# OpenSceneGraph基本渲染理论

Leandro Motta Barros

Start: August 17th 02005

译：FreeSouth

# INDEX

# 1. 基础

在我们讨论 Open Scene Graph (OSG) 之前，我们先来花一些时间来寻找一些关于个别常见有趣问题的相关线索。

## 1.1 什么是场景图

就如它的名字所说的一样，场景图是一个用于组织图形图像数据结构在计算机中显示的应用程序。一个通常的想法是场景往往被分为很多的部分，而出于某种通常的目的，这些部分最终都会被组合到一场，所以，场景图就是一个代表每个结点都可被分割与重组的图。再定义的严格一些，场景图就是一个非循环的图，所以它同时也体现出结点与结点之间的等级关系。

猜想一下，如果您渲染的场景中包含一辆卡车和一条路况还可以的马路。使用场景图的思想来体现该组织如图 1.1 所示：

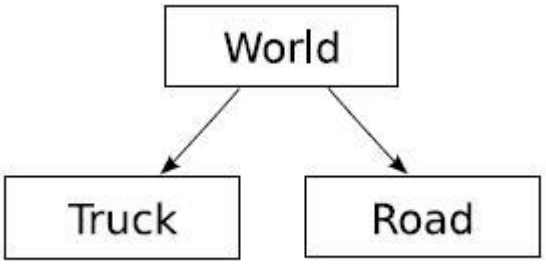


图 1.1

假如你按照上面的图例的渲染一个场景，然而卡车却停在了你不想让它出现的位置。从某种常理上讲，你一定会把它从某处移到合适的位置，幸运的是场景图的结点并不代表几何关系。在这种情况下，你可以申请一个结点，该结点用来代表移动，在场景图中体现为如图 1.2 所示：

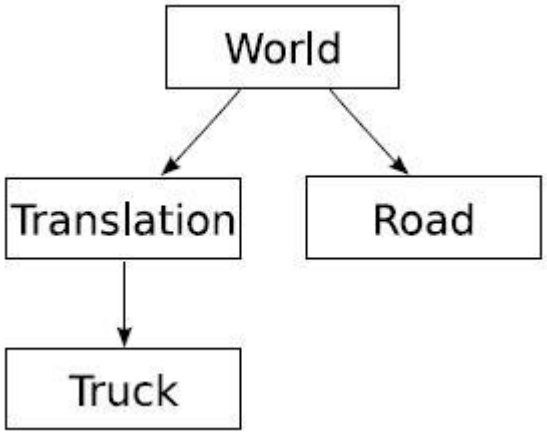


图 1.2：一个场景图，由一条路以及移动后的卡车组成

或许你一直在疑惑为什么看起来明明是树而场景图要称做一种图呢？问得好，确实例子中体现的是这种关系，然而事情总非像例子中所述的那样。现在让我们往图中添加两个盒子，一个在卡车上，另一个在马路上。同样，为了能够使这两个盒子显示在正确的位置，都需要在它们前面加上移动的几何结点。显而易见，在卡车上的盒子可以与卡车同时使用一个移动结点，因此当我们移动卡车时，盒子也会跟着移动。而事实上，因为两个盒子非常相似，故不需要为它们每个盒子都创建一个结点，可以使用一个结点而产生两次引用来达到这个目的，如图 1.3 所示。在渲染其间，盒子将会被访问两次，但是分享的是同一片内存区域，因此盒子在内存中只被加载了一次。

当然场景图的功能不只是单单处理这些简单的问题，然而对于讲清楚场景图的基本概念即：场景图是什么？已经足够了。

故我们现在可以把更多的时间与精力花在第二个基本且重要的问题上。

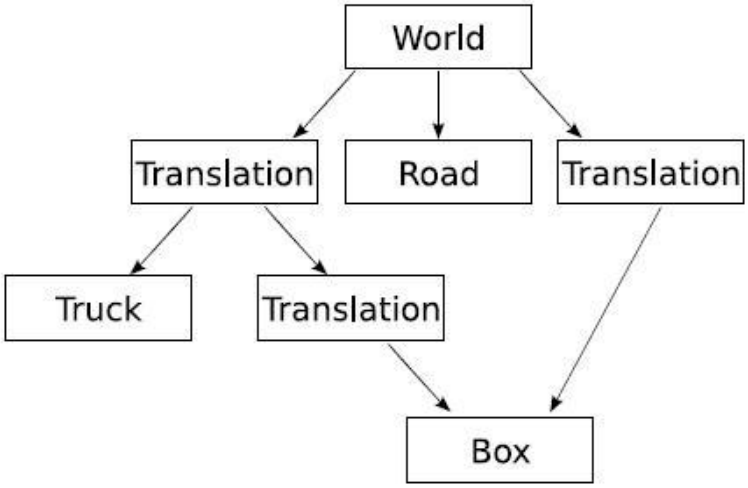


图 1.3：一个场景图，由一条路，一个卡车以及一对盒子组成

## 1.2 第二个问题：谁关心场景图？

任何一个人，只要他需要在计算机中组织他所要渲染的场景，而这些场景在现实中又是非常复杂的，他首先要面临的是效率问题，故明思议：需要在计算机中合理组织，高效显示的用户，关心场景图。

