幕空间反射）

Morgan McGuire, Michael Mara. "Efficient GPU Screen-Space Ray Tracing". JCGT 2014.

<http://jcgt.org/published/0003/04/04/>

//与光栅化阶段插值方式一致 //因为用于比较远近的Texture是由光栅化阶段生成的

//Z in 3Dspace 透视校正插值（原文）

//Depth in ScreenSpace 线性插值（与TerrainRaycast一致，个人认为应当采用）

//Mirror -> Environment Map

//Specular -> Reflection Map

//设ReflectionDirection

= NormalMacroScopic\*dot(ViewDirection,NormalMacroScopic)\*2- ViewDirection

//可以认为ReflectionDirection是对Specular贡献最大的LightDirection

//理想状况下

//根据ReflectionDirection附近的若干个LightDirection采样

//累加BRDF\_Specular \* EnvironmentMap(LightDirection)

//以上做法开销过大

//累加Fresnel \* EnvironmentMap(LightDirection)

//进一步简化，只根据ReflectionDirection采样一次

//计算Fresnel \* EnvironmentMap(ReflectionDirection)

将对EnvironmentMap的采样替换成Screen-Space Ray Tarcing即SSR

误差分析

当前帧的RadianceDirect（辐射亮度-直接光）被用于取代EnvironmentMap

由于Radiance中的Specular部分与ViewDirection有关

严格意义上，ViewDirection应当为ReflectionDirection，而非SurfaceToSensor

基于DDA（Digital Differential Analyzer，数值微分法）

//经典单线程版本

//分歧分支（Divergent Branch）

if( abs(x1-x0) > abs (y1-y0) )

{

slope = (y1-y0)/abs(x1-x0)

for t = 0 to abs(x1-x0)

setPixel(x0 + sign(x1-x0)\*t, y0 + slope\*t);

}

else

{

slope = (x1-x0)/abs(y1-y0)

for t = 0 to abs(y1-y0)

setPixel(x0 + slope\*t, y0 + sign(y1-y0)\*t);

}

//次世代并行版本

//交换坐标轴

//for循环从分歧分支中移出

Long0

Long1

Short0

Short1

bLongX

//分歧分支

bLongX = abs(x1-x0) > abs(y1-y0)

Long0 = bLongX?x0:y0

Long1 = bLongX?x1:y1

Short0 = bLongX?y0:x0

Short1 = bLongX?y1:x1

//并行执行

for t = 0 to abs(Long1- Long0)

//并行执行

slope = (Short1-Short0)/abs(Long1- Long0)

Long = Long0 + sign(Long1-Long0)\*t

Short = Short0 + slope\*t

//分歧分支

bLongX ? setPixel(Long, Short) : setPixel(Short, Long)

//NormalMap

G-Buffer中 同时存储 NormalMacro 和 NormalMeso

//其中NormalMacro用于SSR

//透明表面（在DepthBuffer中不存在Depth）

//在像素着色器（材质系统中设置）中（读取gl\_FragCoord.z（在FrameBuffer坐标系中 在Viewport变换之后））进行SSR

//UE4中材质选项：？？？

//可以将SSR的辐射亮度写入GBuffer- Emissive（自发光）以兼容OIT ？？？

### SSRO（Screen Space Reflective Occlusion， 屏幕空间反射遮挡）

<http://marina.sys.wakayama-u.ac.jp/~tokoi/?date=20161231>

应该在此基础上加上重要性采样

//考虑到Split-Integral Approximation

## 漫反射

### 辐射照度环境映射（Irradiance Environment Map）

//游戏引擎中又叫作间接漫反射（Indirect Diffuse）

仅适用于朗伯漫反射

设各个方向的入射光的辐射亮度LL，视线方向辐射亮度Lv=|w BRDF(l,v) LLcos<l,n> dw

在朗伯漫反射中，BRDF(l,v)为常量，Lv= BRDF(l,v) \* |w LL(w)max(0,cos<l(w),N>) dw

可以基于|w LL(w)max(0,cos<l(w),N>) dw在整个半球方向预计算EN

其中LL(w)可以在辐射亮度环境映射中采样得到

辐射照度环境映射 即不定积分|w LL(w)max(0,cos<l(w),N>) dw其中法线方向N为自变量

可以考虑到Fresnel函数，在这种情况下，ColorDiffuse不再是常量，可以考虑在不定积分中预计算Fresenl

### 球谐函数（Spherical Harmonic）

Peter-Pike Sloan, Jan Kautz, John Snyder. "Precomputed Radiance Transfer for Real-Time Rendering in Dynamic, Low-Frequency Lighting Environments". SIGGRAPH 2002.

