面向模拟演练系统的延迟渲染框架①

唐 宇[1,2], 于 放[2], 孙 咏[2], 王丹妮[3]

[1](中国科学院大学, 北京 100049)
[2](中国科学院 沈阳计算技术研究所, 沈阳 110168)
[3](国网辽宁省电力有限公司 信息通信分公司, 沈阳 110000)

摘 要: 模拟演练系统为企业和应急部门提供了强大的服务, 然而目前的模拟演练系统中存在缺乏现代图形流水线, 多光源情况下存在场景渲染计算量过多, 不能充分利用硬件而导致的渲染效果差等问题. 为了解决这些问题, 本文针对模拟演练系统, 设计和实现了延迟渲染流水线, 它依据 Blinn-Phong 光照模型, 构建了一种压缩的 G-Buffer 布局, 并使用 Light Volume 来对光源进行建模. 最后实验表明, 在多光源情况下, 此文中提出的延迟渲染比传统的正向渲染有更高的效率.

关键词: 模拟演练系统; 延迟渲染; 正向渲染; G-buffer; Light Volume

Deferred Shading Framework for Simulation Drilling System

TANG Yu[1,2], YU Fang[2], SUN Yong[2], Wang Dan-Ni[3]

[1](University of Chinese Academy of Sciences, Beijing 100049, China)
[2](Shenyang Institute of Computing Technology, Chinese Academy of Sciences, Shenyang 110168, China)
[3](LiaoNing Electric Power Company Limited, Shenyang 110000, China)

Abstract: The Simulation drilling system provides a good service for the enterprises and the emergency departments at present. However, it still has quite a few drawbacks. Such as the lack of modern graphic pipeline support, the excessive computation of scene rendering on the condition of multiple lights, the poor rendering effects caused by that it cannot make full use of hardware etc. In order to solve those problems, this paper is designed and implemented with a deferred shading pipeline for simulation drilling system. This pipeline is based on the Blinn-Phong lighting model, which builds a compressed G-buffer layout and uses Light volume to model the light source. The final experiment indicates that under the multiple lights circumstances, the deferred shading expressed in this paper will be more efficient than the conventional forward shading.

Key words: simulation drilling system; deferred shading; forward shading; G-buffer; Light Volume

1 引言

伴随着中国工业的快速发展, 与之对应的安全生产问题越来越突出, 2015 年天津危险化学品仓库爆炸事故的发生, 不仅给我国造成了巨大的经济损失, 更夺走了许多无辜的生命. 这个悲剧的爆炸事件让我们更加深刻地认识到了安全生产的重要性. 在这样的背景之下, 三维模拟演练系统为企业和应急部门提供了强大的服务, 通过对于国家规定的多级危险源进行不同层次的建模, 将真实世界的情况映射到三维的虚拟世界中. 由于三维模拟演练系统的参数配置功能十分强大, 为模拟演练提供了十分灵活的场景演示和处理方式. 三维模拟演练系统的引进提高了应急的效率并且降低了成本, 可以有效加强安全部门和生产部门的合作, 对于团队建设和安全生产的完成具有重大的意义[1].

然而, 目前的三维模拟演练系统中使用正向渲染, 存在缺乏现代图形流水线, 多光源渲染效率低下, 不能充分利用硬件而导致的渲染效果差等问题. 而延迟

① 收稿时间:2016-01-12;收到修改稿时间:2016-03-03 [doi: 10.15888/j.cnki.csa.005350]

渲染可由正向渲染中光照计算的时间复杂度为 O(m*n)降低到延迟渲染中的 O(m+n)(m 代表场景中几何体数量, n 代表了场景中光源的数量). 虽然针对正向渲染的改进如基于 tiled[2]的渲染与 forward+[3], 但它们仍然缺乏像延迟渲染框架这样支持多光源的特性, 延迟渲染框架可以与局部光照计算和全局光照的计算紧密的结合在一起, 并在延迟渲染流水线的最后可以应用各种后处理技术.