## 1.3 一些与 OSG 相关的问题

根椐这一点，我们将围绕“一般”的场景图展开讨论。从现在开始，所有的例子都将会使用到 OSG 场景图，来代替传统的移动结点，我们将会使用真的在 OSG 中继承图中的类来做这些现实的事情。

每一个结点在 OSG 中使用类 osg::Node 来表示，非常的方便快捷。尽管技术上可能需要，但是基于 osg::Node 专门的示例并不是很多。而我们真正的感觉兴趣的是一些基于 osg::Node 的子类，他们具有十分强大的功能。在这一章，我们将会讲到三个子类，它们分别是：osg::Geode, osg::Group, 以及 osg::PositionAttitudeTransform。

Osg::Drawable 类的实例可以用来在 OSG 中渲染一些数据。但是 osg::Drawable 本身并不是结点，所以我们不能直接把它加入到场景图当中去，故，使用一个几何结点 osg::Geode 来代替它变得十分必要。

并不是所有的结点在 OSG 的场景图中都有父结点用来绑定它们。事实上，我们只需要在类 osg::Group 或其子类的实例下面增加新的结点做为他们的孩子。

在 OSG 中，只需要使用 osg::Geode 以及 一个 osg::Group 就可以创建如图 1.1 所示的场景图。结点如图 1.4 所示：

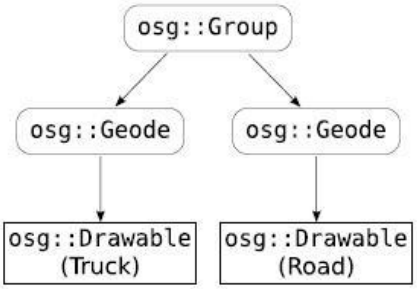


图 1.4：一个 OSG 场景图，由一条路、一个卡车组成。从 osg::Node 派生而来的实例，在图中椭圆形框中所示。而 osg::Drawable 则使用矩形框来表示。

这并不是使用 OSG 中的类来完成图 1.1 中所示功能的唯一方式。比之一个 osg::Drawable 来绑定一个 osg::Geode 而言，OSG 实现图 1.1 的另一种方式如图 1.5 所示：

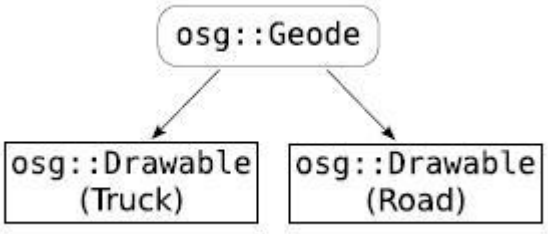


图　1.5：在 OSG 中如图 1.4 所完成的同样功能场景图功能的另一种方法

在图 1.4 与图 1.5 中，实现了同样的功能，即实现图 1.1 所示的功能，同时也会面临同样的问题：卡车可能位于错误的位置，解决方案也和前面一样：移动卡车。在 OSG 中，移动一个结点最简单明了的方式是使用类 osg::PositionAttitudeTransform 来处理。一个 osg::PositionAttitudeTransform 类必须不但与移动相关，还必须与缩放以及属性相关。尽管做的事情可能不尽相同，但是与 OSG 中此类做同样事情也可以使用 OpenGL 中的 glTranslate(), glRotate(), 以及 glScale(函数来完成。使用 OSG 来完成图 1.2 功能如图 1.6 所示：

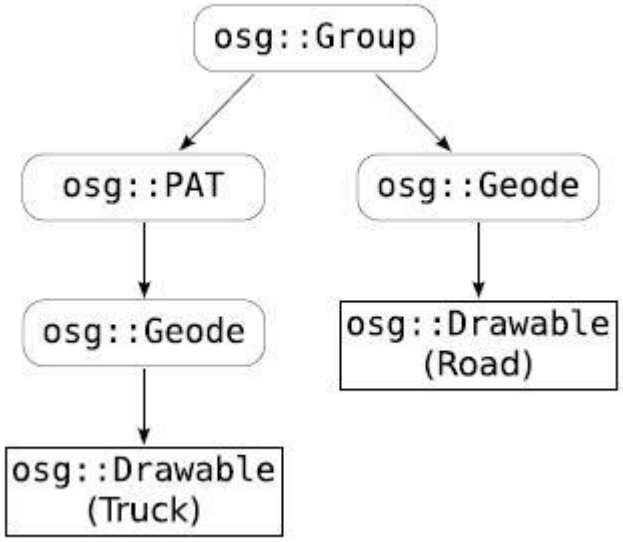


图 1.6：一个 OSG 场景图，由一条路，一个移动过的卡车组成。由于某种编辑方面的原因，　类 osg::PositionAttitudeTransform 被简写成了 osg::PAT