Peter-Pike Sloan. "Stupid Spherical Harmonics (SH) Tricks." GDC 2008.

<http://www.ppsloan.org/publications/StupidSH36.pdf>

球谐函数

原理上与傅里叶变换相似

用相互正交的基函数近似表示目标函数，通过基投影求出各个基函数的系数

在目标函数的频率较低的情况下，球谐函数可以用较少的基函数近似表示目标函数

基于上述方法求出目标函数在各个基函数上的系数的过程是否可以取一个名字？？？球谐函数变换

基于上文

辐射照度环境映射 被定义为不定积分|w LL(w)max(0,cos<l(w),N>) dw其中法线方向N为自变量

定义域是球面的各个方向，即表面法线的方向N

值域是辐射照度

根据实验结果，基于辐射亮度环境映射积分求得辐射照度环境映射的频率始终是较低的（可以理解为辐射照度随着“角度”的变化较慢，即导数较小）

有人证明，使用9个基函数就可以满足任意辐射照度环境映射的精度要求

在实时渲染中，辐射照度环境映射只用于表示间接光，精度要求更低，可以使用4个基函数（1个常量基函数和3个线性基函数）

相对于立方体纹理，用球谐函数存储（只需要存储各个基函数的系数）辐射照度环境映射，可以大幅度降低内存

辐射照度环境映射可以看作是 辐射亮度环境映射即LL(w) 和max(0,cos<l(w),N>) 的卷积

由于在l和N之间的夹角大于90度后，max(0,cos<l(w),N>)的值为0，对积分的值不再有贡献，因此计算整个半球或计算整个球的积分，结果是相同的

卷积定理

函数卷积的傅里叶变换是函数傅里叶变换的乘积

球谐函数也具有类似的性质，可以根据球面卷积（Spherical Convolution）的相关定理快速求出辐射照度环境映射在各个基函数上的系数

1.将辐射亮度环境映射用球谐函数近似表示

积分求出各个基函数的系数

第i个基函数SHi的系数kLi = |w SHi(w) LL(w) dw

其中

w表示球面上的各个方向

w即目标函数/基函数的定义域

SHi(w)表示根据w求出第i个基函数SHi的函数值

在辐射亮度环境映射中w为根据v和n求得的“反射”（光路可逆）光线方向r

LL(w)表示根据 反射光线方向r 对辐射亮度环境映射采样求得相应方向的辐射亮度LL的过程

在计算机中，一般使用离散的方式近似求得积分

即 ∑ |w(t) SHi(r(t)) LL(r(t)) dw

其中

t表示辐射亮度环境映射中的各个纹素

r(t)表示相应纹素的中心所对应的反射光线方向r

w(t)表示相应纹素所对的立体角的大小

以上累加过程在计算机中可以通过并行归约求得

2.辐射照度环境映射可以看作是 辐射亮度环境映射即LL(w) 和cos<l(w),N> 的卷积

与卷积定理类似

在已经求得辐射亮度环境映射的各个系数kLi的前提下，辐射照度环境映射的各个系数kEi可以根据以下公式快速求得

kEi = (4π/(2Index + 1))^(1/2) \* kCi \* kLi

(4π/(2Index + 1))^(1/2)中的Index为频带索引

kCi即函数max(0,cos<l(w),N>)在各个基函数上的系数

在环境映射的分辨率固定在情况下，kLi中的w(t)和SHi(r(t))是固定的，可以和其它项合并进行预计算

根据目前的硬件水平，辐射亮度环境映射可以实时生成，将实时生成的辐射亮度环境映射与以上预计算的数据相乘，可以做到实时生成辐射照度环境映射

### SSAO（Screen Space Ambient Occlusion，屏幕空间环境光遮挡）

### HBAO（Horizon Based Ambient Occlusion，基于水平线环境光遮挡）

//LAπAO(p)

//ILM（工业光魔）使用 辐射照度环境映射+弯曲向量

//1/Z 插值 而非Z 实际上是正确的，透视投影本来就不是线性的 //双曲插值？？？

环境光基于辐射照度环境映射频率较低的事实，将辐射照度定义为常量（即相当于频率为0的极端情况）

Nvidia HBAO+

NVIDIA GameWorks DirectX Samples / Deinterleaved Texturing Sample

<https://developer.nvidia.com/gameworks-directx-samples>

NVIDIA ShadowWorks / HBAO+

<https://developer.nvidia.com/shadowworks>

Texture缓存的空间局部性

将纹理作为贴图 同一条采样指令 相邻像素的UV应当是相邻的

unroll并不一定更高效 虽然指令数减少，但是程序局部性下降，从而影响缓存的命中率

## 体素锥跟踪（Voxel Cone Tracing）

Cyril Crassin, Fabrice Neyret, Miguel Sainz, Simon Green, Elmar Eisemann. "Interactive Indirect Illumination Using Voxel Cone Tracing". SIGGRAPH 2011.

