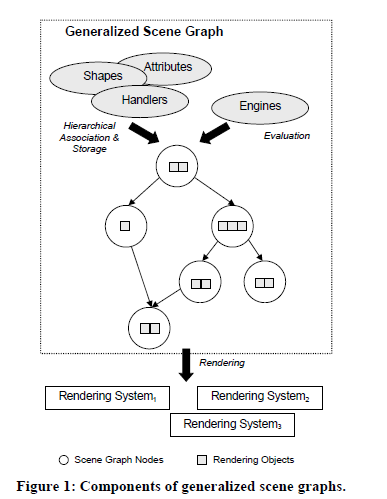
概要

场景图是分层场景模型最基本的数据结构。此通用场景图克服各种当前场景图架构下的限制，如：支持多个不同的3D渲染系统以及集成多个pass的渲染和场景模型定义。其主要的想法就是将场景描述与计算分离。指定场景时场景节点被管理作为渲染对象，并包含如：几何描述、属性、算法等。进行场景节点即渲染对象被渲染引擎计算，引擎将获取算法以描述计算对象的几何和属性。使用通用场景图，大部分的实时渲染技术可以集成到opengl以及一些3D渲染系统，并且不会使其难以控制或影响渲染系统的独立性。

1 介绍

场景图无处不在：大部分高层图形库都提供了场景图程序接口表示3D场景模型以创建3D应用程序。编写3D程序时场景图API应该精简易于使用的。并对渲染性能做到优化，将渲染算法抽象封装。此通用场景图支持不同的上层扩展开发。独立于渲染系统，可无缝地集成多pass的渲染技术和使场景建模更容易。

此通用场景图给予基于一个包含三个主要对象的对象模型（图1）。渲染对象包含3D和2D对象，可视效果和空间变换属性，以及渲染目标的Handle，渲染技术。场景图节点分层组织渲染对象，表示渲染对象的产生和规则。渲染引擎遍历并解析这些在生成的场景图中的节点内容。在场景图计算时，渲染引擎将管理管理一个渲染环境，一系列渲染对象和句柄handle。Handle实现一些算法用于映射属性和几何并用于构建一个目标的渲染系统。我们将handle从几何和属性中分离以支持不同的渲染系统。另外，handle并不会影响这类映射，可支持非图形渲染如：声音渲染。



实现多pass渲染算法这样的技术（这个技术是指渲染中的术语表示完成一个物体基本的渲染流程），这样就需要遍历全部或部分场景图多次；在场景图计算时技术触发器会多次遍历。

再者，通用场景图支持通过预计算分析场景内容建立场景描述模型。这个机制提高了场景描述的可用信和紧凑性，并且这样还可以进行优化让计算耗费降到一个合理的数量级。

由于快速的图形硬件实时渲染技术快速的增长成为可能(e.g.,通过联立的寄存器逐像素片元着色[7])给我们的工作提供了动力。由于计算架构的限制这项技术很难集成到现有场景图系统。此外，我们也在寻找在场景建模层次上的uniform和开放API用于不同的渲染系统。

2 相关工作

OpenInventor[14] 提出了经典的场景图API概念，其可以被其他系统适用（e.g，Java3D和VRML）。作为一个通常的特性，排序和整理场景图中渲染几何并反射底层的渲染管线。OpenInventor，Java3D和VRML专注于实时渲染而不容易适用于其他渲染技术（eg.照片级渲染和非真实感渲染）。此场景图API使用底层3D渲染系统可访问的方法也有不同：OpenInventor 提供完全访问OpenGL接口而VRML和java3D并不支持这样的扩展。

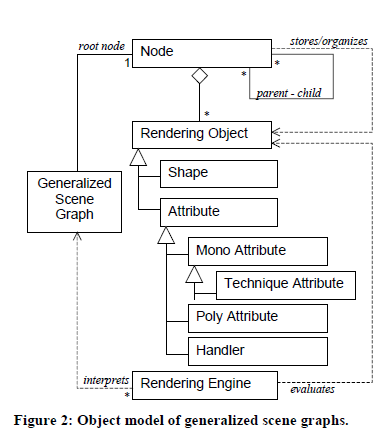
图形架构如PREMO[3]，BOOGA[1], 和 Generic3D[2] 表现可扩展，面向对象计算图形架构，但他们并没有尝试使用通用场景图像我们的一样工作。PREMO可以表示场景类似OpenInventor[15]，但Generic3D提供了一种不同的底层数据结构来定义场景表现类型。然而，这些系统的场景图本身和OpenInventor相比实际并没有多大的提升。

GRAMS[4]是一个小型的图形系统，它将渲染算法和渲染对象分离。几何渲染算法使用效率接近criterion[5]。在我们的通用场景图API中，这样的分离结构被扩展以支持计算属性和多pass算法。

