申请上海交通大学工程硕士学位论文

面向 GPU 优化的渲染引擎研究与实现

院系：计算机科学与工程系

工程领域：计算机应用技术

交大导师：马利庄教授

企业导师：盛斌博士

工程硕士：陈是权

学号：1100332069

上海交通大学电子信息与电气工程学院

2013 年 05 月

Thesis Submitted to Shanghai Jiao Tong University

For the Degree of Engineering Master

GPU EFFICIENCY BASED GRAPHIC ENGINE INVESTIGATION AND IMPLEMENTATION

M.D. Candidate：ChenShiQuan

Supervisor（I）：MaLiZhuang

Supervisor（II）：Sheng Bin

Specialty：Computer Application

School of Electronics, Information and Electric Engineering
Shanghai Jiao Tong University
Shanghai, P.R.China

May, 2013

附件四

上海交通大学
学位论文原创性声明

本人郑重声明：所呈交的学位论文，是本人在导师的指导下，独立进行研究工作所取得的成果。除文中已经注明引用的内容外，本论文不包含任何其他个人或集体已经发表或撰写过的作品成果。对本文的研究做出重要贡献的个人和集体，均已在文中以明确方式标明。本人完全意识到本声明的法律结果由本人承担。

学位论文作者签名：陈是权

日期：2013 年 5 月 28日

附件五

上海交通大学

学位论文版权使用授权书

本学位论文作者完全了解学校有关保留、使用学位论文的规定，同意学校保留并向国家有关部门或机构送交论文的复印件和电子版，允许论文被查阅和借阅。本人授权上海交通大学可以将本学位论文的全部或部分内容编入有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存和汇编本学位论文。

保密□，在___年解密后适用本授权书。

本学位论文属于

不保密☑。

（请在以上方框内打“√”）

学位论文作者签名：陈是权　　指导教师签名：

日期：2013 年 5 月 28 日　　日期：2013 年 5 月 28 日

上海交通大学
学位论文原创性声明

本人郑重声明：所呈交的学位论文，是本人在导师的指导下，独立进行研究工作所取得的成果。除文中已经注明引用的内容外，本论文不包含任何其它个人或集体已经发表或撰写过的作品成果。对本文的研究做出重要贡献的个人和集体，均已在文中以明确方式标明。本人完全意识到本声明的法律结果由本人承担。

学位论文作者签名：

日期：年月日

上海交通大学
学位论文版权使用授权书

本学位论文作者完全了解学校有关保留、使用学位论文的规定，同意学校保留并向国家有关部门或机构送交论文的复印件和电子版，允许论文被查阅和借阅。本人授权上海交通大学可以将本学位论文的全部或部分内容编入有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存和汇编本学位论文。

保密□在年解密后适用本授权书。

本学位论文属于

不保密□。

（请在以上方框内打“√”）

学位论文作者签名：指导教师签名：

日期：年月日日期：年月日

面向 GPU 优化的渲染引擎研究与实现

摘要

近些年来 3D 数据的应用得到了极大的发展，尤其是像游戏、计算机辅助设计等应用的普及使得人们在享受这些应用来到的方便绚丽的同时也开始慢慢关注这些 3D 数字技术背后的支撑技术，越来越多的开发者也开始投身到 3D 应用程序的创业浪潮当中。然而开发一个 3D 程序并不是一件简单的事，应用开发者往往在代码的简洁性和程序的高效性上难以两者兼顾。这时有一个应用程序和底层系统资源中间的桥梁--渲染引擎就显得非常地有必要。目前市场上包括商业的、开源的渲染引擎也不少，但现实是程序开发者在开发中仍然会碰到等各种各样的问题，尤其是性能问题。这其中一个很重要的原因就是这些引擎本身也没有很好有效的利用资源，尤其是图形处理器(GPU)资源。

本文将从基于 GPU 的渲染中数据读写效率入手，研究了如何实现一个高效、充分利用 GPU 性能的 3D 渲染引擎。与传统的 3D 渲染引擎相比，它的特点及创新点体现在：

第一、在物体模块中介绍了一种新的物体单元即批物体单元。这种单元的原理在于将一些具有共同特性的物体一起捆绑送入 GPU 中渲染，较常用的简单单元比起来这种单元能减少对 GPU 调用，提升渲染性能。

第二、在纹理模块上本文介绍了一种纹理集的概念，即当物体材质上的参数都相同仅纹理不同时，可以将各纹理合并到一张大纹理上，连同材质一并送入 GPU，这样可以减少 GPU 从磁盘载入纹理的时间。

第三、在渲染器模块上本文主要介绍了一个对物体排序的思想，这里排序包括两种：以材质排序和以物体距视觉观察点的远近排序，其目的也都是为了减少不必要的 GPU 操作，以提高性能。

第四、在选取系统中本文提出了一种基于 GPU 新的选取算法，这种算法特点在于利用了 GPU 的渲染能力来替代传统选取算法中的求交运算，在效率上有较大提升。

最后本文针对上述的的优化方案进行了测试，结果显示优化后比优化前渲染效率有明显改善。

关键词渲染引擎，GPU 优化，批处理，纹理集，排序，GPU 选取

GPU EFFICIENCY BASED GRAPHIC ENGINE INVESTIGATION AND IMPLEMENTATION

ABSTRACT

With the rapid development of internet and mobile devices, the applications of3D digital mediaarebecomingmore and more popular in recent years.Rather than the beautiful and fantastic rendering images as before, people now care more and more about the background technology for supporting digital media.3D digital applications are now really the hot spot in market. There is also lots of software developers are now joining the tide of such abusiness.While developing excellent3D applications is usually not easy, one of the main reasons is the complexity of the source code, and the performance is also a bottleneck.To resolve this problem, we need a bridge-like component to handle the request between applications and system resources more efficiently. Actually there have already been a lot of such components in the market, which are called as Graphic Engine. While sometimeseven with the graphic engine, people still face to many kinds of problems, especially performance issue, one of the mainreasons is that theenginesthemselves are not efficient enough, in another world, they have to be improved significantly, especially more GPU resources should be used to speed up the algorithms.

To solve the problems above,this thesis introduces how to implement a high efficient graphic engine. The efficiency includes:

1. In mesh component, this thesis introduces a new kind of mesh named as BatchMesh which will reduce GPU draw calls by merging meshes having the same materials.
2. In the texture component, we introduce a new concept named asTextureSet to merge textures for materials whose parameters are the same. This will also help to reduce GPU draw calls
3. In renderer module, the thesis shows you new algorithm to accelerate rendering-ObjectSorting. This sorting is composed of two aspects: y material and the object distance tot camera position, it's also for the purpose ofreducing unnecessary GPU draw calls.
4. Anew way of selection algorithm is introduced - GPU based selection, this algorithmis more efficient than traditional selection system.

Lastly, experiments showthat graphic engine ismuch more efficientby using the above optimizations.

Keywords:Graphic Engine，GPU Optimization，Batch，Texture Set, Sorting，GPU Selection

目录

第一章概述

近些年来，3D数据的应用得到了极大的发展，尤其是像游戏、计算机辅助设计等产业的普及使得人们在享受这些应用带来的方便绚丽的同时也开始慢慢关注这些3D数字背后的支撑技术。越来越多的开发者开始加入到3D应用程序的创业浪潮当中。然而开发一个3D应用程序并非易事，尤其是在开发那些支撑高级渲染效果的大应用（如游戏程序、设计软件等）时应用开发者往往会在代码管理及程序的高效性方面显得非常力不从心，这样导致的后果通常是程序的不稳定性以至于程序的直接崩溃。这个时候我们需要一个专门的处理单元来帮助程序开发者解决这些问题。这个处理单元即是渲染引擎。渲染引擎作为系统底层接口层(API)及应用程序之间的中间层能有效率地为应用程序提供简易操作的API，并能对调用底层系统API进行有效的管理，即简化了程序开发者繁琐调用的系统API的过程，同时也对这系统资源调用进行有效管理，极大地增加了应用程序的稳定性和高效性。然而由于3D程序或者3D技术本身的复杂性，有时候即使利用了渲染引擎程序开发者仍可能会碰到各种各样的问题，这其中一个很重要的原因就是这渲染引擎本身的高效性很多时候没有得到体现，也就是说很多渲染引擎本身存在着很大的优化空间。

1.1 项目的提出

目前市面上各种操作平台（尤其是移动平台）的上3D应用程序越来越多，然而对一个程序开发者来说在操作平台上开发一个3D应用程序目前还主要停留在直接调用系统API的阶段。而这一过程通常会产生如下缺点：

- 代码重复利用率低
- 代码繁重不便于管理
- CPU运算负担过重
- 冗余的GPU调用操作

为了有效的管理系统资源，我们必须避免程序开发人员直接调用系统API，而需要在系统API与应用程序之间再建立一层管理系统，这个管理系统通常就称

之为渲染引擎。

有了渲染引擎后和程序开发者打交道的不再是系统资源而是渲染引擎，因此渲染引擎本身的效率至关重要，然而实际上很多时候渲染引擎本身的效率成为制约其自身发展的一个重要瓶颈。渲染引擎本身效率不高往往体现在以下几个方面：

- 频繁的GPU操作会导致渲染效率低下，而且经常会导致GPU内存消耗过多，造成程序甚至系统的崩溃。
- CPU运算负担过重，渲染引擎中除了底层渲染功能之外，还有很多渲染之上的功能。如碰撞检测、选取功能等。这些功能往往都是在CPU中完成，这无形中给CPU造成了过多的压力。

由此看出设计与实现一个高效的渲染引擎显得非常重要，这也就是本文的动机和研究的内容。

为了解决上述的渲染引擎本身效率的两个问题，本文做了深入地研究。对于GPU 频繁操作的问题，本论文解决的办法是在渲染过程尽可能地减少 GPU 调用，渲染引擎中调用 GPU 的地方很多，其中最典型的有：用来表示物体轮廓的的顶点缓存和索引缓存在送入 GPU 进行渲染时需要拷贝到 GPU 的内存（即显存）中；物体的材质送入 GPU 中会被翻译成一段 GPU 认识的程序，GPU 在切换不同的材质时需要保存前面材质的上下文；材质上的纹理在传给 GPU 中材质的程序段时需要先被载入到 GPU 的内存中；最后的渲染结果需要被写进 GPU 中专门分辟的渲染缓存中，并且在最后对于渲染到相同位置的像素点需要进行深度测试，即有可能需要频繁重复地读写渲染缓存；而对于 CPU 去处负担过重的问题，本文主要研究的是如何运用 GPU 来代替 CPU 做这些运算，GPU 中的并行单元较多，更适于做一些并行较多的计算。

针对以上研究内容本文提出了四种优化方案：一、批物体单元方案。这种方案是针对物体的顶点缓存和索引缓存来优化的，其原理在于将具有某一类共同特性的物体一起捆绑送入 GPU，物体共同特性表现在两个方面：第一是物体的原始顶点缓存和索引缓存相同，而只是其位置大小不同而第二则是物体其运行的特性一致，即看起来像一个整体在运动。二、纹理集方案。这里主要针对 GPU 对于纹理载入的优化。在渲染引擎中我们会碰到一些物体，这些物体的材质相同，

材质的参数里除了纹理其他都相同，在这种情况下可以将这些物体上不同的纹理合成一张大的纹理并边连同材质一起送入 GPU，这样可以减少 GPU 频繁载入纹理的操作。三、材质排序方案，这里主要是为了减少不必要的 GPU 操作。对物体按材质排序可以减少 GPU 切换不同材质时的开销，而按距视觉观察体这的排序可以减少频繁读写渲染缓存的操作。四、基于 GPU 的选取算法。传统的点选的方案主要是由观察点到屏幕选点的射线与场景中的物体进行求交运算。而本文所以介绍的基于 GPU 点选的方案将会放弃求交运算，而是直接通过屏幕的点选到的颜色来直接选择物体。其主要思想是利用 GPU 的渲染能力来替代传统选取算法中的求交算法来提高性能。

1.2 研究的背景和意义

移动互联网的浪潮近些年来风起云涌，越来越多的程序开发者正投身到了移动平台的创业当中。以“愤怒的小鸟”、“水果忍者”等为代表的一些移动设备上的游戏应用在受到人们极大青睐的同时也为3D应用的创业者们引领了方向。

国内3D应用市场虽然比较火热，而开发技术却比较落后，通常一个团队需要花上半年甚至一年的时间来完成一个应用，其原因就在于国内的3D应用开发还大多停留在直接调用系统API的阶段，而非通过成熟的渲染引擎来完成。这其中很大一部分原因是国内渲染引擎的技术本身就比较落后，很多游戏厂商都是直接购买国外商业引擎进行开发。因此研究和实现一个高效率的渲染引擎具有非常大的商业意义。

1.2.1 渲染引擎简价

所谓**渲染引擎**，处理的是用户的渲染动作与操作系统之间函数调用之间的关系问题。不用渲染引擎用户的渲染动作也可以通过调用系统API完成，然而这样的调用正如前面讲的有诸多的缺点。而有了渲染引擎，用户的渲染动作则会通过渲染引擎高效并有序的组织起来，这无论是对于程序开发者在代码管理方面还是对于程序最后的运行效率方面都有很大的帮助。

1.2.2 国内外渲染引擎的现状

图形学的技术在国外研究的比较早，时至今日发展的也较为成熟。国形渲染的技术发展到现在其技术实际上可分为三类：光栅化、光线追踪、辐射度。这三

种技术的起源都来源于国外。光栅化技术由Silicon Graphics公司于1992年开始提出，并开发了一套跨平台的图形渲染的API- OpenGL[1]，随后微软也发展了适用于Windows平台的API- DirectX。光线追踪技术也是美国的Whitted T.1980年提出的。辐射度算法早在1950在蒸汽工程领域就已得到应用，而后由美国康耐尔大学于1984提炼成一门图形学的算法。

国外图形渲染引擎中的应用研究最早起源于游戏引擎，正是由于游戏产业的快速发展，使得人们有需求对游戏代码进行模块化的管理。早在1996年美国Raven Software公司开发的Quake游戏中正式运用到了游戏引擎的概念，在随后二十几年的发展过程中，美国暴雪、欧特克等公司分别在游戏领域和CAD设计领域引领着图形学包括渲染引擎技术发展的最前沿的方向。另外还有诸多优秀开源的渲染引擎，如OGRE、Nebula、Irrlicht等。虽然这些开源引擎的功能不如商业引擎强大，但其都很好地体现了一个图形渲染引擎该有的功能模块和技术，而其源代码开放的优势使得读者在学习引擎技术方面更方便更直接，这也直接导致开源渲染引擎的研究近些年来也成为了一个很重要的热点。

国内对于图形学的研究起步较晚，最开始只有浙江大学、清华大学等一些重点院校有一些图形学的课程，应用也大都停留在研究阶段。如2002年浙江大学CAD&CG实验室开发的桌面型虚拟环境实时漫游系统[41]。图形学在国内的快速发展也起源于游戏领域，自2004年国内游戏业竞争越来越激烈，每款游戏开发的周期越来越短，游戏引擎技术也开始被运用到游戏开发当中，而国外游戏引擎巨额的授权费使得国内的游戏厂商们也开始思考游戏引擎的自己研发。

国内由于游戏产业的迅速发展而带来的游戏引擎的研究热潮，但目前为止国内市场上成熟的游戏引擎还较少，很多开源的引擎效率不高，正是由于这些问题才形成了本文的动机。

1.2.3 渲染引擎优化的意义

目前基于各大主流平台的智能终端，在满足移动用户通话及短信等基本功能需求的同时，还提供了移动互联网、多媒体浏览、游戏娱乐等丰富的功能。然而这些丰富的功能越来越不能满足用户的需求，现在的用户除了应用功能本身以外开始越来越注重视觉和交互的感受，这样也就迫使系统应用除了本身的功能之外还需要对图形渲染方面有更好的要求。系统应用对于图形渲染的需求关系如图