<http://research.nvidia.com/publication/interactive-indirect-illumination-using-voxel-cone-tracing>

## 传统的

### 平面反射（Planar Reflection）

### 环境光（Ambient）

环境光基于辐射照度环境映射频率较低的事实，将辐射照度定义为常量（即相当于频率为0的极端情况）

//Lambert表面

//LightMap/IrradianceMap

### 半球光照（Hemisphere Lighting）

相对于环境光将辐射照度定义为常量

半球光照用天空光、地面光和一个角度来定义辐射照度

### 补光（Fill Light）

在实时渲染中，补光即用只对漫反射产生影响（即不会对镜面反射产生影响）的光源（即直接光）模拟间接光

个人认为补光是一种比较山寨的技术，而且在逻辑层可以实现，并不会对引擎层产生任何影响

前照灯（Head Light）

附加到摄像机的光源，显然不会受到距离的影响

三点布光（ThreePoint Light）

用于影视领域

主体光（Key Light）

补光（Fill Light）

背光（Back Light）/轮廓光（Rim Light）

在实时渲染中，可以使补光与主体光恰好相反，在计算max(0,cos<l,n>)时，恰好相反，可以和主体光合并在同一个Shader中进行计算，从而提高效率，这种做法又被称为双向（BiDirectional）光

环境光（Ambient Light）

用于模拟环境光

# 完整全局光照（Full Global Illumination）

## 镜面反射

### 光线跟踪（Ray Tracing）

递归进行 直到碰撞漫反射表面或达到最大深度

## 漫反射

### 光能传递（Radiosity）

间接反射（InterReflection）->颜色溢出（Color Bleeding）

Enlighten SDK

假定间接光只来自表面的漫反射

### 光照贴图（LightMap）

由于历史原因（Quake II 1997），被称为光照贴图

原理上更应该被称作辐射照度贴图（Irradiance Map），基于辐射度算法进行预计算，用于存储EN

与环境映射的区别在于不基于N而是基于UV进行采样

一般情况下，并不会预计算LightMap与AlbedoMap相乘后的结果

因为LightMap的UV空间可能无法做到与AlbedoMap相同

比如在AlbedoMap中，可能不同位置的UV重复（比如将砖块贴图平铺到墙上），但是这些位置预计算的辐射照度是不同的

### 光能传递法线映射（Radiosity Normal Mapping）

相对于辐射照度环境映射，辐射照度贴图不具有方向性，即无法在运行时传入与法线方向相关的信息

辐射照度贴图根据Li max(0,cos<l,n>) 预计算EN，法线方向n在预计算时已经确定（一般使用顶点缓冲中的法线方向），无法在运行时改变

因此辐射照度贴图不能与高频的法线贴图结合使用

为使用法线贴图，可以考虑使用球谐函数表示辐射照度环境映射

根据前文，在只表示间接光的情况下，可以只使用4个系数

原理上更应该被称作谐函数（Harmonic）贴图

由于历史原因（Half-Life 2 2004），被称为光照贴图

值得注意的是，4个系数表示的是辐射照度环境映射而不是辐射照度

可以在纹理中存储系数，即可以做到每个UV都对应于一个不同的辐射照度环境映射

为了加快计算，辐射照度映射的定义域可以是切线空间中的法线方向，以允许在法线贴图中采样得到切线空间中的法线方向后直接传入到辐射照度环境映射

半球谐函数（Hemisphere Harmonic）

通常法线贴图中的法线仅在半球内（即切线空间中的Z坐标大于0），可以考虑用半球谐函数以进一步提高精度（仍需要4个系数，但精度进一步提高）

### 预计算辐射亮度传输（Precomputed Radiance Transfer）

MSDN Direct3D9 Precomputed Radiance Transfer

http://msdn.microsoft.com/en-us/library/bb147287

# 后处理（Post-Processing）

## FXAA（Fast approXimate Anti Aliasing，快速近似反走样）

Alexander Reshetov. "Morphological Antialiasing”. Intel Labs 2009.

Timothy Lottes. "FXAA". Nvidia WhitePaper 2011.

