**3D原型制作引擎设计**

[**1. 简介**](#_Toc374468422)

[**2. 基本框架**](#_Toc374468433)

[**2.1. 管线模块**](#_Toc374468435)

[**2.1.1. 顶点缓冲管理**](#_Toc374468436)

[**2.1.2. 渲染目标管理**](#_Toc374468437)

[**2.1.3. 纹理管理**](#_Toc374468438)

[**2.1.4. 渲染状态管理**](#_Toc374468439)

[**2.1.5. 着色程序管理**](#_Toc374468440)

[**2.1.6. 遮挡查询**](#_Toc374468441)

[**2.1.7. GPU计时器**](#_Toc374468441)

[**2.2. 资源管理**](#_Toc374468435)

[**2.2.1. 几何资源**](#_Toc374468436)

[**2.2.2. 动画资源**](#_Toc374468437)

[**2.2.3. 材质资源**](#_Toc374468438)

[**2.2.4. 纹理资源**](#_Toc374468439)

[**2.2.5. 着色代码资源**](#_Toc374468440)

[**2.2.6. 管线流资源**](#_Toc374468441)

[**2.2.7. 场景图资源**](#_Toc374468441)

[**2.3. 场景管理**](#_Toc374468435)

[**2.3.1. 模型节点**](#_Toc374468436)

**2**[**.3.2. 网格节点**](#_Toc374468437)

[**2.3.3. 关节节点**](#_Toc374468438)

**2**[**.3.4. 光源节点**](#_Toc374468439)

[**2.3.5. 相机节点**](#_Toc374468440)

[**2.3.6. 粒子节点**](#_Toc374468441)

[**2.3.7. 集合节点**](#_Toc374468441)

[**2.4. 数学模块**](#_Toc374468435)

[**2.4.1. 矢量**](#_Toc374468436)

[**2.4.2. 矩阵**](#_Toc374468437)

[**2.5.3. 四元数**](#_Toc374468438)

[**2.6.4. 空间几何算法**](#_Toc374468439)

[**3. 其他**](#_Toc374468433)

1. **简介**

本引擎的目的是为想快速实现自己渲染算法或快速实现希望的渲染原型的程序开发者提供一个基本框架和可快速定制管线流程的渲染库。不同于一般的3D引擎，此引擎给了了更多的底层接口给用户，并且可以通过配置文件、程序编写实现。并且提供了接口可以让使用者加入自己的组件到引擎。此引擎希望更多给用户细小的程序接口、更加底层和通用，所以接口都以C语言实现。并且使用opengl让用户可以更容易移植。除了用于原型开发，用户完全可以自行扩展并作为自己图形引擎开发自己的项目。

1. **基本框架**

**2.1. 管线模块**

此模块封装了基本的底层图形API，封装管线操作。目标是尽量让其更通用，以方便后期渲染API的切换。OpenGL管线结构包括：VtxSPec，TF，VS，TCS，TES，GS，XFB，Raster, FS, PixOps, FB, CS。这几个阶段，其中现代GPU可编程阶段VS，TCS，TES，GS，FS，CS 这几个阶段在程序开发中十分重要。一个灵活且底层的引擎当然应该支持这些阶段编程。虽然DX和OpenGL总体管线有所不同，但可编程阶段是一致的，不影响程序的设计。并且将复杂的底层图形API封装，提供简单，灵活的接口以管理GPU资源。为了减少内存碎片，和动态内存申请的性能影响，使用内存池对这些数据进行管理。此类实现对所有GPU硬件资源的管理，其生命周期是全局的，我们使用单例模式创建。

**2.1.1. 顶点缓冲管理**

此模块虽然说是顶点缓冲管理，实际上将VAO,VBO,IBO封装到了一个类中，并提供了各自的创建方法。并且一个OGLBuffer 绑定其顶点数据输入结构描述，将VBO和IBO关联到一个结构体中，以内存池管理。VAO另开一个内存池管理。基本方法包括：创建，更新，绑定，销毁，和 空间占用信息获取等方法。

