Smart Pointers
& osg::Referenced Class
& osg::ref_ptr<> Template Class

2008년 여름

박경신

Overview

- Smart Pointer 개요
- OSG Memory Management
- OSG::Referenced Class
- OSG::ref_ptr<> Template Class

Smart Pointer

- 스마트 포인터 (Smart Pointer)는 C++ class이다.
  - syntax 와 semantics상 일반 포인터와 같다.
  - memory management나 locking등 좀 더 유용한 일을 수행한다.
  - 가장 간단한 스마트 포인터의 예로 C++ standard 에 있는 **auto_ptr**

```
// auto_ptr 내부
template <class T>
class auto_ptr {
    T* ptr;
public:
    explicit auto_ptr(T* p = 0) : ptr(p) {}
    ~auto_ptr() {delete ptr;}
    T& operator*() {return *ptr;}
    T* operator->() {return ptr;}
    // ...
};
```


Smart Pointer

- auto_ptr를 사용하여 아래와 같이 바꿀 수 있다.

```
void foo()
{
    MyClass* p(new MyClass);
    p->DoSomething();
    delete p;
}
```

```
void foo()
{
    auto_ptr<MyClass> p(new MyClass);
    p->DoSomething(); // MyClass의 DoSomething 함수를 쓴다
}   // p가 알아서 scope를 벗어나면 clean up해준다
```

Smart Pointer

```
template<class T> class SmartPtr {        // your own SmartPtr class
    public:
        explicit SmartPtr(T * pointee) : pointee_(pointee);
        SmartPtr& operator=(const SmartPtr& other);
        ~SmartPtr() { delete pointee_; };
        T& operator*() const {    // operator overloading
            …
            return *pointee_;
        }
        T* operator->() const {  // operator overloading
            …
            return pointee_;
        }
        operator T*() { return pointee_; }     // conversion
    private:
        T *pointee_;
    };
```


Smart Pointer

```
class Widget {
    public:
        void Fun();
};

int main()
{
    // SmartPtr<Widget> sp = new Widget()과 같은 코드
    SmartPtr<Widget> sp(new Widget);

    sp->Fun();    // (sp.pointee_)->Fun()과 같은 코드
    (*sp).Fun();
}
```


Smart Pointer

- 기존의 포인터 쓰는 방식은

  Widget *p = new Widget;

  즉, 변수 p는 Widget 개체를 위해 할당된 메모리를 **point** 하고 또한 **owns**하고 있는 것이다.
- 따라서, 프로그래머는 **delete p**를 불러서 Widget 개체를 지우고 또한 메모리 해제를 하게 된다.
- 때문에, p로 포인트된 개체의 ownership을 해제하려면 다음과 같이 써야 한다.

  p = NULL; // assign something else to p
- 대부분의 스마트 포인터는 다양한 방법(ownership transfer, reference counting, etc)으로 ownership management를 해준다.

Why would I use Smart Pointer?

- Smart Pointer를 사용해서 우리가 얻는 것은 무엇인가?
- Automatic cleanup – 포인터를 free할 필요 없다.
- Automatic initialization – 스마트 포인터를 NULL 로 초기화할 필요 없다.
- Dangling pointers
- Exception safety – try/catch를 고민하지 않아도 된다.
- Garbage collection
- Efficiency – 스마트 포인터가 가용 메모리의 사용을 좀 더 효과적으로 만들어주며 메모리 할당과 해제 시간을 줄어준다.
- STL containers

Why would I use Smart Pointer?

```
MyClass* p(new MyClass);
MyClass* q = p;
delete p;
p->DoSomething(); // Watch out! p is now dangling!
p = NULL; // p is no longer dangling
q->DoSomething(); // Ouch! q is still dangling!

// auto_ptr은 copy할 때 자신의 ptr를 NULL로 만들어주어 dangling pointer문제 해결
template <class T> auto_ptr<T>& auto_ptr<T>::operator=(auto_ptr<T>& rhs) {
    if (this != &rhs) {
        delete ptr;
        ptr = rhs.ptr;
        rhs.ptr = NULL;
    }
    return *this;
}
```

