OSG

- 这都要归功于osg::Group。现在来看一下osg::Group这个类。osg::Group这个类其实就是一个Node的列表，里面可以保存多个Node，然后它本身继承自Node，可以当成一个Node来使用。下面是它的类图，其中NodeList的类型为 typedef std::vector< ref_ptr<Node> > NodeList; 这就是一个std::vector，其中vector里面的元素为ref_ptr<Node>，相当于Node*，只是多了个引用计数。
- 简单的，我们可以把Group看成是Node的集合，但是又可以把Group当成Node来使用。
- 其实这里使用了设计模式中的组合(COMPOSITE)模式。组合模式有两个要点，一是子类继承自父类， 二是子类有父类的指针列表。
  - NodeVisitor设置的遍历模式为TRAVERSE_ALL_CHILDREN（而不是默认的TRAVERSE_NONE），故调用Group::traverse(NodeVisitor):
  - 表示对其每一个孩子进行遍历，递归调用。相当于深搜。
- 遍历节点树： osg::Node类中有两个辅助函数：
- void ascend(NodeVisitor& nv) //虚函数，向上一级节点推进访问器
- void traverse(NodeVisitor& nv) //虚函数，向下一级节点推进访问器
- NodeVisitor的traverse()函数实现如下：
- inline void traverse(Node& node)
- {
- if (_traversalMode == TRAVERSE_PARENTS)
  - node.ascend(*this);
- else if (_traversalMode != TRAVERSE_NONE)
  - node.traverse(*this);
- }
- osg中，当设置某节点的渲染状态时，该状态会赋予当前节点及其子节点；
- 渲染状态的管理通过osg::StateSet管理，可以将其添加到任意的节点（node、group、geode等）和DrawAble类。
- osgviewer->run()会自动设置一个场景漫游器，该漫游器包含了透视投影矩阵、视口大小、屏幕宽高比以及远近裁剪面、摄像机位置等等参数，如果想要修改远近裁剪面应该首先关闭osg的自动判断远近裁剪面的函数
- 隐藏模型与结点开关
- Ø 隐藏模型
- 隐藏模型其实模型仍在渲染当中，因此损耗并未减少，只不过隐藏了而已，隐藏的确不是个什么好操作，但是有时候对小模型确实也很实用。node->setNodeMask可以设置隐藏与显示。
- Ø 节点开关
- 在OSG当中，专门有一个类来负责打开与关闭结点，该类名为osg::Switch，里面有相应的方法来控制它所管理的结点的打开与关闭。
- 两个方法都能控制模型的显示和隐藏，区别在于隐藏模型方法不会让模型在内存中消失，这样对于小的物体频繁的调用会节省一些时间，而对于有些大的模块在用一次以后可能很久再用第二次，这个时候用节点开关可以将模型销毁，再次使用再调入内存，以防止占用更多的资源。
- 移动/旋转/缩放模型

