DX 11 高级渲染技术下 3D 游戏引擎①

袁 琼, 刘 立

(武汉东湖学院 计算机科学学院, 武汉 430212)

摘 要: 研究了 DirectX 11 高级渲染技术并成功应用于 3D 游戏引擎中. DirectX 11 高级渲染技术包括 MultiTex、CubeMap、NormalMap、ShadowMap 等. 引擎运用了 MultiTex 技术绘制 3D 地形; 使用了 CubeMap 技术进行环境贴图渲染; 使用了 NormalMap 技术进行特殊纹理处理; 使用了 ShadowMap 技术实现了成熟逼真的阴影效果. 文中还详细给出了引擎的解释器框架和多分支渲染核心框架设计, 并对引擎完成了相关技术用例测试, 结果表明了其逼真的图形效果.

关键词: DirectX 11; MultiTex; CubeMap; ShadowMap; 3D 游戏引擎

3D Game Engine with Advanced Rendering Technology in DX 11

YUAN Qiong, LIU Li

(Computer Science Institute, Wuhan Donghu University, Wuhan 430212, China)

Abstract: The advanced rendering technologies of DirectX 11 were studied and applied to the 3D game engine. Advanced rendering technologies of DirectX 11 included MultiTex, CubeMap, NormalMap and ShadowMap, etc.. MultiTex was used to implement 3D terrain rendering, CubeMap was used to do environment map rendering, NormalMap was used to do special texture processing and ShadowMap is used to achieve the effect of mature realistic shadow. The paper also gave a detailed framework of interpreter and multi-branch of rendering core framework design, and completed the related technical cases to test on the engine. The result showed that the engine was true to life and effective in processing graphics.

Key words: DirectX 11; MultiTex; CubeMap; ShadowMap; 3D game engine

1 引言

计算机软硬件发展日新月异, 现代计算机软件以游戏为代表的高品质程序为达到逼真的图形效果, 纷纷加强了程序对 GPU 图形显卡的应用. 程序与显卡之间的接口目前有两种, 一个是 OpenGL[1], 另一个则是 DirectX(简称 DX)[2], 在 Windows/Xbox 平台游戏开发, 绝大多数均采用 DX, 在移动平台(跨平台)则多用 OpenGL, 它们除了 API 有所不同, 就其实质而言, 算法与基础理论均是大同小异. 在计算机领域, 最复杂的莫过于次世代游戏[3], 它集合了几乎所有计算机相关技术, 包括对多而大的文件的读写加密解密处理、复杂的 CPU/GPU 实时运算、以及声光输出处理、最严格的优化需求等, 几乎榨干计算机硬件的一切性能. 区别于普通的游戏软件, 次世代游戏一方面能带给用户更加完美的体验, 另一方面更能促进普通游戏行业与相关产业(如 3D 仿真、CG 影视等)发展.

本文的研究正是在这种背景下正式提出的, 目标是研究 DX11 高级渲染技术并应用于 3D 游戏引擎的开发中.

2 DX 11高级渲染技术

本文引擎所使用的每一个高级条目都属次世代技术类, 又称为高级技术类. 包含 MultiTex、CubeMap、NormalMap、ShadowMap[4]等.

① 基金项目:武汉东湖学院青年基金(2014)

收稿时间:2015-07-13;收到修改稿时间:2015-09-21

2.1 MultiTex

MultiTex 即多纹理混合, 如果要求绘制一幅包含草地、沙地、泥地、青石板等细致的广袤地形图, 怎么做, 绘制一幅真实超大纹理铺在地表多边形上吗? 在 2D 时代, 有很多的确就是这么干的, 然而 3D 时代, 单纯依赖拼块的方式已经行不通, 因为在超大仿真地图绘制上, 不可能也不允许使用如此高昂的代价来完成, 这时候就需要运用到 MultiTex 技术, 本文以引擎工程中的地表编辑器(如图 1 所示)中实际的一副地表(如图 2 所示)为例进行论述.