Vision架构[13],专注于基于全局光照计算的图形运算。完全将几何对象和属性对象分开用基于对象设计甚至更加底层的系统架构[12]。在我们的工作中，几何与属性的分离被我们扩展：属性列表不止包含视觉属性和空间变换属性还包含渲染算法；一个通用的属性管理器Handle,依赖于渲染系统类型的类型属性。更者，我们通过引擎抽象和参数化计算进程并应用到场景图。

此论文概要了通用场景图的渲染方面强调通用和抽象图像内容的描述-交互和动画是相互互补的问题（e.g.事件句柄，控制管理）这里我们并不考虑。

3 通用场景图

一个通用场景图由场景节点和渲染目标组成。图2大概表示了此对象模型。

渲染目标

渲染目标列出为一下列表：

* 几何包含3D和2D通用对象同时也可表示体数据。几何对象不提供任何渲染的方法；它们只包含描述几何数据和状态的参数。这保证了它们的接口不会被限制到单种渲染技术。
* 属性包含可视属性（e.g,颜色，材质，纹理），空间变换属性（e.g.,旋转，相机状态，公告板），特性（e.g.,细节层次，拾取，标志符过滤），还有光照属性（e.g,光源，光照开关）。
* Mono属性储存在栈上，栈顶表示当前活动属性（e.g，颜色,材质,绘制风格）。
* Poly属性用集合来储存，ie,超过一个属性的类型也能被同样地启用（e.g.光源，拾取标志符）。
* 技术指定mono属性其封装里多pass的渲染算法。
* Handle指定mono属性其映射几何和属性用其组成一个特定的渲染系统，性能计算，或者转换渲染对象为一系列的渲染对象，以简化渲染。Handle包含如：几何绘制器，几何简化器，光线碰撞，属性绘制器等。
* 渲染引擎计算几何，管理属性栈和集合，解释（每个对象设定的）句柄。一个渲染引擎表示一个通用的渲染环境。

场景图节点

场景图节点组织渲染对象以分层的方式：它们也可生成和限制渲染对象而且它们都是自动进行场景建模。

通用场景图分辨在两个类型的场景图的遍历：evaluation和inspection。Evaluation解析和映射场景节点内容（e.g,图片整合）；渲染引擎执行evaluation。 而inspection只做场景图内容和图结构查找（e.g,场景图存储）。两者的设计都基于观察者设计模式。

Evaluation

Evaluation的时候，场景图节点和属性与渲染引擎进行数据交流。场景图节点可表示使用渲染引擎并处理渲染对象的“小型渲染程序”（图3）。在一个场景节点，渲染对象和子图可以被存储到不同类型的列表。下面的伪代码大概描述了节点类启用和禁用流程：

class Node {

private: List<Object> children;

public:

void eval(E : Engine) {

unapply(apply(children,E),E);

}

void inspect(V : Visitor) {

for each child c do {

V.explore(c)

if(c is a node) { c.inspect(V) }

}

}

};

proc apply(C:List, E:Engine):Stack {

S:Stack = {}

iterate c through C {

if(c is a shape) E.eval(c)

else {

S.push(c)

if(c is a mono attribute) E.push(c)

else if(c is a poly attribute) E.add(c)

else if(c is a handler) E.install(c)

}

}

return S

}

proc unapply(S:Stack,E:Engine) {

while S not empty {

c Ä S.pop()

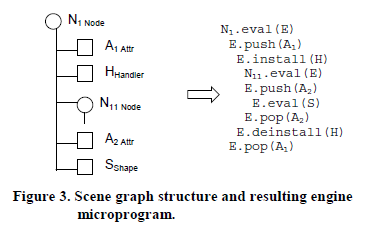
if(c is a mono attribute) E.pop(c)

else if(c is a poly attribute) E.remove(c)

else if(c is a handler) E.deinstall(c)

}

}



Mono属性被压栈到渲染引擎环境。Poly属性包含在他们各自的集合。Handle放在环境的handle列表中。在任意情况下，查找对应的属性handle或几何handle并调用映射或应用渲染对象到地城渲染引擎。在场景图节点，子节点可以和孩子节点列表任意混合-子节点开始于一个相互嵌套的evaluation。在任意情况下，一个节点的属性只会影响子节点，不会影响兄弟节点。

渲染引擎

一个渲染引擎负责联立几何和属性并调用handle。对于mono属性，引擎维护属性栈泛型列表。第一次一个指定类型的属性被使用，其类别的栈将会被创建；栈顶元素表示当前启用的属性。对于Poly属性，将使用一个泛型的集合；其元素不会被排序。