- 移动/旋转/缩放其实都是对矩阵进行操作，在OSG当中，矩阵可以当作一个特殊的结点加入到root当中，而矩阵下也可以另入结点，而加入的结点就会被这个矩阵处理过，比如移动过/旋转过/缩放过。在OSG中控制矩阵的类为osg::MatrixTransform。
- Ø 移动
- osg::Matrix::translate
- Ø 旋转
- osg::Matrix::rotate
- Ø 缩放
- osg::Matrix::scale
- 优化：
  - OSG中对于海量数据场景管理效率有些低效，例如某个应用场景是渲染上百万个站点（点数据），根据OSG的场景树架构，则是将每个点作为一个GeoNode挂载到场景树上，这样导致树最底层节点数十分庞大，因此将所有点压进一个Geometry中，然后将它ADD到geonode作为一个Node.
    - drawcall是CPU对底层图形绘制接口的调用命令GPU执行渲染操作，渲染流程采用流水线实现，CPU和GPU并行工作，它们之间通过命令缓冲区连接，CPU向其中发送渲染命令，GPU接收并执行对应的渲染命令。
    - 这里drawcall影响绘制的原因主要是因为每次绘制时，CPU都需要调用drawcall而每个drawcall都需要很多准备工作，检测渲染状态、提交渲染数据、提交渲染状态。而GPU本身具有很强大的计算能力，可以很快就处理完渲染任务。
    - 当DrawCall过多，CPU就会很多额外开销用于准备工作，CPU本身负载，而这时GPU可能闲置了。
    - 解决DrawCall：过多的DrawCall会造成CPU的性能瓶颈：大量时间消耗在DrawCall准备工作上。很显然的一个优化方向就是：尽量把小的DrawCall合并到一个大的DrawCall中，这就是批处理的思想。下面是一些具体实施方案：
    - 合并的网格会在一次渲染任务中进行绘制，他们的渲染数据，渲染状态和shader都是一样的，因此合并的条件至少是：同材质、同贴图、同shader。最好网格顶点格式也一致。
    - 尽量避免使用大量小的网格，当确实需要时，进行合并。
    - 避免使用过多的材质，尽量共享材质。
    - 合并本身有消耗，因此尽量在编辑器下进行合并
    - 确实需要在运行时合并的，将静态的物体和动态的物体分开合并：静态的合并一次就可以，动态的只要有物体发生变换就要重新合并。
  - OpenGL实现个性化渲染效果；
  - Unity内置了Draw Call Batching技术
  - 拾取问题，RTT或者判断存数据库，然后计算判断；
- QT
  - 优良的跨平台特性
  - 面向对象
  - Qt 的良好封装机制使得 Qt 的模块化程度非常高，可重用性较好，对于用户开发来说是非常 方便的。 Qt 提供了一种称为 signals/slots 的安全类型来替代 callback，这使得各个元件 之间的协同工作变得十分简单。
  - 信号和槽
    - 是一种高级接口，应用于对象之间的通信，他是QT的核心特性，也是QT差别于其他工具包的重要地方。信号和槽是QT自行定义的一种通信机制，他独立于标准的C/C++语言，因此要正确的处理信号和槽，必须借助一个称为moc（Meta

Object Compiler）的QT工具，该工具是个C++预处理程式，他为高层次的事件处理自动生成所需要的附加代码

- 信号
  - 当某个信号对其客户或所有者发生的内部状态发生改动，信号被一个对象发射。只有定义过这个信号的类及其派生类能够发射这个信号。当一个信号被发射时，和其相关联的槽将被即时执行，就象一个正常的函数调用相同。信号-槽机制完全独立于所有GUI事件循环。只有当所有的槽返回以后发射函数（emit）才返回。如果存在多个槽和某个信号相关联，那么，当这个信号被发射时，这些槽将会一个接一个地执行，不过他们执行的顺序将会是随机的、不确定的，我们不能人为地指定哪个先执行、哪个后执行。
- 槽
  - 是普通的C++成员函数，能被正常调用，他们唯一的特别性就是非常多信号能和其相关联。当和其关联的信号被发射时，这个槽就会被调用。槽能有参数，但槽的参数不能有缺省值。
  - 既然槽是普通的成员函数，因此和其他的函数相同，他们也有存取权限。槽的存取权限决定了谁能够和其相关联。同普通的C++成员函数相同，槽函数也分为三种类型，即public slots、private slots和protected slots。
- 元对象编译器moc（meta object compiler）
  - 对C++文件中的类声明进行分析并产生用于初始化元对象的C++代码，元对象包含全部信号和槽的名字及指向这些函数的指针。 moc 读C++源文件，如果发现有Q_OBJECT宏声明的类，他就会生成另外一个C++源文件，这个新生成的文件中包含有该类的元对象代码。例如，假设我们有一个头文件mysignal.h，在这个文件中包含有信号或槽的声明，那么在编译之前 moc 工具就会根据该文件自动生成一个名为mysignal.moc.h的C++源文件并将其提交给编译器；类似地，对应于mysignal.cpp文件moc 工具将自动生成一个名为mysignal.moc.cpp文件提交给编译器。
  - 元对象代码是signal/slot机制所必须的。用moc产生的C++源文件必须和类实现一起进行编译和连接，或用#include语句将其包含到类的源文件中。moc并不扩展#include或#define宏定义,他只是简单的跳过所遇见的所有预处理指令。