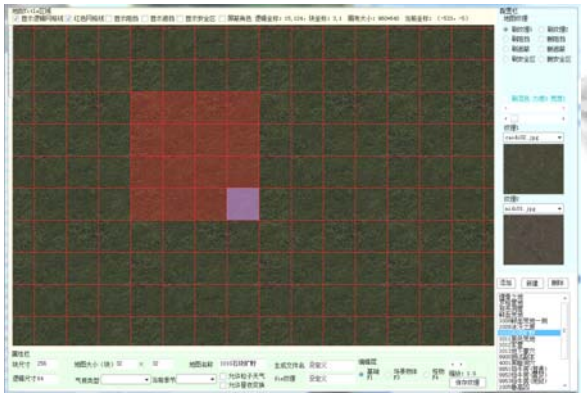

图 1　地图纹理编辑器

图 2　局部最终成像(左)与 fix 混合纹理(右)

图 3　两幅混色纹理素材一(左)与素材二(右)

图 2(左)这幅地形图只使用了图 3 中的两种素材, 然而做到了无缝拼接的效果, 其实现过程如下:

首先, 假设有一块多边形地图块域采用了图 3 中的素材一和素材二, 按照逐像素着色方式分别读取素材中每个点 p 的颜色值, 假设素材一在点 p 对应的颜色为 C1, 素材二在点 p 的颜色为 C2; 然后载入其匹配的 fix 混色纹理点 p 的颜色, 假设找到该点对应位置颜色 C3; 最后依据以下公式, 得到其最终颜色成像:

$$Cfinal=C1*g+C2*(1-g)$$

其中 g 为 C3 的灰度值.

2.2 CubeMap

计算机图形学中纹理不仅仅是贴图, 它还包含很多特殊的纹理. 贴图只是 2D 纹理, 还有 3D 纹理, CubeMap 就是典型的一种 3D 纹理, 主要用于模拟天空盒以及环境反射.

图 4　具备环境反射技术的物体绘制

如图 4 所示, 环境反射就是让一个模型反射周围场景, 就好像镜子的效果, 它是相当逼真而且代价极低的一种技术. 使用过程包括两个部分, 一是制作立方体环境贴图, 二是应用到渲染计算阶段.

图 5　一个预制作的立方体环境贴图展开样式

立方体环境贴图可以如图 5 中预制作, 然后直接投入使用, 也可以实时渲染制作. 实时制作代价会稍高一些, 如果不缩减立方体环境贴图每个面的分辨率, 理论上渲染代价额外高 6 倍. 不过一般会将其缩放得很小, 因为作为反射, 不需要太高分辨率. 得到立方体环境贴图后就要应用它.

在本引擎中, 环境贴图渲染的实现方案如下:

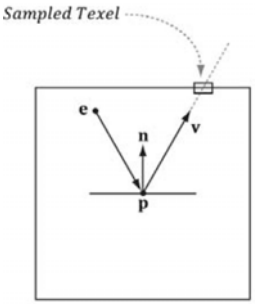

图 6　算法草图

如图 6 所示, 请把外框想象作立方体环境贴图, P 是要渲染的点, e 是视点位置, 首先要得到 V 这条向量, 然后把 V 看作查找向量, 找到 SampledTexel.

在着色器中核心代码:

```
e= normalize(PointPos-CameraPos);
V=reflect(e,n);
SampledTexel=gCubeMap.Sample(samLinear,V);
```

这里的 samLinear 是一种点采样方式.

图 7 水面反射天空的渲染

图 7 的水面即采取了这种办法反射了环境天空的效果.

2.3 NormalMap

图 8 NormalMap 图例

本引擎中采用了 NormalMap 后的效果如图 8 所示, 注意手臂和树干部分的不光滑特性.

图 9 手臂 NormalMap 技术使用前(左)后(右)对比

如果没有 NormalMap, 如图 9(左)所示, 你看到的人物手臂会非常平滑, 丝毫不会有凹凸不平的材质感; 同样, 你看到的树干树叶也不会呈现高光效果, 你看到的水面将是平的或者呈明显多边形化. NormalMap 也是一种特殊纹理技术, 是一项次世代图形基础标准. 其实现过程如下:

首先需要法线贴图, 然后获取纹理的正切空间、顶点的正切空间、在切线空间和物体空间之间变化, 解压缩获得新的法线扰动值等, 不同于以往的纹理只可以用于 2D 表面, 它可以应用到 3D 表面的特殊纹理. 作为凹凸纹理的扩展, 它使每个平面的各像素拥有了高度值, 包含了许多细节的表面信息, 能够在平平无奇的物体外形上, 创建出许多种特殊的立体视觉效果. 法线贴图是最近比较被关注的技术, 也将成为以后 CG 领域的一大主流技术, 也就是游戏界被称为的次世代技术. 图 10 中给出了图 8 中使用到的 NormalMap 纹理(局部).