- **Create a new copy** of the object pointed by p
- **Ownership transfer:** it transfer the responsibility for cleaning up ("ownership") from p to q
- **Reference counting**
- **Reference linking**
- **Copy on write**

Why would I use Smart Pointer?

```
class Base { /*...*/ };
class Derived : public Base { /*...*/ };
Base b;
Derived d;
vector<Base> v;
v.push_back(b); // OK
v.push_back(d); // error

// 다른 class 의 개체를 사용하고자 한다면 포인터를 사용해야 함
vector<Base*> v; v.push_back(new Base); // OK
v.push_back(new Derived); // OK too
// you need to cleanup!
for (vector<Base*>::iterator i = v.begin(); i != v.end(); ++i)
    delete *i;
```


Why would I use Smart Pointer?

```
// 스마트 포인터를 사용하면 자동으로 clean up해 줌
class Base { /*...*/ };
class Derived : public Base { /*...*/ };
Base b;
Derived d;
vector< linked_ptr<Base> > v;
v.push_back(new Base); // OK
v.push_back(new Derived); // OK too
// cleanup is automatic
```


Memory Management

- 일반적인 시나리오 경우, OSG 응용 프로그램은 scene graph의 root node로 포인트를 하고 있다.
- 그리고, root node 의 포인터가 scene graph의 모든 노드(node)로 reference하고 있다.

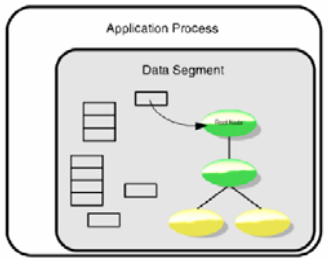

Figure 2-1
Referencing a scene graph
Typically, an application references a scene graph with a single pointer storing the address of the root node. The application doesn't keep pointers to other nodes in the scene graph. All other nodes are referenced, directly or indirectly, from the root node.

Memory Management

- OSG 응용 프로그램에서 scene graph 사용이 끝나면 메모리누수 (memory leak)을 피하기 위해 각 노드마다 차지하고 있던 메모리를 delete해야 한다.
- 전체 scene graph를 traverse하면서 각 노드(와 데이터)를 delete하는 일은 매우 복잡한 일이다.
- 이 문제의 해결방법으로 reference counted memory를 사용한 automated garbage collection을 사용해야 한다.
- 모든 OSG 노드는 reference counted 되어 있어서, reference count가 0으로 줄면 개체를 delete한다.
- 따라서, OSG의 전체 scene graph를 지우려면 root node 포인터 하나만 지우면 된다.

Memory Management

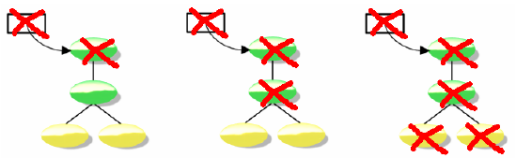

Figure 2-2
Cascading scene graph deletion
OSG's memory management system deletes the entire scene graph when the last pointer to the root node is deleted.

Memory Management

- 모든 OSG 노드와 scene graph 데이터 클래스는 공통 베이스 클래스인 osg::Referenced에서 파생(derived)되어 있다.
- osg::Referenced는 integer타입 reference count와 이것의 증가(increment)와 감소(decrement) 방법을 가지고 있다.
- OSG는 osg::ref_ptr<>라 불리는 smart pointer template class를 정의하고 있다.
- ref_ptr<> 변수를 사용하여 heap에 할당되어 있는 OSG 노드와 scene graph 데이터의 주소를 저장한다.
- 프로그램에서는 ref_ptr<> 변수로 referenced object 주소를 할당했을 때, reference count 가 자동적으로 증가한다.