- 物体坐标系，世界坐标系，相机坐标系，投影坐标系以及屏幕坐标系.我要讨论的就是这些坐标系间的转换
- 计算机3D图形学最最最基本的目标就是:将构建好的3D物体显示在2D屏幕坐标上
- OpenSceneGraph(OSG) 是一个开放源码，跨平台的图形开发包，它为诸如飞行器仿真，游戏，虚拟现实，科学计算可视化这样的高性能图形应用程序开发而设计。它基于场景图的概念，它提供一个在 OpenGL 之上的面向对象的框架，从而能把开发者从实现和优化底层图形的调用中解脱出来，并且它为图形应用程序的快速开发提供很多附加的实用工具。
- 读者需要具备相当的图形学基础理论和openGL程序开发的知识储备，需要了解openGL中的顶点和顶点数组、渲染状态、纹理映射、显示列表、VBO等知识。osg中应用了多种现代化图形学理论的概念和实现方法，包括了多线程渲染技术、动态数据调度技术、纹理烘焙技术、场景裁剪技术、动画技术、模块化的插件设计技术等。
- 场景图形的关键在于场景节点树的构建；而对于经验丰富的OPenGL开发者而言，真正影响OpenGL绘制管道工作效率的并非数据的组织结构，而是渲染状态的组织结构和切换频率。OSG采用构建状态树的方法，有效地减少了状态切换的次数和冗余的可能性，保证了渲染流程的顺畅执行，其效率因而高过大部分同类型的渲染引擎。
- 场景图形树结构的顶部是一个根节点，从根节点向下延伸，各个组节点中均包含了几何信息和用于控制其外观的渲染状态信息。根节点和各个组节点都可以有零个（实际上是没有

执行任何操作）或多个子成员。在场景图形的最底部，各个叶节点包含了构成场景中物体的实际几何信息

- Osg程序使用组节点来组织和排列场景中的几何体
- 场景图形通常包含了多种类型的节点以执行各种各样的用户功能，例如开关节点可以设置其子节点可用或不可用，细节层次节点（LOD）可以根据观察者的距离调用不同的子节点，变换节点可以改变子节点几何体的坐标变换状态
- 场景图形特征：
  - 1. 提供底层渲染API中具备的几何信息和状态管理功能之外，还兼备以下的附加特征和功能：
  - 2. 空间结构：
  - 3.场景拣选，投影面剔除和隐藏面剔除。
  - 4.细节层次：
  - 5.透明
  - 6.状态改动最少化
  - 7.文件I/O
  - 8.更多高性能函数：全特征文字支持，渲染特效的支持，渲染优化，3d模型文件读写支持，跨平台输入渲染及显示设备的访问.
- 场景图形渲染方式：
- 三种遍历操作
  - 1. 更新
  - 2.拣选
  - 3. 绘制
- 1.osg库：包含了用于创建场景图形的场景图形节点类，用作向量和矩阵运算的类，几何体类，以及用于描述和管理渲染状态的类，3d图形程序所需的典型功能类，例如命令行参数解析，动画路径管理，以及错误和警告信息类。
- 2.osgUtil库：osg工具库包含的类和函数，可以用于场景图形及其内容的操作，场景图形数据统计和优化，以及渲染器的创建。它还包括了几何操作的类，例如delaunay三角面片化，三角面片条带化，纹理坐标生成等。
- 3.osgDB 库：建立和渲染3d数据库的类和函数：允许用户程序加载，使用和写入3d数据库，它采用插件管理的架构。osgDB维护插件的信息注册表，并负责检查将要 被载入的osg插件接口的合法性。OsgDB：：DatabasePager实现应用程序从文件中读取各部分数据库信息时，在不干扰当前渲染的前提下以后 台线程的方式进行。
- 4.osgViewer库：包含了场景中视口及可视化内容的管理类。定义了一些视口类，可以将osg集成到许多视窗设计工具中，如MFC，win32等。
- 涉及到osg的内存引用计数策略。
  - Referenced：所有场景图形节点和osg的许多其他对象的基类。它实现了一个用于跟踪内存使用情况的引用计数。如果某个继承自Referenced类的对象引用计数数值到达0，那么系统将自动调用其析构函数并清理为此对象分配的内存。
  - ref_ptr<>：模板类ref_ptr<>为其模板内容定义了一个智能指针。模板内容必须继承自Referenced。
  - Object：纯虚类，一切需要I/O支持，拷贝和引用计数的对象的基类。
  - Notify：osg库提供了一系列的控制调试，警告和错误输出的函数。用户可以通过指定一个来自NotifySeverity枚举量的数值，设定输出的信息量。
- 交运算：osgUtil库
  - 通过提供大量用于场景图形交运算，使用如下类可以获得场景图形中被拾取部分的信息：