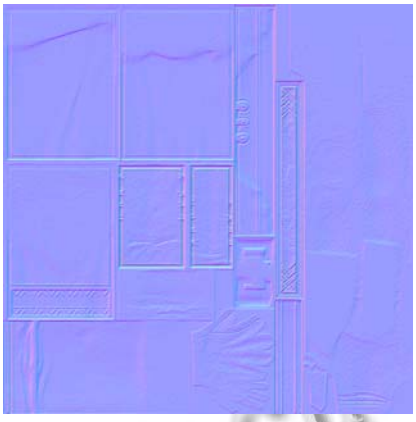

图 10 NormalMap 纹理(局部)

2.4 Shadow Map

这也是一个复杂技术, 可以用来实现现有最成熟逼真的阴影效果. 与之对应的是 Shadow Volume, 它原理更为复杂, 运算复杂度与场景复杂度有关而不可预测.

对于方向性灯光, 如太阳光, 其过程如下:

(1) 需要把视点(camera view)的视椎体(camera frustum)搬到光源的 view space.

(2) 求得 view matrix 的各个参数: farZ 参数为在 view space 中视椎体的 maxZ-minZ; nearZ 为 0.0; upVector 是方向光的任意一个垂直向量; lookAt 是视椎体的“质心”.

(3) 计算 view matrix, 把 veiw matrix 搬到平行投影坐标系(orthographic projection space).

(4) 再次渲染场景的过程中, 将每个片断(像素)

变换到前述眼坐标系中, 并缩放到[0, 1]的范围内以便查询纹理.

(5) 以当前片断在眼坐标中的 S、T 坐标查询深度纹理获得深度值, 将此深度值与当前片断的 R 坐标进行比较, 若 R 坐标大于深度值, 则当前片断在阴影中; 否则当前片断受光照.

具体在实现时, 还会遇到各种意想不到的问题, 比如 Z-fighting、各矩阵调节等.

3 3D游戏引擎的设计与实现

3.1 引擎解释器框架

引擎[5][6]的质量决定整个最终产品所能达到的品质, 采用 C++编码能确保更加容易发挥硬件性能, 在主循环过程中能承受更大压力, 更小的延迟. 这也意味着将有更加稳定的帧率.

引擎核心采取与世界水平保持一致的 DX11 API 版本[7]与 HLSL5.0, 相比 DX9, 虽然可参考内容更加匮乏, 但更容易扩展, 对未来硬件更加契合.

引擎还运用到XNA数学库, 用以方便地支持各种矩阵操作运算.

音频视频这两项不是本文重点讨论内容, 故而采取 Win32 下的 MCI.

为了模拟 3D 世界, 一些物理数学概念也必须了解, 包括各项矩阵转换、物理光照、力学系统等. 它们将用于展现模型世界到屏幕、模拟光照、阴影以及物体的受力运动、粒子群等.

引擎中的模型系统处于精简考虑, 采用自定义的文件系统, 模型原件来自 3DSMAX, 通过 PandaDirectX 导出.x 文件, 经引擎编码编辑器修改, 从文本格式文件修改为二进制文件. 纹理由 Nvidia 开发的 Normal Texture Tools 通过 Photoshop 制作.

本系统的引擎解释器框架如图 11 所示.

0. 读取配置文件

-包括开启游戏时的分辨率, 是否是全屏, 渲染质量等设定.

-预载入运镜摄像机轨道配置文件、粒子系统的产生、运动配置文件.

1. 依据配置文件, 进行初始化窗口 InitWindow 和初始化 3D 设备 InitDecice 操作.

InitWindow 通过调用 AdjustWindowRect 函数创建 Win32 窗体.

InitDecice 创建 DXGI_SWAP_CHAIN_DESC 并填充结构, 通过 D3D11CreateDeviceAndSwapChain 执行交换链、设备、渲染环境的创建, 随后创建设定各种不同的深度模板纹理视图、各 depthstencilstate、rasterozer 裁剪以及 Shader 载入编译, 然后为各个 Shader 创建纹理与常量缓冲区.

2. 初始化脚本系统, 它将执行 main 入口脚本内的内容, 包括载入播放音视频、模型、地图等.

本文设计的脚本系统是一种解释性语言. 脚本中的所有关键字、函数、逻辑系统均实际在 C++程序中写好, 通过匹配, 按照既定参数读取方法读取脚本并执行, 在效率上, 唯一额外的操作就是 I/O.

