Using OSG in Your Application

2008년 여름
박경신

Overview

- Rendering
- Viewer Class
- Dynamic Modification
  - Data Variance
  - Callbacks
  - NodeVisitors
  - Picking

Rendering

- 렌더링 컨트롤을 위해 다음과 같이 해야 한다.
  - OpenGL model-view matrix를 변경하는 view management 코드를 만들어야 한다.
  - Window와 OpenGL context을 생성한다. (만약 응용프로그램에서 필요하다면 multiple windows 와 contexts를 관리해야 한다)
  - Paged DB를 사용하려면 osgDB::DatabasePager를 사용한다.
  - Update, cull, draw traversal을 구현하기 위해 osgUtil::UpdateVisitor, orgUtil::CullVisitor, osgUtil::RenderStage 개체를 instantiate한다.
  - Main loop에서 events를 처리한다.
  - Model-view matrix 를 갱신하는 view 를 부른다.
  - 프레임버퍼 렌더링 전에 glClear()를 부른다. 그리고 update, cull, draw를 불러서 렌더링을 하고, swap buffers 한다.
  - 스테레오 (stereo)나 멀티파이프 (multipipe) 렌더링을 필요하다면 추가적인 코드를 작성한다.
  - 마지막으로, platform-independent 하게 코드를 작성한다.

Rendering

- 이 과정은 매우 복잡하므로, OSG는 utilities와 libraries를 제공한다.
  - osgUtil::SceneView – SceneView 는 update, cull, draw traversals를 wrapped한 클래스이다. (그러나, DatabasePager는 쓰지 않음)
  - Producer – Producer는 멀티 파이프 렌더링을 지원하는 external camera library이다.
  - osgProducer – osgProducer 는 OSG와 Producer를 응용프로그램 사용에 맞게 통합한 라이브러리이다.

Rendering

- OSG 2.0부터 core OSG 에 osgViewer를 추가했다.
- osgViewer는 display management, event handling, rendering과 같은 응용프로그램에 필요한 다양한 기능을 가진 a set of viewer classes를 가지고 있다.
- osgViewer는 osg::Camera 클래스를 사용하여 OpenGL model-view matrix를 바꾼다.
- osgViewer는 DatabasePager 를 full support 한다.

Rendering

- osgViewer는 다음 Viewer와 CompositeViewer를 지원한다.
  - Viewer 클래스
    - 여러 개의 동기화된 카메라로 멀티 모니터에 하나의 뷰 렌더링을 관리한다.
    - 그래픽 시스템 기능에 따라 자체적인 window(s)과 graphics context(s)을 생성한다.
    - Single Viewer-based application은 하나 또는 여러 개의 디스플레이에서 실행된다.
  - Composite Viewer 클래스
    - 같은 scene에 대한 multiple views를 지원하고 다른 scene에 대해 multiple cameras를 지원한다.
    - 각 view의 렌더링 순서를 지정함으로써 하나의 렌더링의 결과를 다른 곳으로 넘겨줄 수 있다.
    - 이 클래스를 사용하여 HUDs, prerender textures, display multiple views in a single display를 생성한다.

Viewer Class

- 예제:

```
#include <osgViewer/Viewer>
#include <osgDB/ReadFile>

int main(int, char **)
{
    osgViewer::Viewer viewer;
    viewer.setSceneData(osgDB::readNodeFile("cow.osg"));
    return viewer.run();
}
```


Changing the View

- Viewer는 osg::Camera 개체를 생성해서 OpenGL model-view matrix를 관리한다.
- Camera 콘트롤 2가지 방법
  1. Viewer 에 camera manipulator를 attach한다.
     - By default, Viewer::run()이 osgGA::TrackballManiuplator 를 생성해서 카메라 콘트롤을 한다.
     - osgGA는 Viewer::setCameraManipulator() 와 같이 지정할 수 있는 여러 개의 manipulators를 정의하고 있다.
  2. 사용자가 camera projection과 view matrices를 정의한다.
     - 사용자가 update view 와 render frame을 구현해야 한다.