Memory Management

- Referenced에서 파생한 개체의 포인터를 저장한 코드는 반드시 일반적인 C++ 포인터 변수보다는 ref_ptr<>에 포인터로 저장되어야 한다.
  - 그 변수로 할당된 메모리를 자동적으로 deletion해줄 수 있도록.
- ref_ptr<>는 operator overloading을 사용해서 ref_ptr<> 변수는 일반적인 C++ 포인터 변수와 같이 작동한다.
  - ref_ptr<>에서 operator ->()와 operator *()를 overloading
- 예외적인 경우로, 일반적인 C++ 포인터 변수를 Referenced에서 파생한 개체 저장에 "일시적" 사용이 가능하다 (만약 그 heap memory address가 궁극적으로 ref_ptr<>에 저장이 된다면). 그러나 일반적으로 ref_ptr<>을 사용하는 것이 안전하다.

OSG Referenced Class

- osg::Referenced class
  - reference counted block of memory로 구현되어 있다.
  - 모든 OSG nodes와 scene graph data (state information, arrays of vertices, normals, texture coordinates 등을 포함)은 이 클래스로부터 파생되었다.
- osg::Referenced 클래스의 3가지 주요 기능
  - Reference count member variable인 **_refCount**는 클래스 생성자 (constructor)에 0으로 초기화되어 있다.
  - Public methods인 **ref()** 와 **unref()**는 _refCount 변수의 증가 (incrementing)과 감소 (decrementing)을 담당하고 있다. 그리고, unref()는 _refCount가 0가 되면 개체를 delete한다.
  - 클래스 소멸자 (destructor)는 protected 와 virtual이다. 즉, 스택 (stack)에 생성하거나 명시적 파괴 (explicit deletion)는 불가능하다.

OSG ref_ptr<> Template Class

- osg::ref_ptr<> template class
  - Referenced 타입 개체의 스마트 포인터 (smart pointer)로 구현되어 있다.
  - Reference count를 관리한다.
  - Referenced 개체는 마지막 ref_ptr<>가 referencing하고 있는 것이 사라지게 되면 스스로 자신을 delete한다.
- ref_ptr<>의 주요 기능
  - Private 포인터 **_ptr** 은 managed memory address에 저장된다. **get()** 함수를 이용하여 _ptr 값을 받는다.
  - 프로그래머가 ref_ptr<>을 일반 C++ 포인터처럼 사용할 수 있도록 **operator->()**나 **operator=()** 같은 다양한 함수를 제공한다.
  - ref_ptr<>이 NULL이면 **valid()** 함수는 true를 반환한다.

OSG ref_ptr<> Template Class

- 프로그램에서 ref_ptr<> 변수로 주소를 할당할 때, ref_ptr<> assignment operator인 operator=()가 Referenced::ref()를 불러서 reference count를 자동적으로 증가시킨다.
- ref_ptr<>변수가 Referenced::unref()를 부르면서 reference count를 감소할 때의 두 가지 경우는 다음과 같다.
  - 클래스 소멸자에서 ref_ptr<> deletion 할 때
  - operator=()에서 reassignment할 때

OSG ref_ptr<> Template Class

- OSG의 osg::Geode 클래스와 osg::Group 클래스를 사용하는 예제
  - Geode는 렌더링을 위한 geometry를 가지고 있는 OSG leaf node이다.
  - Group은 여러 개의 children을 가진 하나의 node이다.
- 예:

```
#include <osg/Geode>
#include <osg/ref_ptr>
…
osg::ref_ptr<osg::Geode> geode = new osg::Geode;
If (!geode.valid())
   // ref_ptr<> is invalid, throw exception or display error
```

OSG ref_ptr<> Template Class

- 다른 예제

```
#include <osg/Geode>
#include <osg/Group>
#include <osg/ref_ptr>
…
{
   // Create a new osg::Geode object. The assignment to the
   // osg::ref_ptr<> increments the reference count to 1
   osg::ref_ptr<osg::Geode> geode = new osg::Geode;
   // Assume 'grp' is a pointer to an osg::Group node. Group uses a
   // ref_ptr<> to point to its children, so addChild() again increments
   // the reference count to 2
   grp->addchild(geode.get());
} // The 'geode' ref_ptr<> variable goes out of scope here.
  // This decrements the reference count back to 1
```