1-1所示：

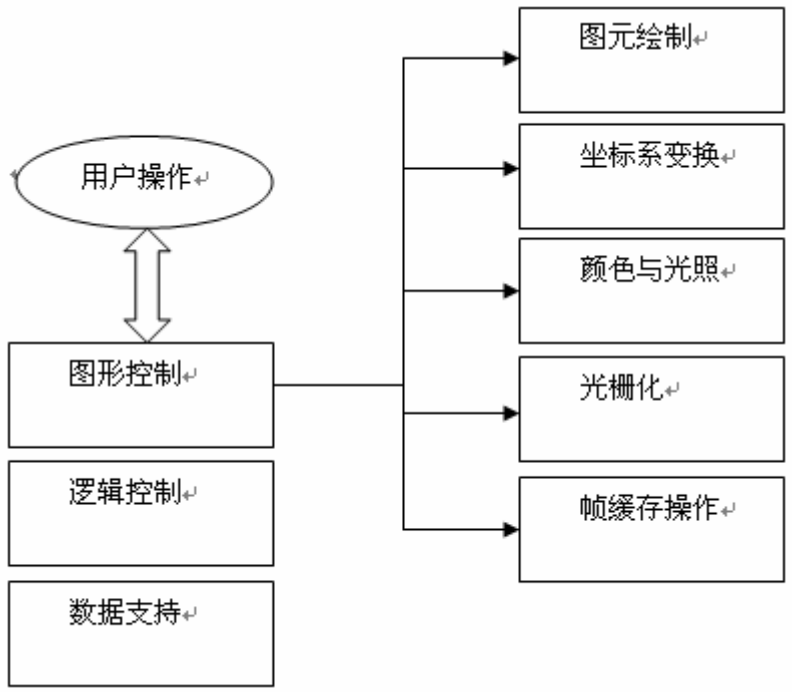


图 1-1 系统应用对与图形渲染的需求关系

Fig.1-1 Thedemand relationship between mobile application and graphic engine

在人们由注重功能本身向注重视觉等感观体验的转变这样的环境下，用计算机来表达的三维世界蕴藏着巨大的市场潜力，而图形引擎在计算机三维世界的地位不言而喻，所以研究图形引擎技术有着举足轻重的地位。正如终端系统中用户与文本及图形交互的需求模型所示（图1-2）：

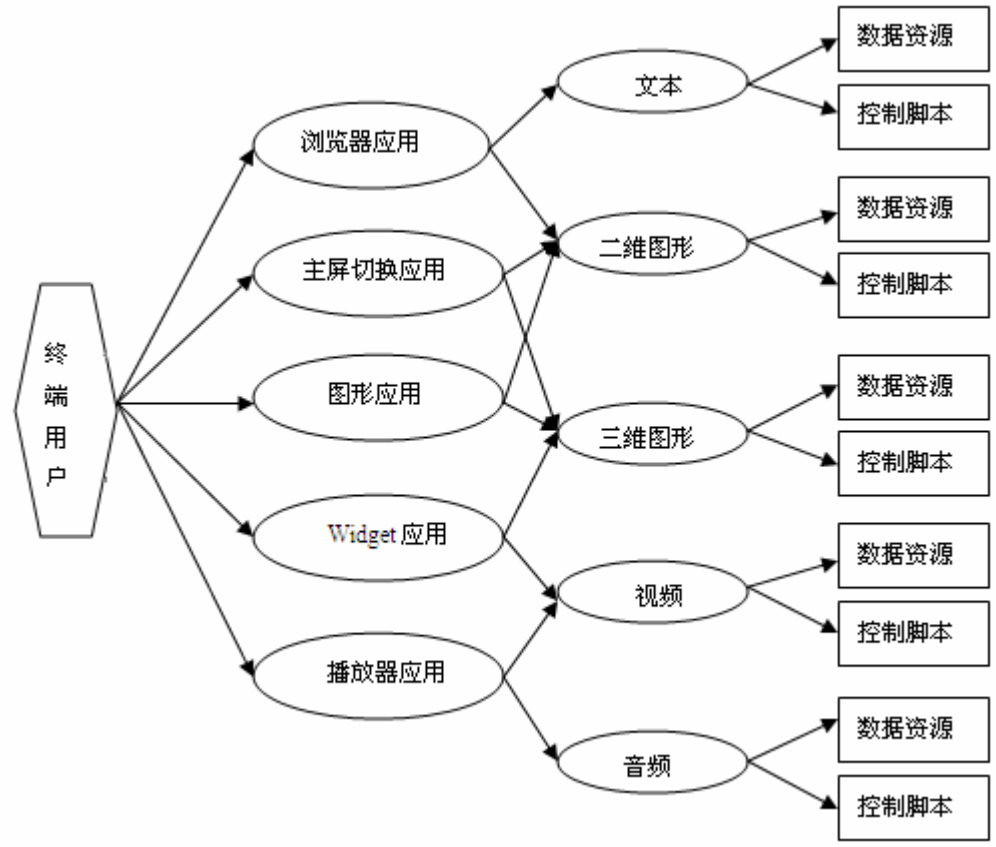


图 1-2 平台用户需求模型用例图

Fig.1-2Theuser case diagram of requirement model for end users

图形引擎作为一种底层工具在图形软件开发中扮演着十分重要的作用，所以研究渲染引擎将有十分重要的意义。

1.3 主要研究内容与论文结构

本论文将从研究渲染引擎的模块入手，分析各个模块和GPU数据操作的关系，以此提出一些基于GPU的优化方案。

论文第二章介绍渲染引擎的基本概念，以及组成渲染引擎的各个模块等。在这些模块当中我们将分别指出当前实现这些模块的当算算法以及本文将要介绍的改进算法。

从第三章起至第六章将会详细地分析和介绍这些改进算法。

第三章主要介绍物体模块。这一章首先介绍下通用简单物体单元，后面会着重介绍一种新的为减少对GPU调用的批物体单元的原理和实现方式。

第四章主要介绍纹理模块。在纹理模块中主要的是一种纹理集的算法，其原

理是减少一些琐碎小纹理载入GPU的开销，而合成一张大纹理可减少对GPU的调用提高性能。

第五章主要介绍渲染器模块。在这一章会了解一渲染器模块的职能并提出一种对物体按材质及距视角观察者距离远近排序的思想用来提升渲染性能。

第六章主要介绍选取系统。首先先了解一下选取算法的原理，然后会简单介绍了传统选取算法的是如何实现的并指出其弊端，接下来会引出这一章的重点即基于GPU的选取方式。

最后一章本文进行了对前面的几个优化方案进行了测试并给出了一些使用建议。

第二章渲染引擎的优化分析

本章将深入了解下渲染引擎的概念，以及级成组成渲染的模块。在介绍渲染引擎的模块时，将着重介绍本文有提出创新的几大模块：**批物体单元**、**纹理集**、**物体排序**、**基于GPU的选取系**统等。这些模块的创新之处都是围绕着GPU数据读写效率展开的，在物体模块合并顶点缓存，在纹理模块合并小纹理，在渲染器模块对物体排序来达到减少GPU调用的目的;在选取系统中利用GPU的特性来来做一种新的选取算法。

2.1 渲染引擎简介

2.1.1渲染引擎概念

随着3D产业的迅猛发展及竞争的越来越激烈，各大公司都在想更快更高效地推出其3D应用，可以说3D应用程序开发的周期越来越短。为了缩短项目周期，减少开发成本，人们将3D程序代码中一些通用的部分抽出来做成模块，并设计好与应用程序的接口，这就是所谓的渲染引擎[40]。

2.2.2渲染引擎模块

渲染引擎的架构实际上是由渲染系统及数据系统两层组成[39]，如图2-1所示

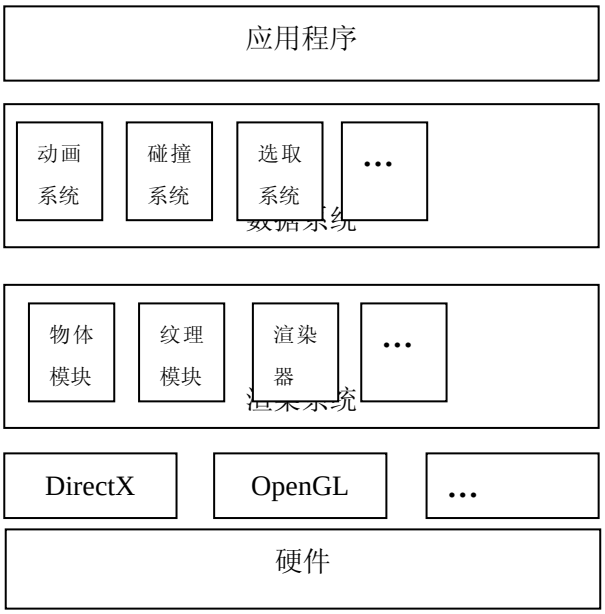


图 2-1 渲染引擎模块

Fig.2-1Components of GraphicEngine

2.1.1 渲染系统

渲染系统可以分成几个模块：

物体模块

物体模块主要负责是用户模型的创建。我们知道3D场景中的物体是由一系列的点（称为顶点）组成的，那么如何有效地管理这一系列格式，位长都有可能不同的点就是物体模块要做的事情。最简单的物体模块就是线性存储这一系列的点组成一个物体单元。

本文将要介绍一种新的物体单元即**批物体单元**。这种单元实际上是基于简元的，它是指当简单单元的物体材质相同时，可以将这些物体合并成一个大的物体单元，这样在渲染这些物体时我们只需要调用一次GPU的渲染函数即可，以此来达到提高性能的目的。

材质模块

材质是描述物体的表面对光的反映的一种程度[29]，通俗地讲它描述的是物体的表面特性，如木头是一种材质，金属是另一种材质。它们对光的吸收及反射的程序是不同的。图形学上材质表现为Shader中一些描述材质的计算公式，光源在计算中会被传到Shader中与材质进行计算，从而得到最终的颜色。

材质的参数指的是输入给Shader中技术材质计算公式的参数值。例如红色的金属和绿色的金属它们从材质上来讲是一种材质，它们所在Shader中的计算公式是相同的，不同的是它们材质的参数：一个是红色，一个是绿色。

纹理模块

纹理实际上可以算是材质上一个参数，但由于其往往需要占用较大的系统资源，引擎中我们往往会将纹理单独作为一个模块来管理。纹理模块通常需要处理的事情有：从磁盘读取图片文件、管理纹理内存空间(更新、编辑等)、纹理地址模式和采样问题等。

本文在纹理模块中将要介绍的是一种**纹理集**的概念，原理是当很多物体的材质相同，只有纹理不同的情况下，我们可以将这些材质上不同的纹理写到一张大的纹理（可以称为纹理集）中去，这样我们就只需要将这张大纹理连同这材质一次送入GPU即可，减少了GPU调用。

渲染器模块

渲染器模块主要处理是渲染流程的问题，如何时调用物体剪裁和相机剪裁以及每个渲染过程的顺序问题。本文在渲染器模块中主要提出的是**物体排序**的思想。这里的排序主要包括两个方面：第一是按材质，主要原理是GPU在切换不同的材质 (即Shader)时会带来很大的开销，如果我们将材质进行排序，即相同的材质顺序地送入GPU，这样就会减少切换Shader时产生的开销；另一种是按物体距视觉观察点的远近(即Z值大小)排序，这样做的目的是尽可能的减少重复读写和覆盖渲染缓存的动作。

2.1.2 数据系统

数据系统中常见的有动画系统，碰撞系统，选取系统等。本文将要着重介绍的是选取系统。

选取系统

在3D图形学中，经常需要对一些物体做一些编辑或高亮显示（以示区别其他物体），这时候我们就需要用到选取功能。选取的原理很简单，即由在屏幕上设定选取点，然后返回由该选取中点选定的物体。传统的选取算法是由观察点到屏幕上的选取点引出一条射线，然后将这条射线与3D世界里的物体进行求交运算，有交的物体即被选中的物体。这是传统的选取做法，利用的是CPU的计算能力。

在本文中，我们将介绍一种**基于GPU的选取算法**，它适用于最常见的一种选取类型，即点选可见的物体。这种算法的原理是将场景中物体赋上一种唯一的颜色，并将其渲染到一个离线的帧缓存中，这个颜色是与物体的编号一一对应，并能正反查找的。渲染完后我们会根据选取点的位置找到其在这个离线缓存中的相对位置，即取出该位置的颜色值，最后通过该颜色值找到选中物体的编号，这样就完成了用GPU来做选取的算法。

2.2 本文要用到的图形学术语

GPU

GPU英文全称Graphic Processing Unit，中文翻译为“图形处理器”。GPU是相对于CPU的一个概念，它所处理的是计算机中与图形专门相关的运算[34]。

DirectX/OpenGL

DirectX/OpenGL是目前运用得最广的两个性能卓越的三维图形标准。通俗地来它们是与底层硬件打交道的一个3D图形库，上层所有的渲染动作都通过DirectX/OpenGL来调用底层的硬件来完成的。对一个游戏引擎[36]来讲，目前较为流行的是Windows平台上用Directx而其它平台用OpenGL，所以两者需要支持。

材质

材质是描述物体表面特性一表达方式，如瓷砖是一种材质，水泥是一种材质，金属又是一种材质。不同材质之间对光的反应公式是不同的。

Shader

在图形学中Shader可以做两种理解：第一种做着色器理解，是指一组供计算机图形资源在执行渲染任务时使用的指令。程序员将着色器应用于图形处理器(GPU)的可编程流水线中，来实现三维应用程序[26]。目前可用的着色器主要有顶点着色器(vertex Shader)和像素着色器(pixel Shader)。Shader的另一种解释是指Shader文件，一个Shader文件通常包含有完整地用来渲染的指令代码，当然最重要的是一个顶点着色器和一个像素着色器（以及其它一些变更的声明等）。将一个Shader文件通过DirectX或OpenGL编辑成Shader命令再结合物体、光照、相机模型就形成了渲染的全部要素。在材质上表现形式上我们也用一个Shader文件来表示，在这个Shader文件中（主要是顶点和像素着色器中）记录了此种材质与光照/相机等信息的计算公式（不同的材质会有不同的计算方式）。

Shader Parameter

Shader文件中我们记录上主要是材质与光照、相机等信息的计算公式，而ShaderParameter则是指这些计算公式的输入值。比例木头和金属是两种材质（即它们的计算公式不同），而红色的金属和绿色的金属则属于同一种材质，两者与光照相机等信息的计算公式相同，所不同的是它们的材质的参数不同，一个输入是红色，一个输入是绿色。

Texture

纹理实际是材质的一部分，但由于纹理的技术细节较多，我们拿出来单独讨论。首先纹理实际上包括凹凸贴图，体纹理等多种，这里我们只讨论那种最为普通的2D的纹理贴图。为物体表面贴上纹理的过程实际是一个纹理映射的过程。

在物体创建过程中我们会指定物体的每个顶点所对应的纹理上的纹理坐标，在将三维世界物体的顶点经过投影变换变成屏幕上二维的点后与同样是二维世界的纹理上的像素点做比例映射，最后得出该该顶点的纹理值。需要注意，无论是什么样的的纹理，它们采用的坐标都是一样的，通常水平方向的叫u, 垂直方向的叫v，u和v都是归一化的坐标，即从0到1，即任何纹理(2D)的左上角坐标为(0，0)，右下角坐标为(1，1)。