Changing the View

```
osgViewer::Viewer viewer;
viewer.setSceneData(osgDB::readNodeFile("cow.osg"));
viewer.getCamera()->setProjectionMatrixAsPerspective(40., 1., 1., 100.);
// create a matrix to specify a distance from the viewpoint
osg::Matrix trans;
Trans.makeTranslate(0., 0., -12.);
// rotation angle (in radians)
double angle(0.);
while (!viewer.done()) {
    // create the rotation matrix
    osg::Matrix rot;
    rot.makeRotate(angle, osg::Vec3(1., 0., 0.));
    angle += 0.01;
    // Set the view matrix (concatenation of rotation and translation matrixes)
    viewer.getCamera()->setViewMatrix(rot * trans);
    // draw the next frame
    viewer.frame();
}
```


Setting Camera

- Camera는 projection matrix를 지정하는 함수들
  - void setProjectionMatrix(const osg::Matrix& matrix);
    // glMatrixMode(GL_PROJECTION);
    // glLoadMatrix(m);
  - void setProjectionMatrixAsOrtho(double left, double right, double bottom, double top, double zNear, double zFar);
    // glOrtho
  - void setProjectionMatrixAsOrtho2D(double left, double right, double bottom, double top);
    // gluOrtho2D
  - void setProjectionMatrixAsFrustum(double left, double right, double bottom, double top, double zNear, double zFar);
    // glFrustum
  - void setProjectionMatrixAsPerspective(double foby, double aspectRatio, double zNear, double zFar);
    //gluPerspective

Setting Camera

- Camera 개체의 view matrix를 지정하는 함수
  - setViewMatrix(const osg::Matrixf &matrix)
  - setViewMatrixAsLookAt(const osg::Vec3 &eye, const osg::Vec3 &center, const osg::Vec3 &up)

Setting the Clear Color

- Camera는 clear color를 지정하는 interface를 제공한다.

```
// set the clear color to black
viewer.getCamera()->setClearColor(osg::Vec4(0., 0., 0., 1.));
```

- By default, Camera는 depth 와 color buffers를 clear 한다.
- Camera::setClearMask()를 사용하여 적당한 OpenGL buffer flags를 넣어준다.

```
// clear the color, depth, and stencil buffers at the start of each frame
viewer.getCamera()->setClearMask(GL_COLOR_BUFFER_BIT |
      GL_DEPTH_BUFFER_BIT | GL_STENCIL_BUFFER_BIT);
```

Dynamic Modification

- OSG는 애니메이션이나 dynamic scene을 생성을 위한 scene의 동적인 변경을 가능하게 해준다.
- Cull traversal은 draw traversal에서 처리할 render graph에 geometry와 state을 저장한다.
- osgViewer는 스레드 안전 (thread safety)를 위해 lock을 강제하지 않는다. 그대신, 응용프로그램에서 cull과 draw traversal 밖에서 scene graph를 변경시킨다.
- **Update traversal 에서만 scene graph를 변경시킨다.**

Dynamic Modification

- Dynamic scene graph modification을 위해서
  - OSG에게 scene graph의 어느 부분이 변경되어야 할지를 알려줘야 한다 – 이것은 개체 (Node, Drawable, StateSet, 등)에 **data variance**를 지정하는 것으로 가능하다.
  - Node 와 Drawable 개체에 callback를 지정해야 한다; OSG는 이렇게 지정된 callback 함수를 specific traversal에서 실행시킨다. Node나 Drawable을 update traversal에서 변경시키고자 한다면 **update callback**을 지정해야 한다.
  - 응용 프로그램은 미리 scene graph의 어느 부분이 변경될지를 모를 수 있다. 때문에 사용자가 scene graph에서 관심있는 node를 찾거나 마우스나 다른 입력 방식에 의해 **node**를 선택해야 한다.

Data Variance

- osgViewer는 draw traversal 이 완료되기 전에 main loop가 지속될 수 있는 스레드 모델을 지원한다.
  - 즉, draw traversal이 active할 때에 Viewer::frame()이 반환할 수 있다.
  - 이전 프레임의 draw traversal은 다음 프레임의 update traversal과 겹칠 수도 있다.
- osg::Object::setDataVariance()으로 conflict resolution의 방법을 제공하고 있다.
- **setDataVariance()**를 사용하여 개체의 data variance를 지정한다.
  - 초기화 상태에서 data variance는 **UNSPECIFIED**이다.
  - Data variance는 **STATIC** 이나 **DYNAMIC**으로 바꿀 수 있다.