    - Intersector：纯虚类，定义了相交测试的接口。执行相交测试时，应用程序将继承自Intersector的某个类实例化，传递给IntersectionVisitor的实例，并随后请求该实例返回数据以获取交运算的结果。
    - IntersectionVisitor：IntersectionVisitor搜索场景图形中与指定几何体相交的节点，而最后的测试工作由Intersector继承类完成。
    - LineSegmentIntersector：继承自Intersector，用于检测指定线段和场景图形之间的相交情况，并向程序提供查询相交测试结果的函数。
    - PolytopeIntersector：与LineSegmentIntersector类似，该类用于检测由一系列平面构成的多面体的相交情况。当用户点击鼠标，希望拾取到鼠标位置附近的封闭多面体区域时，PolytopeIntersector类尤其有效。
    - PlaneIntersector：与LineSegmentIntersector类似，用于检测由一系列平面构成的平面的相交情况。
- 优化：osgUtil库
    - Optimizer：用于优化场景图形。
    - Statistics和StatsVisitor：StatsVisitor返回一个场景图形中节点的总数和类型，Statistics返回渲染几何体的总数和类型。
    - 几何体操作：osgUtil库
    - Simplifier：使用此类用来减少Geometry对象中几何体的数目，这有助于低细节层次的自动生成。
    - Tessellator：根据一组顶点的列表，生成由前述列表描述的多边形，即一个osg：：PrimitiveSet
    - DelaunayTriangulator：实现了Delaunay三角网格化运算，根据一组顶点的集合生成一系列的三角形。
    - TriStripVisitor：遍历场景图图形并将多边形图元转化成三角形和四边形条带。
    - SmoothingVisitor：SmoothingVisitor可生成顶点法线，也就是所有共享此顶点的面的法线平均值。
    - 纹理贴图生成：包含了建立反射贴图，中途向量贴图，高光贴图。
- LOD（level of detail）
    - 是指根据物体模型的结点在显示环境中所处的位置和重要度，决定物体渲染的资源分配，降低非重要物体的面数和细节度，从而获得高效率的渲染运算。在OSG的场景结点组织结构中，专门提供了场景结点osg::LOD来表达不同的细节层次模型。其中，osg::LOD结点作为父节点，每个子节点作为一个细节层次，设置不同的视域，在不同的视域下显示相应的子节点。
- 数据分页
    - 在城市三维场景中可以采用数据分页的方式进行动态调度。这里“分页”的意思是随着视口范围的变化，场景只加载和渲染当前视口范围内数据，并将离开视口范围内的数据清除内存（可以设定不同的数据卸载策略），不再渲染。保证内存中只有有限的数据量，场景的每一帧也只有有限的数据被送到图形渲染管道，从而提高渲染性能。
- 动态调度
    - OSG源代码中提供PagedLOD来进行模型的动态调度。在不同的视域下，PagedLOD动态读取不同细节层次的结点模型，实现了分页LOD显示。OSG内部采用osgDB::DatabasePager类来管理场景结点的动态调度，场景循环每一帧的时候，会将一段时间内不在当前视图范围内的场景子树卸载掉，并加载新进入到当前视图范围的新场景子树。OSG采用了多线程的方式来完成上述工作。

幕布 - 思维概要整理工具