Address Mode

Address Mode指的是当物体上的顶点做指定的纹理坐标超过[0，1]的时候如何处理的问题。我们设想一下，如果我们现在需要画一个棋盘（即黑白相间的格子），通常的办法就是找一张棋盘的纹理贴到一个平面上，即形成了棋盘的画面，然而如果棋盘面积较大，我们找一张棋盘相同大小的纹理有时也是困难的。其实我们只需要这个棋盘纹理中的一部分，然后再将其重复的铺展开来形成整个棋盘。这便是一种Address Mode。其它种类的AddressMode还有Mirror（顾名思义，反过来再平铺），Clamp（超过0-1的部分按边界颜色延伸）等。

Filtering

由前述我们可以得知，物体的顶点所对应的纹理上的坐标是由物体创建时确定的，而对于三个顶点之内的三角形内部的点我们是通过映射完成的（通常是通过线性比例来确定的），这样一来就存在一个问题，即当映射的点不正好是纹理上的某个像素中心时（纹理上的像素也是离散的）如何返回该点的颜色值，解决这个问题就需要用到滤波即Filtering。

MipMap

MipMap实际产是一种特殊的纹理映射技术，它的原理是根据观察者距离的不同，以不同的分辨率将单一的材质贴图以多重图像的形式代表平面纹理：尺寸最大的图像放在最前面的位置，而相对较小的的图像则后退到背景区域。MipMap每层的图像实际上是由硬件生成，第一层为原纹理，第二层为原纹理的一半，第三层为第二层的一半，一直分到最后一层为1x1像素的纹理为止。每一层又称为一个MipMap水平(Level)。

VertexBuffer/IndexBuffer

在图形学领域我们通常用物体的轮廓来描述物体就够了，而不需要对物体的

内部（即实心）进行描述。在实时渲染中我们又通常用一系列的三角形面片(即一连串的三角形的顶点)来表达这些轮廓。存储这些三角形顶点的数据结构称为顶点缓存。顶点缓存的存储位置即可以在内存中，也可以在显卡的显存中。

FVF

顶点格式（Flexible Vertex Format）指是每个顶点所包含的属性信息，如顶点的三维坐标，颜色，顶点法线和纹理坐标等。有了顶点格式，我们就能从顶点缓存的数据流中知道各个顶点的起始位置和结束位置。

Target

在图形学中，一个Target指的是显存中的一块区域，它存储的是用来渲染的渲染数据，根据其用途我们可以将其划分为可显示的Target(Presentable Target)及离线Target(Offscreen Target)。

Bounding Box

Bounding Box也称包围盒，即以一个体积稍大但特性简单的几何体包裹在复杂的几何对象外围，以此用来简化一些复杂运算。最常见的包围盒有AABB(Axis-Aligned Bounding Box)包围盒、包围球等[32]。包围盒最常见的运用场景即是做一些碰撞计算：当包围盒都与被计算的物体不交相时，那实际物体则肯定不相交，这样便可以加速一些运算。

2.3 本章小结

本章我们了解了渲染引擎的组成分析，一个渲染引擎通常由一个底层的渲染系统以及上层的数据系统组成。在渲染系统中我们主要介绍了几个模块：物体模块、纹理模块及渲染器模块，而在数据系统中我们着重介绍了选取系统，在这几个模块和系统中我们都分别提出了一些利用GPU的特性来进行优化的方案，这也是本文的特点和创新所在。最后我们还介绍了一些本文中要用到的图形学的术语，以方便后面我们对优化方案的阐述。

第三章批物体单元

本章将深入了解物体模块的细节，包括什么是物体模块、通用简单模块的设计与实现等，本章后半段将会用大部分篇幅来介绍一种新型的物体单元，即批物体单元（BatchMesh），这种物体单元又分静态和动态两种：静态批单元相当于一个容器可以将各大小不同的物体单元合并成一个大的单元，而合并的条件是这些物体的材质必须相同而且运行性质一致（不会相对运动）。而动态批单元则是用一份顶点缓存和多个运动矩阵形成多个物体实例。无论静态还是动态批单元的作用在于减少对GPU的调用，提升渲染的性能。

3.1 物体模块的概念

由渲染的原理我知道，在图形学中一个物体实际上是由模型轮廓再加上外表的材质所组成。描述一个模型的轮廓有很多种办法：如B样条曲线，贝塞尔曲线等，而在实时渲染中用得最普遍的还是三角形面片的形式，即一个模型轮廓由一系列的三角形面片组成(三角形面片的多少取决于要表达的轮廓的细节程度)。而三角形面片又是由三维空间中的顶点组成的，所示我们可以说物体模型实际就是一系列顶点的组合。

在渲染时这一系列的顶点会被存放在一个缓存中，即顶点缓存(vertex buffer)。为了节约存储空间我们还需要用到一个索引缓存。索引缓存的意思如图3-1。如图中两个三角形，如果全部都用顶点表示，那么我们就需要6个顶点，但实际上我们只需要4个顶点，然后用索引表示(0，1，2)为一个三角形，(0，2，3)为第二个三角形。

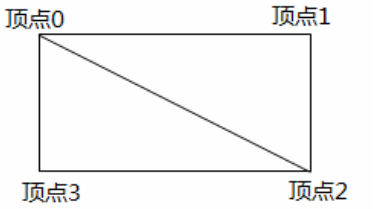


图 3-1 索引缓存概念

Fig.3-1Concept of Index buffer

在渲染引擎中，一个物体模块表现为一个类，这个类的作用就是处理用户输入的顶点及索引，一个最基本的物体模块必须满足以下功能：

第一、 允许用户输入和更新顶点缓存和索引缓存

第二、 允许用户输入和改变顶点格式

第三、 返回顶点的步长

3.2 简单物体单元

所谓简单物体单元即满足上述所说的三点要求即是最简单的物体单元。不过在渲染引擎中，为了可扩展性我们通常会声明一个物体单元的基类，而简单物体单元则是继承这个基类的实体类，除此之外，如果用户想要实现一个新的物体单元，即继续这个基类完成一些必要函数即可。

3.3 批物体单元

在实际的渲染应用中我们经常碰到需要创建很多相同或相似的实例：例如广场上的很多人、地面上的很多花、车子上几个相同的轮胎等。这些实例中的物体一个共同点就是它们本身的物体模型都是相同的，它们之间不同点在于有些是一起动的（如地面上的花，车子的四个轮子通常都是一起运动的），有些是分开动的（如广场上的人可能会朝不同的方向走动）。

根据这些物体模型的特点，我们可以将这些物体进行一些批处理，批处理的意思是将这些物体一起送入GPU以减少GPU调用。根据这些物体间的相对关系，我们又可以将这种批处理分成两种：即动态和静态。动态批处理如广场上的人，虽然它们的个体的物体模型都相同，但每个物体运动的方向轨迹可能不同。动态批处理的原理主要依赖DirectX和OpenGL中的API，利用的是GPU的功能。而静态批处理如地上的花，它们的个体的物体模型相同，而且它们不会相对运动，而只会像一个整体一样一起运动。静态批处理的思想建立一个大的物体单元把这些材质相同的物体合成一个大的物体单元。无论静态还是动态做的目的都是减少GPU调用以达到提高性能的目的。

3.3.1 动态批处理

动态批处理的思想如图所示3-2所示：

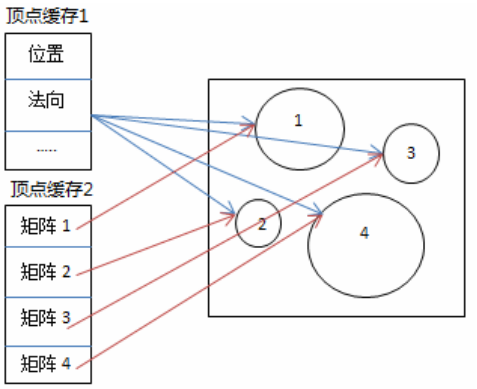


图 3-2 动态批处理示意图

Fig.3-2Dynamic Batching

这里的关键点在于维护了两个顶点缓存，顶点缓存1用来存放物体的顶点信息，而顶点缓存2则是用来存放每个物体特有的信息，如大小、位置、运动矩阵等。顶点缓存2的中每个矩阵和顶点缓存1中的顶点信息结合起来就成了一个实例。

要了解该技术的详细过程我们需要了解的一个概念是IA(input assembler)。IA实际相当于在顶点着器和顶点缓存区之间的一个处理单元。通常我们准备好顶点缓存区后并不是直接传给顶点着色器，而是先送入IA，在经过IA处理后的数据再送入顶点着色器。

由图3-2我们可以看出动态批处理的基本思想将存储每个实例不同的信息的顶点缓存区与存储实例中相同的信息的另一个顶点缓存区结合起来生成不同实例的顶点信息。举个简单的例子，场景中有很多球体，每个球体的原始顶点相同，只是位置不同。按照传统的做法伪代码如下：

```
for( i : 0 to sphereNum )
    {
        set world matrix
        draw sphere i
    }
```

上面的方法也是最笨拙的方法，即在画每一帧的时候对所有物体遍历，每画到一个物体时再将这个物体变换的矩阵（大小、位置等信息）赋给这个物体。仔细观察会发现其实这个过程某种程度上存在着很大的重复，即每次设置物体变换矩阵的过程实际上是很相似，即相当于更新某个固定的变量。技术上我们完全可以将这个变量一起存储到顶点信息当中，如同顶点信息中的法线向量、纹理坐标一样，即顶点着色器的输入多了一个矩阵(通常是4x4的float形式)，然后在顶点计

算时将这个矩阵计算在内即可。由前面IA的介绍我们知道顶点着色器输入实际上是IA送入的，所以我们要解决的问题变成怎么样利用IA生成我们想要的信息：

假设需要渲染100个球体，每个球体256个三角形。按照前面最笨拙的办法，我们可以在CPU端写入256*100（*表示相乘，下同）个三角形的顶点数据，正如我们前面讲了这做太低效了。正确的做法是利用DirectX的API来帮助我们完成。

利用DirectX接口我们需要两个顶点缓冲区，一个来描述球体的顶点信息，即256个三角形的信息，共768个顶点。另一个用来描述位置信息，一个位置信息实际上就是一个变换矩阵，这个矩阵存储在顶点中，即需要100个顶点存储。两个加起来还不到1000个顶点，相对于上面的250*3*100=768000个顶点少了上千倍。下面看IA如何处理使得这不到1000个顶点即可达到我们想要的结果。首先我们需要设置输入格式：

```
//the vertex layout
D3D10_INPUT_ELEMENT_DESC layout[] =
{
{ "POSITION" , 0 , DXGI_FORMAT_R32G32B32_FLOAT , 0 , 0 , D3D10_INPUT_PER_VERTEX_DATA , 0 } ,
{ "mTransform" , 0 , DXGI_FORMAT_R32G32B32_FLOAT , 1 , 0 , D3D10_INPUT_PER_INSTANCE_DATA , 1 }
};
```

我们需要将两个顶点缓存区设置成如上的格式。第一个元素描述的是球体的顶点信息；第二个元素描述的是位置信息。我们注意到第五个参数，D3D10_INPUT_PER_VERTEX_DATA/D3D10_INPUT_PER_INSTANCE_DATA，这里面如果是前者，IA会把顶点缓存中的每个顶点当做顶点处理。而如果是后者，IA实际上是把顶点缓冲区中的每个元素当做实例来处理的。那么最后IA可以为我们生成顶点数*实例数* numberOfInstance个顶点数据，这也正是我们想要的数据。

有了这些数据，我们就可以进行顶点处理了。看看VS中有什么变化：

```
//the input struct of the vertex Shader
struct     VS_INPUT
{
    //the instance id
    uint     uInstanceID : SV_InstanceID;
    //the position of the particle
    float3     vPosition : POSITION;
    //the instanced position
```

```
    float3    vTransform : mTransform;
};
//the default vertex Shader
VS_OUTPUT    DefaultVertexShader( VS_INPUT input )
{
    //the output of the vertex Shader
    VS_OUTPUT vs_out;
    //transform the vertex
    vs_out.vPosProj = mul( float4( ( input.vPosition + input.vTransform ) , 1.0f ) , ViewProjMatrix );
    //pass the normal
    vs_out.vNormal = input.vPosition.xyz;
    //copy the color
    vs_out.vColor = ColorBuffer[input.uInstanceID % 8];
    //return the output struct
    return vs_out;
}
```

我们注意到我们为每个顶点变换的矩阵是ViewProjectionMatrix（投影变换），而不是WorldMatrix（世界变换），原因是每个顶点的World Matrix都已经存储到了顶点结构中的vTransform中。实际上input.vPosition+ input.vTransform就相当于做了WorldMatrix的变换。

通过上面的步骤，就可以渲染出来不同位置的球体了。至此我们也就完成了动态批处理的过程。

3.3.2 静态批处理

静态批处理原理如图3-3所示：

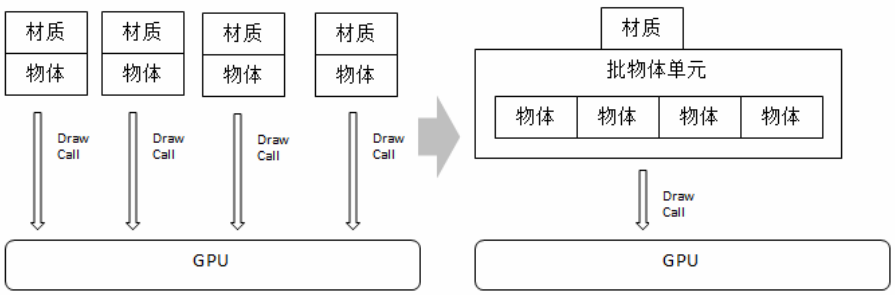


图 3-3 简单物体单元和静态批物体单元对 GPU 的调用

Fig.3-3 GPU Draw call for simple mesh and instanced mesh

静态批处理的思相与动态批处理不同，静态批处理的物体之间相互关系固定，也就是说它们之间变化的矩阵是一样，因此我们实际可以将这些物体合成一个大的物体。

静态批处理的意思是对将这些一起运动（即不相对运动）的物体为它们建立

一个大的顶点缓存将各小物体的顶点缓存拷贝到这块大的顶点缓存区中，即将各小物体捆绑形成一个大物体。为了实现静态批次，先创建一个用来存储捆绑后大物体的顶点缓冲区（当然也包括索引缓冲）。需要保证这个这个缓冲足够大，足以储存我们希望处理的所有实体。由于我们只对缓冲进行一次填充，并且不再做修改。在Direct3D中我们可以用D3DUSAGE_WRITEONLY标志来提示驱动程序把缓冲放到速度最快的可用显存中：

```
    HRESULTres;
    res = lpDevice ->CreateVertexBuffer( MAX_STATIC_BUFFER_SIZE, D3DUSAGE_WRITE, 0,
D3DPOOL_MANAGED,&mStaticVertexStream, 0 );
```

接下来实现Commit()方法。它将把需要渲染的经过坐标变换的几何体数据填充到顶点和索引缓冲中。以下是Commit方法的伪代码实现：

```
Foreach GeometryInstance inInstances
    Begin
    transform geometry in mGeometryPack toworld space with instance mModelMatrix
    Apply other instnce attributes(likeinstace color)
    Copy transformed geometry to the VertexBuffer
    Copy indices ( with the right offset)to the Index Buffer
    Advance current pointer to the VertexBuffer
    Advance currect pointer to the IndexBuffer
    End
```

接下来就只需要调用真正的画函数方法，提交这些准备好的数据即可。

静态批处理是渲染大量实体最快的方法，它的不便之处在于它只适用于物体不相对运动的情况而且还有些其它限制：

占用内存过多；根据几何包大小和希望渲染的实体数量，内存数量可能会变得很大。注：当显示不够时，我们可以把数据分页放到AGP Memory中，但这样会降低效率，因些应该尽量避免。