因此, 本文首先设计了延迟渲染的流水线, 并分析了 Blinn-Phong 光照模型, 依据光照模型和三维模拟演练系统的特性设计了一种高效的 G-buffer 布局, 并使用 Light Volume 对光源进行了建模. 文章的第三章实现了模拟演练系统的延迟渲染框架, 并与正向渲染做了对比. 文章的最后论述了延迟渲染中全局光照算法的集成, 并展望了延迟渲染的未来.

2 延迟渲染

早期的固定管线架构的 GPU, 由于其计算能力的限制, 只可以使用有限的光源来照亮场景, 并且对于光照的计算不够灵活. 现代的 GPU 由于可编程管线的广泛使用, 在着色器中使用不同的光照算法提供了可能[4]. 延迟渲染的一大优势就是光照计算与场景的复杂度相分离, 可以大大提高场景中光照的速度.

2.1 延迟渲染流水线

延迟渲染的流水线如图[1], 首先使用着色器对场景渲染, 不同于正向渲染的情况, 这里只进行基本的场景渲染并将结果输出到G-Buffer进行缓冲区的填充. 之后以所有光源的 Light Volume 为几何体进行光照计算, 并将结果填充到 Light-buffer. 在延迟渲染流水线的最后, 对整个屏幕的每一个像素点进行一次着色运算, 使用 G-buffer 和 Light-buffer 为输入, 将对应的颜色值混合在一起. 同时最后的全屏渲染也可以应用一些后处理技术,如 HDR[5], SSAO[6].

图 1 延迟渲染流水线

2.2 Blinn-Phong 光照模型

尽管可以通过物理建模的方式解决光线和材质的相互作用问题, 但这种方式计算量太过庞大, 并不适合在实时渲染中使用. 一种可行的方式是使用 Phong 模型, 它使用环境光, 漫反射和镜面反射来对光和几何体的表面进行光照建模即:

$$I = I_a + I_d + I_s \tag{1}$$

为了计算这三个分量, 我们假定对于几何体表面某一点 p, 有 4 个向量, p 点处的法向量 n, p 点指向光源的向量 l, p 点指向观察者的向量 v, 沿着 l 入射在 p 点处的反射向量 r. 设 f 衰减值为:

$$f = \frac{1}{a + bd + cd^2} \tag{2}$$

其中, a、b、c 为常数, d 为光源与目标点之间的距离. 环境光分量为:

$$I_a = K_a L_a \tag{3}$$

漫反射光分量为:

$$I_d = fK_d (l \cdot n) L_d \tag{4}$$

镜面反射光分量为:

$$I_s = fK_s L_s (r \cdot v)^{\alpha} \tag{5}$$

为了避免计算反射向量 r, 在 Blinn–Phong 模型中使用半角向量来计算镜面反射光分量:

$$h = \frac{l + v}{|l + v|} \tag{6}$$

$$I_s = fK_s L_s (h \cdot n)^{\alpha} \tag{7}$$

由式(1)、(3)、(4)、(7)可得 Blinn–Phong 模型中的光照计算公式为:

$$I = f\left(k_d L_d \max(l \cdot n, 0) + k_s d_s \max\left((h \cdot n)^{\alpha}, 0\right)\right) + k_a L_a \tag{8}$$

ka、kd、ks 分别代表了环境光、漫反射、镜面反射的材质系数, La、Ld、Ls 代表了光源的环境光、漫反射、镜面反射的光照强度, 指数 a 是高光系数. 在多光源下, 只对漫反射和镜面反射做累加, 并存储在 Light-Buffer 中. 而环境光的计算放在最后的全屏渲染中.