为了保持完整性，图 1.7 是使用 OSG 的方式来完成如图 1.3 所示的“一般”场景图。

# 1.4　超级指针和 OSG

*Save the whales. Feed the hungry. Free the mallocs.*

*— fortune(6)*

令人悲观的是，事实上有相当一部分的 C++使用者对于使用超级指针一无所知或者相当畏惧。由于在现实中越来越广泛的使用超级指针，故值得我们花一些时间来对它大加解释一番。不要惧怕跳过这一章，如果超级指针听起来就像希腊一样熟悉（当然，你并不是希腊，故继续吧）。

让我们来以一个定义开讲：一个*资源*必须在它被使用之前被申请，而在它使用之后不久被释放，也许我们有很多的应用程序在使用同一片资源，一个不需要了而另一个却还需要使用这片资源。这两种情况都涉及到页面（在打开前必须处于关闭状态）和数据的移动（在移动前可能被加锁或做一些包裹处理）。在 OpenGL 当中，也同样有一些关于申请与释放资源的例子（一个示

例是纹理函数的名称被命名为 glGenTextures()， 它就必须被函数 glDeleteTextures()释放）。

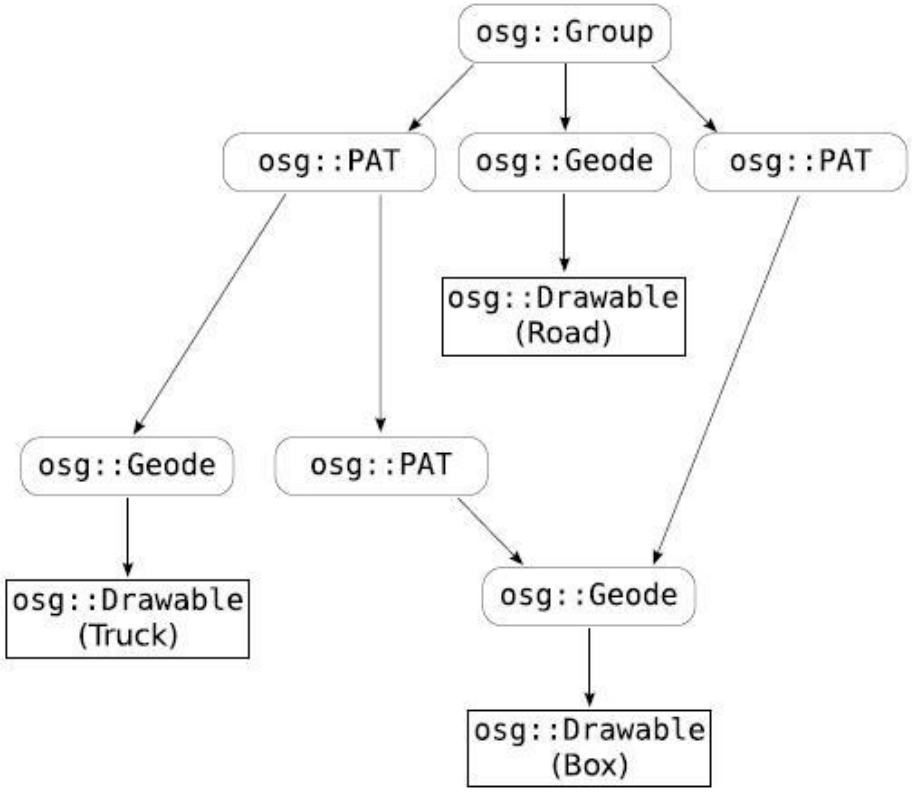


图 1.7： 一个 OSG 场景，由一条路、一个卡车以及一对盒子组成。

完美义者一般都设想以下理想情况：存在这样一些程序员，他们相信他们仅仅用手写代码就可以完成在任何时候需要释放内存时就释放该释放的内存，以及从来不会忘记用手写这些代码。这种设想往往会直接导致内存泄漏。好消息是：在有一些规则的制约之下，这种泄漏事故完全可以交由 C++编译器去检查处理，这种做法往往比我们肆意妄为要可靠的多。