不支持多种LOD(Level Of Detail)；LOD指的是物体的顶点信息会根据其距视觉观察点的远近来改变，如距离较近时表达物体信息的顶点较多，而当距离较远时顶点会相对较少。如果我们采用了静态批处理，由于物体的顶点信息已经拷贝到了大的物体的顶点缓存中，很难再根据距离远近来做出改变，如果非要处理的话，我们需要为每个LOD建立多个大顶点缓存区，这样才可以才可以保证每个小物体根据不同的LOD获得想要的顶点信息，但这样就违背了我们当初设计静态批处理简单高效的原则。

3.4 测试结果

最后对静态和动态批处理都进行了测试，在动态批处理测试过程中，应用画了1000000个带漫反射光照的多边形。我们测试的目的是看多边形的数目对于FPS(Frame Per Second，即帧率)的影响，如图3-4所示，没有进行批处理和动态批处理的对比。

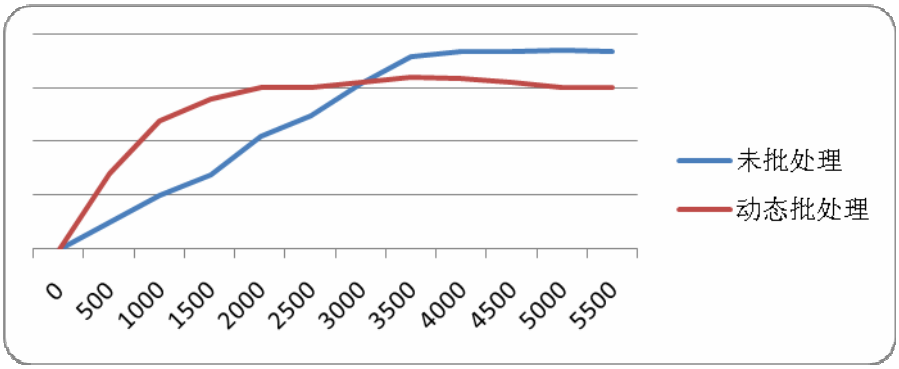


图 3-4 动态批处理的测试结果

Fig.3-4Dynamic batching test result

从结果中我们可以看出在一起进行批处理的多边形的数目较小时动态批处理很快能体现出自己的优势，FPS甚至比未批处理时高出几倍。而一旦当多边形数目较大时，由于GPU对每个批处理中的矩阵处理的开销使得动态批处理的性能下降甚至不如未批处理的情形，因此动态批处理应尽量用于小场景或批处理物体数目不多的情形。

静态批处理测试结果如图3-5所示：由于其不受GPU的限制，其性能方面要优于动态批处理(它们用于不于的情形)，而且也不会随着批处理的数量增大而碰到瓶颈限制。但需要注意的是当静态批处理的物体数量较大时，对系统内存的要求会越来越高，往往会靠成程序崩溃的现象或强迫系统内存与硬盘进行页交换，对性能会造成间接影响。

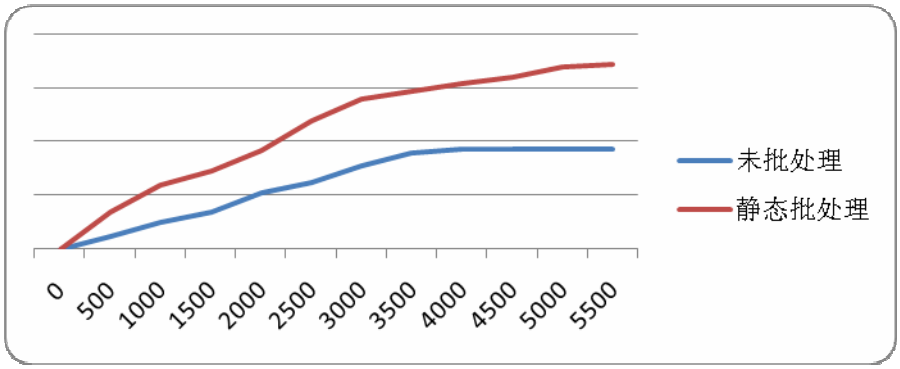


图 3-5 静态批处理的测试结果

Fig.3-5Static batching test result

3.5 本章小结

本章主要介绍了渲染引擎的一个重要模块-物体模块。文章一开始介绍了物体模块的概念以及功能等，随后本文引出来批处理的概念，即将多个满足一定条件的物体一起送入GPU，以达到减少GPU调用的目的。批处理又分为动态批处理和静态批处理，它们分别用于物体间相对可以运动以及相对之间关系固定的场景。动态批处理运用的是DirectX或OpenGL的API来实现的，运用的是GPU的能力。而静态批处理刚是在CPU端将各物体的顶点缓存合成一个大的顶点缓存。它们虽然方面不同，运用场景不同，但作用都是为了减少GPU调用。最后我们对两种批处理都进行了测试。测试显示出批处理较未批处理体现出了较好的性能优势。

第四章纹理集

本章将会着重介绍纹理模块中的纹理集的概念。纹理集主要适用于当物体的材质相同，而材质上的参数除了纹理之外其余都相同（典型的应用如人物造型，人物造型上各个部位材质是相同的，都为皮肤，然而它们所带的纹理可能有所不同）的情形。在这种情况下如果将物体及材质一个一个地送入GPU中势必会导致效率低下，GPU从磁盘上读取纹理文件的过程往往较慢，而且需要每次都激活Shader来读取纹理文件。纹理集的一个思想是用预处理的方式将这些材质上的小纹理合成一张大纹理，这样就只需要一次性地送入GPU，减少了频繁读取纹理的过程，提升效率。

4.1 材质和纹理的关系

在理解纹理集前我们首先要了解一下材质和纹理[38]两个概念。正如我们第二章术语中所讲的材质实际是描述物体表面的一种特性，图形学上来讲它实质上是一组计算物体表面对光的反映的数学公式，存储这个数学公式的文件称Shader。通常我们也理解成一个材质就是一个Shader。而同一材质之间的对光计算的公式相同所不同的只是传统Shader的参数不同。纹理实际上是这个输入参数中的一个，但纹理与其它参数（如颜色）不同，纹理还涉及到磁盘的读取，CPU和GPU内存的存储以及纹理映射时的一些细节技术，所以渲染引擎中通常都会将纹理单独做为一个模块。

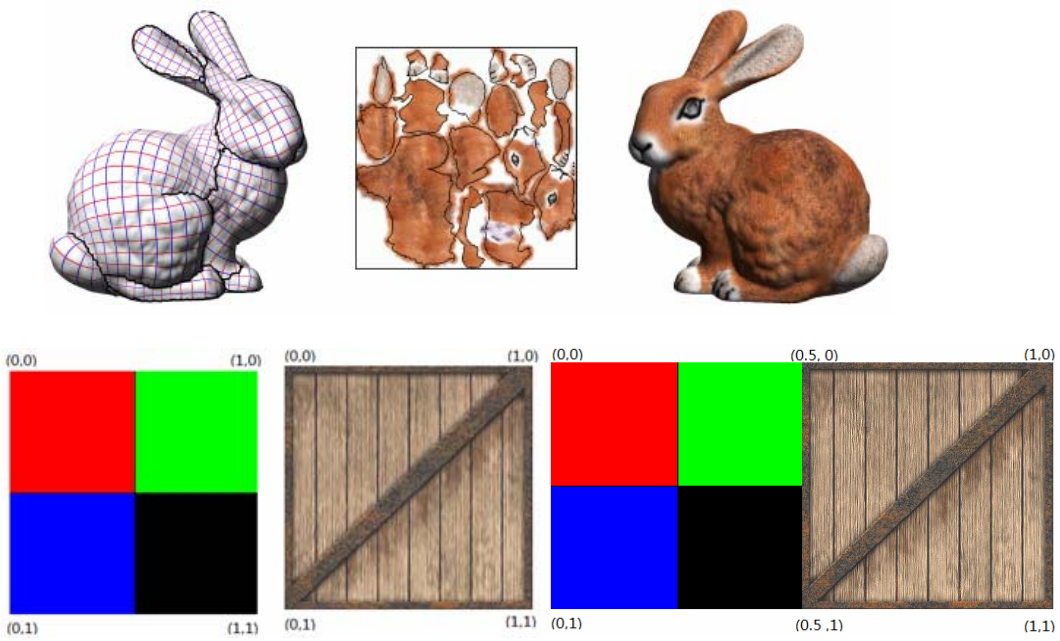
纹理实际上包括很多种，通常最常见是2D贴图，还有3D，及体纹理。而根据其产生的方式不同，又有文件纹理和内存纹理（即由计算在内存中计算出来的纹理，而非从磁盘读取）。纹理从本质上是内存中的一块区域，里面存储的是纹理的颜色值[19]。

4.2 纹理集概念

纹理集技术的应用前提是物体是材质（即Shader）相同，材质的参数上唯一不同的是材质（其它如颜色等都相同）。由于Shader相同，我们可以将这个Shader

文件一次性送入GPU，然后再将每个参数上的材质传给Shader时，需要激活Shader，而如果每个物体的纹理不同时，每渲染一个物体时就需要激活这个Shader来接受参数，这样就无形地带来了开销，而将每个物体上的不同材质合并成一张大的纹理后，然后再将这个大纹理一次性传给Shader，这样就节省了激活Shader的时间，每个物体上的小纹理分别对应这张大纹理上的一部分。

下面来举例说明这个减少GPU调用的过程：我们现在需要画两个物体，按通常的做法我们会将第一张纹理赋给（调用SetTexture()函数）第一个物体，然后开始画（调用DrawPrimitive()函数），再将第二张Texture通过SetTexture函数赋给第二个物体，再调用DrawPrimitive()画第二个物体。这样整个流程就用到两次Draw Call(即两次对GPU的调用)。而如果将两个纹理合并到一个张纹理集中里，如图4-1所示则无需要在两次DrawPrimitive()之间再次调用SetTexture(DirectX/OpenGL API)函数，也就是说我们只需要一次DrawPrimitive即可，即将Draw Call从两次减少到了一次。



(a) (b)

图 4-1 分开的小纹理(a)和合并后的纹理集(b)

Fig.4-1Separated textures (a) and texture set (b)

纹理集技术主要分几步：

第一步、创建一个纹理集类，其主要功能是用来将小纹理合并成一张大的纹理。

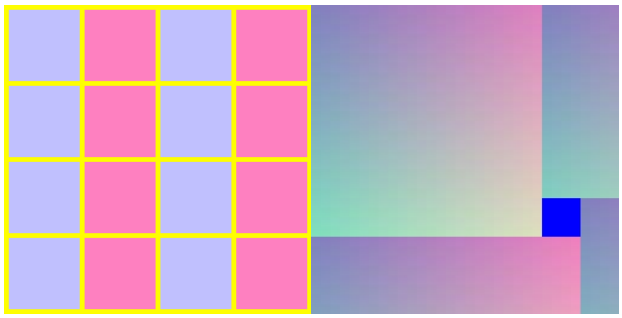
第二步、将每个物体上的小纹理传给纹理集类来创建一张大纹理，并将每个物体上小纹理替换成这张大的纹理。

第三步、根据每张小的纹理在纹理集中的相对位置更新UV坐标。

第四步、将序列化的DrawPrimtive()合并成一个DrawPrimitive()即用一次Draw Call即可。

这其中有几个技术要点我们需要解决：纹理集空间问题、纹理寻址问题、纹理过滤问题以及Mip-Map问题。

第一、纹理集空间问题是指如何将大小不同的小纹理有效地加入到纹理集中并且能保证访问过程快速而有效的。如何将各小纹理合并到大纹理中，这其中包括规则(Uniform)和不规则(Non-uniform)两种方式：规则的纹理集意思是指纹理集中分配给各小纹理的大小是固定的。这种方法的优点在于方便管理，计算方便，而缺点则是消费空间；而不规则纹理集则是指按各小纹理实际的尺寸最优化地存储到纹理集中。这样优点则是节省空间，而缺点则是计算起来较为复杂。如图4-2所示：



（a） (b)

图 4-2 规则纹理集(a)和不规则纹理集(b)

Fig.4-2Uniform texture set (a) and Non-uniform texture set (b)

这两方式各有优缺点，应用开发者需按实际的需要来进行选择。如需要渲染一个人物体模型，人物模型各部分的纹理差异通常是较大的，如耳、眼、嘴、手、身体等部位的纹理样式大小都不尽相同，在这种情况下应使用不规则纹理集。而如果需要渲染一个如玩具其它模型，其模型的各个部分大小都差不多，使用的纹理尺寸也相当，在这种情况下用规则纹理则会即节省空间又能保证访问速度。值得一得的是在不规则纹理集中如何将大小不一要的纹理以最优排列放在纹理集中实际上一个数学问题，并且已有成熟的算法，本论文不多赘述。

第二、纹理寻址问题，术语叫做Address Mode。其处理是实际上让纹理UV超出(0，1)范围该如何处理的问题，比如是平铺、超过部分留空白等。单个纹理的时候当UV超出(0，1)的时候处理很简单，但当纹理被放到纹理集中时当访问原来纹理超出(0，1)的时候则可能会与邻居纹纹理冲突。如图4-3所示：

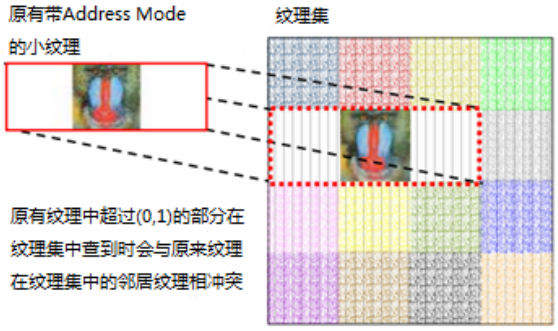


图 4-3 带有 Address Mode 的纹理在纹理集中示意图

Fig.4-3Address Mode texture in texture set

处理这个问题一种比较简单的办法就是将带Address Mode的小纹理按照Address Mode方式展开后再放入纹理集。例如需要用到的某个纹理的UV值是(2，3)，则我们可以将这个纹理横向重复两次竖向重复三次，然后再将重复后的纹理放入纹理集中。这样所带来的弊端则是非常的占用空间，尤其是当UV值较大时，纹理集的尺寸将是变得非常大。

在实际应用过程中，我们通常不会使用这种方法，而是会在像素着色器中(Pixel Shader)中重写Ad

```
//在pixel Shader中自己实现Address Mode(本例中为Clamp方式)
ps_2_x
dcl_2d s0
dcl_2d s1
dcl t0.xyzw
dcl t1.xyzw

// c0 contains: left, top, width, height
// c1 contains: 1/width, 1/height, 0.5, 1
// c2 contains: kMinU, kMinV, kMaxU, kMaxV
sub r0.xy, t1, c0
mul r0.xy, r0, c1                // bring coordinates into normalized local texture coord [0..1]
frc r0.xy, r0                    // if texture repeats then coords are > 1, use frc to bring
// these coords back into [0, 1) interval.
mad r0.xy, r0, c0.zw, c0.xy // transform coords back to texture atlas coords

// clamp to inside texture (to avoid bi-linear filter pulling in foreign texels)
max r0.xy, r0.xy, c2.xy
```

```
min r0.xy, r0.xy, c2.zw

dsx         r1, t1                  // use the original coords for mip-map calculation
dsy         r2, t1
texldd_pp r0, r0, s1, r1, r2

mov_pp      oC0, r0
```

图 4-4 在 pixel Shader 中自己实现 Address Mode(Clamp)

Fig.4-4 Implement Address Mode (Clamp) in pixel Shader

在Pixel Shader中自己实现Address Mode的主要思想是在使用Address Mode时临时换回原有小纹理(0，1)的范围，而在使用完Address Mode完之后再变换回来。即Address Mode中超出(0，1)的部分仍然会到原来小纹理中去查找，这样就不会与纹理集中的邻居纹理形成冲突。