OSG ref_ptr<> Template Class

- 전 슬라이드의 예제에서는 코드에서 오랫동안 geode를 유지하지 않기 때문에 사실 ref_ptr<>이 필요하지 않다.
- osg::Group 부모 노드에서 내부적으로 ref_ptr<>이 osg::Geode가 차지하는 메모리를 관리하기 때문에 일반 C++로 충분하다.

```
// Create a new osg::Geode object (regular C++ pointer).
// Don't increments the child Geode increment the reference count 1
osg::Geode *geode = new osg::Geode;
// Internal ref_ptr<> in Group increments the child node Geode
// reference count to 1
grp->addChild(geode);
```


OSG ref_ptr<> Template Class

- OSG에서 Referenced 개체는 반드시 ref_ptr<> 변수에 선언이 되어야 한다. 일반 C++ 포인터를 사용할 시 주의할 것.

```
{
   osg::Geode * geode = new osg::Geode;
}
// don't do this, memory leak
```

- Referenced 에서 파생된 개체는 명시적으로 지울 수 없다.

```
osg::Geode geode1 = new osg::Geode;
delete geode1; // compile error: destructor is protected

{
   osg::Geode geode2;
}
// compile error: destructor is protected
```


OSG ref_ptr<> Template Class

```
osg::ref_ptr<osg::Geode> geode = new osg::Geode; // OK

int i;
osg::ref_ptr<int> rpi = &i;      // NO! int is not derived from Referenced

{
      // 'top' increments the Group count to 1
      osg::ref_ptr<osg::Group> top = new osg::Group;
      // addChild() increments the Geode count to 1
      top->addChild(new osg::Geode);
}     // The 'top' ref_ptr goes out of scope, deleting both the
     // Group and Geode memory
```

OSG ref_ptr<> Template Class

- 함수로 개체의 주소를 반환 (Returning the address of an object)할 때 주의해야 한다. 잘못 주소를 반환했을 경우, ref_ptr<>가 저장하고 있는 메모리 주소가 반환 스택에 돌려지기 전에 스코프를 벗어나게 된다.

```
// Don't do this. It stores the address as the return value on the call
// stack, but when the grp ref_ptr<> goes out of scope, the reference
// count goes to zero and the memory is deleted. The calling function
// is left with a dangling pointer.
osg::Group* createGroup() {
    // Allocate a new Group node
    osg::ref_ptr<osg::Group> grp = new osg::Group;
    // return the new Group's address
    return *grp;
}
```


OSG ref_ptr<> Template Class

- 위의 문제를 해결하려면 다음과 같이 사용해야 한다.

```
osg::ref_ptr<osg::Group> createGroup()
{
    osg::ref_ptr<osg::Group> grp = new osg::Group;

    // return the new Group's address. This stores the Group address
    // in a ref_ptr<> and places the ref_ptr<> on the call stack as the
    // return value
    return grp.get();
}
```


OSG ref_ptr<> Template Class Summary

- Referenced에서 파생된 개체를 ref_ptr<> 변수로 할당하는 것은 자동적으로 Reference::ref()에서 reference count를 증가시킨다.
- 만약 ref_ptr<> 변수가 다른 것을 포인트하게 되거나 또는 지워지게 되면 Referenced::unref() 를 불러서 reference count를 감소시킨다. 그리고, 만약 reference count가 0가 되면 unref()는 개체에 할당된 메모리를 delete하게 된다.
- Referenced 타입의 개체를 할당할 때, 항상 ref_ptr<>를 지정해서 OSG의 메모리 관리가 정확히 될 수 있도록 한다.

Reference

- http://www.informit.com/articles/article.aspx?p=31529
- http://ootips.org/yonat/4dev/smart-pointers.html