模式识别 Z U L

基于边缘检测（索贝尔算子）

只融合凸而不融合凹，否则不能消除锯齿

这是一种“现象”，具体原因不明

目前已知，在MLAA中，要对一个像素的4条边都进行模式识别并融合，但在FXAA中，只对对比度最高的一条边进行

## DOF（Depth Of Field，景深）/Defocus Blur

物理上应当是一个像素向周围像素分散（Scatter）

假定相邻像素的深度相同，改用聚集（Gather），与高斯滤波类似

Second, though the results are convincing in certain circumstances, the approach does not correctly approximate Depth of Field in scenes with a great deal of depth complexity. Depth-of-Field is actually a "scatter" effect -- an object out of focus will actually contribute to adjacent (non-occluded) pixels -- rather than a "gather" effect. Because of this, taking the blurred version of surrounding pixels is not correct if those pixels are either in focus or part of the foreground.

This problem cannot be completely avoided with the method suggested here; however, it can be greatly mitigated by creating two SATs: one for foreground objects, and one for background objects. Each SAT would be generated from a filtered version of the input image which only contained values for pixels in its assigned range band, and black everywhere else (meaning that those pixels would not contribute to the SAT). One would also have to add a "coverage" value to the alpha channel to distinguish between "empty" pixels and actual "black" ones, and then use that total value instead of the analytic area when computing the average of a summed area; however, this would be at minimal additional cost, given that we are already storing a four-channel format texture.

### 区域求和表（Summed Area Table）

[Hoobler 2011] Nathan Hoobler. “High Performance Post-Processing”. GDC 2011.

分歧（Divergence） 与SIMT有关

波阵面（WaveFront） 即流多处理器(Stream Multiprocessor) NV中32个核心 AMD中64个核心

同一SM中的核心必须执行同一指令，但不同SM中的核心可以执行不同指令

一般认为，线程组与SM对应

// Assume WAVE\_SIZE = wavefront size for the hardware (NV:32,AMD:64)

[numthreads(WAVE\_SIZE,2,1)]

void DivergeCS(uint3 tID : SV\_DispatchThreadID)

{

float val;

// Divergent

if (tID.x%2)

val = ComplexFuncA(tID);

else

val = ComplexFuncB(tID);

// Not divergent

if (tID.y%2)

val =+ ComplexFuncC(tID);

else

val =+ ComplexFuncD(tID);

Output(tID, val);

}

[flatten] if分支的两部分都进行计算，并不会提升性能

预加载到组共享内存 //组共享内存的访问速度高于全局内存

将一维的 前缀和（Prefix Sum）算法（实际上即parallel\_scan） 应用到二维

#version 450 core

layout (set=0, binding=0) uniform samplerBuffer srvIn;//vec4

#define GROUP\_SIZE 64

shared float sSums[GROUP\_SIZE];

layout(local\_size\_x = GROUP\_SIZE, local\_size\_y = 1, local\_size\_z = 1) in;

void PrefixSumCS()

{

sSums[gl\_LocalInvocationID.x] = texelFetch(srvIn,int(gl\_GlobalInvocationID.x)).x;//预加载到组共享内存

memoryBarrierShared();//内存依赖

barrier();//执行依赖

for(int t=1; t<GROUP\_SIZE; t\*=2)//时间复杂度 log2N 可以unroll

{

sSums[gl\_LocalInvocationID.x] +=

(gl\_LocalInvocationID.x >= t)?(sSums[gl\_LocalInvocationID.x-t]):0.0f;//if flatten

memoryBarrierShared();//内存依赖

barrier();//执行依赖

}

//使用结果

//比如，求平均值

}

## TAA（Temporal Anti Aliasing，时域反走样）/ 运动模糊（Motion Blur）

Matthew Eldridge, Homan Igehy, Pat Hanrahan. "Pomegranate: A Fully Scalable Graphics Architecture." SIGGRAPH 2000.

<https://graphics.stanford.edu/papers/pomegranate/pomegranate.pdf>

Matthaeus Chajdas. "Unlock the Rasterizer with Out-of-Order Rasterization." AMD GPUOpen 2016.

<https://gpuopen.com/unlock-the-rasterizer-with-out-of-order-rasterization/\>

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

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

MSAA FXAA——空间性（Spatial）反走样

Motion Blur——时间性（Temporal）反走样（TAA）

Nvidia TXAA是MSAA和TAA的整合，可以考虑用FXAA和TAA实现同样的效果

## TXAA

TXAA即Nvidia对MSAA+TAA的改进版本

其中：TAA（时域反走样）即Motion Blur（运动模糊）

而FXAA和MSAA都属于SAA（空间上的反走样），与TAA并没有任何关系，由于FXAA比MSAA要高效得多，可以认为MSAA已经被淘汰

模拟信号--采样（Sample）-->数字信号--过滤（Filter）🡪模拟信号

MSAA实现

不可用原因 应当对BRDF(l,v)En进行插值 而非GBuffer中的值

SV\_COVERAGE

pixel

subpixel 即采样点

超采样/过采样

antialiasing algorithms that compute more than one full sample per

pixel are called supersampling (or oversampling) methods.