在 C++中资源管理的基本思想十分有必要在这里讨论一下，但是比较细致入微的讨论则超出了本文的能力范围。说起范围，“自动变量”的范围（就是变量在栈中分配）在 C++的资源管理当中占有十分重要的地位：程序设计语言规则保证当一个对象被踢出某个范围时析构该对象的操作就必须被执行或是自动被执行。这是怎样避免内存泄漏的呢？花一点时间来看一下下面的类：

```
class ThingWrapper
{
public:
   ThingWrapper() { handle_ = AllocateThing(); }
   ~ThingWrapper() { DeallocateThing (handle_); }
   ThingHandle& get() { return handle_; }
private:
   ThingHandle handle_;
};
```

它分配了一些**事物**在构造函数中，又释放掉这些事物在**析构**函数中。因此，我们可以在我们需要的时候这样使用这些**事物**：

```
   ThingWrapper thing;
   UseThing (thing.get());
```

示例中演示了一个**事物扩展**如何分配一个**事物**（在**事物扩展**的构造函数当中）。但是最美妙的事情是当这些**事物**不被使用时它们会被自动的释放掉，析构函数在这时会被自动的执行。这就是内存自动管理的机制。

这个**事物扩展**类是使用 C++程序设计语言技术当中一种被叫做“资源使用即初始化”(RAII)的技术。一个超级指针仅仅是使用 RAII 自动管理堆内存的小类。类似于**事物扩展**类，我们可以假定一些如 AllocateThing()和 DeallocateThing()的函数来完成这些

功能，超级指针就是一个典型的在构造函数中分配内存，使用 C++的标准操作在析构函数中来删除内存的技术。

在**事物扩展**的事例当中，事物拥有者使用 AllocateTing()来分配该事物， 然后拥有者便有责任来管理他们的分配与释放操作。在 OSG 当中，有一些稍微有些复杂的细节来实现这些操作，也就是一些事物可能有多个拥有者。比如，在场景图 1.7 所示当中，类 osg::geode 中含有 box 绑定到两个父结点身上，到底哪个负责他们的分配与释放操作呢。

在这种情况之下，只要还有一个引用指向该资源，该资源就不应该被释放。所以，大多数的 OSG 对象都有一个内部的计数器用来计算有多少个指针指向它。当没有指针指向该资源（同样，可以称做对象）时，它的计数器会变为 0，当计算器为 0 时就可以放掉对象。

幸运的是，我们程序员不需要手工的来维护这些引用计数：这就是超级指针为什么存在的原因。在 OSG 当中超级指针被一个称为 osg::ref_ptr<>的类模版来实现。无论在什么时候，只要该对象被引用，则在 osg::ref_ptr<>当中的计数器会自动增加， 通过这种方式实现了资源的自动管理，当不需要使用它时，它会在不久后释放，在申请它时会自动分配空间。

下面这个例子会示例如何使用 OSG 的超级指针，例子中有一些代码如下所示：

SmartPointers.cpp

```
1 #include <cstdlib>
2 #include <iostream>
3 #include <osg/Geode>
4 #include <osg/Group>
5
6 void MayThrow()
7 {
8 if (rand() % 2)
9 throw "Aaaargh!";
10 }
11
12 int main()
13 {
14 try
15 {
16 srand(time(0));
17 osg::ref_ptr<osg::Group> group (new osg::Group());
18
19 // This is OK, albeit a little verbose.
20 osg::ref_ptr<osg::Geode> aGeode (new osg::Geode());
21 MayThrow();
22 group->addChild (aGeode.get());
23
24 // This is quite safe, too.
25 group->addChild (new osg::Geode());
26
27 // This is dangerous! Don't do this!
28 osg::Geode* anotherGeode = new osg::Geode();
29 MayThrow();
30 group->addChild (anotherGeode);
31
32 // Say goodbye
```

```
33 std::cout << "Oh, fortunate one. No exceptions, no leaks.\n";
34 }
35 catch (...)
36 {
37 std::cerr << "'anotherGeode' possibly leaked!\n";
38 }
39 }
```

关于上面示例，我们值得注意的是一件事情：我们 1.3 节所示的关于如何组织场景图形成了初步的认识。（现在，为了满足大家的好奇心，下一节我们将更进一步来谈论大家都感兴趣的事情）。该示例给出了安全使用 OSG 超级指针的两个方法以及避免使用一种危险的方法来使用它们：

- 第 20 行到 22 行，示例了一个比较安全的方式来使用超级指针，一个 osg::ref_prt<>（叫做 aGeode）在第 20 行被初始化为 osg::Geode(资源)。在这一点上，关于 geode 的引用计数器在堆内会被置 1（因为 osg::ref_ptr<>中只分配了一个 Geode,故可以命令为 aGeode）。