2.3 G-Buffer 的优化存储

G-Buffer 全称是 Geometry Buffer, 它是包含了场景中颜色, 法线, 顶点坐标, 深度值的多个缓冲区集合, 通常每一个子缓存区是一块纹理. 由于 G-Buffer 包含了多个依赖于分辨率的纹理数据, 在渲染中可能占据了大量的带宽. 例如为了保持精度常用纹理的内部数据格式为 RGBA16, 则每一个像素点需要 64bit 的数据. 对于 1024*768 大小的一块纹理, 共需要 50331648bit = 6MB 的数据. 传统 G-Buffer 按照表 1 的布局方式, 它共占据 20.25MB 的显存. 在一个要求 60 帧的场景下, G-Buffer 每秒要消耗 1.215GB 的带宽! 所

以对于 G-Buffer 的布局需要慎重的考虑.

表 1　传统 G-Buffer 布局

数据类型	数据格式	RGBA 通道位数				总 bit
颜色值	RGBA16	16	16	16		64
高光系数					16	
法线	RGBA16	16	16	16	空	64
顶点坐标	RGBA16	16	16	16	空	64
深度值	DEPTH24	24				24

首先注意到的是 RGBA16 格式的像素点占据了 64bit 的数据, 如果改为 RGBA8 的数据, G-Buffer 的大小将变为原来的 1/2, 但是随着精度降低, 图形可能失真, 下文中提出了一种压缩 G-Buffer 布局的方式, 如表 2 所示, 在一个要求 60 帧的场景下, 压缩的 G-Buffer 每秒只需要消耗将近 0.495GB 的带宽, 效率提高了 59%.

表 2　压缩后 G-Buffer 布局

数据类型	数据格式	RGBA 通道位数				总 bit
颜色值	RGBA8	8	8	8		32
材质 ID					8	
法线	RGB10A2	10	10			32
高光系数				10	空	
深度值	DEPTH_24	24				24

2.3.1 顶点坐标和深度值的存储

与传统的 G-buffer 布局不同, 优化的 G-buffer 布局并不存储顶点坐标, 可以根据顶点的深度信息来重构顶点坐标.

为了便于理解, 在光照计算阶段, 假设摄像机镜头发出了无数的射线, 如果射线与某一面片的顶点相交, 则对于摄像机镜头来说, 这个顶点是可见的, 则最后在屏幕上对应像素显示的为相交顶点. 基于以上的理解, 摄像机镜头的射线可以看为方向向量, 则对于屏幕上的像素点一定有一个与其对应的方向向量, 并且某一个面片的顶点在这个方向向量所指的方向上.

在延迟光照中, 顶点信息主要用来计算光照, 将光源看为几何体, 对于可能在几何体内的顶点才重构其顶点坐标, 来计算光照, 在几何体外的顶点, 默认为光源对其不产生影响, 重构其顶点信息是没有必要的. 对于几何体最后影响的像素中, 每一个像素都有一个对应的方向向量, 根据在这个方向向量上几何体的顶点, 可以轻易地重构面片顶点坐标, 如图 2 所示, 在视锥体俯视图中, C 为照相机位置, P1 为当前几何体的顶点, P2 为需要重构的顶点, 由于 $FarClip \neq 0$ 且 $P1_z \neq 0$ 则有

$$\mu_1 \overrightarrow{CV} = \overrightarrow{CP1} \tag{9}$$

观察坐标系下, $C=(0,0,0)$, 且 $V_z = -FarClip$ 则

$$\mu_1 = \frac{P1_z}{V_z} = \frac{P1_z}{-FarClip} \tag{10}$$

由式(1)、(2)可得

$$V_x = P1_x \frac{-FarClip}{P1_z}, V_y = P1_y \frac{-FarClip}{P1_z} \tag{11}$$

同理 $\mu_2 \overrightarrow{CV} = \overrightarrow{CP2}$, 为了重构 $P2$ 的值, 只需要知道 μ_2 的值. 且

$$\mu_2 = \frac{P2_z}{-FarClip} \tag{12}$$

对于每一次的几何体顶点坐标的重构, 如果知道 μ_2 的值, 只需要每次重新计算向量 CV, 然后与 μ_2 相乘即可. 可以观察到 $P2_z$ 实际是在填充 G-buffer 时观察坐标系下的深度值, 由此可以将 μ_2 直接存储在深度缓冲区中, 可以避免重复计算.