3. 消息循环、渲染循环、程序结束的 While 结构, 如果收到按键、鼠标消息则跳到 4 执行对应操作, 可能包括脚本的操作. 如果收到退出游戏、关闭程序的消息则执行 6 退出. 其他情景下跳转到 5 执行渲染.

4. 在这里执行引擎脚本中预设的操作, 例如镜头操作、读取新场景等. 在执行完毕后将立刻再执行 3 的主循环过程. 为了保证渲染帧数的稳定性, 此处要求的延时不能超过 16ms, 如果超过这个值, 游戏将不可避免的产生跳帧、抖动感.

5. 通过帧数限制决定是否进行渲染, 如满足条件则对场景按照要求进行渲染. 待渲染结束后返回主循环等待下一帧. 通常为了保持 60fps, 需要控制一个渲染循环时间在16.67ms内. 无论是否进行渲染, 函数均会执行 Timer 操作, 用以执行延时类函数.

6. 当收到结束程序的消息则执行退出消息. 在此处需要负责清空内存、保存存档等操作.

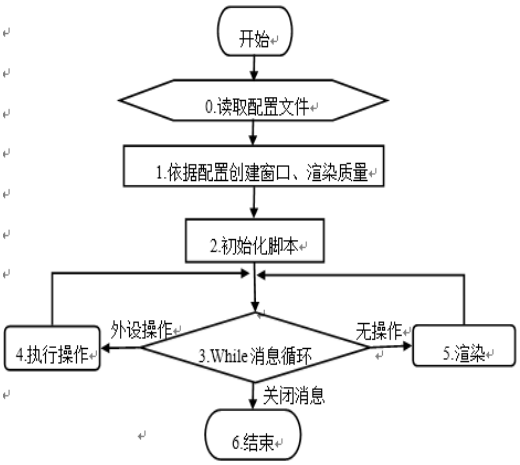

图 11　引擎解释器框架

3.2 多分支渲染核心框架

将内存中的数据在处理后传输给 GPU 的显存, 通过一系列技术手段将画面以预期的方式呈现在屏幕上, 其核心在于各式各样的算法与技巧.

本系统的渲染核心框架如图 12 所示.

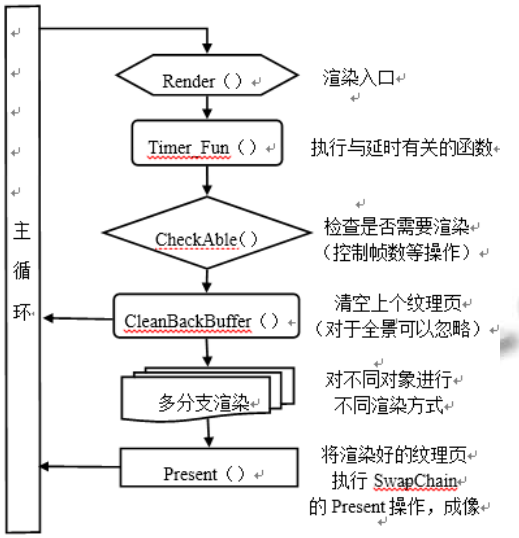

图 12 多分支渲染核心框架

Timer_Fun()将执行一些在脚本中定义的延时相关函数, 它能实现一些预定动画、预定时触发的动作等. 因此无论这一帧是否执行, 它都要处理, 在主线程中, 它通常不适合执行大量复杂的操作.

CheckAble()中需要计算当前帧是否达到绘制点, 否则程序会执行极限频率 Render()操作, 对于超过 60fps, 人眼已经无法感受到差别, 考虑到 GPU 负载发热、用户将程序置于后台等行为, 有必要对帧率进行限制.

CleanBackBuffer()操作是用以清理上一个交换链上的纹理页, 它不是必须的, 因为在大部分情景下, 会完整绘制整个画面. 如此, 旧页面会被覆盖. 它能降低少量 GPU 负担.

多分支渲染是整个引擎最复杂, 也是衡量引擎质量权重最大的部分. 下面将具体阐述.

3.3 多分支渲染过程及其效果

本项目使用修改的 Shadow Map 技术, 首先从光源方向构建 ShadowTransform, 对所有物体(Clip 掉近透明像素)进行渲染到最低 1024、最高达 4096 分辨率, 渲染 Radius 依 Camera 所见范围动态调整在 20-110 的超级采样深度图中, 然后再进行正常渲染, 将像素点于灯光视角深度信息比对, 并对边缘进行多次采样执行高斯模糊计算实现平滑.