  在稍后的第 22 行，geode 被当做一个孩子加入到一个组当中。这个操作一旦执行结束，组对 geode 的引用计数就会增加 1，变为 2。

  现在，有哪些比较烂的事情发生了呢？如果我们在第 21 行调用 MayThrow()之后，会发生干什么事情呢？好的，aGeode 将会超出范围，然后被析构掉。然而，在它的计数减至 0 之后，它仍会合适的处理该 geode,这时也没有内存泄漏。
- 第 25 行或多或少的与前面几行做了相同的事情，不同的是 geode 被一个简单的 new 语句分配以及只用了一小行便添加到组当中。这是相当安全的，因为在彼此之间发生了如此多的坏事情（毕业，这里也没有彼此）。
- 坏的、错误的、危险的以及应该被责备的方式来管理内存是从第 28 到第 30 行的做法。它看起来和第一种是一样的，但是 geode 是被 new 分配的且存储在一个普通的指针当中。如果 MayThrow()在第 29 行被抛出，没有人会调用 Delete,这时 geode 就发生了泄漏。

  这里有另外一件事情颇值得一提：osg::Referenced 的构造函数有时候并不都是公有的，所以你有可能不能说直接的删除 anotherGeode，因为该类是从 osg::Referenced 派生而来。从 osg::Referenced 派生而来的类实例(像 osg::Geode)采用 osg::ref_ptr<>进行管理只是简单的意味着该类会自动的管理它的内存空间。

  所以，做正确的事情，永远不要使用第三种方法来写代码。

# 2. 两个 3D Viewer

在这一章，我们将创建一些 OSG 程序来在屏幕上显示一些数据。这两个 3D 模型 viewer 都表明了很多的概念以及基本理论。

## 2.1 一个最简单的 viewer

第一个 viewer 是一个非常简单的。基本上而言，它所做的所有事情就是从硬盘上读入数据，类似于使用命令行一样读入数据，然后在屏幕上渲染一下。不卖关子，现在来看一下实际代码：

VerySimpleViewer.cpp

```
1 #include <iostream>
2 #include <osgDB/ReadFile>
3 #include <osgProducer/Viewer>
4
5 int main (int argc, char* argv[])
6 {
7 // Check command-line parameters
8 if (argc != 2)
9 {
10 std::cerr << "Usage: " << argv[0] << " <model file>\n";
11 exit (1);
12 }
13
14 // Create a Producer-based viewer
15 osgProducer::Viewer viewer;
16 viewer.setUpViewer (osgProducer::Viewer::STANDARD_SETTINGS);
17
18 // Load the model
19 osg::ref_ptr<osg::Node> loadedModel = osgDB::readNodeFile(argv[1]);
20
21 if (!loadedModel)
22 {
23 std::cerr << "Problem opening '" << argv[1] << "'\n";
24 exit (1);
25 }
26
27 viewer.setSceneData (loadedModel.get());
28
29 // Enter rendering loop
30 viewer.realize();
31
32 while (!viewer.done())
33 {
```

```
34 // Wait for all cull and draw threads to complete.
35 viewer.sync();
36
37 // Update the scene by traversing it with the the update visitor which
38 // will call all node update callbacks and animations.
39 viewer.update();
40
41 // Fire off the cull and draw traversals of the scene.
42 viewer.frame();
43 }
44
45 // Wait for all cull and draw threads to complete before exit.
46 viewer.sync();
47 }
```

这个例子非常的简洁，但是非常有说头。第一：注意一下，OSG 可以像使用 OPENGL 一样，是跨平台的。因此使用 OPENGL 图形上下文来创建一个窗口不是 OSG 所管的事情。这是第一个示例，在这个示例当中（以及在接下来的其它示例当中）这个工作将会被一个名为 OpenProducer(或简单的称为 Producer)的库来完成。Producer 是一个高校的、合适的、稳定的以及与 OSG 结合非常易用的开源库。

所以我们的第一个例子在第 15 行是一个基于 Producer 的 viewer。在第 16 行使用它的标准操作建立 viewer，这个标准操作中包含了一些非常有用的特征与功能。

OSG 知道如何读（写）一些 3D 模型以及图像，与之相关的函数和操作都封装在名字空间 osgDB 之下。第 19 行使用了这些函数，osgDB::readNodeFile()函数含有一个参数，表示模型的路径，返回了一个 osg::Node 型的指针。这个返回值包含了所有的 3D 信息，包括点、多边形、法线以及纹理映射。

osgDB::readNodeFile()返回的结点被加入到场景图当中。事实上，在这个简单的示例当中，它就是所有的场景图：注意到第 27 行意思是我们告诉 viewer 我们将要使用这个 view，而且将会自动添加到结点到这个 viewer 当中调出渲染。

在第 30 行调用了 realizes（实现）viewer 的窗口，因此，它使用了 OpenGL 上下文。如大多数程序一样，这个小程序干得相当漂亮。从这行开始，我们需要调用一些额外的操作来使程序保待运行状态。这个循环在第 32 行一直延续到第 43 行结束，是一个典型的基于 OSG 和 Producer 的主循环。最后，在第 46 行简单的等待别的线程完成任务之后开始退出。