**2.1.2. 渲染目标管理**

渲染目标是指每个pass 结果输出到的目标缓存。在OpenGL中实现我们可以通过FBO，RBO，联合纹理来实现向纹理的渲染。利用其实现的离屏渲染我们可以实现许多渲染算法如：shadow map，radiosity shading，reflection 以及 各种后期处理特效。当然现在在各自游戏引擎里十分流行的延迟渲染都得用到此对象。此类我实现FBO的绑定管理，绑定到默认FBO让结果渲染到屏幕。此类将OGLTexture类以组合模式加入，以保证可以访问纹理的方法。 内存池管理渲染目标。基本方法包括： 创建，获取缓存，获取纹理句柄，绑定当前渲染目标，绑定当前被渲染的纹理， 缓存基本属性设置。

**2.1.3. 纹理管理**

纹理采样着色是一种非常简单有效的几何表面着色方式。每张纹理我们除了提供图片数据，还要对其绑定一个采样器以设置纹理的采样方式。纹理是占用GPU内存资源的主要资源所以本引擎提供了纹理压缩设定。为提高纹理贴图时的性能和视觉质量，MIPMAP是必不可少的。并且在一些渲染算法中可以用mipmap做到一些高性能且有趣的效果如：以mipmap等级实现面环境反射时模拟曲面粗糙度。由于纹理占用大量的内存空间，引擎给用户提供了其总内存占用的方法。同上以内存池管理纹理缓存。此引擎最多支持最多同时启用15个纹理对象，使用数组保存当前活动的纹理。提供基本的方法包括：纹理创建，纹理更新，纹理绑定，纹理销毁，获取纹理数据，添加绑定活动纹理单元，绑定采样器状态，获取纹理属性等。

**2.1.4. 渲染状态管理**

渲染状态包括：光栅化状态，混合状态，深度模板状态。每种类型的状态将使用结构体保存其状态描述，并且每种状态都将保存前一个和当前状态以便状态回滚。因为需要维护大量的状态描述，所以为节省运行内存，描述结构体中各个参数使用分位存储并联合hash，进行快速状态切换比较，如果当前状态和前一次状态相同就没有必要想GPU发送更多的命令，以节省CPU资源。各个状态描述使用大量枚举表示。并且在类中提供所有状态设置，获取方法。

**2.1.5. 着色程序管理**

引擎支持顶点着色器，几何着色器，曲面细分着色器，像素着色器，以及计算着色器。其中由于计算着色器和其他阶段关联不大，所以单独处理。着色程序也是绘制的核心，用户编程的时候回大量编写着色程序以实现渲染算法。引擎在原有OpenGL的着色程序语义上进行了扩展，将常量，采样器，渲染状态加入到了引擎定义的shader代码语义中。其语义更接近fx及cg语言着色程序。着色程序分为两个内存池，分别储存用户定义常规着色程序和计算着色程序。其中用户自义着色程序中几何与曲面着色程序为可选程序。类中基本方法包括：着色程序的创建，销毁，绑定，程序编译的日志获取，常量，采样器设置，句柄获取，顶点属性句柄获取等。

**2.1.6. 遮挡查询**

遮挡查询的目的是利用GPU查询几何空间中几何元件被遮挡结果。利用此类查询几何元件的遮挡情况，我们可以选择不绘制以减轻计算负担。

**2.1.7. GPU计时器**

查询GPU时间辍。

**2.2. 资源管理**

资源是一些用于渲染场景的数据对象，如：纹理或着色代码等。资源的一个重要特点就是可以重用。这就意味着这些资源只需要被读取一次并且可以被多个对象引用。每个资源通过名称指定并且在同类型资源中不可重名，它们可以被其他对象引用。

所有资源通过一个资源管理器进行管理。资源管理器包含一个资源表以确保每个资源只被读取一次并且之后可以被重复使用。它还支持资源的查找、访问、移除。想对于其它引擎，使用了一个单独的资源管理器对于每个资源都没有类型划分，访问每个特定资源的基类指针。为了保证处理资源时的健壮性，管理器使用了引用计数。对于一个资源只有在它没有被其它对象引用的时候才可以被移除。这样还可以保证未被使用的资源更好的被收集并做释放处理。

引擎使用延时资源读取。这意味着资源的创建和读取是异步的。这样的处理方式带来的好处就是这些数据被添加后不用立即启用的情况下可以创建一个进程进行后台加载。这些资源被创建前被初始化为特定默认数据类型且能够被其他对象有效的引用。这些资源成功创建将可以读取和填充期望数据。这个阶段基础数据结构被建立和初始化。一般情况下在数据读取之后只能改变数据的值而不能改变结构。如对于一张纹理只能改变每个像素的颜色值却不能改变纹理的维度和图片色彩位数。当资源已经被读取了的情况下不能再次读取，当卸载后可以重新读取但是要重新设置初始状态并直接创建。