基于引擎对应的游戏类型, 渲染按照层级模式反向渲染, 即优先渲染子对象, 其后考虑渲染父对象, 在渲染子对象时, 同时进行相对屏幕的 Z 轴排序, 优先渲染距离屏幕近的, 完全被遮蔽的物件则直接进行剔除操作, 与一般 2D 图形的画家算法不同, 这样做节省顶点处理和像素处理, 能极大限度的降低像素着色器的负载, 从而达到在不影响画面的情况下优化性能的目的.

首先进行 Shadow 渲染, 以最低质量标准绘制场景, 不绘制半透物体, 从灯光视角对物体进行深度渲染, 将纹理保存, 然后通过这张灯光视角下的深度纹理, 执行光照渲染. 测试效果如图 13 所示, 左图无 Shadow 阴影, 右图开启 Shadow 阴影, 光源在图示的右上角.

图 13 Shadow 渲染效果对比图

在渲染同层次物件时, 优先处理完全不透明物体, 随后处理半透明物体, 在对半透明进行渲染时, 关闭其深度写入, 以此解决渲染顺序问题.

渲染完成后, 进入 PostRendering 阶段, 在此阶段, 将执行 HDR 渲染. HDR 渲染技术全称是 High dynamic range, 即高动态范围渲染, 由于现代计算机屏幕理论最大只能呈现 0-255 总计 256 个阶的亮度色域, 然而世界实际亮度范围却远超于此, 人眼对此会有一个瞳孔适应过程, 类似于相机曝光, 将所见场景亮度压缩到可适应区域. HDR 为模拟这样一个适应过程应运而生. 将场景渲染到一副较小解析度的纹理中, 计算得到该纹理的最大亮度/最小亮度/以及平均亮度, 接下来, 将实际画面中的所有亮度动态调节映射到 0~1 区间中来, 以此实现该效果. 通常 Bloom 也会在此同步运算, 即获得纹理中亮度大于某个值的点颜色保留, 其他则设置为黑色, 再将此纹理高斯模糊处理, 最后同 HDR 结果相叠加, 即可产生闪光区效果. 如图 14 所示, 左图无 HDR, 右图开启 HDR 渲染, 右图能看清楚更多细节.

再如图 15 所示, 左图无 HDR+BLOOM, 右图开启 HDR+BLOOM 渲染, 右图能看到水面的波光粼粼.

图 14　HDR 渲染效果对比图

图 15　HDR+BLOOM 渲染效果对比图

接着是景深渲染, 用以模拟摄像机镜头, 人眼晶状体视距. 如图 16 所示, 左图无景深效果, 右图为开启景深, 右图呈现出了远处模糊, 近处清晰的逼真效果.

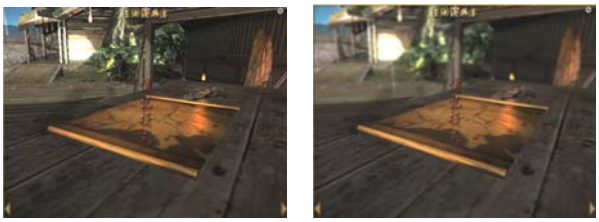

图 16　景深渲染效果对比图

要达到这样的效果, 需要保存当前页面的深度视图, 并将其存入纹理中, 接下来读取该深度纹理, 通过脚本或者默认的视图中心, 得到默认“期望视距”, 再在像素着色器中, 根据该期望视距和实际深度值的比较, 进行不同程度的高斯模糊, 即可实现上述效果.

最后, 在开启 NormalMap、CubeMap、ShadowMap、HDR、Bloom、景深后同一场景渲染对比效果如图 17 所示, 左为仅开启多光源光照, 右为开启所有特效, 明显右图效果更逼真.

以上实现效果只是和未进行渲染的图像进行比较, 下面以阴影为例考察其他引擎处理效果进行对比.

如图 18 所示, 来自于 COMCAP 平台下的鬼泣 4 中人物场景效果. 第一副图片中, 长椅上没有主角尼禄的阴影, 第二幅图中, 主角的武器、身后的建筑均没有在背部产生应有的阴影.