## 2.2 另一个简单的（可能有 BUG）的 3D Viewer

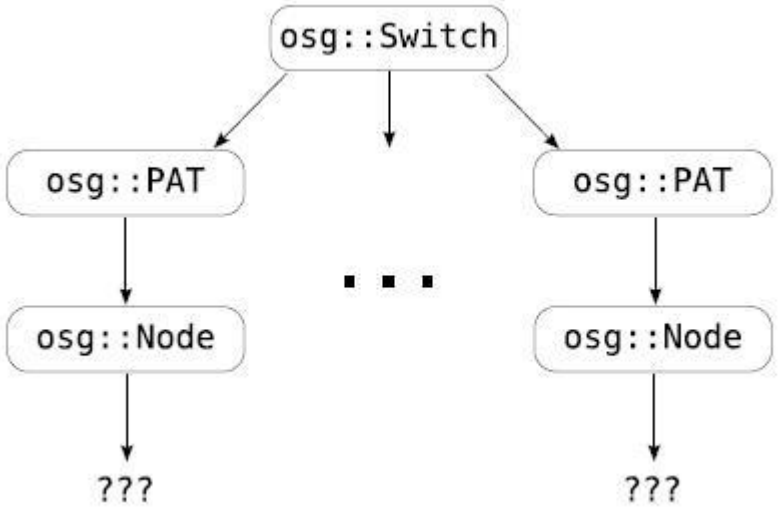


图 2.1： OSG 场景图使用了“简单但是有点小 BUG 的 Viewer”.为了一些排版上面的原因，osg::PositionAttitudeTransform 被简写为 osg::PAT

SimpleAndBuggyViewer.cpp

```
1 #include <iostream>
2 #include <osg/PositionAttitudeTransform>
3 #include <osg/Switch>
4 #include <osgDB/ReadFile>
5 #include <osgGA/GUIEventHandler>
6 #include <osgProducer/Viewer>
7
8 osg::ref_ptr<osg::Switch> TheSwitch;
9 unsigned CurrentModel = 0;
10
11 class ViewerEventHandler: public osgGA::GUIEventHandler
12 {
13 public:
14 virtual bool handle (const osgGA::GUIEventAdapter& ea,
15 osgGA::GUIActionAdapter&)
16 {
17 if (ea.getEventType() == osgGA::GUIEventAdapter::KEYUP)
18 {
19 switch (ea.getKey())
20 {
21 // Left key: select previous model
22 case osgGA::GUIEventAdapter::KEY_Left:
23 if (CurrentModel == 0)
24 CurrentModel = TheSwitch->getNumChildren() - 1;
25 else
```

```
26 --CurrentModel;
27
28 TheSwitch->setSingleChildOn (CurrentModel);
29
30 return true;
31
32 // Right key: select next model
33 case osgGA::GUIEventAdapter::KEY_Right:
34 if (CurrentModel == TheSwitch->getNumChildren() - 1)
35 CurrentModel = 0;
36 else
37 ++CurrentModel;
38
39 TheSwitch->setSingleChildOn (CurrentModel);
40
41 return true;
42
43 // Up key: increase the current model scale
44 case osgGA::GUIEventAdapter::KEY_Up:
45 {
46 osg::ref_ptr<osg::PositionAttitudeTransform> pat =
47 dynamic_cast<osg::PositionAttitudeTransform*>(
48 TheSwitch->getChild (CurrentModel));
49 pat->setScale (pat->getScale() * 1.1);
50
51 return true;
52 }
53
54 // Down key: decrease the current model scale
55 case osgGA::GUIEventAdapter::KEY_Down:
56 {
57 osg::ref_ptr<osg::PositionAttitudeTransform> pat =
58 dynamic_cast<osg::PositionAttitudeTransform*>(
59 TheSwitch->getChild (CurrentModel));
60 pat->setScale (pat->getScale() / 1.1);
61 return true;
62 }
63
64 // Don't handle other keys
65 default:
66 return false;
67 }
68 }
69 else
```

```
70 return false;
71 }
72 };
73
74
75 int main (int argc, char* argv[])
76 {
77 // Check command-line parameters
78 if (argc < 2)
79 {
80 std::cerr << "Usage: " << argv[0]
81 << " <model file> [ <model file> ... ]\n";
82 exit (1);
83 }
84
85 // Create a Producer-based viewer
86 osgProducer::Viewer viewer;
87 viewer.setUpViewer (osgProducer::Viewer::STANDARD_SETTINGS);
88
89 // Create the event handler and attach it to the viewer
90 osg::ref_ptr<osgGA::GUIEventHandler> eh (new ViewerEventHandler());
91 viewer.getEventHandlerList().push_front (eh);
92
93 // Construct the scene graph
94 TheSwitch = osg::ref_ptr<osg::Switch> (new osg::Switch());
95
96 for (int i = 1; i < argc; ++i)
97 {
98 osg::ref_ptr<osg::Node> loadedNode = osgDB::readNodeFile(argv[i]);
99 if (!loadedNode)
100 std::cerr << "Problem opening '" << argv[i] << "'\n";
101 else
102 {
103 osg::ref_ptr<osg::PositionAttitudeTransform> pat(
104 new osg::PositionAttitudeTransform());
105 pat->addChild (loadedNode.get());
106 TheSwitch->addChild (pat.get());
107 }
108 }
109
110 // Ensure that we have at least on model before going on
111 if (TheSwitch->getNumChildren() == 0)
112 {
113 std::cerr << "No 3D model was loaded. Aborting...\n";
```

```
114 exit (1);
115 }
116
117 viewer.setSceneData (TheSwitch.get());
118
119 TheSwitch->setSingleChildOn (0);
120
121 // Enter rendering loop
122 viewer.realize();
123
124 while (!viewer.done())
125 {
126 viewer.sync();
127 viewer.update();
128 viewer.frame();
129 }
130
131 // Wait for all cull and draw threads to complete before exit
132 viewer.sync();
133 }
```