上述计算需要在光照计算的片断着色器中进行, 顶点 P1 通过几何体的顶点在顶点着色器中通过插值而得出.

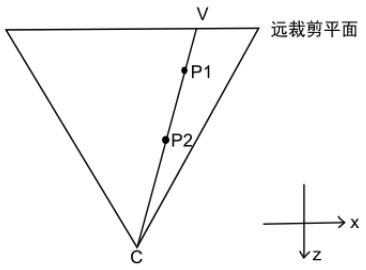


图 2　视锥体俯视图

G-buffer 中深度值使用一个内部数据格式为 R24 的纹理来存储, 即深度值用 24 位的浮点数来表示, 这样可以保持精度. 在填充深度缓冲区时, 依据重构的要求将在观察坐标系下的顶点 z 值除以远裁剪面值 $depth = z / -FarClip$, 并将 depth 存储于缓存区.

2.3.2 法线的存储

法线中相应的每一项数据使用 8bit 来存储可能会带来精度的问题, 这里可以使用等面积方位投影来将法线由三项压缩为两项, 法线编码与解码的步骤如表 3. 由于法线编码后只需要存储两个浮点数, 为了保持精度, 纹理的内部数据格式选为 RGB10A2, 即 RGB 三个通道占用 10bit, A 通道占用 2bit. RG 通道存储编码后的法向量, B 通道可以存储其他的参数, 如高光系数.

这样纹理占用的大小不变, 但是法线的精度上升, 并多存储了一项信息.

表 3 法线的编码与解码

```
1.  //对法线的编码操作
2.  vec2 encode (vec3 n)
3.  {
4.      return n.xy / sqrt(n.z * 8.0 + 8.0) + 0.5;
5.  }
6.  //法线的解码操作
7.  vec3 decode(vec2 enc)
8.  {
9.      vec2 fenc = enc * 4.0 - 2.0;
10.     float f = dot(fenc,fenc);
11.     float g = sqrt(1.0 - f / 4.0);
12.     vec3 normal;
13.     normal.xy = fenc * g;
14.     normal.z = 1.0 - f / 2.0;
15.     return n;
16. }
```

2.3.3 颜色的存储

对于颜色使用 RGB 三个通道都是 8bit 是足够的, 即红, 绿, 蓝每一个通道都有 256 个渐变值, 这样大概能表示 1680 万种颜色, 这对于大多数应用场景是足够的, 剩下的 8bit 的 A 通道可以用来存储材质 ID.

2.3.4 材质属性的存储

传统 G-buffer 布局下材质属性可以编码在 G-buffer 中, 由于 G-buffer 已经十分巨大, 这样的解决方式并不可取. 而在压缩后的 G-buffer 布局中, 为了在光照计算中使用材质属性, 可以在某一场景初始化时, 将场景中的所有材质与一个材质 ID 关联起来, 在 G-buffer 中先将材质 ID 输出, 在进行光照计算时, 依据材质 ID 来索引材质属性. 将材质数据上传到显存有两种方式, 其一可以将所有的材质属性填充到一个一维的纹理, 在着色器中对其采样取出数据. 另一种方式是在着色器中使用一个 uniform block, 通过 Uniform Buffer Objects 来上传数据. 通常后一种方式有更快的速度, 这是由于 Uniform Buffer 通常是显存中局部的缓冲区, 而纹理是显存中全局缓冲区.

当然, 由于材质 ID 所占用的 a 通道只有 8 位, 理论上同一场景材质数量不能大于 256, 但是在三维模拟演练系统中, 这样的数量是远远大于需求的.