第三、纹理过滤问题(Texture Filtering)是指当在纹理中的采样点不正好是像素中正中心时该如何返回像素值的问题，Texture Filtering的方式通常有Nearest Neighbor、Bilinear Filtering、Trilinear Filtering、Anisotropic Filtering等。

任何一个在纹理上的采取点周围都可以找到与其相邻的四个像素点的中心(如图4-5中的C0，C1，C2，C3点，M点为采样点)，如果是Neareast Neighbor即选取与白色的点的最相近的一个像素中心点的颜色值做为采取点的值，图中即为C0。而如果是Bilinear Filtering则是先将横向像素（C3C2和C0C1）加权平均（权重为其到采取点的距离，这里的距离不是点到点的距离，而是横向距离），然后得到的两个颜色值再根据其到采取点的纵向值加权平均得出最后的颜色值[31]。

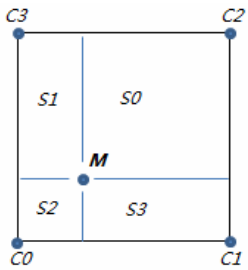


图 4-5 双线性滤波示意图

Fig.4-5 Bilinear sampling diagram

如图4-5采样点M周围的4个整数坐标点C0、C1、C2、C3对M点的权重依次为

$$\frac{S_0}{S_0+S_1+S_2+S_3}\ 、\ \frac{S_1}{S_0+S_1+S_2+S_3}\ 、\ \frac{S_2}{S_0+S_1+S_2+S_3}\ 、$$

$\frac{S_3}{S_0+S_1+S_2+S_3}$。M坐标点的像素值(对于RGB模式图像来说每个颜色分量R、G、B的计算方法都一样)为：。

运用到纹理集后所出现在问题主要体现在边界上，当采样点在采到纹理最周边的宽度为1的像素时，在如果在原有小纹理中进行，进行加权平均的像素点有可能会超出纹理的范围即空白区域，而如果在纹理集中运行刚会采到旁边的纹理中。解决这个问题我们有两种解决办法：第一种办法就是在其原来小纹理周边加上一圈1x1的空白像素再加到纹理集中，如图4-2中(a)所示。这样当采样到原小纹理边缘时到则会采到这1x1的空白外围，不会邻居纹理冲突。第二种办法是像Address Mode一样在Pixel Shader中自己写代码做过滤，过滤的思想与Address Mode相同即将纹理集的坐标临时换成原有小纹理的坐标，等过滤结束后再换回来。代码可参照图4-4。

第四、Mip-Map问题。Mip-Map实际上也可以算是纹理过滤的一种，它所处理的也是采样点的像素值的问题。只不过Mip-Map较前面的最邻、双线性等过滤方式有所不同，它还需要解决一个细节层次的问题。简单来说当观察视角距物体较近时我们希望能看到更多的细节，而当视角较远时则可以减少细节。纹理也一样，当视角距离较近时，我们希望纹理的分辨率较高，视角距离较远时，分辨率可以较低。纹理的分辨率集中现在纹理中像素的多少，比如一个16x16纹理，当我们视角较近时，则16x16的像素则可能全部用到，而当视角较远时我们可能只需要8x8的纹理。如果使8x8的像素看起来与16x16的像素看起来感觉是一张纹理，这就是Mip-Map所要解决的问题。

Mip-Map的原理是首先将纹理由硬件划分成很多Mip层，原纹理为最上一层，其下每一层是上一层的一半。如128x128的纹理，其Mip层依次为128×128， 64×64，32×32，16×16，8×8，4×4，2×2，1×1。在实际应用中我们会通过观察视角与物体的距离计算出一个细节距离(LOD，Level Of Detail)，这个细节距离是一个浮点值，也就是说它会落在某两个整数Mip层之间，在带有Mip-Map的纹理在计算时通过LOD与最近的两个Mip层之间的距离做加权计算。就是Mip-Map的计算过程。

Mip-Map运用到纹理集后出现的问题主要发现生在2x2的Mip层上，如图4-6所示：

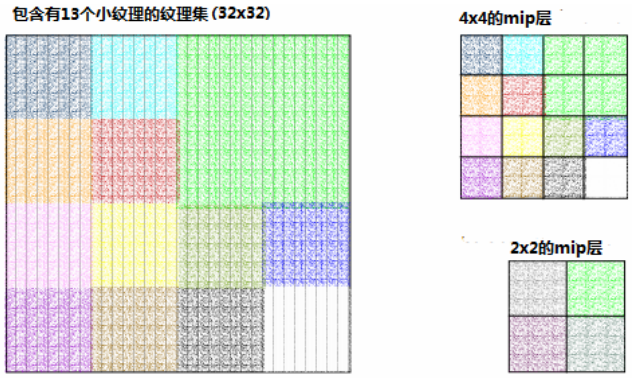


图 4-6 纹理集的 mip-map，出错 2x2mip 层

Fig.4-6Texture set with mip-map, pollution in 2x2 mip-level

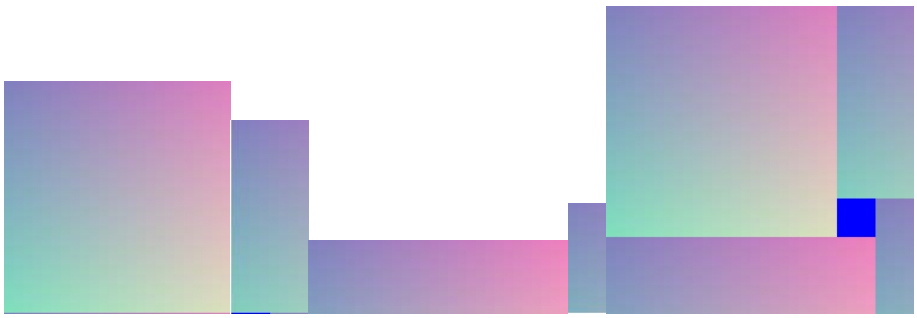
从图4-6我们可以看出32x32的纹理集到4x4的Mip层还是正常的，因为其每个小格子里的像素都是原来纹理中每个单独块像素平均得到的，它并没有与相邻的纹理产生冲突，而在接下来的2x2层就产生了污染，因为每个方块纹理在进行下一层计算时已经涉及到了相邻纹理。

为了解决这个问题，一种办法是将Mip-Map链层（链层指的是由最原始纹理直至至最小纹理层所形成的链）的长度（即Mip层的层次，如128x128的纹理其Mip-Map链层的长度为8）限制到最小纹理的上一层上，而非1x1的层。另一种解决办法就是只允许相同（或者相似）大小的纹理合并到同一张纹理集中。

4.3 测试结果

测试一：

测试输入为四个不同纹理，以不规则的进行加入到纹理集中，如图4-7所示



（a）(b)

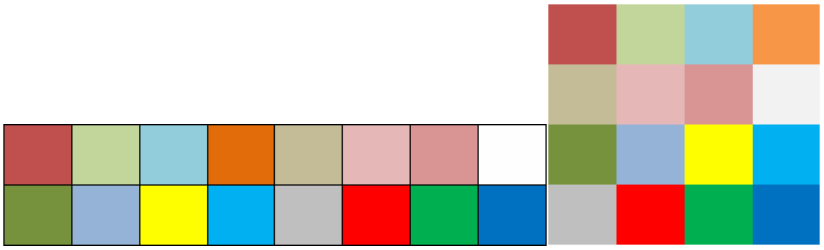
图 4-7 合并的纹理(a)及合并后的纹理集(b)–不规则

Fig.4-7 Un-merged textures (a) and Texture set (b) - Non-Uniform

结果：未合并：0.254s(渲染四个Quad的总时间，单位为秒)；合并：创建纹理集时：0.128s渲染时间：0.182s

测试二：

测试输入为16张大小相同的小纹理，为了方便测试，每种纹理都利用Procedural Program产生一种单一颜色，如图4-8所示：



(a) (b)

图 4-8 合并的纹理(a)及合并后的纹理集(b)–规则

Fig.4-8 Un-merged textures (a) and Texture set (b) – Uniform

结果：未合并：0.238s(渲染四个Quad的总时间，单位为秒)；合并：创建纹理集时：0.042s渲染时间：0.166s

测试二：

测试输入为需要合并的小纹理的张数（由少到多，为了便于观察结果我们使用的是规则的纹理集模式），输出为合并后渲染时间与未合并渲染的时间比。如图4-9所示：

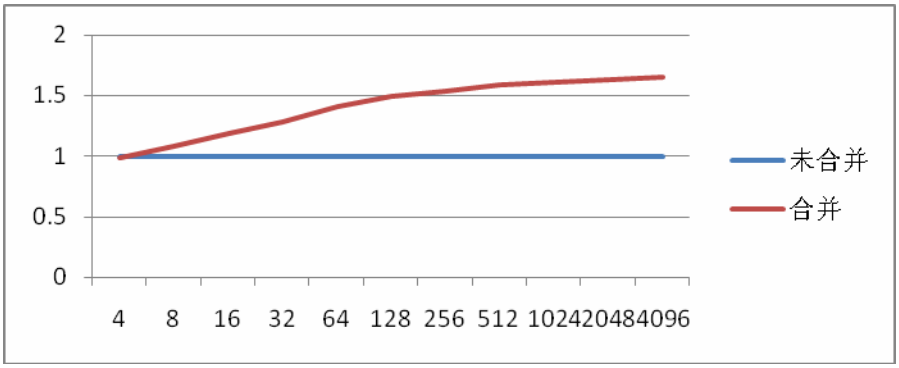


图 4-9 合并的纹理张数对优化性能的影响

Fig.4-9How the number of merging texture affects the final rendering performance

说明：从上述结果中我们可以看出纹理集创建是需要占用一定的时间，尤其是在不规则纹理集的情况下由于需要将各小纹理按最优方式合并成一张大纹理集，这个算法本身就需要花占用一定时间。但这不影响我们最后的效率，因为在用法上创建纹理集是在预渲染阶段完成的，即用户先用纹理集算法组织成大纹理后再进

行渲染，而且当合并的纹理越多，优化后效率的提升越明显（纹理越多纹理集创建时所产生的影响就会越小）。

4.4 本章小结

本章我们主要介绍了纹理模块中纹理集的概念，纹理集的思想是将一组小的纹理合并到一张大的纹理中以节省Shader因每次载入新的纹理而进行激活的时间以及频繁地从磁盘上读取的开销。随后我们介绍了运用纹理集过程中需要处理的几个关键问题，包括如何以最优结果将小纹理放入纹理集中、如何处理Address Mode问题、Texture Filtering问题以及Mip-Map问题。文章分别对这几个问题给出了分析以及解决方案。最后我们对纹理集的结果进行了测试，结果显示除去纹理集创建时候间，运用纹理集比不运用纹理集性能高出近30%，并且合并的小纹理数量越多，所获得的性能优势越大。

第五章物体排序

本章将深入了解渲染器模块，包括渲染器的概念以及其在渲染流程中的作用等，并且还将引入一个对物体进行排序的思想。这里排序主要包括两个方面：第一是按材质排序，其目的是为了减少GPU在切换不同材质需要保存和重新激活上下文的开销。另一方面是按物体Z值大小排序，这样做的目的是为了尽可能的减少在帧缓存上重复画的动作。材质排序的算法非常简单，而对于物体按Z值排序我们无需非常精确，而只需要根据物体的包围盒中心的Z值用来排序即可。

5.1 渲染器的作用

渲染器虽然称作一个模块，但它实际做的事情是将各个实际功能模块（如果相机模块、光照模块等）组合起来，形成一个可以渲染的管线。我们可以为不同的渲染功能渲染场景建立不同的渲染器，一个典型的应用就是可以为透明场景和不透明场景建立两种渲染器。这两种渲染器的流程大致相同，比如先调用物体剪裁组件，相机组件，光照组件，再调用纹理组件，渲染状态组件，渲染组件等。而它们的不同这处则在于透明场景我们还需要一个专门处理透明物体的组件（可以称之为透明组件）。用户只需要必要的组件加到渲染器中再调用渲染器的接口进行渲染即可，渲染器内部会为用户进行组织、调整这些组件以达到最优的管线效果。

5.2 按材质排序

材质排序的主要思想是GPU在切换材质(Shader)时往往非常耗时，其主要原因是切换Shader时往往会导致大量的状态缓存(Cache)失效。比如:

1. Shader中Z Test和Alpha Test会决定是否启用Heirachical Z，如这类的Shader切换发生，则需要把之前的Z 缓存都释放再重新填充；

2. Shader对Texture Sampler Slot使用的不同可能会导致不用的Texture Cache Layout，往往需要将整个Texture Cache重置；

3. 有些比较长的Shader，其指令并不能放在GPU的内存中上，那么切换时就需要从外存中载入，这个开销相对较小；

总结来讲，Shader切换时往往会伴随着渲染状态，渲染器资源和顶点和索引缓存的无效，以及大量变量的更新，也就是说整个管线都需要重新去填充，所以往往开销非常大。

为了避免从一种材质切换到另一种材质上给GPU带来的开销，我们需要尽量将同一种类型的材质一起送入GPU中，这即是材质排序的思想。材质排序的过程非常简单，但带过来的后果却是相当有益处的。材质排序示意图如图5-1，从图中我们可以轻易看出在未排序时需要进行材质切换5次，而在排序后则只需要两次即可。

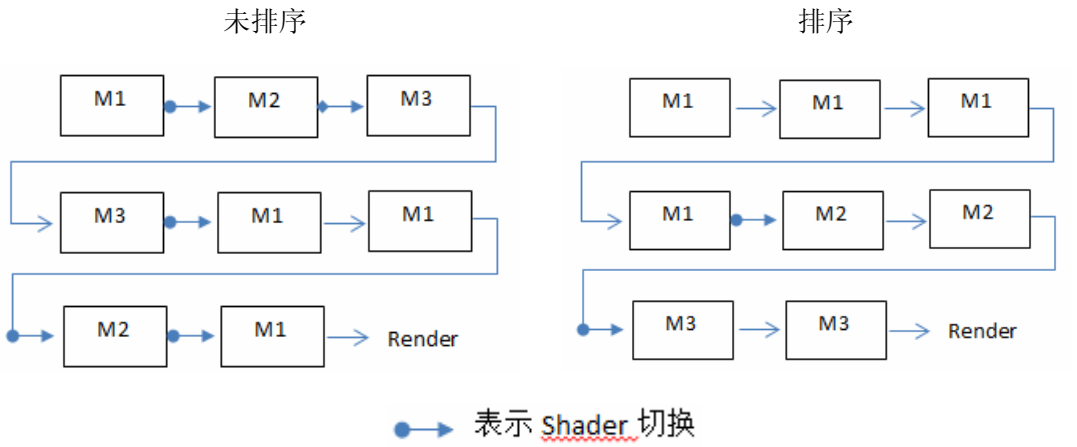


图 5-1 材质排序与非排序比较

Fig.5-1 Comparison between material sorted and not sorted rendering

排序的算法我们已知有很多种，这里我们需要了解的是如何判断为同类材质。如第二章所提出的材质实际是可以看成一个Shader，里面存储的是描述物体表面特征的计算公式，如木头和金属是两种计算公式即是两种材质。另一种对于像红色的木头和黑色的木头，它们的计算公式相同应该属于同一种材质，它们的不同在于材质的参数不同。排序时我们应当是当同一类的材质排在一起，而对于材质的参数则不在考虑范围之内(对于不同的材质GPU带来的是切换的开销，而对于同一种材质GPU所需要的是激活的开销，激活的开销远小于切换的开销，所以无需要再对材质的参数进行排序)。在判断相同材质的具体实施时我们可以在渲染引擎中以一种哈希表的形式用来存储材质以及对应的编号。材质可以是渲染中已经定义好的也可以是用户自己创建的，但无论是哪种材质都需要调用引擎的材质

