****

**本科毕业论文（设计）中期报告**

**题 目： 一种动画引擎的设计与实现**

**专 业 计算机科学与技术**

**学 生 徐昊奕**

**学 号 1190200826**

**指导教师 吴锐**

**日 期 2023.3.1**

**哈尔滨工业大学教务处制**

一、论文（设计）工作是否按开题报告预定的内容及进度安排进行

是。

论文工作主要围绕动画引擎展开，主体内容为在理解动画引擎的整体架构的基础上探索设计三大核心模块：渲染模块、动画模块、碰撞模块。

原先的进度安排为：在课题的中期前完成并完善渲染模块，完成动画模块。在结题前完成并完善三个动画引擎的核心模块——渲染模块、动画模块与物理碰撞模块，同时，在这三个模块中尝试引入创新的技术。  
 现在的进度为完成并完善了自己的小型离线软渲染器，并在其中尝试了光线追踪算法；学习了新的图形api语言——openGL并使用它完成了动画引擎模块，并在其中尝试了动画混合和运动模糊；对物理模块的设计创新进行了预研，是要超出进度安排的。

二、已完成的研究工作及成果

总体上，在开题之后，完成并完善了自己的小型离线软渲染器，并在其中尝试了光线追踪算法；完成了动画引擎模块，并在其中尝试了动画混合和运动模糊；对物理模块的设计创新进行了预研。下面，将针对已经完成的三部分内容进行展开。

1. **基础渲染模块**

三维场景渲染的本质涉及以下几个步骤：描述一个虚拟场景，这些场景一般是以某数学形式表示的三维表面，；定位及定向一个虚拟摄像机，为场景取景；设置光源，光源会与环境中的物体交互作用并反射，最终到达虚拟摄像机的感光像面；描述场景中物体表面的视觉特性，这些视觉特性决定光线如何与物体表面产生交互作用；求解渲染方程即求解着手方程对于每个位于影像矩形内的像素，渲染引擎会找出经过该像素而聚焦于虚拟摄像机的一天或多条光线，并计算其颜色及强度。

在我的渲染器中，我首先从obj格式的文件中导入场景的数据。Obj文件是一种文本文件，可以直接用写字板或是文档编辑器进行查看和编辑修改。它是一种3D模型文件，不包含动画、材质特性、贴图路径、动力学、粒子等信息，主要支持多边形模型。它的文件结构为，注释行以符号“#”为开头，空格和空行可以随意加到文件中以增加文件的可读性。有字的数据行都由关键字开头，简单解释了这一行的数据时是什么。其中关键字对应的解释为以字母＂v＂开头的行是顶点数据，即顶点在模型空间中对应的三维坐标值；以字母＂vt＂开头的行是顶点数据的贴图/纹理坐标点，其值为u, v即纹理坐标，指的是纹理图片如果被放在屏幕上显示时，即在屏幕空间中以屏幕左下角为原点的坐标；以字母＂vn＂开头的行是顶点数据之正则化后的法线向量，表示顶点的朝向；以字母＂f＂开头的行是表示面，以3个点以点自身的“v/vt/vn”的索引形式组成。清楚了上面说的格式后，我很容易地获取到了顶点数据，并把顶点数据放入了下面的流程图中。



图 1 渲染流程

这张流程图便是我的小型软渲染器的主体结构。下面我将按实现的流程详细展开。

首先，是顶点处理阶段。这个阶段会执行顶点变换和顶点着色的工作。通过模型矩阵、观察矩阵和投影矩阵(MVP矩阵)计算出顶点在裁剪空间下的位置，以便后续阶段转化为标准化设备坐标系下的位置。同时，这一步中也会计算出顶点的法线和纹理坐标等。同时，在这个阶段也可能会进行顶点的着色计算，如平面着色 (Flat Shading)和高洛德着色 (Gouraud Shading)都是在顶点着色器中进行着色计算。其中，核心的部分是顶点坐标的变换，需要从模型空间一步步通过矩阵变换到屏幕空间，大致流程如下所示：

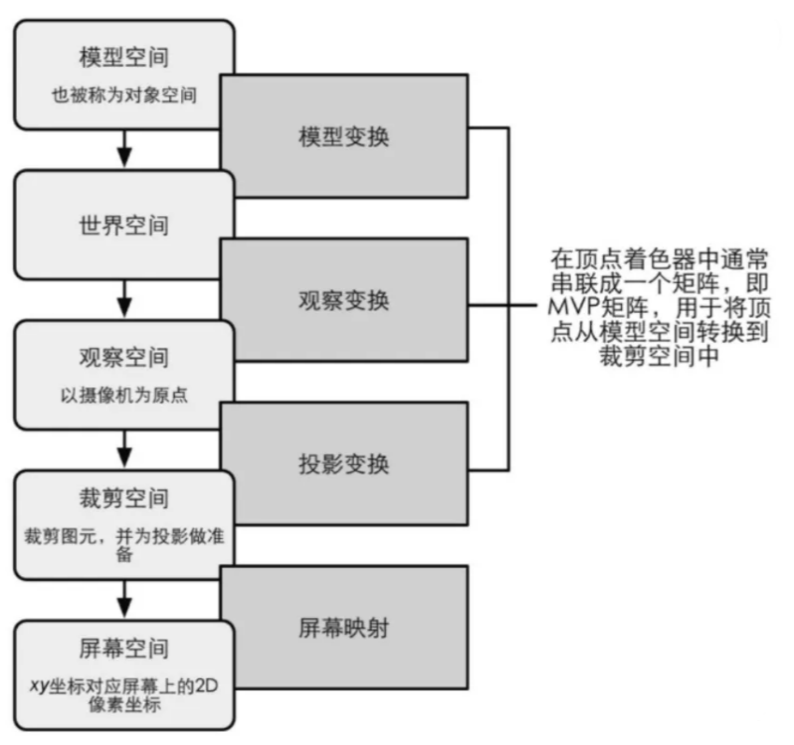


图 2 空间变换

在上面的图中，右边的每一个块都代表一个矩阵，第一个模型变换对应一个复合变换，就是将平移、旋转、缩放组合起来的几何变换，又多个矩阵串联呈现，变换顺序是先缩放，再旋转，最后平移，矩阵公式如下所示：