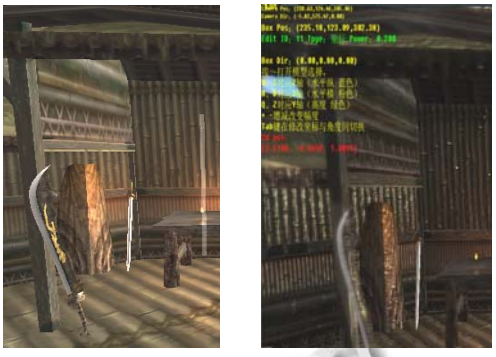

图 17　最终渲染对比图

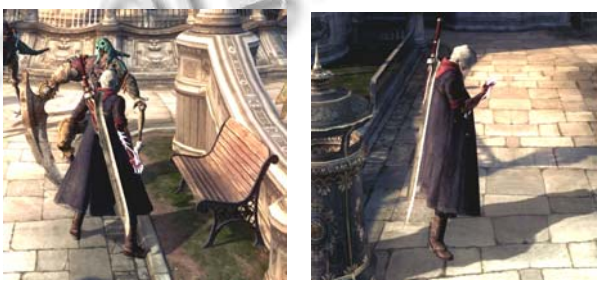

图 18　观察鬼泣 4 的纹理

事实上, 由于鬼泣 4 中没有动态光源, 地图所见大部分的建筑阴影, 经破解后观察纹理发现均是美工提前渲染好的效果(非实时渲染), 如图 19 所示, 如有动态光源, 这种办法显然就不适用了.

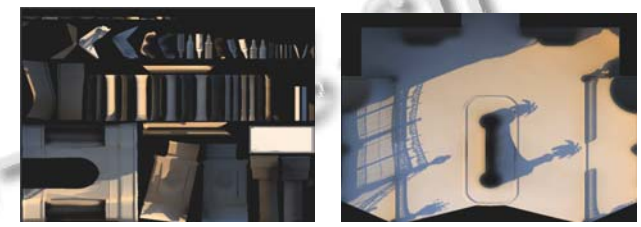

图 19　使用动态光源前后

图 20 来自于 steam 平台下的 DOTA2. 放大图片观察, “卡尔”背部明显可见到因 Z-Fighting 导致的阴影锯齿, 右图近视角可见地面阴影锯齿明显.

图 20　DOTA2 中“卡尔”场景效果图

可见 , 本文在 DX11 高级渲染技术下的影子在表达效果上较很多 3D 游戏均有所改进, 例如发行于 steam 平台下的 DOTA2 及 COMCAP 下的鬼泣 4 等等.

4 小结

本文在 DirectX 11 高级渲染技术的基础上实现了一个 3D 游戏引擎, 该引擎的开发具有如下特点:

使用 C++制作了游戏引擎解释器核心, 它将执行负担最重的最终客户端, 直接呈现出游戏效果, 同时大量基于 HLSL 的引擎研发技术将在此程序中得以体现.

使用 C 开发了游戏编码工具, 执行开发过程中极其复杂的处理请求. 例如 3D 模型格式转换, 纹理拼接、代码快速加密技术等.

使用 C#进行了各项引擎编辑器工具集的快速开发, 它具备 UI 界面系统, 便于操作, 各项配置系统由它产生与维护, 例如 UI 系统编辑器、关卡、地图设计系统等.

然而, 游戏引擎研发包含了大量 3D 图形核心技术, 包括阴影算法、光照系统、视差、法线纹理、模型系统、以及大量 Post-Rendering 渲染技术[8][9]等, 都较为复杂, 将作为后期进一步研究的方向.

参考文献

1 董槐林,徐天茂等.OGRE 引擎中基于BVH的角色动画的实现研究.计算机与现代化,2011(10).

2 Sherrod A, Jones W. Beginning DirectX 11 game programming. Course Technology PTR, 2011.

3 Valdetaro A, Nunes G, Raposo A. Understanding shader model 5.0 with directx11. Course Technology PTR, 2010.

4 Pinheiro RB. Introduction to multithreaded rendering and the usage of deferred contexts in DirectX 11. Course Technology PTR, 2011.

5 耿卫东.三维游戏引擎设计与实现.杭州:浙江大学出版社, 2008.

6 陈占锋,谭会辛.基于 Direct 的飞行射击游戏引擎实现要点.重庆工学院学报,2008,22(10).

7 李建波.DirectX 3D HLSL 高级实例精讲.北京:清华大学出版社,2013.

8 Walsh P. DirectX 10 3D 游戏编程深度探索.北京:清华大学出版社,2010.

9 Luna FD. Introduction to 3D game programming with DirectX 11. Mercury Learning & Information, 2012.