类来进行实例化，我们可以材质类的构造函数中对材质进行注册，注册的材质即可获得一个内部哈希表的编号和存储位置。在渲染器中进行渲染之间我们对物体上的每个材质到哈希表中进行查找，获得编号相同的即为相同的材质。

5.3 按物体 Z 值排序

由渲染管线的知识可知，当一个3D世界的物体经过投影变换后并最终渲染到显示帧缓存中时，其相应的深度时也也会存放在一个深度缓存中，如图5-2所示。

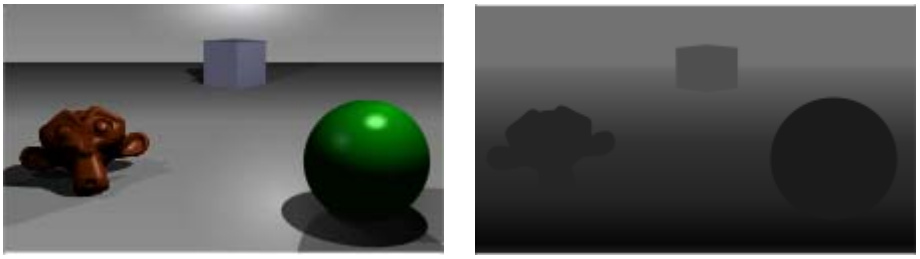


图 5-2 显示帧缓存（左）和深度帧缓存（右）

Fig.5-2Color buffer (left) and Depth buffer (right)

深度值是3D世界区别2D世界的一个重要标志。这个深度值实际上是经过相机投影变换后的Z值（即与视觉观察点的垂直距离），所以这个深度缓存也通常称为Z缓存(Z Buffer)。深度缓存通常是显存中的一块二维区域，其区域中每个元素对应着屏幕上的每个像素值。如果场景中的另一个物体也必须渲染在同一个像素点的位置，则需要判断两者深度值，深度值较小的即更靠近视觉观察点的像素值则会覆盖显示缓存中当前的像素值。同样的这个较小的深度值会被写进深度缓存中覆盖当前的深度值。这即是3D世界中的可见性问题。通俗的理解是近的物体挡住了远的物体，这个过程在图形中也常被称为深度测试或深度剪切(Z-Test，Z-Culling)。

对于3D场景来讲，物体的分布往往是错综复杂的，而且视觉观察点的位置也会经常发生变化，这就意思着物体之间相互遮挡（不仅物体之间的，也会发会生在物体不同的部位上，如正方体前面的面挡住后面的面）的情况会经常发生，为了保证物体可见性问题我们需要频繁地进行深度测试和重写显示缓存和深度缓存。然而重复的读写缓存操作则是耗时的(在移动平台上尤其明显)。

为了减少上述的频繁的重写缓存的问题，我们提供的一种解决方案是对物体按其深度值排序，即距离视觉观察点物体较近的物体排在前面先进行渲染，较远

的物体排在后面后进行渲染。较近的物体在先渲染时占据了显示缓存，于是后渲染的物体就有会很大可能性其深度值要大于当前的深度值，于是则没有必要再重写显示缓存中的值，即减少了一次重写显示缓存的动作和一次重写深度缓存的动作。

与材质排序的一样，按Z值排序的重点或难点不在排序算法上，而是如何决定物体距离视觉观察点的远近。

首先我们利用物体的包围盒来近似表示物体，在实时计算机图形学中包围盒(Bounding Box)通常是一个与轴对齐的矩形体。在判断物体距离视觉观察点远近时我们是根据包围盒的中心点来判断，如图5-3所示。

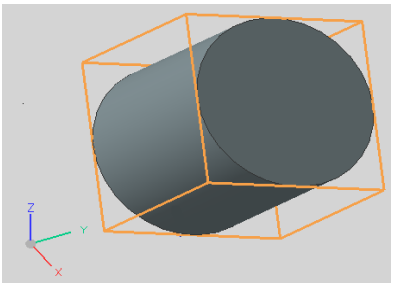


图 5-3 包围盒

Fig.5-3Bounding box

用包围盒的中心来判断物体远近是一个简单易行而且计算量非常小的方法，它能够快速的并且大可能性地将物体的远近关系表示出来。

其次我们这里所说的深度值(Z值)并不是指世界坐标中的Z值，而是经过投影变换后的Z值。所以我们比较两个物体包围盒的中心距离观察点的远近，首先需要对这两个中心点进行坐标变换(即从原的世界坐标系转变成视觉观察点所在的坐标系)，公式如图5-4所示：

$$Cdir = cross(Cright,\ \ Cup)$$

$$Cz = Cdir.x * x + Cdir.y * y + Cdir.z * z$$

图 5-4 视觉变换

Fig.5-4View Transformation

其中Cright、Cup为相机的参数；Cright为向右的朝向，相当于坐标系的x轴, Cup为向上的朝向，相当于坐标系的y轴；Cdir相当于坐标系的z轴，由前面两个参数计算得来。x、y、z为原坐标系下的点。Cz为新坐标系统下点的Z值。Cross表示叉乘。

通过新坐标系统下的Z值我们就能判断包围盒距离视觉观察点(相机原点)的远近，从而确定两个物体谁先被送入GPU进行渲染。

需要注意的是，包围盒的方式实际上只能确定物体大概的远近关系，而无法做到精确。（事实上有些情况我们是无法判断两个物体远近的，如一个物体由前往后穿过另一个物体，在种情况我们无论用什么方法都无法决定两个物体哪个在前哪个在后）。而实际上我们也无需精确，对物体按Z值排序的思想是利用简单快速的算法来确定物体远近的大致关系，从而减少因为Z值判断而重写显示缓存和深度缓存的可能性。我们的目的是尽可能的减少这种可能性，而不是全部避免（有些情况下也无法避免，如上述的两物体交叉的例子）。物体（无论有没有遮挡有情况）送入GPU先后的顺序不会影响最后渲染结果，只会影响重复读写显示缓存和深度缓存的动作。所以按物体包围盒中心的Z值排序的思想能确定渲染结果的正确性。

值得一提的是，上述两种排序时共同需要注意的一个问题就是透明材质的问题，由渲染的管线原理我们知道当场景中有透明材质时我们需要进行三步渲染：第一步将场景中所有不透明物体过滤出来先画；第二步再将透明物体渲染出来；第三步则是第二步中结果和第一步中的结果按照透明度做混合形成最终的结果。在有透明场景中的物体中仍按原有方案进行排序则有可能导致的结果是对于已经排好序的物体由于透明的要求重新排序到其它的位置中去渲染。所以当场景中有透明物体时我们需要进行材质排序策略是：在不透明物体中和透明物体中分别做排序。

5.4 测试结果

我们对几种场景做了测试，主要是针对场景中可以排序的材质的多少对最终可以提高的性能做对比，如图5-5横轴表示的是排序前后材质切换的比例，比如100:1表示排序前需要进行100次材质切换，而排序后刚只需要进行一次。纵轴则是渲染时间归一化后的结果，未排序前的结果始终是1，排序后的结果小是1则表示性能优于未排序，反之则劣于。从测试结果我们可以看出在200:1时材质排序的优势开始得到体现，当节省的切换越来越大时，所获得的优势也会越来越大。而当达到近50000:1时后继续的获得的优势渐渐趋缓，这是由于当排序过程本身所带

来的开销的影响。

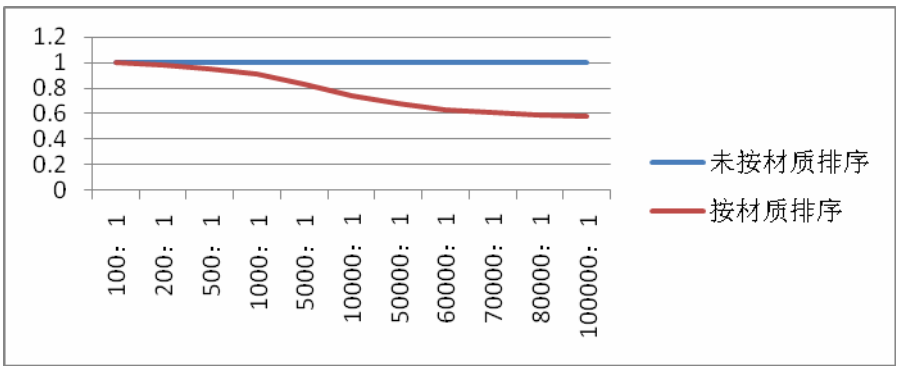


图 5-5 材质排序结果的测试结果

Fig.5-5Test result of material sorting rendering

对于按Z值排序的测试，主要是针对被遮挡的次数（即节省的写显示缓存和深度缓存的次数）对最终渲染时间的影响。由图5-6可以看出当节省的次数比较少的时候，由于计算包围盒中心点的Z值所带来的开销会影响最终的渲染时间。但当节省的次数超过几千时（实际上千百像素的遮挡情况下渲染的场景是非常容易的），排序的算法开始体现优势，当节约的次数达到十万数量级时效果比较明显。

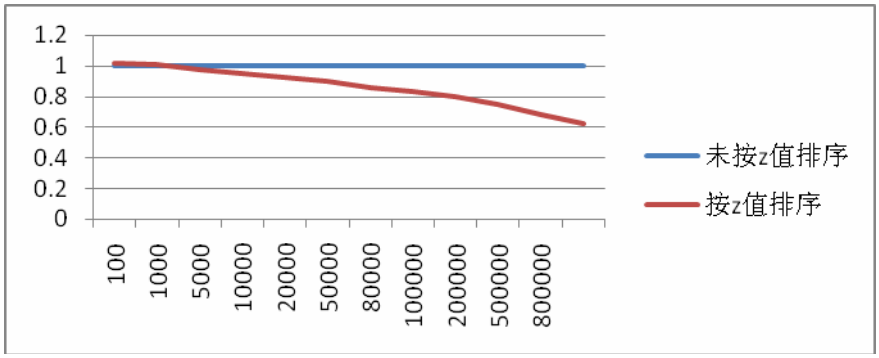


图 5-6Z 值排序结果的测试结果

Fig.5-6 Test result of z sorting rendering

5.5 本章小结

本章我们主要介绍了两种对物体排序的算法：按材质排序和按物体距视觉观察点远近排序。按材质排序主要是将材质相同的物体顺序地送入GPU，以减少GPU在切换不同材质时所带来的开销。而按物体距视觉观察点远近排序则是通过排

序，将距视觉观察点较近的物体尽可能的先送入GPU进行渲染，较远则后送入，这样做的目的是为了减少两个像素在渲染到同一位置时因Z值测试而需要重写显示缓存和深度缓存的操作。两种排序的最终目的其实都是为了最后渲染时减少对GPU开销。两者排序的实现过程也非常简单，而最后的性能收益却相当可观，是我们在实现渲染引擎时需要考虑的一个重要问题。

第六章选取系统

本章将介绍一种基于GPU的选取算法。这种算法适用于选取中最常见的一种情形即点选取可见物体。传统的的选取算法主要是利用CPU进行射线（观察点到屏幕选取点）和物体（由三角形面片组成）的求交运算，这种算法精确但比较耗时。本文提出的基于GPU的选取算法，主要思想是将物体先预渲染到一个离线的帧缓存中，然后根据采取点的颜色值与离线帧缓存中的物体颜色进行匹配查到最后确定选中的物体。这种算法利用了GPU的性能较传统的算法性能上有较大提升。

6.1 选取系统原理

我们在图形学应用过程中经常会碰到这样的需求，即需要对屏幕上的某个或某些物体做高亮显示以便进行放大/缩小、更改颜色等操作。在这个时候选取某个或某些物体的过程就叫做**选取**。选取实际上分很多类，按选取区域不分，可以分为点选、框选、多边形选等；按选取对象的不同，可分为点选、线选、面选、物体选；按被对象可见性不同，又可划分为只选取可见物体和可选取所有物体。然而在实际应用中有一类情形是最为常见，即点选可见物体。这个类情形占到了所有选取用例的80%，所以研究如何加快点选可见物体是非常有意义的。

从渲染的角度来讲，选取的过程实际就是一个从2D的平面的选取区域来判断后面3D世界的哪些物体被选中的一个过程。如图6-1所示，渲染的过程是将视觉体(viewing frustrum)里的3D物体投影到屏幕(viewplane)的一个过程，而选取的过程正好相反，即在屏幕选取某个物体（例如图示中的带有花纹纹理的圆球），然后对这个物体进行高亮显示（如赋值另一种颜色）。从表面看来我们是对屏幕二维世界进行操作，而实际上其背后的技术则是通过屏幕上的选取区域找到其背后视觉体中的对应的三维世界的物体。

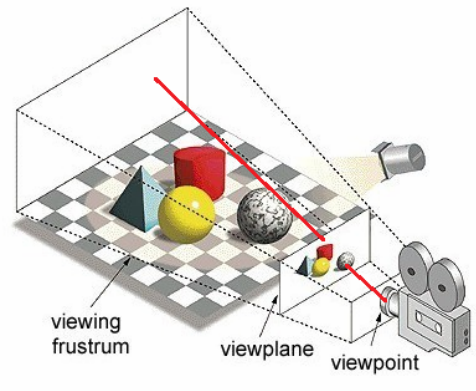


图 6-1 选取原理示意图

Fig.6-1 Diagram of selection

6.2 传统选取算法

传统的选取算法实际上是一系列的求交运算。它是以视觉观察点为原点，由观察点到屏幕上的选取点为方向引出一条射线，这条射线经过屏幕（viewplane）进入三维世界，然后将这条射线与三维世界的物体进行求交运算，与之相交的物体即被选中。

由第二章的物体的组成我们知道实时渲染中中我们通常是以三角形面片来表达物体的轮廓。所示物体与射线的求交运算就就变成了一系列三角形与射线的求交运算。下面是目前最为通用（效率最高）的射线与三角形的求交算法，通过了解它我们可以知道它的计算量

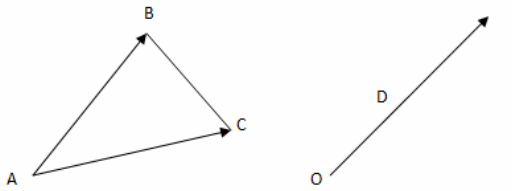


图 6-2 3D 空间中的三角形和射线

Fig.6-2 Triangle and ray in 3D Spaces

如图 6-2，空间中三角形 A(A_x，A_y，A_z)，B(B_x，B_y，B_z)，C(C_x，C_y，C_z)，则三角内(包括三角形的顶点和边)的任意一点可表作为

$P(P_x, P_y, P_z) = A + \beta (B - A) + \gamma (C - A)$，其中约束条件为：$\beta + \gamma \leq 1$ 且

且

设射线原点为 O(O_x，O_y，O_z)，方向为 D(D_x，D_y，D_z)，则射线上的任意一点(包括原点)可表作为 T(T_x，T_y，T_z) = O + tD，其中约束条件为：t >= 0

求射线与三角形相交实际上则是求下面一个方程的解。

$A + \beta (B - A) + \gamma (C - A) = O + tD$　　展开为：

$A_x + \beta (B_x - A_x) + \gamma (C_x - A_x) = O_x + tD_x$

$A_y + \beta (B_y - A_y) + \gamma (C_y - A_y) = O_y + tD_y$
$A_z + \beta (B_z - A_x) + \gamma (C_z - A_z) = O_z + tD_z$