观察变换的矩阵如下所示：

投影变换的视图如下所示：

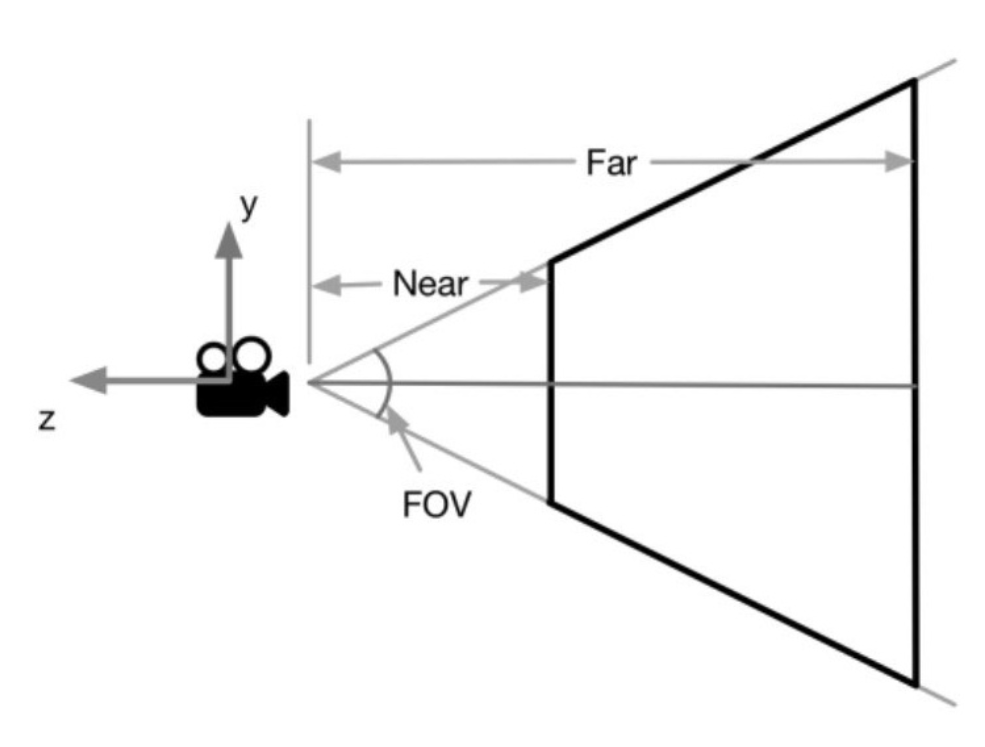


图3 投影变换

对应的矩阵如下所示

改变顶点处理中对应的矩阵比对图如下所示：

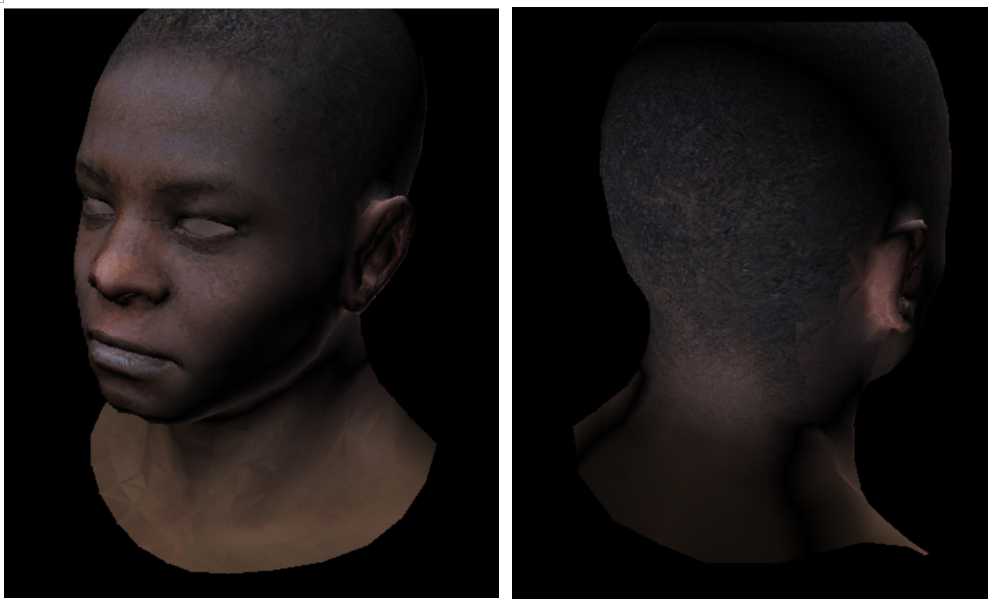


图4 改变矩阵变量得到的不同渲染结果

在顶点处理阶段中第二重要的便是顶点着色（Vertex Shading）。我们看到不同的物体可以呈现出不同的材质以及明暗程度，这就是着色器在使物体表面每一个像素点能都够呈现出不同的颜色。在我的渲染器中，我是用的是最常用的Blinn-Phong反射光模型，它能够用简单的数学原理尽可能解释物体表面呈现出不同颜色及明暗程度的原因。在现实世界中有三种光照类型：高光：物体完全反射光源照射到表面的光照到人眼；漫反射光：物体朝着任意方向反射相同亮度的光照到人眼；环境光：物体表面完全背对光源，但其它物体反射光照照射到该物体表面上，再通过该物体反射光照到人眼。我们聚焦在物体局部的某一个点上。