所有的资源都继承Resource子类，并实现其虚函数。并将所有资源用ResourceManager进行管理。ResourceManager可以提供资源类型注册，资源添加，移除，读取，查找等操作。

**2.2.1. 几何资源**

几何资源从二进制文件读取且通过模型和网格储存多边形数据。这些资源包括三角形索引，一些顶点属性数据流如：顶点位置，法线向量，切线向量，纹理坐标，节点权重。并且还可以对模型添加骨骼动画相关数据：为变形目标添加一个单独的数据流。几何资源格式可参考：geo\_ani.vsdx文档的几何文件结构图表。

**2.2.2. 动画资源**

动画数据资源包含动画节点的关键帧数据。动画分为网格和关节两个节点类型。这些节点通过定义名称并存储每帧本地位置变换向量，缩放向量，旋转四元数。

动画资源格式可参考：geo\_ani.vsdx文档的动画文件结构图表。

**2.2.3. 材质资源**

材质大体定义了网格或渲染对象的可见外观。一个材质引用一个shader并且绑定其中的全局唯一变量。这些全局唯一变量可以是纹理贴图，浮点类型点向量，通过这些数据类型可以传递任意指定应用数据到shader。

**2.2.4. 纹理资源**

纹理资源可以使用2D纹理贴图和Cube贴图。2D纹理就是一张简单的图片。2D纹理贴图通常被着色器使用来定义面的显示效果，不过可以编码为非图片数据如：法线向量或查找表。Cube贴图通过六张图片产生一个正方体表示的贴图，cube贴图表示一个点周围的环境。所以可以通过一个三维向量指向不同的方向来观察纹理。

**2.2.5. 着色代码资源**

在引擎中shader有扩展语义，相比一般概念本引擎的shader并不是都在显卡上面执行。产生这个想法的原因是由于模型在不同的阶段需要使用不同的shader渲染。生成阴影贴图就需要其他shader代码计算出的环境光数据。所以在引擎中一个shader资源是一个程序集并执行在不同的环境下。单独的一对顶点着色和片元着色程序代码被称作一个shader环境。管线通常使用一个渲染环境作为当前步骤。真正的着色代码将被通过GLSL在显卡上执行。代码子文件是纯文本文件可以添加任意代码。代码文件可以被shader格式文件引用，并且其中的着色代码可以被重用。

**2.2.6. 管线流资源**

管线由XML文件储存对场景的渲染步骤。管线包含一个渲染列表，决定了每个阶段哪个几何图形被渲染并输出什么缓冲数据。每个管线可以输出任意数量缓冲区即：渲染目标。渲染目标临时储存渲染结果并将被下一渲染阶段使用。可编程管线十分强大，使用一连串shader，可以实现许多的现代渲染技术如：延时渲染和其他的高级后期处理效果。

**2.2.7. 场景图资源**

场景图资源储存在XML文件中，里面定义了场景图的分支。每个XML元素通过名称表示一个场景节点。XML的节点结构直接可以映射到场景层次。通过引擎API可以访问所有的场景节点属性并通过修改XML属性进行相应配置。引擎一个特点就是将所有的3D模型表示为场景图文件。这样做可以就像网格一样访问模型的部件，使用场景图API而不需要添加其他函数。

**2.3. 场景管理**

场景图用于表示将被渲染的场景的逻辑或空间结构。通常场景图服务于加速基于场景搜索和查找操作，比如：可见性剪裁或碰撞检测。在引擎中场景图通过一个树结构来实现。每个节点可有拥有任意数量的孩子节点但是引擎强制添加了一些限制，通过定义层级规则组成决定了哪个节点类型可以连接在哪个父节点。

其他所有的抽象实体如材质等均表示为节点属性。这个思想使得图中的节点数量相对较少并且在树中启用操作更快。每个节点有两种变换，一个是局部变换一个是全局变换。局部变换，也叫对象变换，它相对于父节点进行变换并由程序明确指定变换参数。全局变换（世界变换）通过遍历父节点的所有子节点并通过层级累加的方式自动计算相应变换。所以当一个节点被变换，其子节点也要相应变换。例如：如果想将场景完全旋转，只需要将根节点进行变换剩下引擎将自动旋转所有子节点。