用矩阵表示，即

$$= \begin{bmatrix} Ax & Ox \\ Ay & Oy \\ Az & Oz \end{bmatrix}$$

其解为

其中

$$|DETER| = \begin{vmatrix} & \\ & \end{vmatrix}$$

由上我们可以看出射线与三角形求交的过程是涉及到一系列的矩阵运算，当一个物体上的三角形面片达到一定数量级时，这种计算将是非常耗时的。

当然，我们可以采取一些办法来加速这种运算，比如先将射线与物体的包围盒求交，如果射线与包围盒不相交，则无需与物体的三角形面片计算。尽管如此，有时候我们还是不能避免正面的矩阵计算。

6.3 基于 GPU 的选取算法

目前图形学领域也有很多研究 GPU 下的选取计算，但方案大多仍然基于传统选取算法的思想，所不同的是它们利用 GPU 的性能来加速上述传统算法中的矩阵运算[34]。本文要介绍的基于 GPU 的选取算法与这些方案不同，其主要思想是将 CPU 计算的压力转移到 GPU 端。不再需要求交运算，而是通过 GPU 渲染的功能将建立一个物体颜色与编号的对应关系，再通过反查找找到被选中的物体。整个过程如下：首先先忽略场景的每个物体本本身的材质，给每个物体赋上一种特殊的材质，这种材质的特点是输入一个编号，而输出一种单色（所谓单色，是指物体中所在有顶点的颜色都相同，即从外观上上这个物体所有部位都一种颜色）。并且这个输出颜色对于每个物体来讲必须是唯一的，即每个物体上的颜色不能雷同；其次对场景中的所有物体进行一个编号，并将传给这个特殊材质。再带有这种特殊材质的物体渲染到一个离线的帧缓存中；最后通过选取点的颜色到

这个离线帧缓存中查找，与这个颜色相同的物体即为被选中的物体。详细步骤如下：

第一步、申请一块离线的帧缓存。这个帧缓存与实际用作渲染的帧缓存在大小、格式等参数完全一样，唯一不同的是这个帧缓存不做显示用。
这个离线帧缓存并不影响最终的渲染结果，而是在当用户需要做选取动作时才申请这样一块帧缓存，这个帧缓存仅供内部使用，对用户来讲是透明的。也即是说当用户口进行选取动作时我们会在内部进行第二遍渲染，这个渲染跟用户正常的渲染是独立的。

第二步、创建一个材质（Shader），这个材质会根据不同的 ID 输出一种不同的单色。材质的核心代码如图 6-3 所示：

```
//·Pixel·shader.
float4·PS_HWSelection_Shader(VS_TO_PS·In,·uint·primitiveID·:·SV_PrimitiveID)·:·SV_Target
{
····int·colorID·=·gRenderItemPrimitiveBase·+·primitiveID;
····float4·color;
····color.x·=·(colorID·&·0x000000FF)·/·255.0f;
····color.y·=·((colorID·&·0x0000FF00)·>>·8)·/·255.0f;
····color.z·=·((colorID·&·0x00FF0000)·>>·16)·/·255.0f;
····color.w·=·1.0f;

····return·color;
}
```

图 6-3 3D 输出唯一色的顶点着色器代码

Fig.6-3Pixel Shader code for output the unique color

如第一步所提到的，当用户进行选取动作时我们会有两遍渲染，而这个材质则是用在进行离线的渲染的时候，它与物体原有的材质并不冲突，也不会替代原有的材质。从对应关系来讲，这个新创建的材质实际上是建立了一种物体编号与物体颜色的对应关系表，这个对应关系表的数据内容如表 6-1 所示：

表 6-1 物体编号与颜色映射表

Tab.6-1 Mapping table between Object ID and output Color

编号	颜色
0	0x00000000
1	0x00000001
…	…
255 （2^8 - 1）	0x000000FF
256	0x00000100
257	0x00000101
…	…
65535 （2^16 - 1）	0x0000FFFF
65536	0x00010000
…	…
16777215 （2^24 - 1）	0x00FFFFFF

在这个对应关系表中物体的编号与颜色都是唯一而且是可逆的。这样做的目的是为了方便我们在接下来的步骤能够进行反查找，即通过颜色查找被选中的物体的编号。

第三步、根据选取点在屏幕上的位置找到其在离线帧缓存中相对应的位置，并读取该点的颜色值。
由渲染的原理我们可知，视觉体摄影的近平面即是最后三维世界物体成像的投影平面，我们在创建相机时通常都会将这个投影平面设置与最终显示的屏幕成等比例（否则最终渲染画面会有变形或被拉扯的感觉）。例如最终显示的窗口屏幕大小为 400：300 像素，那么在第一步申请帧缓存时，帧缓存的大小也通常设置成 400：300，投像平面的长宽比为 4：3。这样选取点在选取屏幕上的位置只需要按等比例找到帧缓存上的相应位置然后读取该点的颜色值即可。

第四步、根据该点的颜色值在表 6-1 中通过反查到找到物体对应的编号。
这种反查找可以利用如下公式获得：

```
int objectID = (colorID.z * 255.0f & 0x00FF0000) << 16 + (colorID.y * 255.0f & 0x0000FF00) << 8 + (colorID.x * 255.0f & 0x000000FF);
```

得到物体编号后返回该编号的物体即是被选中的物体。
整个过程可以用流程图来表示如图 6-4 所示：

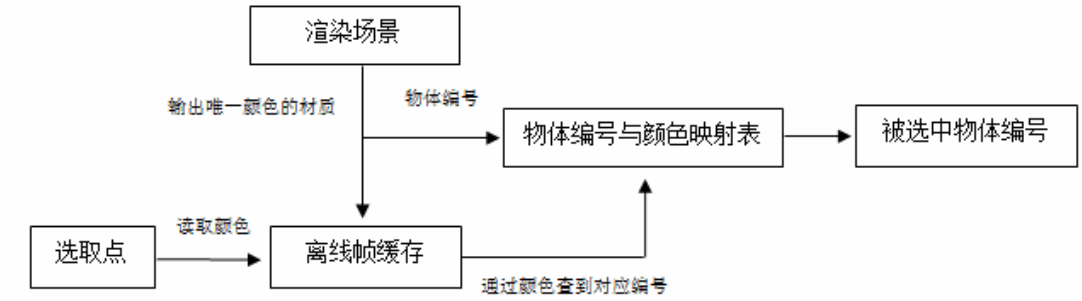


图 6-4 GPU 选取算法流程图

Fig.6-4 Diagram of GPU selection algorithm

值得说明的是从表6-1我们可以看出物体最大的编号支持到16777215（2^24 - 1）（2^24表示2的24次方），大于此数的编号从公式来看又会回到从0开始编号，这样便会形成冲突，即会有两个物体有相同的编号(也即相同的颜色)，这样在第四步进行反颜色到物体编号反查找时就会出错。为了解决这个问题，我们需要对物体进行分批编号分批渲染，每批的最大数目不超过（2^24 - 1）。这样就不会引起编号冲突。实际上（2^24 - 1）已经是一个很大的数，对于通常的渲染场景中来讲这个数字已经完全足够。

6.4 基于 GPU 的选取算法与传统选取算法对比

基于GPU的选取算法充分利了GPU的特点，利于GPU的渲染功能快速建立了一个选取需求与被选中物体的对应关系，然后通过反查找能很快速地帮助用户找到被选中的物体。它相对于传统的选取算法等于是将原有的CPU的压力转移到了GPU上，而且转移后过来的压力也不会对GPU造成很大的负担(仅仅是渲染一遍的代价，而没有涉及过多的GPU计算)。这种算法也有一些局限性：首先，它只支持选取可见物体，如果用户想选取的还包括不可见物体，刚这种算法不适用，因为在渲染的过程中不可见的物体（如被其它物体遮挡的物体）会在渲染的过程由于Z值测试而被剪裁掉，所以无法在最后的颜色与编号的映射表中体现出来；其次，它最常用的场景中点选整个物体，本章最开头讲的实际上选取分很多种，如点选、框选、多边型选等，还有选取整个物体、选物体的某个面、某条线等等，对于整个物体选取的类型，无论它是点选还是框选等类型，只要是选择可见物体我们都可以考虑运用上述的基于GPU的选取算法，而对于这之外的其它类型的选取算法我们也原理上也是可以用GPU来实现，但实现起来的代价较大，甚至会比传统的算法代价更大，所以对于选取可见物体并且是选取整个物体之外的其它类型的选取我们还推荐用传统的算法来计算。基于GPU的选取算法与传统的选取算法对比总结可见表6-2：

表 6-2 基于 GPU 的选取算法与传统选取算法对比

Tab.6-2 Comparison between GPU based selection and traditional selection

基于 GPU 的选取算法	传统的选取算法
利用GPU资源	利用CPU资源
速度较快	速度较慢
只适用于点选整个物体的情形	适用于所有选取情形，包括框选、多边型选、面选、线选等
只适用于选取可见物体	可用于选取不可见物体

6.5 测试结果

我们测试的主要有三个场景：

第一、 测试场景中的物体多少对于GPU选取算法的影响；

第二、 测试GPU能力对于GPU选取算法的影响；

第三、 对比CPU对于传统算法的影响以及GPU对于基于GPU选取算法的景程；

以下是测试的详细数据：

测试目的:由于GPU选取是需要将场景渲染到一个离线的帧缓存中去，所以场景中的物体所占据屏幕的大小势必会影响渲染效率。

说明:下图中的横轴是场景中的物体所占据屏幕大小，纵轴表示的是渲染时间。

结论:当场景中的物体所占据屏幕比例越小的时候,GPU选取所体现的优势越大，随着物体所占据屏幕比例增加,GPU选取的优势会慢慢变小,而当一种极端场景，即场景中的物体占满屏幕的时候，GPU选取的性能还会略低于传统的传统选取，这是由于GPU的选取在渲染的时候开销过大的缘故,但从其结果看即使是这个极端情况两者的最终结果实际相差不多，是可以接受的。如图6-5所示：

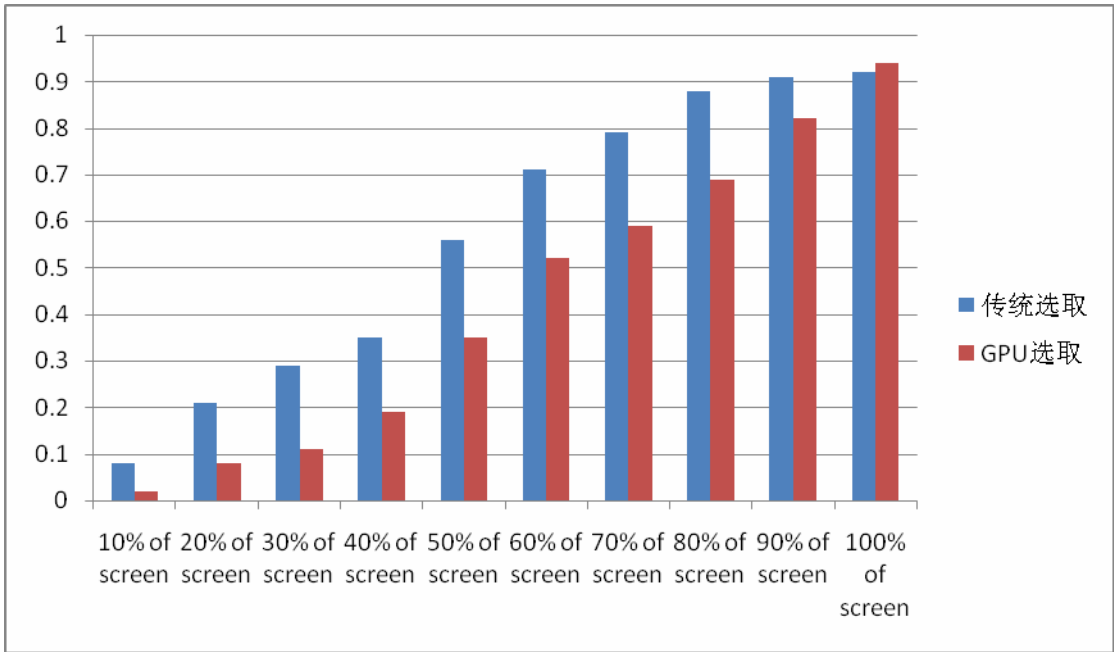


图 6-5 传统选取与 GPU 选取对于不同场景的测试对比

Fig.6-5Comparison between traditional selection and GPU selection on different render scenes

测试目的：测试GPU能力对GPU选取算法的影响。

说明：SM（Shader Model）是衡量GPU能力的一个很重要的标准，纵轴表示的是渲染时间。

结论：GPU能力越高，GPU选取效率越高，如图6-6所示：

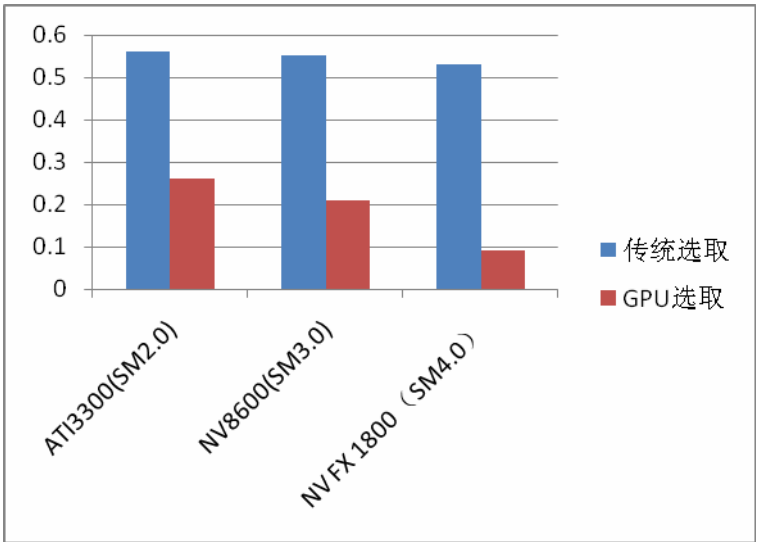


图 6-6 传统选取与 GPU 选取在不同 GPU 下的对比

Fig.6-6 Comparison between traditional selection and GPU selection on different GPUs

测试目的：测试CPU对传统选取算法的影响，也是从侧面测试需要什么样的CPU和什么样的GPU能达到相当的选取效率。

说明：横轴表示测试机CPU所带的核数多少；纵轴表示的是渲染时间。

结论：CPU能力越高越，传统选取效率越好，8核的CPU才差不多和一个SM3.0的GPU选取的效率相当。如图6-7所示：

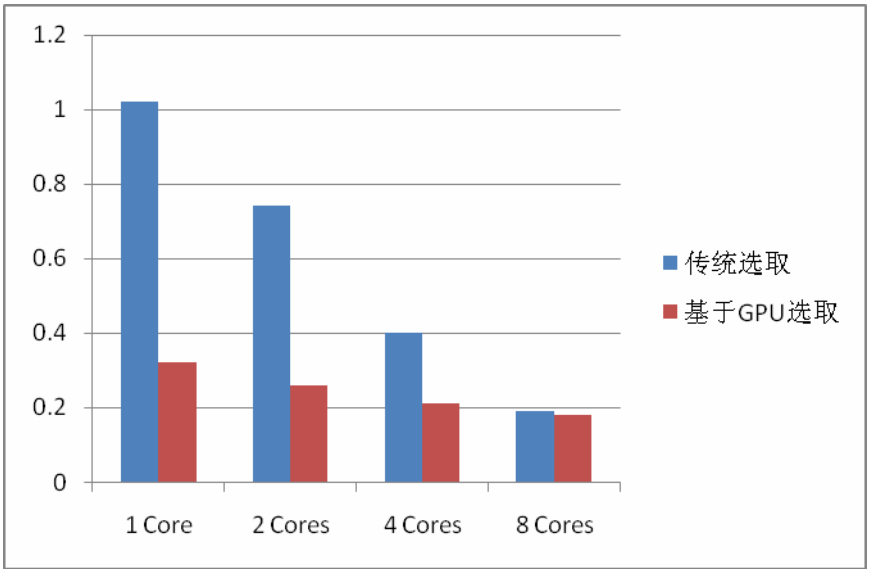


图 6-7 传统选取与 GPU 选取对于不同 CPU 的测试

Fig.6-7Comparison between traditional selection and GPU selection on different CPUs