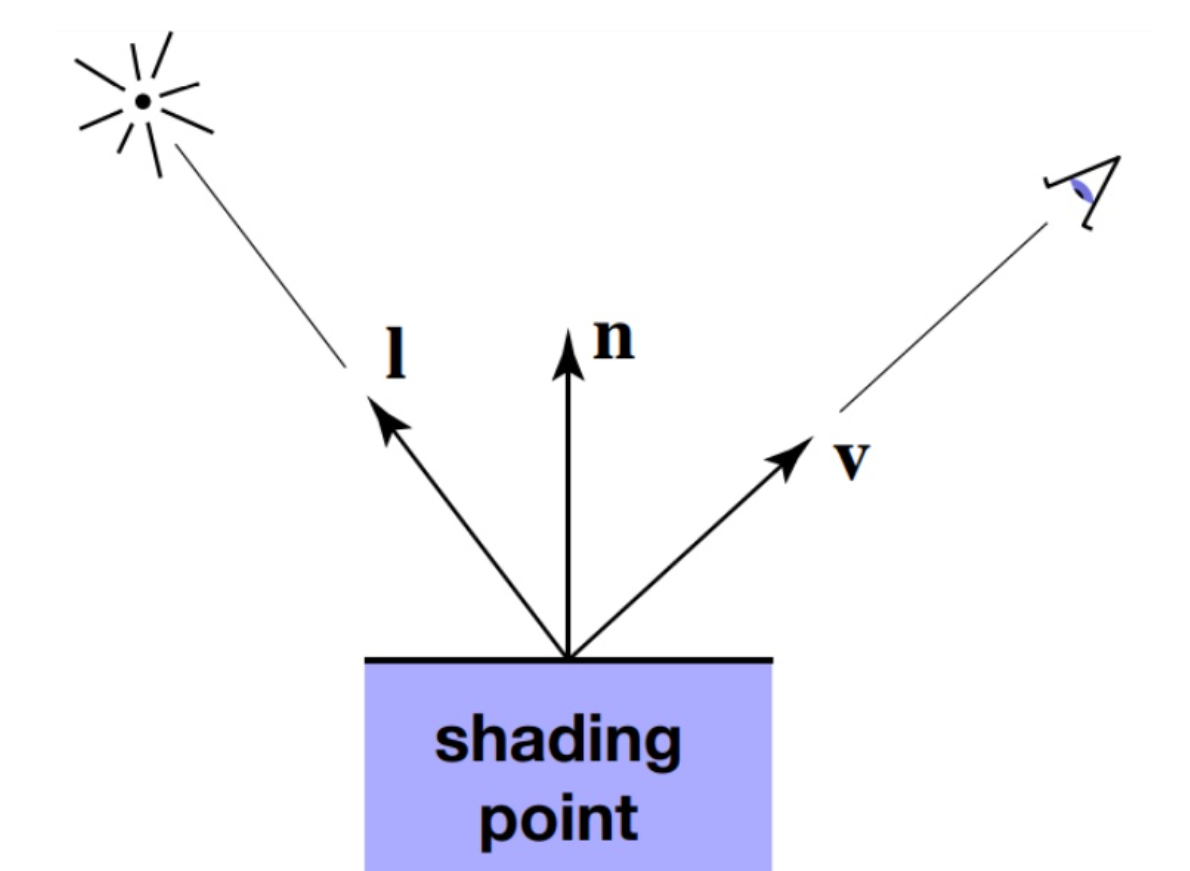


图 5 光照图示

图中的向量l代表光源的直射方向，向量v代表视线方向，向量m代表物体表面的法线方向。漫反射光是物体表面向四面八方反射相同强度的能量，所以：无论从哪个视角看同一个点，都应该呈现出相同的颜色。所以，Blinn-Phong反射光模型的整体公式如下所示，其中k值都是对应光照的参数，h是半程向量，也就是光源方向l和镜面反射方向R的角平分向量：

将上面的三个公式计算得到的值就是最后片元对应渲染出的颜色

接下来是光栅化，光栅化是指从二维顶点所处的屏幕空间到屏幕上的像素的转换。在我的渲染器中，光栅化包括遍历三角形，并在三角形周围建立边界框，对于边界框中的每个图元，使用下面的质心公式计算当前点P是否在三角形ABC中并得到至质心坐标中：

若上面的式子有解，且满足,那么点P是三角形ABC中的点，得到的质心坐标可以用于后面对片元P的纹理，光照强度，深度缓冲Z-buffer等进行插值。

在上面，我们引入了深度缓冲Z-buffer，它的用处是，当我们看向一个模型时，内表面会被外表面所遮挡，也就是说内表面离我们眼睛的距离会大于外表面离我们眼睛的距离，所以我们判断一下两个平面离我们眼睛的距离，在渲染时，只需要渲染离我们眼睛近的那一个。

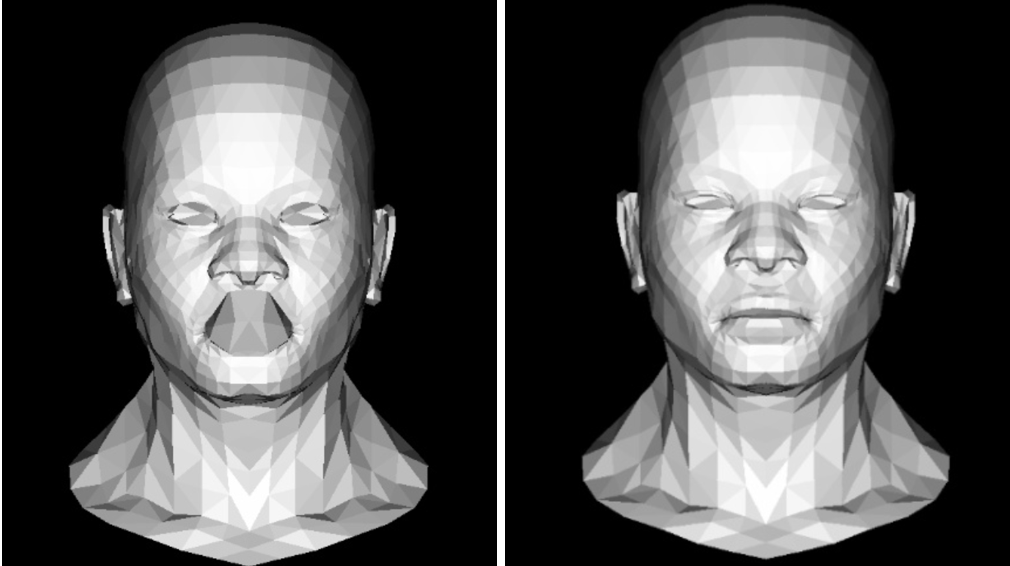


图6 左图未引入Z-Buffer右图引入Z-Buffer

我的渲染器的最后一步是片元着色。通过上一步，我们可以得到三角形片元中的每一个图元，并得到它的质心坐标，通过质心坐标结合顶点着色器得到的三角形中的三个顶点的纹理，光照强度，深度缓冲等信息，我们就可以带入质心公式得到每个三角形中片元对应的信息，也就是渲染出最后的图像，如下图所示（这里还用了透视矫正，但因为篇幅原因不再赘述）：