累加缓冲

可以用 几何着色器+SV\_RenderTargetIndex 实现

pixel shader 对pixel计算一次 而非对各个subpixel计算一次

MSAA（Multi）

区分 着色样本 和 位置样本

centroid sampling /centroid interpolation（GPU硬件完成）

着色样本的位置会随着改变，

CSAA（Coverage） 将SV\_Coverage与采样分离

FXAA(Fast approXimate Anti Aliasing，快速近似反走样)

SW S SE

W M E

NW N NE

采样得到以上9个像素的方式

FXAA\_GATHER4\_ALPHA

在信号处理中 时域 对应 频域

在反走样中 时域 对应 空间上（即空域）

时间上走样

空间上走样

非真实感渲染（Non Photorealistic Rendering）

# 轮廓边缘渲染/“勾边”

计算机图形学中关于轮廓的定义

## 表面角度

当dot(v,n)接近0（即小于某个阈值时）时，输出轮廓颜色

否则，discard（节省带宽（GPU与内存通信））

根据dot(v,n)采样1D纹理（避免分歧分支）

注：当dot(v,n)小于0时，三角形为被背面，一般被剔除

以辐射亮度环境映射（Environment Map）实现，在边缘处有一个与轮廓颜色相同的

## 过程式几何体（Procedural Geometry）

1.绘制正面

2.绘制背面（深度偏移 Nearer）

# 第三部分 过程式动画 / 物理模拟

## 传统的-基于力的模拟（Force-Based Simulation）

### 布料（Cloth）

NVIDIA Direct3D SDK 10.5 Code Samples / Cloth Simulation

<https://developer.download.nvidia.com/SDK/10.5/direct3d/samples.html#Cloth>

Tae-Yong Kim, Nuttapong Chentanez, Matthias Müller-Fischer. "Long Range Attachments - A Method to Simulate Inextensible Clothing in Computer Games." Eurographics Association 2012.

<https://developer.nvidia.com/research>

PhysX Clothing

<https://docs.nvidia.com/gameworks/content/gameworkslibrary/physx/guide/Manual/Cloth.html>

NVIDIA Clothing

<https://docs.nvidia.com/gameworks/content/gameworkslibrary/physx/nvCloth/UsersGuide/Index.html>

### 破坏物（Destruction）

Matthias Muller, Nuttapong Chentanez, Tae-Yong Kim. "Real Time Dynamic Fracture with Volumetric Approximate Convex Decompositions." SIGGRAPH 2013.

<https://developer.nvidia.com/research>

NVIDIA Blast

<https://developer.nvidia.com/destruction>

### 流体（Fluid）

NVIDIA Flow

<https://developer.nvidia.com/nvidia-flow>

## 次世代-基于位置的模拟（Position-Based Simulation）

### 粒子（Particle）

Jan Bender, Matthias Müller, Miles Macklin2. "Position-Based Simulation Methods in Computer Graphics." EUROGRAPHICS 2015.

<https://developer.nvidia.com/research>

Miles Macklin, Matthias Muller, Nuttapong Chentanez, Tae-Yong Kim. "Unified Particle Physics for Real-Time Applications." SIGGRAPH 2014.

<https://developer.nvidia.com/research>

NVIDIA Flex

<https://developer.nvidia.com/flex>

# 第四部分 图形调试

## Visual Studio 图形分析器

### UE4 Shader Debug

//Engine\Source\Developer\Windows\ShaderFormatD3D\Private\D3D11ShaderCompiler.cpp

1.D3D10\_SHADER\_DEBUG

1-1.DEBUG\_SHADERS:=1

1-2.CFLAG\_Debug => r.Shaders.Optimize:=0 => CvarOptimizeShaders:=0

2.D3DStripShader

CFLAG\_KeepDebugInfo => r.Shaders.KeepDebugInfo:=1 => CvarKeepShaderDebugData:=1

3.