6.6 本章小结

本章首先介绍了选取的概念以在实际产品中的分类等，其次简单阐述了传统选取

的做法以及其弊端，传统选取算法在于由视觉观察点到屏幕选取点的射线与三维世界中的物体进行求交运算，这种算法通常是在CPU中完成，而且比较耗时，当前也有一些利用GPU做选取的算法也是基于求交运算这个思想，它们的利用GPU的特性来加速这种求交运算（核心为矩阵计算）的过程。而本文中后段介绍了一种基于GPU的选取算法则主要是抛弃这种求交运算而利用GPU的渲染功能，最终通过物体编号和渲染颜色的反查找得到被选中物体，这种算将选取的压力从CPU端减少并转移到了GPU端，显示出了很好的效率。

第七章总结和展望

本章将对第三章到第六章的四种优化方案进行总结，包括它们的测试结果以及适用场合以及在实际工程应用中应注意的问题等等。后们还将介绍笔者写作本论文中研究过的其他方面的渲染引擎的优化以供读者研究。最后展望渲染引擎未来发展的技术。

7.1 总结

随着计算机技术的不断发展，计算机图形学技术的不断进步，越来越多的三维渲染引擎应运而生。网络游戏及3D产业的迅速发展更是让这些技术有了商业驱动的动力。本文通过实际的学习经验，设计和实现一个基于GPU读写效率优化的渲染引擎。本文的主要工作如下：

首先本文综述了三维渲染引擎的发展以及其背景知识，并指出来当前渲染引擎的一些问题，由些引出渲染引擎优化的点。

其次针对上述的问题分别从四方方面阐述渲染引擎的优化点，包括：在物体模块的批物体单元；纹理模块里的纹理集；渲染器模块的物体排序；选取模块里的基于GPU的选取算。前面这三点都是围绕着一个方面的而展开优化的，即减少GPU的调用。而最后一点则是借助GPU的渲染能力来帮助复杂计算。

最后对这些优化点进行了深入测试，数据显示改进后的渲染引擎性能较之前有较大的性能提升。

7.2 其它优化点

除了以上优化的方面之外，在写作本论文中笔者还研究其他方面的优化，提出来供读者参考。包括：阴影算法的改进、在移动平台上用定点算法来替代浮点运算。

7.2.1 阴影算法的改进

阴影算法的改进主要是针对当前很多渲染引擎中为了达到逼真的效果而采用的模板阴影。模板阴影主要利用CPU计算出物体对于光阴而生阴影的轮廓，这种算

法较为耗时，而利用阴影图的算法则是利用GPU的能力先将阴影从光源的视角渲染出来，然而再将其与场景一起从相机正常视角一起渲染出来。

模板阴影算法由Frank Crow 1997年提出，它的是主要原理是计算场景的物体对于光源视频的轮廓，由光源到轮廓延伸出去从而知道阴影区域。

轮廓线的计算是模板阴影算法中的关键。其主要思想是首先测试光线同场景中物体上每个三角形的法线的夹角是否小于90度，即光线矢量三角形法线适量的点积要大于零。而在余下的所有面向光线的三角形中，把每一个三角形面的每条边可以先保存起来，不过在保存之前先把该条边同已保存的每条边先进行逐一比较，如果找到的边同它的两个顶点相同，但顺序相反，则在保存的存储中移除这条边。如找不到则把该条边加到存储中。这样就得到一个物体面向光的轮廓线。最后我们轮廓线上的每个点延着光线方向延长，其所形成的面同场景中去做模板测试，得到阴影区域。

从上述步骤中我们可以看出要计算模板阴影就必须针对场景物体中所有三角形片面针对光源视角进行计算，而这样的计算是非常耗时的。

而阴影图的算法则是利用GPU的渲染功能将阴影从光源视角进行预渲染，最后将其形成的阴影与场景一起以相机视角正常渲染出来。其步骤可分为：

第一步：以光源所在位置为观察点渲染场景，这里重要是将渲染后的场景的深度值保存起来（比如一张纹理中）。

第二步：正常渲染场景，比较深夜值值和从第一步中保存深度图的纹理中读出来的值得大小判断遮挡，如果有遮挡的话证明这个位置产生阴影，输出一个不同的颜色做为阴影颜色即可。

阴影图的算法实际已经较为成熟，在近几年的新的渲染引擎几乎都有运用，它能在性能及渲染效果上达到一相很好的平衡。

7.2.2 利用定点来代替浮点运算

在移动平台上运用的是ARM处理器，而ARM处理器最大的问题在于其浮点运算能力较差，所以通常在移动平台上的程序(尤其是浮点计算较多的游戏程序)开发并不鼓励大规模的浮点运算，而采用定点运算来代替浮点运算的方法。

浮点数在ARM处理器上比较慢的原因是因为ARM处理器不支持硬件浮点运算，它所有的浮点运算都是在浮点运算模拟器上进行，而由于浮点运算的浮点运算所产生的汇编指令远比硬件支持的浮点运算要长。所谓定点数，通俗地说，就是小数点固定的数。以人民币为例，我们日常经常说到的如123.45¥，789.36¥等，默认的情况下，小数点后面有两位小数，即角、分。浮点数，一般说来，就是小数点不固定的数。

使用定点运算实际也就是使用整数来进行计算。但指定固定数目的位元做为数值的分数部份。就好像是指定某一数字，其千位数以下为分数。若要表示0.500，只要乘以1000，便得到500这个数值。如500 + 500 = 1000 (可视为：0.500 + 0.500 = 1.000)。

在具体应用上我们可以参照OpenGL ES中对于定点数的类型的声明，找到渲染引擎中与OpenGL ES的结合的部分即上层传给底层OpenGL ES的数据，将其转成定点数再传给OpenGL ES。

需要说明的是由于时间等诸多方面因素本文并没有对渲染引擎中的数据进行全部的定点转换和测试，只是对其中的矩阵相乘的一个小函数进行扩大，测试结果显示有近5%的性能提升。我们相信如果将渲染引擎中与OpenGL ES结合的数据全部转成定点这个性能提升度应该会很大。

7.3 展望

近些年3D产业的发展日新月异，3D从业者们也开始意识到了渲染引擎在3D开发技术中的地位。越来越多的引擎开发技术也开始在各种各样的3D技术峰会（如Siggraph）上呈现。游戏引擎是3D渲染引擎技术发展的一个突出的实例，近些年来游戏产业发展迅猛，各大公司发布新游戏的时间越来越短，这也得益于渲染引擎技术的成熟。我们必须承认国内渲染引擎技术（包括游戏引擎）的发展还是刚刚起步，技术还不够成熟，其完善也需要一段时间，但是中国的3D产业正如一个冉冉升起的新星迅速崛起，并逐步成为一个新兴的产业群和增值的新亮点，随着3D技术的不断发展，笔者相信渲染引擎也将会迎来一个全新的发展机会，也必将成为全新的研究热点，未来的渲染引擎技术也将会使用户更快、更方便、更高效地运用3D技术来获取更加逼真和生动的渲染效果。

参考文献

[1] AaftabMubshi.OpenGL 2.0 Programming Guide，Pearson Education:2008,115-126

[2] Rodriguez J.Typhoon Labs OpenGL Tutorial, 2003, 42-75

[3] Nurminen A, Helin V.Technical Challenges in Mobile Realtime 3D City Maps with Dynamic Contents，Proc of IAESTED Software Engineering, 2005, 244-268

[4] McConnel Steve.Code Complete, Microsoft Press, 1993, 66-90

[5] CHANG J M, GEHRINGGER E.A high – performance memory allocator for objected oriented system. IEEE Transactions on Computers, 1996, 45(3),357-366

[6] Andre LaMothe.Tricks of the Windows Game Programming, Electric Power Press, 2004,455-468

[7] Dave Astle，Dave Durnil.OpenGL ES Game Development, 2005,120-156

[8] OpenGL ES Common/Common-Lite Profile Specification, 2004,56-87

[9] Cok Keith，Roger Corron.Developing Efficient Graphics Software: The Yin and Yang of Graphics, 2000,289-320

[10] LindholmErik,MarkKilgard. A User-Programmable Vertex Engine, 2001,266-280

[11] James Fung,Steve Mann.Open VIDIA,parallel GPU computer vision,ACM international conference on Multimedia,2005,311-345

[12] An approximate image-space approach for interactive refraction ACM Transactions on Graphics Chris Wyman, 2005,178-180

[13] WolfgangHeidrich,Hans-Peter Seidel,Efficient Rendering of Anisotropic Surfaces Using Computer Graphics Hardware, Image processing Workshop, 1998,203-250

[14] Kari PulliTomiAarni Ville MiettinenKimmoRoimelaJaniVaarala.Mobile 3D Graphics with OpenGL ES and M3G, Morgan Kaufmann, 2007,111-123

[15] Dave Astle Dave Durnil.OpenGL ES Game Development, Course Technology PTR, 2004,23-34

[16] LengyelEric. Mathematics for 3D Game Programming and Computer Graphics, Charles River Media, 2002,300-358

[17]ArvoJames.“A Simple Method for Box-Sphere Intersection Testing” in Graphics GemsAndrew S. Glassner(ed.), AP Professional, 1990,120-129

[18] Mortenson. M.E. Mathematics for Computer Graphics Applications, Second edition, Industrial Press, 1999,201-280

[19] Ebert David S, F. Kenton Musgrave, Darwyn Peachy, Ken Perlin, and Steven Worley. Texturing and Modeling: A Procedural Approach，second edition, AP Professional 1998,40-68

[20] Badouel Didier,“An Efficient Ray-Polygon Intersection” in Graphics Gems, Andres S Glassner(ed.), Ap Professional, 1990,69-101

[21] WattAlan and Mark Watt. Advanced Animation and Rendering Techniques, ACM Press, 1992,2-11

[22] WooAndrew.“Fast Ray-Box Intersection” In Graphics Gems，Andrew S. Glassner(ed.) AP Professional, 1990, 119-133

[23] Ulrich, Thatcher.“Loose Octrees” In Game Programming Gems,Mark Deloura (ed.) Charles River Media, 2000, 33-51

[24] Willard F.Bellman Lighting the stage,Art and Practice,Broadway Press,2001, 23-45

[25]江峰. 3D 游戏引擎研究与实现 [D]. 浙江大学，2005（2），3-9

[26]张继开. 三维图形引擎技术的研究 [D]. 北方工业大学, 2004(5), 11-19

[27]邓世根. 移动通信弹射设备中三维图形引擎技术研究. 成都电子科技大学, 2006, 2-4

[28]官酩杰. 基于 OpenGLES 的移动平台图形渲染引擎研究与实现. 北京交通大学, 2010(6), 9-59

[29]Wolfgang F. Engel. Direct3D 游戏编程入门教程. 北京：人民邮电出版社, 2002. 7-9

[30]埃肯因. 实时计算机图形学 [M]. 普建涛，译. 北京：北京大学出版社, 2004, 1-20

[31]彭群生鲍虎军金小刚编著，计算机真实感图形的算法基础. 北京：科学出版社, 2002, 177-190

[32]杨钦徐永安. 计算机图形学. 清华大学出版社, 2005(3), 228-229

[33]唐泽圣. 三维数据场可视化. 北京：清华大学出版社, 1999, 1-9

[34]吴恩华柳有权. 基于图形处理器 (GPU)的通用计算 [J]. 计算机辅助设计与图形学学报. 2004(5), 34-41

[35]伍毅鲁东明. 采用 GPU 的计算机辅助壁画修复技术. 计算机工程. 2006(8), 32-34

[36]Luckylai. 游戏引擎演化史. http://www.gameres.com, 2005

[37]Andre LaMothe. Windows 游戏编程技术. 中国电力出版社, 2005, 10-55

[38]左鲁梅黄心渊. 纹理映射技术在三维游戏引擎中的应用. 计算机仿真, 2004(10), 21-28

[39]耿卫东. 三维游戏引擎设计与实现[m]. 杭州：浙江大学出版社, 2008, 55-91

[40]Lucifer 著. 游戏引擎系统. http://wwwchinagamedev.nedcgd/develop/3D/200202/EngineSys.htm, 2005(10), 10-31

[41]蒋燕萍. 虚拟环境漫游中的关键技术：[硕士学位论文]. 北京：北方工业大学，2003, 62-63

致谢

在结束这篇论文前，我必须要非常郑重地感谢：我的导师马利庄教授、论文项目的指导老师盛斌助理教授以及上海交通大学硕士研究生办公室主任张忠能教授。马老师对我的论文选材方面给出了高瞻远瞩的指导意见，使我的论文在内容方面有了方向性的指导；盛老师则在项目上给予我非常大的帮助，并且非常耐心地帮助我一次又一次地讨论和修改论文，不仅是论文的内容，甚至包括论文的段落格式的修改等；张老师则每次在关键时候时时刻刻提醒着我论文的进度，催促我按时按量按质地完成论文。在这里我要真诚地表达我深深的谢意。其次我还要感谢我的家人在我准备这篇论文的时间给予了我莫大的支持。没有你们，我是不可能完成这个项目，也不可能按时完成这次毕业设计，非常感谢！

攻读学位期间发表的学术论文

上海交通大学硕士学位论文答辩决议书

姓　　名	陈是权	学号	1100332069	所在学科	计算机技术
指导教师	马利庄	答辩日期	2013-05-28	答辩地点	上海交通大学新建楼2029室
论文题目	面向GPU优化的渲染引擎研究与实现				

投票表决结果：3 / 3 / 3 （同意票数/实到委员数/应到委员数） 答辩结论：☑通过 ☐未通过

评语和决议：

陈是权同学所完成的学位论文《面向 GPU 优化的渲染引擎研究与实现》，以 GPU 为基础，讨论与研究了 3D 渲染引擎中数据及渲染系统优化的问题。论文选题具有重要的理论意义和应用价值。

论文取得了下列研究成果：对于渲染引擎的三个模块及一个系统分别提出了优化方案，这些方案包括：在物体模块对于顶点数据相同的一些物体我们分别可以采用动态和静态两种批处理方式使其渲染帧率加快；在纹理模块介绍了一种纹理集算法以加速引擎中处理琐碎小纹理的过程；而在渲染器模块中提出了对物体进行两种排序：按材质排序以减少 GPU 切换材质的开销，而按物体 Z 值排序则可以减少 GPU 中重复读写深度缓存的操作；最后我们在选取系统中抛弃传统算法，提出了一种利用物体渲染颜色与编号进行查找和反查找来达到选取目的方法。上述方案经过测试，测试结果显示其优化后的方案较之前或传统算法都有明显的帧率提升。

论文框架合理、条理清晰，叙述清楚，所提出的技术可行、有效。表明作者已具有较扎实的基础理论和深入的专门知识，具备解决工程项目实际问题的能力。答辩时叙述清楚，条理清晰，并能正确回答问题。

答辩委员会经讨论后认为，该论文达到了工程硕士学位论文的要求，经无记名投票表决，一致同意通过该生的学位论文答辩，并建议授予其工程硕士学位。

答辩委员会成员签名	职务	姓名	职称	单 位	签名
	主席	何援军	教授	上海交通大学	何援军
	委员	姚天昉	副教授	上海交通大学	姚天昉
	委员	张忠能	高级工程师	上海交通大学	张忠能
	委员				
	委员				
	秘书	张日英	高级工程师	上海交通大学	张日英