图 7 最后渲染出来的图像

在完成了上面的这些步骤之后，我把顶点着色器和片元着色器抽象出来，使得使用这个渲染器的用户可以自由定制获得的渲染效果，例如若在片元着色器中，选择若颜色处在一定范围内边渲染成同一种颜色，这便是卡通渲染效果，若在顶点着色器中，改变顶点在用矩阵进行不同空间转换对应的矩阵的参数，那么就可以得到不同大小或是不同角度的渲染图像。



图8 使用同一个软渲染器不同可编程片元着色器得到的不同渲染效果

1. **拓展渲染模块**
   1. **阴影贴图**

阴影贴图（Shadow Map）是一种传统的阴影生成算法，其核心思想是通过额外引入一张记录了深度信息的图，使得每一个着色点会根据是否在阴影中而产生着色结果的变化，来达到产生阴影的目的。在拿到深度后，我们就可以通过对比当前shadowmap上的片段深度与采样深度来判断该片元着色点是否被遮挡。

其中，阴影的生成还可分为硬阴影和软阴影，后者阴影的边缘会随着这单程度产生变化。在我的代码中，我使用了PCF计算软阴影。的计算思路是，在计算一个着色点的visibility项时，我们不单单考虑当前着色点与其对应的阴影贴图上一个像素的深度对比，而是将阴影贴图上该像素周围的像素深度也进行对比，最后按照某种方式加权平均，最后可以得到一个（0，1）范围内的值，而不是传统ShadowMap中的非0即1。

* 1. **环境光遮蔽**

环境光遮蔽（Ambient Occlusion，简称 AO）是全局光照明的一种近似替代品，可以产生重要 的视觉明暗效果，通过描绘物体之间由于遮挡而产生的阴影， 能够更好地捕捉到场景中的细 节，可以解决漏光，阴影漂浮等问题，改善场景中角落、锯齿、裂缝等细小物体阴影不清晰 等问题，增强场景的深度和立体感。

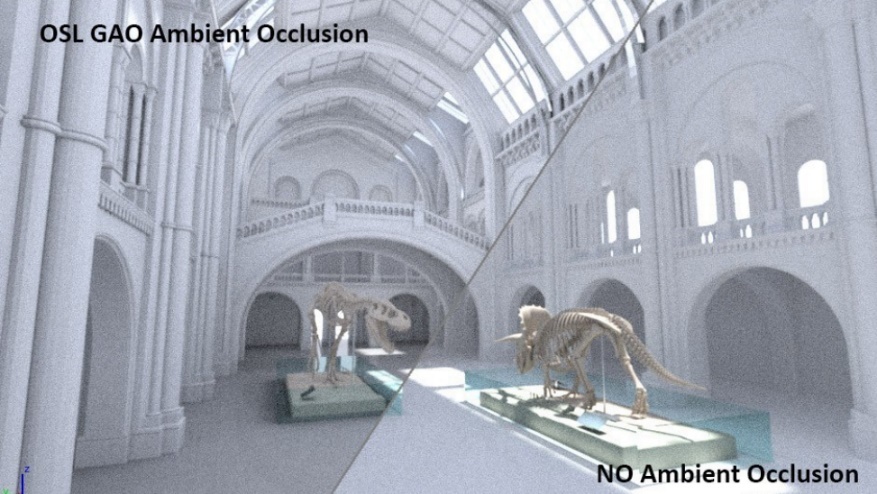


图 9 有无环境光遮蔽渲染效果对比

一种OA的实现方式是随机投射出一些光线，将光线通过每个三角面图元的重心点，构建出通过重心点的半圆，然后计算出所有三角形共用顶点的平均occlusion。这样的实现方式性能开销过大，虽然更加真实，但是只适用于理想渲染，对于动画引擎需要实现的实时渲染，我们通常采用屏幕空间环境光遮蔽技术（SSAO），也就是使用屏幕空间的法线和深度值来计算每个屏幕像素的环境光遮蔽率。

* 1. **光线追踪**

在完成上面的光栅化渲染后，我也尝试实现了光线追踪这个全局渲染方式。光线追踪是和我们上面所说的光栅化对应的概念。与光栅化渲染不一样的是，光线追踪把一个场景的渲染任务拆分成了从摄像机出发的若干条光线对场景的影响，这些光线彼此不知道对方，但却知道整个场景的信息。每条光线会和场景并行地求交，根据交点位置获取表面的材质、纹理等信息，并结合光源信息计算光照。

它的伪代码如下所示：

|  |
| --- |
| **Algorithm 1:** Ray Tracing |
| **输入:** 光线的原点，光线的方向，场景中所有物品的信息  **输出:** 屏幕上每个像素应该呈现的颜色  **算法步骤**： |
| for each pixel of the screen  {  Final color = 0;  Ray = {starting point, direction};  Repeat  {  for each object in the scene  {  determine closest ray object/intersection;  }  if intersection exists  {  for each light in the scene  {  if the light is not in shadow of anotherobject  {  add this light contribution to computed color;  }  }  }  Final color = Final color + computed color \* previous reflectionfactor;  reflection factor = reflection factor \* surface reflectionproperty;  increment depth;  } until reflection factor is 0 or maximumdepth is reached  } |

在上面的伪代码中，我们可以看到有一个递归，这个递归的思想是当我们从视点向成像平面上的像素发射光线，找到与该光线相交的最近物 体的交点，如果该点处的表面是散射面，则计算光源直接照射该点产生的颜色；如果该点处 表面是镜面或折射面，则继续向反射或折射方向跟踪另一条光线，如此递归下去，直到光线 逃逸出场景或达到设定的最大递归深度。

我最后实现的光线追踪效果如下图所示。从上面的伪代码中，我们会发现光线追踪是需要对于每一组光线和物体判断他们是否相交的，这样的判断对于动画中用到的模型是非常复杂的，因此在我的算法中，我将一般的模型简化成球体，这样只要通过球体与直线的基本相交公式就可以计算出场景中的球体与光线是否相交。