2.4 **Light Volume**

Light Volume 是将光源看为一个几何体, 只对几何体影响到的像素进行光照计算. 很自然的, 对于点光源可以使用球体, 聚光灯可以使用棱锥, 环境光源和平行光源可以使用全屏四边形. Light Volume 的一个重要特性就是对光照计算的优化, 如图 3 所示, 可以对光源进行视锥体检测, 所以光源 1 不参与运算. 同时, 由于橘色球体并没有光源和它相交, 所以它也不参与光照的计算.

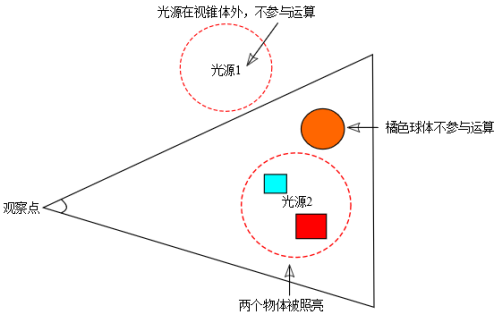


图 3 Light Volume 对光照计算的优化

将 Light Volume 进行光照计算的漫反射分量和镜面反射分量分别写到两个缓冲中, 并开启 blend, 对多个 Light Volume 的计算结果进行累加.

3 实验及其结论

本文在基于模拟演练的系统之上配合 OpenGL Shading Language[7]实现了上述的延迟渲染框架, 并在多光源场景下与传统的正向渲染就行了对比. 工作站使用了 Intel Core I5 的处理器, 并配有 8GB 的内存, 显卡为 GTX 660.

3.1 **G-Buffer 可视化**

本文实现了优化的 G-buffer 存储形式, 如图 4、5、6 所示, 使用了模拟演练中经典的工厂场景, 分别为优化后的 G-buffer 中法线与高光系数、颜色与材质 ID、线性的深度值的可视化.

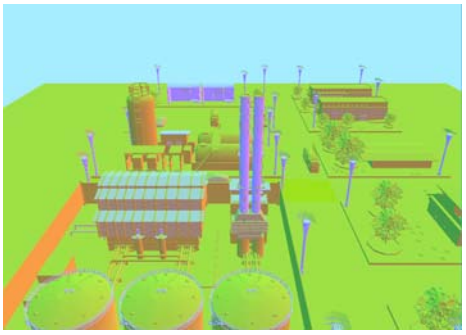


图 4 RGB10A2 的可视化

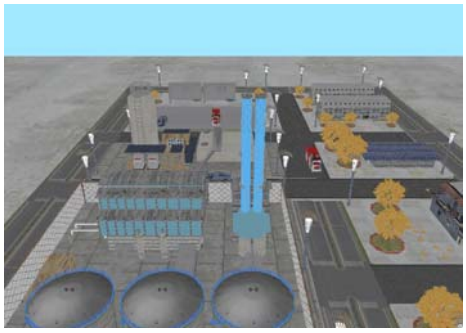


图 5　RGBA8 的可视化

图 6　DEPTH_24 的可视化

为了测试法线编码后质量，进行了对比测试，图 7 为不经过编码的原始法线，图 8 为编码后再解码的法线. 可见肉眼无法直接观测到显著的区别，因此图 9 为图 8、9 差异的热度图，可以观察到法线编码解码后的变化十分微小，这样的情况在三维模拟演练中任何的任何场景都是不存在问题的.