CFLAG\_ForceRemoveUnusedInterpolators => r.D3D.RemoveUnXXXXXX:=0 => CVarD3DRemoveUnusedInterpolators:=0

DEBUG\_SHADERS in //Engine\Source\Developer\Windows\ShaderFormatD3D\Private\D3D11ShaderCompiler.cpp

CvarXXXXXX in //Engine\Source\Runtime\Engine\Private\ShaderCompiler\ShaderCompiler.cpp

## RenderDOC

在Win10中支持CallStack

DIA2::GetModule

复制msdia140.dll并用NoRegCoCreate

修改renderdoc.vcxproj

修改AfterBuild中的各Copy项

# 第五部分 性能优化

<http://developer.nvidia.com/dx12-dos-and-donts>

## Draw Call Batch与乱序光栅化

MSDN / Direct3D 9 / Accurately Profiling Direct3D API Calls

<https://docs.microsoft.com/en-us/windows/desktop/direct3d9/accurately-profiling-direct3d-api-calls>

一般认为Draw Call Batch是为了解决CPU瓶颈，MSDN上的Direct3D9文档中对此进行了相关的讨论。

但实际上，Draw Call Batch还可以提高GPU绘制三角形时的并行度，还可以在某种程度上解决GPU瓶颈。

我们知道GPU在本质上是一个并行的系统，比如，“不同Fragment Shader之间不具有任何依赖”可以认为是GPU并行性的一个典型代表；但是由于应用层的某些逻辑的需要，导致了GPU在一定程度上具有串行性。

比如，经典的绘制透明物体的算法需要在应用层将物体从前到后或从后到前排序，并依次调用Draw Call，这意味着“在结果上”GPU绘制三角形的顺序应当与应用层调用Draw Call的顺序一致。

这意味着应用层按顺序调用Draw Call其实隐含着某种依赖关系。

我们之所以强调“在结果上”是因为：在实际实现中，GPU仍并行地绘制三角形，只不过在输出结果前进行了相关的处理来保证Draw Call靠后的三角形一定覆盖Draw Call靠前的三角形。

"Real Time Rendering Fourth Edition Chapter 23 Graphics Hardware Section 9 Architecture"中对GPU处理三角形时所进行的排序进行了详细的讨论。

显然同一个Draw Call中的三角形不需要排序（应用层的逻辑不关心它们的绘制顺序），因此Draw Call Batch可以提高GPU的并行度，从而可以在某种程度上解决GPU瓶颈。

Matthaeus Chajdas. "Unlock the Rasterizer with Out-of-Order Rasterization." AMD GPUOpen 2016.

<https://gpuopen.com/unlock-the-rasterizer-with-out-of-order-rasterization/>

AMD最近给出了一个乱序光栅化的Vulkan扩展，试图通过摒弃排序来提升GPU的并行度。

//PS：AMD的乱序光栅化扩展说明了OIT（顺序无关透明）算法的重要性，是时候启用一波OIT算法了（<http://research.nvidia.com/publication/phenomenological-scattering-model-order-independent-transparency>）233333333

但是，通过摒弃排序来提升GPU并行度的方式貌似只适用于Sort-Last Fragment架构的PC平台的GPU；Sort-Middle Tiled架构的Mobile平台的GPU在Fragment Shader之前进行排序，并通过FPK(Forward Pixel Kill)的方式对Fragment Shader的执行次数进行优化，想要摒弃排序貌似是不大可能的。

## Update Constant Buffer的驱动实现

在使用经典的图形API（比如Direct3D11）时，一个典型的场景渲染过程可以用伪代码描述如下：

Map Constant Buffer For View //For example, View Transform, Projection Transform

for each Mesh

{

Map\_Write\_Discard Call For [Constant Buffer For Per Mesh Data] //For example, Model Transform

...... //For example, CPU Write to It

Unmap Call For It

Draw Call For Mesh

}

尽管应用层绘制了很多个Mesh，但是应用层只创建了一个Constant Buffer。

那么问题来了！当我们在绘制第2个Mesh时，该Constant Buffer中存放着第1个Mesh的相关数据，很可能正在被GPU读取；但是，我们仍可以用CPU将第2个Mesh的相关数据写入到该Constant Buffer中，并不会与GPU的读取发生冲突，这是为什么呢？

注：虽然下文中的讨论是针对Direct3D的Map\_Write\_Discard，但是，在使用OpenGL时，应用层时也是在只创建一个UniformBuffer的情况下绘制多个Mesh；这意味着OpenGL中的DRAW\_DYNAMIC的实现也是如此；虽然按照OpenGL规范，DRAW\_DYNAMIC只是应用层的Hint，并不保证驱动实现，但是应用层的用法已经假定了驱动的实现（否则应用层在绘制时需要分配和Mesh个数一样多的Uniform Buffer了（滑稽脸））。

MSDN / Direct3D11 / How to: Use dynamic resources

<https://docs.microsoft.com/en-us/windows/desktop/direct3d11/how-to--use-dynamic-resources>

MSDN上的Direct3D11文档中对Map\_Write­\_Discard的实现方式进行了说明：当应用层以Write\_Discard的方式Map一个Buffer时，驱动会返回一块新的内存区域给应用层，从而避免了CPU与GPU之间的读写冲突。

Holger Gruen. "Constant Buffers without Constant Pain". NVIDIA GameWorks Blog 2015.