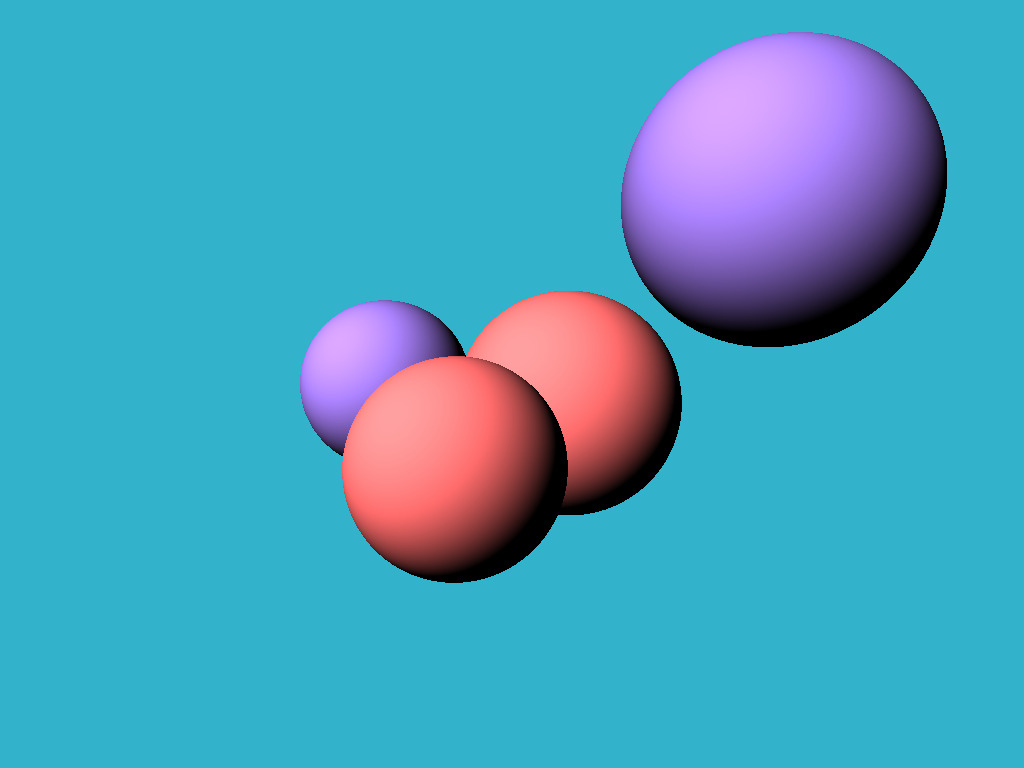
****

图 10 实现的光线追踪算法渲染出的图像

那么在一般的情况下，我们如何实现对于光线与模型是否碰撞的判断呢。模型可以抽象成网格（mesh），网格是由一个个三角面片元组成的，那么问题就可以转化成如何判断光线是否穿过模型的任意一个三角面。遍历模型的三角面显然是开销很大的方式，对于动画引擎这样需要实时计算出当前图像的应用，显示是不合适的，因此有了以下的解决方式。首先是AABB包围盒和OBB包围盒，他们使用一个包围盒包住该物体，在与该物体的三角面计算求交之前先判断光线是否与包围盒相交，倘若连包围盒都与光线没有交点的话，那么显然不会与物体的三角面有交点。倘若有交点，我们便可以使用例如KD树，层次包围树BSP或是层次包围盒BVH这样的结构将三角面图元存储下来，这样的存储结构相当于对世界空间或是模型空间进行了划分，优化了碰撞检测的速度。

在光线追踪的基础上，Kajiya在1986年提出了路径追踪，该方法的基本思想是从视点发出一条 光线，光线与物体表面相交时根据表面的材质属性继续采样一个方向，发出另一条光线，如 此迭代，直到光线打到光源上（或逃逸出场景），然后用蒙特卡洛方法，计算光线的贡献，作为像素的颜色值。而使用蒙特卡洛方法对积分的求解是无偏的，只要时间足够长，最终图像能收敛到一个正确的结果。

1. **基础动画模块**

对于动画引擎来说，五种基本动画经常被用到：精灵/纹理动画、刚体层次结构动画、蒙皮动画、顶点动画与变形目标动画。

刚体层次结构动画指角色本身由多个刚体部件构成，并按照某种层级结构进行部位之间的约束。由于这种层级约束的存在，使得角色可以根据约束状态实时计算出当前的姿势，然后在时间轴上插值不同姿势，从而达到播放动画的效果。这种技术不需要蒙皮操作。而缺少蒙皮操作带来的后果是，部位与部位之间会产生明显的裂缝。顶点动画（per­vertex animation）。动作设计师直接操作顶点，定义顶点在运行时的变化。这样的技术理论上可以产生任何能想象到的效果。现代游戏品 质下，游戏角色普遍都会包含几千甚至几万个图元或者顶点，如果使用顶点动画，会产生非常巨大的数据量，严重影响实时性能。因此，研究会着重于动画引擎最常用也是最重要的蒙皮动画。



图11 左图为刚体层级结构动画右图为顶点动画

蒙皮动画让动画师使用相对简单二点骨头系统，去设定精细三维角色网格的姿势。当骨头移动时，三维网络的顶点就跟随移动。其中骨骼网格渲染组件是连接渲染器和动画系统的桥梁。虽然这些组件能非常紧密地合作，但它们的接口还是有明确定义的.动画系统生成骨骼中所有骨头的姿势，这些姿势以矩阵调色板形式传至渲染引擎。之后，渲染器利用矩阵表转化顶点，每个顶点用一个或多个矩阵生成最终混合顶点位置。此过程称为蒙皮。

下面的图片便是我们自己做的独立游戏中，美术使用spine软件制作的蒙皮动画。为了实现在游戏中，角色更贴近显示。角色对于状态机的每种状态都应该有对应的生动动画，负责美术的同学为了更方便制作出这么多套动画，而不是每个动画的每一正都单独画一遍就会使用蒙皮动画。首先，美术同学会先画出人物静止站立的图片；其次通过像spine之类的绑骨软件按层级添加骨骼，使得骨骼基本与模型重合。选中模型，增加修改器。然后添加所有骨骼。这一步就是俗称的蒙皮。一般情况下，如果已经把骨架调整得跟模型基本吻合，系统会自动产生比较准确得蒙皮权重。现在，只要调节关节的角度，便可以让角色动起来，实现想要的动画片段。