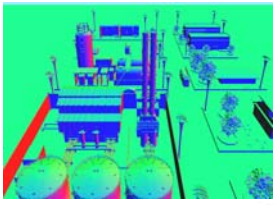
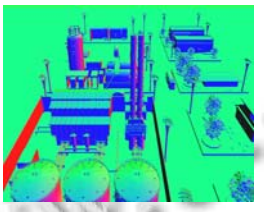


图 7　原始法线图



8　解码后的法线

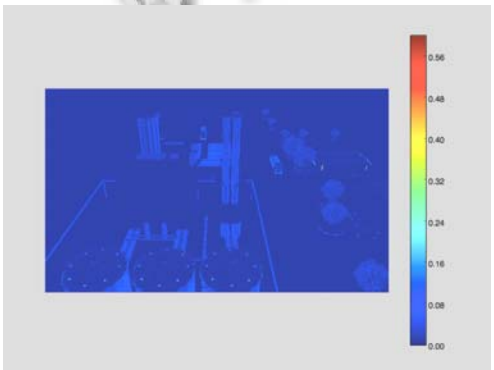


图 9　图 8、9 差异热度图

3.2 延迟渲染框架的效率

延迟渲染框架的最大优点之一就是多光源的支持，为了测试在多光源情况下框架效率，在场景添加了多个点光源，如图 10 所示，可视化了光源的 Light Volume，图 11 为点光源的作用结果，图 12 给出了在多个点光源下的渲染效果图.

最后图 13 给出了在单个光源情况下，延迟渲染和正向渲染在三维模拟演练系统中的效率. 由图可知，在单光源下，延迟渲染的效率要低一些，这是由于延迟渲染需要额外的资源来合成最后的结果. 图 14 给出了在三角形面片数量为 100K 的场景中不同光源数量的作用效果，随着光源数量的增加，延迟渲染仍然能够保持相对较高的帧数.