引擎的通常类型包括：

* 整合引擎负责绘制器和简化器将渲染对象映射到渲染引擎。
* 几何引擎负责分析计算光线和对象的碰撞，对象与对象的碰撞，对象与环境的碰撞，e.g,用于3D交互。

分析引擎负责将渲染对象的使用频率和应用到几何上的属性信息收集。E.g,用于场景编辑。

Handle表

Handle表是引擎的渲染环境的一部分，handle由一个二维数组栈存储。此栈根据他们提供的服务（e.g，绘制，交互，计算，几何简化）和他们处理的目标渲染对象存储(e.g,球体，立方体，材质)。对于每个服务目标对，其栈顶元素表示当前活动handle。以这样的实现方式将多个批次封装到对象中。

对于每个支持的渲染系统，必须实现一个特定的渲染引擎。在整合时，渲染引擎配置是原生几何和属性绘制器的handle表。在场景图，handle可以像属性一样被重写，即每个节点或子图的几何和属性可以被重新配置。

特定渲染引擎可以提供优化实现方法来拓展底层渲染引擎的特定功能。例如:opengl引擎可以重载所有和几何变换的方法，因为opengl可直接访问这些信息。

设计规范

下面列出了泛型场景图的设计原则特性：

* 类似Vision[12]严格将几何和属性分离。另外，提供给泛型渲染环境的属性没有类型限制。
* 严格分离定义和实现。Handle封装了所有类型的应用到渲染对象的算法。因此，几何与属性类十分轻量，且算法也可进行根据优化或者特定应用实现而部分替换。
* 分离结构与内容。场景图提供结构；渲染对象提供内容。两者可相互独立被扩展。
* 场景图可以被理解为参数化的场景内容描述。一个场景图只能被给定渲染引擎使用。泛型场景图的内容可以因为不同的目的被不同的渲染系统解释。
* 几何和属性依赖于环境。例如：一个公告板属性，其朝向总是朝向观察方向，其定义和当前的模型观察变换相关。

4 渲染接口

泛型场景图支持多渲染系统。对于每个系统，我们实现handle用于内嵌几何和属性，这些属性用于系统特定特性和特定渲染引擎。

获取几何

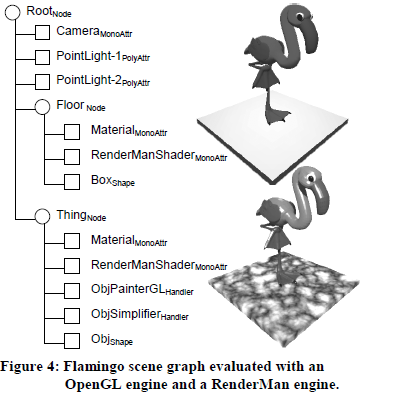
两个类型的handle负责几何：几何绘制器映射几何到渲染系统的原生结构，几何简化器负责解析几何到一系列底层的渲染对象（包括属性）。如果几何没有提供绘制器，引擎将会查找简化器。如果简化器没有，则此几何将被忽略。

渲染引擎在整合阶段安装对应handle。如果handle需要被替换，可将其直接包含到泛型场景图。通常，我们替换handle用于优化渲染算法。例如：一个应用可能需要切换OpenGL1.1与OpenGL1.2两个不同版本实现。

另外，泛型场景图可内重用与当前现存图形代码。例如：考虑opengl解析和显示函数来绘制“obj”格式对象：我们需要包含以下代码：

* 一个几何类表示模型几何对象
* 一个opengl几何绘制类
* 一个简化类负责转换“obj”格式对象到一个渲染对象，其由材质属性和几何内容-这样opengl渲染器马上就可以支持了

因此，此泛型场景图并不会影响现有API，可以做到十分有效率的图形代码。图4举例了一个场景图样例使用“obj”几何类。



获取属性

在渲染系统中属性差异很大。因此，泛型场景图允许存储任何类型的属性。除非handle已经被安装不然其属性不会被加入运算。这样，渲染系统可忽略这些属性。

泛型场景图API定义一个集合的标准属性（e.g,可视效果和空间变换属性），并且提供特定属性用于特定渲染引擎。例如：OpenGL特定属性覆盖大部分opengl功能。对于RenderMan一个shader属性接口编译成RenderMan 着色文件（图4）。

特别是杜宇特定渲染器属性，其包含场景图的常规属性，促进生产高质量动画：因为所有渲染目标系统细节可以表示为泛型场景图所以手动导出场景描述现在不是必须的。对所有内嵌属性，默认转换属性为合理可映射属性（e.g,材质和光源）。类似于几何，属性可以直接运算或进一步解析为更底层的几何渲染对象，其可允许建立复杂的属性集合。例如：一个“重大”属性可以被解析为opengl的一个明确的线风格或材质。而相比于非真实感渲染器可能会直接解释属性。

5 多pass渲染

多pass渲染特性