<http://developer.nvidia.com/content/constant-buffers-without-constant-pain-0>

NVIDIA GameWorks Blog上的一篇文章对Map\_Write\_Discard的驱动实现进行了说明：驱动根据某个配额（目前是128MB）预先分配一大块内存，当应用层调用Map\_Write\_Discard时，驱动以类似于线性分配器的方式不断将指针后移，并将新的内存区域返回给应用层。

同时该文章中指出：驱动将应用层对UpdateSubresource的调用转发到Map\_Write\_Discard，UpdateSubresource和Map\_Write\_Discard在本质上是等价的。

//分界线---------------------------------------------------------------------------------------------------------------------------

MSDN / Direct3D12 / Fence-Based Resource Management / Ring Buffer

<https://docs.microsoft.com/en-us/windows/desktop/direct3d12/fence-based-resource-management>

在使用次世代的图形API（比如Direct3D12）时，Map\_Write\_Discard不再被API支持，需要在应用层实现相关的语义，MSDN上的Direct3D12文档中给出了一种基于RingBuffer的实现方式（本质上即模仿驱动实现Map\_Write\_Discard的方式，只不过配额（即上文中的128MB）可以根据应用层自身的实际情况进行定制）。

## Visual Studio GPU性能探查器

## 着色器资源绑定

Matt Pettineo. "Bindless Texturing For Deferred Rendering And Decals." Blog at WordPress.com 2016.

<https://mynameismjp.wordpress.com/2016/03/25/bindless-texturing-for-deferred-rendering-and-decals/>