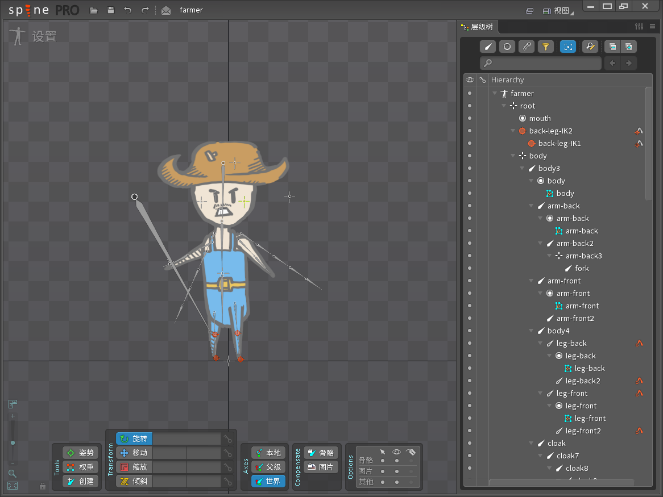


图 12 自己制作的游戏中2d美术的蒙皮动画

下方是动画引擎unreal中的蒙皮动画模块。我们可以看到，在上面的美术同学的工作完成后，程序同学也就是我会把制作出来的模型导入到动画引擎中，导入之后的效果如下所示。

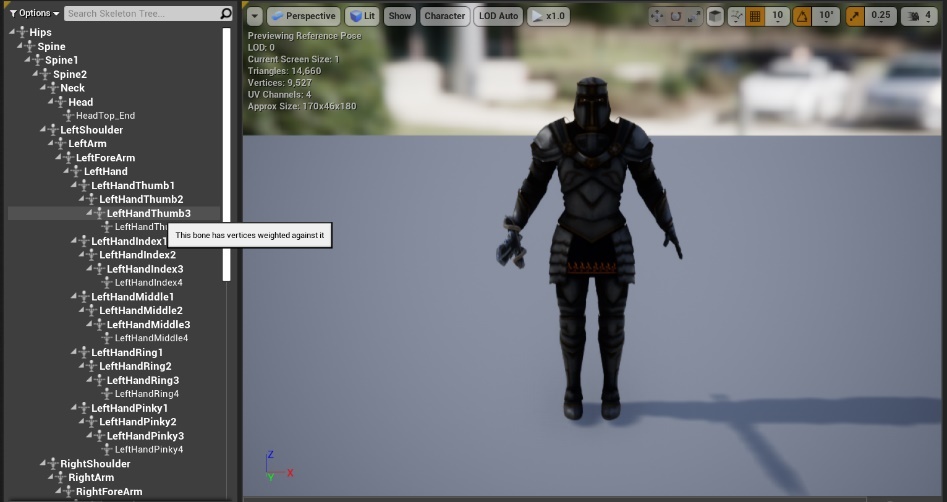


图 13 unreal中3d模型的骨骼层级

上面便是一般的动画制作或是游戏制作中使用蒙皮动画技术优化动画制作与动画表现的流程，下面我将阐述在我的动画引擎中，这部分蒙皮动画的实现。

因为之前做出的软渲染器难以满足我们制作蒙皮动画的需求，用所以这一部分采OpenGL这个市面上几乎最常见的图形API制作。接下来我将讲解我对于这个模块的实现。  
 首先，我们使用OpenGL中的Assimp库导入模型。在此之前，我们需要了解一下蒙皮动画技术中的一些基础概念——骨骼、关节、姿势与蒙皮。在蒙皮动画技术中，每一个模型都需要赋予一套骨架。如同上面图示的unreal中的模型，骨架由刚性关节以层次结构所构成。所谓刚性，即物体不会发生形变。但是实际上关节并不是真实存在的物体，更合理地是把关节理解为一个新的坐标系。动画设计师控制的是关节，而骨骼只是连接在父子关节之间的假象对象，用来辅助人类理解骨架。一个关节的姿势定义为 相对于某个参考系的位移、旋转、缩放，通常可以表示为4\*4或者4\*3的矩阵，所以整个骨架的姿势可以用矩阵数组来表示。

在知道了上面的基础概念之后，我们便可以正式使用OpenGL结合它的Assimp库将格式为md5mesh格式的模型导入到我们的动画模块中。用Assimp导入的模型数据是按下面的方式组织起来的：

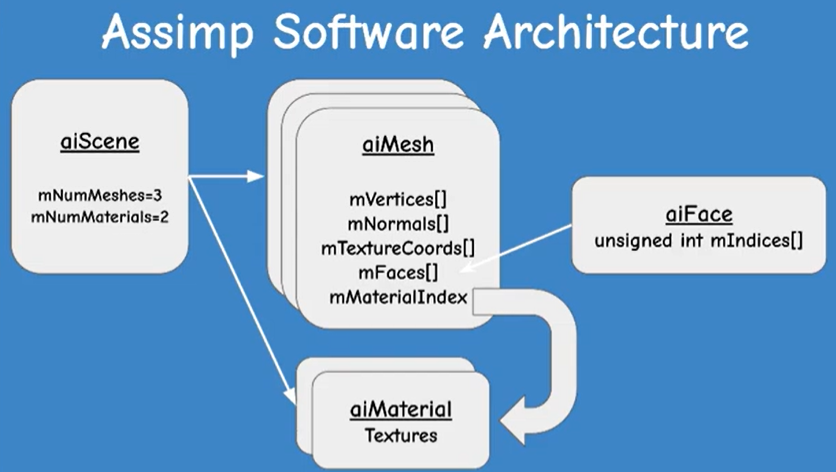


图14 用Assimp导入的数据格式

在每一个网格mesh数据中，我们还需要读入关节、姿势的数据。关节、姿势通常相对于父关节来指定，这样做可以用父关节带动子关节，从而产生非常自然的运动效果。数学上，一个关节就是一个仿射变换。第j个关节的变换矩阵为：

所以，一套由N个关节构成的骨架而言，它的某一个姿势可以表示为矩阵集合：

最后是最重要的一部分——蒙皮，其中在实现蒙皮动画中最关键的一点是某个点受关节影响，则这个点在关节坐标内的空间坐标保持不变（关节空间）。也就是说，模型上的每一个顶点，都需要保存它受哪一个关节影响，以及受影响的程度。大部分引擎都会把影响顶点的关节数量限制在4个以内。如果顶点受多个关节影响，一般做法是利用骨骼权重（BoneWeight）对骨骼矩阵（BoneMatrix）进行线性叠加，然后再进行空间变换。