# 3. 进入 StateSets

在 OSG 当中这是一个非常非常重要的类，可惜的是我们现在在来得及提起它：osg::StateSet.它如此重要以至于我非要用单独的一单来讲它才行。但是，为了让大家知道 osg::StateSets 的重要性，我必须来讲解一下 OpenGL 是如何工作的。OpenGL 的背景知道我们会在下面一节进行讨论。如果你已经对于现在的内容非常累了，你可以跳过题为"OpenGL-状态机"一节直接跳到第 3.2 节，否则保持状态，继续下一节。

## 3.1 OpenGL-状态机

OpenGL 可以被粗略的看做把一些点移动后转化为像素显示在屏幕上。本质上来讲，程序可以这样理解："Hey，OpenGL 请为我处理这一系列 3D 空点中的点点"。然后，短暂的处理之后，它答道："完成，结果就显示在你的屏幕上了"。这不是一个百分百精确描述 OpenGL 如此工作的语言，但是对于本章内容的理解，已经足够了。

所以，OpenGL 所做的工作就是把一些点变为像素显示在屏幕上。猜想我们通过了四个点让 OpenGL 来处理。名称我们暂时命名为：v1, v2, v3 以及 v4.像素之间是如何组织的呢？两条线段（v1-v2 和 v3-v4）?或许三条线段（v1-v2, v2-v3 和 v3-v4）?为什么没有其它的呢？

从另一个角度来讲，我们换种新思维，为什么不为这些像素上色呢？上什么色呢？是不是有一些色彩光源照在了这些点上呢？如果有的话，总共有多少个光源呢？他们分别位于哪里呢？是否有纹理呢？

我们还准备问一些关于时间与空间理论的大问题，算了，就此为止吧。一个非常重要的问题，非常值得你注意的问题是：尽管 OpenGL 是一个能把点转换为像素显示的机器，但是我们有很多种的方式来做这个转换。以及，或许，我们必须配置 OpenGL 以使它可以做我们想让它做的事情，但是我们该如何配置这个机器呢？

分割以及个个击破处理之，这里有数以吨计的设置，而且它们是没有交叉的。这就意味着，我们可以改变，比如，光照设置而不改变纹理设置。当然这里有一个这些设置的交集，比如，一般而言，光照与纹理在显示真实性方面相互合作，共同作用。更重要的思想是：他们可以独立的设置。

从现在开始，我们把这些 OpenGL 的设置起一些合适的名字：属性以及模式（现在属性与模式的曲别并不是十分的重要）。所以 OpenGL 有一个属性和模式的集合，以及这个集合可以用来精确的定义如何设置 OpenGL 的属性和模式。但是人们马上就发现写类似于“设置属性与模式”的字串是一件非常麻烦的事情，故 State 应运而生。

这个解释只是一个标题而已，也就是仅仅对 State 这个单词进行的解释。OpenGL 可以被看做一个状态机器，所有状态机器的细节都是用来定义顶点是如何被转化为像素的。如果我们绘制一个绿的物体，而现在我们又想绘制一个蓝的物体，我们就必须改变 OpenGL 状态来实现这个操作。如果我们绘制这些物体时希望有光线，或者没有光线。我们也必须改变 OpenGL 状态机器设置。同样，在纹理映射和众多场合，状态机都是十分重要的概念。