GL\_NV\_bindless\_texture

//1. Fetch From System Memory

Old Object Bind

Application

| Object Name

Driver <—Fetch Object State(For Example GPU Address)—System Memory

| GPU Address

GPU Command Buffer

New Object Bind

Application

| GPU Address //Driver Feedback GPU Address To Application When Object Creating

Driver //No Longer Fetch From System Memory

| GPU Address

GPU Command Buffer

//2.Memory Resident

//To Do

Another Issue: Reduce Texture Sampling的次数

texture array //Limit Maxium Of Texture Count

virtual mapping //Too Complex

Direct3D12

根签名

根常量\根描述符\描述符表

Microsoft Direct3D Shader Compiler——fxc.exe——编译输出字节码

ID3D12Device::CreateRootSignature

描述符堆/创建

ID3D12Device::CreateDescriptorHeap

//Direct3D12中的CPU描述符在Vulkan中对应于FrameBuffer

描述符堆/写入

ID3D12DescriptorHeap::GetGPUDescriptorHandleForHeapStart

ID3D12Device::GetDescriptorHandleIncrementSize

ID3D12Device::CreateConstantBufferView

ID3D12Device::CreateShaderResourceView

ID3D12Device::CreateUnorderedAccessView

命令列表/绑定

//根签名

ID3D12GraphicsCommandList::SetGraphicsRootSignature

ID3D12GraphicsCommandList::SetComputeRootSignature

//根常量（与描述符堆无关）

ID3D12GraphicsCommandList::SetGraphicsRoot32BitConstant

ID3D12GraphicsCommandList::SetComputeRoot32BitConstant

//描述符堆

ID3D12GraphicsCommandList::SetDescriptorHeaps

//着色器资源 –> 引用描述符堆

ID3D12GraphicsCommandList::SetGraphicsRootConstantBufferView

ID3D12GraphicsCommandList::SetGraphicsRootShaderResourceView

ID3D12GraphicsCommandList::SetGraphicsRootUnorderedAccessView

ID3D12GraphicsCommandList::SetComputeRootConstantBufferView

ID3D12GraphicsCommandList::SetComputeRootShaderResourceView

ID3D12GraphicsCommandList::SetComputeRootUnorderedAccessView

ID3D12GraphicsCommandList::SetGraphicsRootDescriptorTable

ID3D12GraphicsCommandList::SetComputeRootDescriptorTable

Vulkan

描述符集布局

vkCreateDescriptorSetLayout

描述符集布局\*1 -> 描述符集布局绑定\*n

流水线布局

vkCreatePipelineLayout

流水线布局\*1 -> 描述符集布局\*n + 推送常量\*n

GLSL源代码

layout(set=\_, binding=\_) \_\_\_\_

layout(push\_constant) \_\_\_\_

set->描述符集布局

bind->描述符集布局绑定

push\_constant->推送常量

描述符池

vkCreateDescriptorPool

描述符集/创建

vkAllocateDescriptorSets

描述符集 1<–>1描述符集布局 个人认为粒度有问题？？？

描述符集/写入

vkUpdateDescriptorSets

流水线

//流水线布局

vkCreatePipeline

命令缓冲/绑定

//流水线布局 + 描述符集

//流水线布局 应当兼容 创建时的流水线布局

//描述符集 应当兼容 流水线布局中的描述符集布局

vkCmdBindDescriptorSets

//推送常量（与描述符集无关）

vkCmdPushConstants

Vulkan — Direct3D12

推送常量 — 根常量

X — 根描述符

描述符集布局 — 描述符表

流水线布局 — 根签名

描述符池 — 描述符堆

描述符集 — 描述符堆（中的区间）

流水线 — 流水线状态对象

Descriptror Binding — GPU Descriptor Handle

Vulkan Shader Resource Binding

<http://developer.nvidia.com/vulkan-shader-resource-binding>

bound DescriptorSets stay active as long as the PipelineLayout for that binding slot is matching

//即同一流水线布局可以在不同的流水线之间复用

//图形流水线和计算流水线之间互不干扰

// example for typical loops in rendering

for each view {

bind view resources // camera, environment...

for each shader {

bind shader pipeline

bind shader resources // shader control values

for each material {

bind material resources // material parameters and textures

for each object {

bind object resources // object transforms

draw object

}

}

}

}

vkCmdBindDescriptorSets

尽可能地复用绑定，减少vkCmdBindDescriptorSets的调用次数

可以用firstSet和descriptorSetCount调节

描述符集用于控制粒度

改变频率不同的资源组织到不同的描述符集（Descriptor Set）

Advanced use of Descriptor Tables

<http://msdn.microsoft.com/en-us/library/dn859250>

改变频率不同的资源组织到不同的描述符表（Descriptor Table）

Differences in the Binding Model from Direct3D 11

[http://msdn.microsoft.com/en-us/library /dn859252](http://msdn.microsoft.com/en-us/library%20/dn859252)

Driver Resource State Tracking Separated From Binding

资源屏障

比如 延迟渲染中 RTV->SRV资源屏障中

CPU GPU Mapped Memory Synchronization Separated From Binding

在资源Map但未UnMap的情况下，GPU根据相关资源是否被绑定确定是否等待CPU

## 内存管理

Vulkan Memory Management

<http://developer.nvidia.com/vulkan-memory-management>

Adam Sawicki. "Memory management in Vulkan and DX12." GDC 2018.

## Sort-Last Fragment Architecture

Christoph Kubisch. "Life of a triangle - NVIDIA's logical pipeline" NVIDIA GameWorks Blog 2015.

<https://developer.nvidia.com/content/life-triangle-nvidias-logical-pipeline>

## Sort-Middle Tiled Architecture

Matthew Eldridge, Homan Igehy, Pat Hanrahan. "Pomegranate: A Fully Scalable Graphics Architecture." SIGGRAPH 2000.

<https://graphics.stanford.edu/papers/pomegranate/pomegranate.pdf>

Matthaeus Chajdas. "Unlock the Rasterizer with Out-of-Order Rasterization." AMD GPUOpen 2016.

https://gpuopen.com/unlock-the-rasterizer-with-out-of-order-rasterization/

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

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

Kurt Akeley, Pat Hanrahan. "Parallelism and communication." Stanford University, CS448A, Real-Time Graphics Architectures, Fall 2001.

<http://www.graphics.stanford.edu/courses/cs448a-01-fall/lectures/lecture9/parallel.2up.pdf>

SubpassInput (Vulkan) <=> ProgrammableBlending (Metal)

Image Usage Transient Attachment (Vulkan) <=> StorageModeMemoryLess (Metal)

Vulkan RenderPass和Mobile GPU Tiling Architecture

Clear操作 阻止从External Memory加载

MSAA开销较低

可以用MSAA取代FXAA

基于MSAA的OIT可用

基于MSAA的Alpha-Coverage可用

Resolve操作在On-Chip Memory（可以理解为Cache）中完成

MSAA Texture在On-Chip Memory中 每个像素可以有多个采样点写入

Non-MASS Texture在External Memory中 每个像素只有1个采样点写入

缺点：

需要整个场景的数据

如果三角形个数过多，会导致MultiPass发生

## Staging Buffer

Vulkan Quake Port

<https://github.com/Novum/vkQuake>

R\_StagingAllocate

Vulkan DOOM 3 Port

<https://github.com/DustinHLand/vkDOOM3>

状态机 用 函数指针 取代 分支

AMD GPUOpen Vulkan Memory Allocator

github google filament