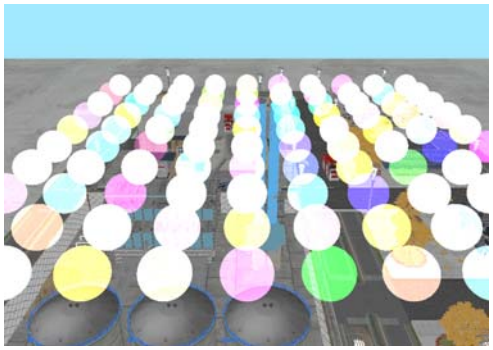


图 10　多个点光源

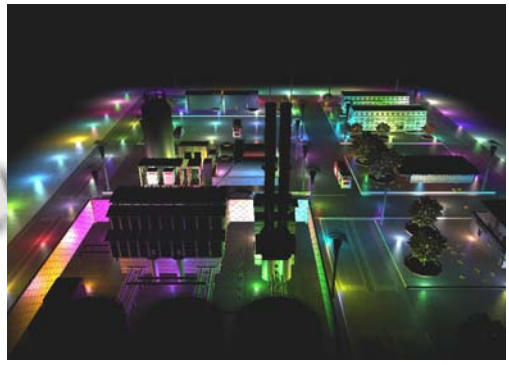


图 11　点光源作用结果

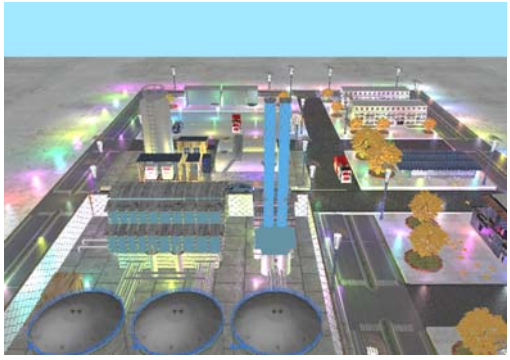


图 12　渲染效果图

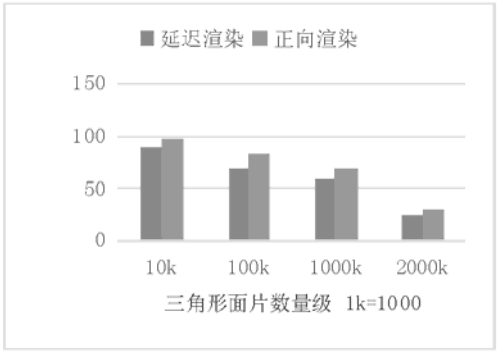


图 13 单光源帧数对比

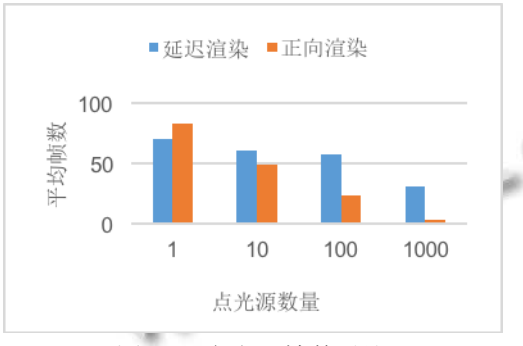


图 14 多光源帧数对比

3.3 全局光照算法集成的讨论

现代流行的图形引擎为了提供更加真实的场景渲染效果, 在延迟渲染框架集成了全局光照的算法. 其中一种高效并易于实现的方式为使用 Reflective Shadow Map[8]. Reflective Shadow Map 是在 Shadow Map 基础上的一个扩展, 它先以光源为观察点来渲染整个场景, 并存储渲染后场景中的深度信息、法线信息、位置信息和光照强度信息. 由于是以光源为观察点, 所以对于场景中的每一个像素都是一个潜在的光源反射点, 可以把这个光源反射点看为一个新的点光源, 称为像素光源(Pixel Light). 所以对于存在于 G-buffer 中的某一个点, 将点投影到 shadow map, 并通过随机采样的方式选取对应的像素光源来参与运算. 由于每个点都需要多个像素光源, 引入了大量的计算, 因此一些算法引入重要性采样, 并生成某些区域来作为光源所影响的范围.

环境光遮蔽也是一种十分有用的模拟全局光照的手段, 该算法认为一个物体的表面含有许多的凸起和凹槽, 由于 G-buffer 提供了所需要的颜色、深度和法向量的信息, 所以在框架中集成算法也十分的便捷, 只需要在特定的着色器中提供相关实现.

4 结语

延迟渲染框架的优异之处在于将场景的渲染和光照效果的计算相分离, 它更适应于现代的图形流水线, 虽然增加了显存的使用量, 但是减轻了显卡寄存器的压力, 大大提高多光源情况下的光照计算的效率, 也因此对于虚拟光源这类的全局光照算法有十分优秀的支持. 同时延迟渲染可以作为一种实验性的平台来提供多种算法的组合, 各种后处理技术的应用便是一种体现. 近年来, 移动平台的崛起, 针对移动平台的延迟渲染框架和一些定制化的硬件被提出, 可见在未来的一段时间内针对延迟渲染的研究仍然有重要的意义.

参考文献

1 梅玉龙.应急演练计算机三维模拟系统研究.中国安全生产科学技术,2012,(4):92–97.

2 Olsson O, Assarsson U. Tiled shading. Journal of Graphics, GPU, and Game Tools, 2011, 15(4): 235–251.

3 Engel W. GPU Pro 4: Advanced Rendering Techniques. CRC Press, 2013: 115–135.

4 Angel E, Shreiner D. 交互式计算机图形学.北京:电子工业出版社,2012.

5 Chen LT, Cai HB, Wang GJ, et al. HDR-based deferred lighting technology. Journal of University of Electronic Science & Technology of China, 2007, 36(5): 817–820.

6 Hoang TD, Low KL. Multi-resolution screen-space ambient occlusion. Proc. of ACM Symposium on Virtual Reality Software & Technology. 2010. 101–102.

7 Shreiner D.李军,徐波,译.OpenGL 编程指南.北京:机械工业出版社,2010.

8 Dachsbacher C, Stamminger M. Reflective shadow maps. Proc. of the 2005 Symposium on Interactive 3D Graphics and Games. ACM. 2005. 203–231.