每个节点还有对应的规则绑定盒以及相应的子节点。对于每个节点类型都有相应的函数并通过不同的参数进行创建。大部分参数在创建节点后是可以改变的但是关系到节点内部结构的数据是不可改变的。基础公共属性对于所有场景节点均可通过函数进行访问。为了获取和修改特定属性，API提供的函数通常使用一个句柄来访问相应类型场景节点。

**2.3.1. 模型节点**

模型要么是静态模型要么是骨骼模型。这个节点引用一个几何资源其中包含多边形数据。模型节点实际上是对于网格和关节的抽象容器，并且可以利用动画资源产生动画。模型本身是不可渲染的必须通过网格使得模型可见。通常模型节点的关节和网格是通过导出或转换工具创建并存储在场景图资源中的。

**2.3.2. 网格节点**

网格节点定义了一组多边形集合和一个单独的材质。每个网格表示为几何资源的一部分且通过联立材质进行渲染和绘制。

**2.3.3. 关节节点**

一连串关节节点用于表示骨骼动画的关节。网格，关节节点可以和父模型在场景图层次中单独存在。每个铰接件有个索引属性在模型的几何资源中来定位一个绑定矩阵。通过绑定矩阵可以进行蒙皮变换。

**2.3.4. 光源节点**

灯光源用作设置场景光照。这个节点通过一系列参数设置灯光属性如：灯光颜色，光照半径，聚光灯可见域等。通过相应参数还可以决定阴影。有意思的一点是一个材质可以指定一个灯光源。这里必须使用延时渲染因为要统一使用特定灯光源着色器进行后期处理步骤才能完成一个灯光的表示。灯光源提供一个光照环境属性来定义哪个着色器环境被用于计算光照强度。

**2.3.5. 相机节点**

相机在引擎中作为一个特殊的场景节点，场景总是通过一个虚拟相机的透视来渲染。通过这样的方式相机拥有场景图系统的灵活特性。如：可以将一个相机附加到一个动画模型，这样就可以根据模型动画自动变换相机。

**2.3.6. 粒子节点**

粒子节点用于配置粒子系统。通过粒子特效文件用于创建基于粒子的描述并且通过一系列参数配置属性，如：产生速度 或 最大粒子数量等。

**2.3.7. 集合节点**

一个集合节点是对于其他节点的一个通用容器。通常集合节点为根节点。集合节点有两个属性通过定义这两个属性可以设置对于这个节点的最大和最小可视距离。通过这两个属性可以非常容易地实现一个简单的LOD方案。

**2.4. 数学模块**

本引擎封装的数学库比较轻量，但基本满足常用计算操作。3D环境包含大量的数学计算，精度和效率都显得十分重要，但往往二者不可兼得，所以我们要在两者之间权衡。由于当前的硬件都对浮点运算做了加速，并且支持硬件上的矢量运算，我们几乎所有的计算都使用了浮点数。当然除了基本数学单位的封装，还封装了数学单位的转换与基本运算。为了方便表示空间几何我们还封装了基本几何类型如：平面，射线，点等。

**2.4.1. 矢量**

矢量包括2,3，4维矢量，利用矢量我们可以用来表示空间中的顶点位置，平移，旋转，缩放等。矢量运算要比标量复杂的多。除了重载基本的比较和+-\*/运算符外，还要实现点乘，叉乘，归一化，线性插值等方法。

**2.4.2. 矩阵**

矩阵在3D计算中大量运用，因为它能够表示大量的信息，且利于计算机进行运算。矩阵可以方便表示线性变换如平移，旋转，缩放，也可以表示相机的投影，空间切换。矩阵运算包括：转置，求值，求逆矩阵等等。

**2.4.3. 四元数**

本引擎使用四元数，因为四元数占用空间小，计算效率高。四元数不存在使用欧拉角时产生的万向锁的问题。在两个旋转向量之间四元数可以十分方便的做球面线性插值。这些优点保证了四元数在3D变换中的权威地位。不过在着色程序中四元数不方便计算，通知我们将旋转量保存在矩阵中，在需要旋转是再转换成四元数旋转后再转换回矩阵。本引擎除了提供球面线性插值方法，还提供了一个更快的旋转线性插值方法叫做nlerp。

**2.4.4. 空间几何算法**

这里提供了射线与三角形，射线与AABB的交叉算法。以及点到AABB的最短距离算法。这些方法是最常用的基本算法。需要使用到这些算法的最基本的功能有：LOD，几何，面的拾取，剔除等。