其中骨骼矩阵描述了顶点从模型空间变换到关节空间的变换矩阵。蒙皮之前，顶点是定义在模型空间的。假设P受关节B影响，权重为1。根据上面我们提到的关键点，当我们调整骨架姿势时，需要保证P点在B关节空间中保持不变。于是，骨架从T字形静止站立姿势到目标的过程中，每个顶点需要经过下面的矩阵变换：

在我们导入的数据中，顶点和关节的关系如下

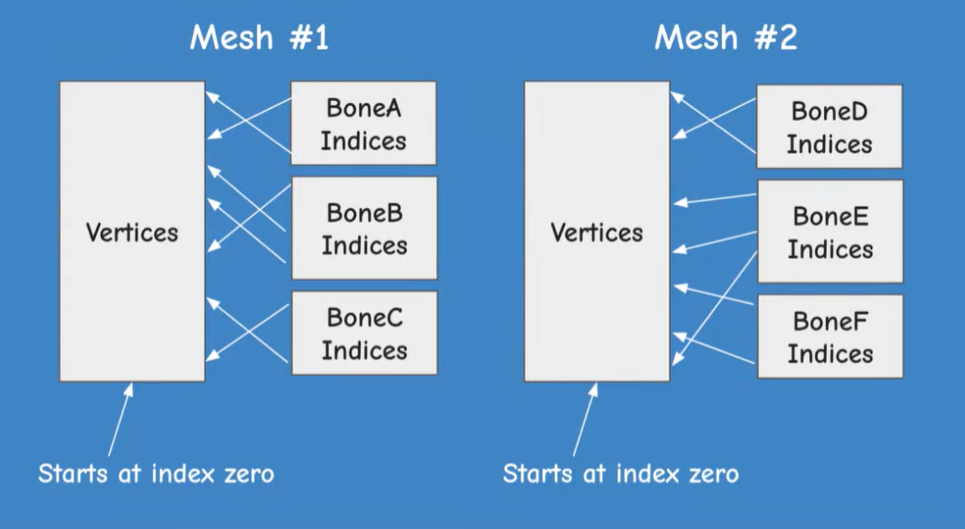


图15 骨骼与顶点的对应关系

按照上面的数据结构，我们便可以从导入的md5mesh文件读入蒙皮动画需要的数据：



图16 从模型文件中得到的数据

其中，蒙皮权重描述了某个顶点如何受多个关节点影响。这个权重一般需要由动画制作同学在专业的软件中确定。蒙皮权重如果没有调整好，会导致模型在关节连接处在某些动作下出现明显的瑕疵。如果一个顶点受多个关节影响，一般采用的线性权重蒙皮算法，通过加权求和的形式计算最终在模型空间的坐标。如下所示：

为了确保我们对于每个顶点都得到了正确的权重，我在真实渲染出静态模型之前首先使用一个着色器去遍历每个骨骼，对于受到当前骨骼影响的每个顶点，用不同颜色表示当前骨骼对于当前顶点的骨骼权重。

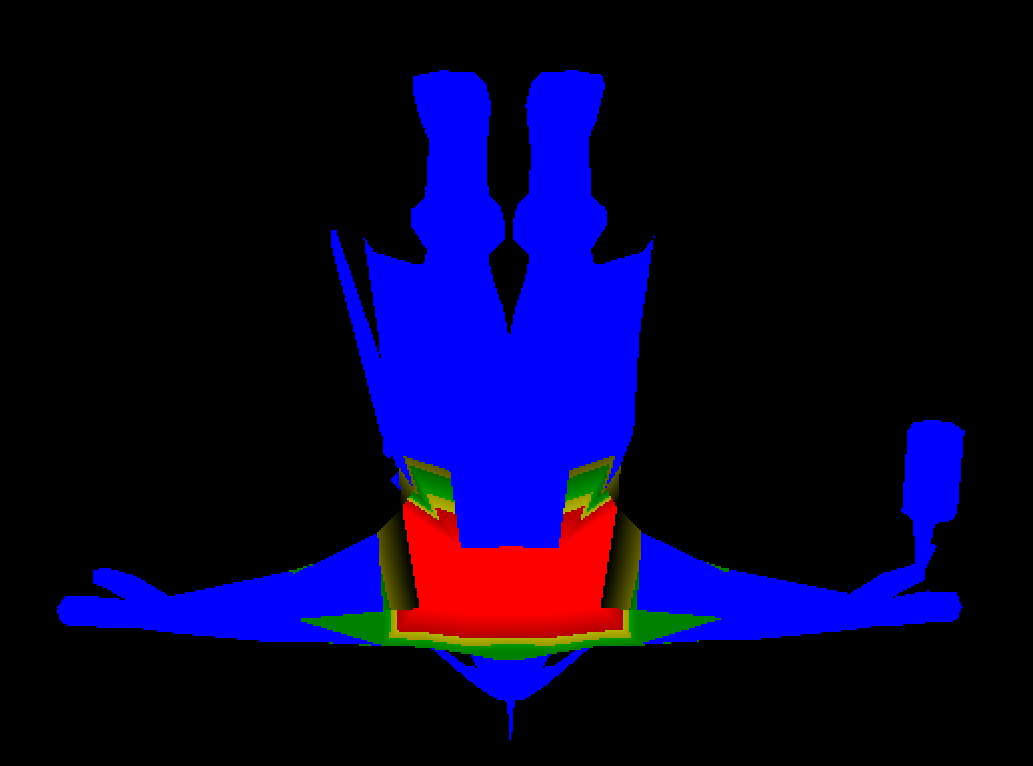


图17 0号骨骼对于其影响的点的骨骼权重

确认上面的骨骼权重无误后，我先渲染出了T字形静态模型。这一步和在我之前的渲染模块的渲染方式是相同的，因此这里只简单介绍下同样的算法在openGL结合Assimp库中的实现。的动画模块中，我以网格Mesh为单位进行渲染，其中Mesh结构体中包含了顶点缓冲，索引缓冲以及材质的索引。材质（Texture）指的是贴图纹理了，而网格实体是可以共享材质的因此我们还要为每个纹理分别设置相应的向量，对应的，每个网格便会有一个材质索引来为自己绑定合适的纹理。在获取了上面的网格结构体之后，我们可以通过解析这个结构体获得当前网格上的顶点，这些顶点对应三个相互独立的数组，分别是顶点位置数组、顶点发现数组、顶点对应的纹理坐标数组。这三个数组囊括了后续渲染所需要的顶点信息，这些信息是我们可以对每一个顶点构建定点结构体。除了顶点信息，从网格结构体中我们还可以得到模型三角面的信息，这个信息的形式是顶点的索引，代表这个三角面是由哪几个顶点组成的。对于每个mesh，读到了上面的信息便可以使用库开启深度测试进行渲染。