显尔易见的问题是：当我们使用 OSG 时，如何设置和改变 OpenGL 状态？这个问题我们在下一节回答。

## 3.2 OSG 和 OpenGL 状态

## 3.3 一个简单的（无 BUG）3D Viewer

SimpleAndBuglessViewer.cpp

```
1 #include <iostream>
2 #include <osg/PositionAttitudeTransform>
3 #include <osg/Switch>
4 #include <osgDB/ReadFile>
5 #include <osgGA/GUIEventHandler>
6 #include <osgProducer/Viewer>
7
8 osg::ref_ptr<osg::Switch> TheSwitch;
9 unsigned CurrentModel = 0;
10
11 class ViewerEventHandler: public osgGA::GUIEventHandler
12 {
13 public:
14 virtual bool handle (const osgGA::GUIEventAdapter& ea,
15 osgGA::GUIActionAdapter&)
16 {
17 if (ea.getEventType() == osgGA::GUIEventAdapter::KEYUP)
18 {
19 switch (ea.getKey())
20 {
21 // Left key: select previous model
22 case osgGA::GUIEventAdapter::KEY_Left:
23 if (CurrentModel == 0)
24 CurrentModel = TheSwitch->getNumChildren() - 1;
25 else
26 --CurrentModel;
```

```
27
28 TheSwitch->setSingleChildOn (CurrentModel);
29
30 return true;
31
32 // Right key: select next model
33 case osgGA::GUIEventAdapter::KEY_Right:
34 if (CurrentModel == TheSwitch->getNumChildren() - 1)
35 CurrentModel = 0;
36 else
37 ++CurrentModel;
38
39 TheSwitch->setSingleChildOn (CurrentModel);
40
41 return true;
42
43 // Up key: increase the current model scale
44 case osgGA::GUIEventAdapter::KEY_Up:
45 {
46 osg::ref_ptr<osg::PositionAttitudeTransform> pat =
47 dynamic_cast<osg::PositionAttitudeTransform*>(
48 TheSwitch->getChild (CurrentModel));
49 pat->setScale (pat->getScale() * 1.1);
50
51 return true;
52 }
53
54 // Down key: decrease the current model scale
55 case osgGA::GUIEventAdapter::KEY_Down:
56 {
57 osg::ref_ptr<osg::PositionAttitudeTransform> pat =
58 dynamic_cast<osg::PositionAttitudeTransform*>(
59 TheSwitch->getChild (CurrentModel));
60 pat->setScale (pat->getScale() / 1.1);
61 return true;
62 }
63
64 // Don’t handle other keys
65 default:
66 return false;
67 }
68 }
69 else
70 return false;
```

```
71 }
72 };
73
74
75 int main (int argc, char* argv[])
76 {
77 // Check command-line parameters
78 if (argc < 2)
79 {
80 std::cerr << "Usage: " << argv[0]
81 << " <model file> [ <model file> ... ]\n";
82 exit (1);
83 }
84
85 // Create a Producer-based viewer
86 osgProducer::Viewer viewer;
87 viewer.setUpViewer (osgProducer::Viewer::STANDARD_SETTINGS);
88
89 // Create the event handler and attach it to the viewer
90 osg::ref_ptr<osgGA::GUIEventHandler> eh (new ViewerEventHandler());
91 viewer.getEventHandlerList().push_front (eh);
92
93 // Construct the scene graph
94 TheSwitch = osg::ref_ptr<osg::Switch> (new osg::Switch());
95
96 for (int i = 1; i < argc; ++i)
97 {
98 osg::ref_ptr<osg::Node> loadedNode = osgDB::readNodeFile(argv[i]);
99 if (!loadedNode)
100 std::cerr << "Problem opening '" << argv[i] << "'\n";
101 else
102 {
103 osg::ref_ptr<osg::StateSet> ss (loadedNode->getOrCreateStateSet());
104 ss->setMode (GL_NORMALIZE, osg::StateAttribute::ON);
105 osg::ref_ptr<osg::PositionAttitudeTransform> pat(
106 new osg::PositionAttitudeTransform());
107 pat->addChild (loadedNode.get());
108 TheSwitch->addChild (pat.get());
109 }
110 }
111
112 // Ensure that we have at least on model before going on
113 if (TheSwitch->getNumChildren() == 0)
114 {
```

```
115 std::cerr << "No 3D model was loaded. Aborting...\n";
116 exit (1);
117 }
118
119 viewer.setSceneData (TheSwitch.get());
120
121 TheSwitch->setSingleChildOn (0);
122
123 // Enter rendering loop
124 viewer.realize();
125
126 while (!viewer.done())
127 {
128 viewer.sync();
129 viewer.update();
130 viewer.frame();
131 }
132
133 // Wait for all cull and draw threads to complete before exit
134 viewer.sync();
135 }
```