图18 渲染得到的静态T姿势模型

在知道了导入静态T形姿势的原理后，我们使用类似的步骤导入md5mesh格式的模型动画文件。接着，对于关键帧进行采样获取当前帧对应的骨骼矩阵，采样完成后，我根据当前时间换算得到采样时间，利用采样时间，找到对应的动画片段和前后关键帧；根据前后关键帧，插值得到骨骼矩阵也就是上面公式中的来让顶点坐标从模型空间变换到骨骼矩阵也就是上面公式中的，相当于线性迭代多个骨骼对vertex的变换和权重。利用上面的骨骼矩阵，我们可以进行蒙皮变换，如下所示：

最后，把上面变换后得到的顶点再使用顶点着色器进行MVP变换和光照计算，便可以渲染出完整的蒙皮动画。



图19 最后渲染得到的蒙皮动画中的四帧

**4、动画模块拓展**

**4.1** 运动模糊

运动模糊是快节奏的3D游戏中比较常用的技术，给运动的物体添加模糊效果，视觉上可以提高速度感，提高玩家的真实感体验。运动模糊技术的中最重要的一个概念是：运动向量，运动向量描述的是渲染出的同一个像素，在两帧之间的移动方向和距离。

具体的算法实现为，当正常渲染完毕之后，再渲染得到的结果上再进行一次渲染，也就是后处理。在第二次渲染中，从当前像素出发，沿着运动向量在当前帧的颜色缓冲上采样附近像素，并将采样的颜色取颜色均值，就得到当前像素下物体的运动模糊颜色值。最后得到的结果如下图所示：



图20 经过运动模糊之后得到的图像

**5、物理模块预研**

碰撞检测对每个动画于游戏都很重要。没有碰撞检测，物体会互相穿透且无法在虚拟世界里合理地互动。一些动画包含真实或半真实的动力学模拟。这些在游戏业界称为刚体动力学模拟；因为动画中通常只考虑刚体的运动，以及产生运动的力和力矩。研究运动的物理分支是运动学，而研究力和力矩的是动力学。

其中，刚体是理想化、无限坚硬、不变性地固体物理；动力学是一个过程，计算刚体怎样在力地影响下随时间移动及相互作用。刚体动力学模拟令游戏中地物体能高度互动及混沌地自然移动，这种效果通常难以用预制的动画片段达成。动力学模拟需大量使用碰撞检测系统，以正确地模拟物体的多种物理行为，包括从另一物体弹开，在摩擦力下滑行、滚动，并最终停止。

物理模块是十分复杂的模块，我初步指定使用OpenGl和bullet库相结合或实现这个模块。后者的引入是由于物理模块所需要的GJK、EPA等算法的实现十分繁琐。首先，我会定义一系列物理模型，例如胶囊体、球体、平面等。其次，我会在模块中引入动力学基础是上面的物理模块的碰撞可以产生正确的交互，其中既要考虑物体基于质心的线性运动，还要考虑物体围绕质心的旋转运动，其中冲量的概念十分重要。冲量是指力在时间上的累计效应。有了上面的两块内容，我们就可以在每个固定时间dt（这里注意不是每帧）更新计算物体的碰撞与对应的表现，这里的更新大致可以分为下面四块：

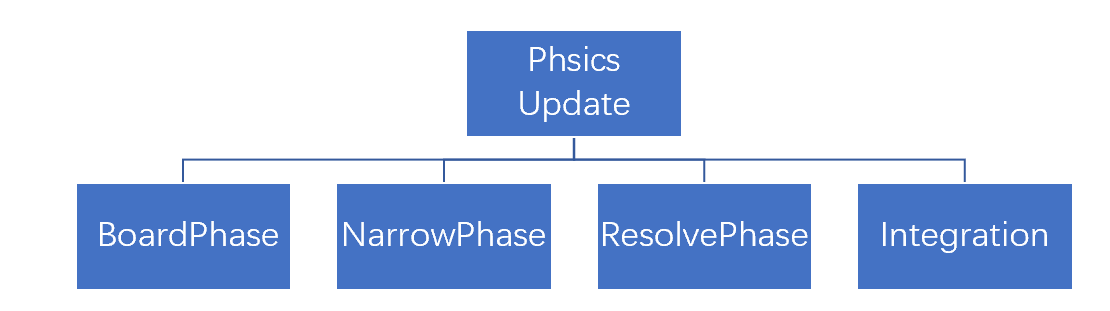


图21 物理模块更新的四个阶段

具体每个模块de 作用，我将在最后的结题报告中再做详细的阐述。

三、后期拟完成的研究工作及进度安排

继续优化已经完成的渲染模块和动画模块，探索更多这两方面性能与效果的改善方法。同时，继续研究物理模块，完成对于物理模块的设计。最后，从整体去剖析各个动画引擎，特别是开源的各个动画引擎，探索动画引擎的整体架构。

四、存在的问题与困难

缺乏对于动画引擎架构的整体把握，现在会更注重于模块的探索与设计。虽然有些著名的动画引擎的源码是开源的（例如现在最流行的动画引擎之一——虚幻引擎），但是代码量庞大，不容易阅读。对于需要应用的openGL这门图形学api不够熟悉，它相对于我之前熟悉的shaderlab来说，接口更加繁杂，使用更加困难，若不使用任何现成的库，创建一个普通的三角形就需要200多行。

五、论文按时完成的可能性

论文大概率可以按时完成。

六、主要参考文献

[1]Gregory J. 游戏引擎架构[M]. 2. 电子工业出版社, 2019.

[2] Ginsburg D, Purnomo B. OpenGL ES 3.0编程指南[M]. 2. 机械工业出版社, 2015.

[3] Eric Haines, Naty Hoffman, Tomas Akenine-Moller. Real-Time Rendeirng[M]. 2. A K Peters/CRC Press, 2018.