Data Variance

- OSG는 반드시 모든 DYNAMIC Drawable과 StateSet 개체를 처리한 후에 draw traversal이 반환하도록 하고 있다.
- Draw traversals은 반환된 후에도 여전히 render graph를 처리할 수 있다. 그러나, 그 시점에서는 오직 STATIC 데이터만 render graph에 남아있다.
- Render graph는 오직 Drawable과 StateSet 개체의 reference 만 가지고 있다.
- 만약 프로그램이 스위치 노드 (switch on/off a child) 경우와 같이 노드를 변경하고자 할 때 노드의 data variance는 DYNAMIC으로 지정되어야 한다.

Callbacks

- OSG는 Node와 Drawable objects에 callback을 지정할 수 있도록 한다.
  - Node callbacks은 update과 cull traversals에서 불려진다.
  - Drawable callbacks은 cull과 draw traversals에서 불려진다.
- NodeCallback을 사용하려면, 다음과 같이 한다.
  - NodeCallback에서 파생된 새로운 클래스를 생성한다.
  - **NodeCallback::operator()()** 함수를 override한다.
    - 프로그램의 dynamic modification을 수행한다.
  - **Node::setUpdateCallback()**을 사용하여 변환하고자 하는 node에 이 파생 클래스의 새로운 instance를 attach 한다.
- OSG는 각 update traversal에서 파생 클래스의 operator()()함수를 불러서 node를 바꾼다.

Callbacks

- OSG는 operator()()함수에 2개의 파라메터를 보낸다.
  - First parameter는 프로그램의 callback에 관련된 Node – operator() 함수에서 dynamic modification할 callback의 node
  - Second parameter는 osg::NodeVisitor 주소

```
class RotateCB : public osg::NodeCallback {
public:
      void operator()(osg::Node* node, osg::NodeVisitor* nv) {
      ….
          traverse(node, nv);
      }
};
…
// attach callback to a node
node->setUpdateCallback(new RotateCB);
```


Callbacks Example

- Update callback을 사용한 dynamic modification 예제
  - 다음은 하나의 cow를 두 개의 MatrixTransform 노드에 attach하고, NodeCallback에서 파생한 클래스를 이 두 개의 MatrixTransform에 각각 attach한다. 그리고, update traversal에서 NodeCallback이 각 cow를 rotate 시킨다.

Figure 3-1
Dynamic modification using an update callback
This figure shows the output of the Callback example program. The code dynamically rotates the cow on the left around its vertical axis, while the cow on the right remains unmodified.

Callbacks Example

- RotateCB::operator()() 는 traverse()을 가지고 있다. traverse()는 update traversal (osgUtil::UpdateVisitor)가 현재 그룹 노드의 children를 traverse 하게 한다.
- NodeCallback은 pre- 또는 post-traversal processing을 수행하는데, 프로그래머가 traverse() call과 관련있는 code를 어디에 넣느냐에 따라 달려있다.

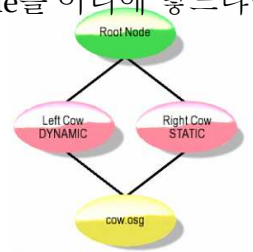

Figure 3-2
The Callback example program scene graph
This figure shows the Callback example program's scene graph hierarchy. Note the two **MatrixTransform** nodes have different data variance.

Callbacks Example

```
osg::ref_ptr<osg::Node> CreateScene() {
   osg::Node * cow = osgDB::readNodeFile("cow.osg");
    cow->setDataVariance(osg::Object::STATIC);

    osg::ref_ptr<osg::MatrixTransform> mtLeft = new osg::MatrixTransform;
    mtLeft->setName("Left Cow\nDYNAMIC");
    // will modify this node during the update traversal
    mtLeft->setDataVariance(osg::Object::DYNAMIC);
    osg::Matrix m;
    m.makeTranslate(-6.f, 0.f, 0.f);
    mtLeft->setMatrix(m);
    mtLeft->addChild(cow);

    osg::ref_ptr<osg::MatrixTransform> mtRight = new
    osg::MatrixTransform;
    mtRight->setName("Right Cow\nSTATIC");
    mtRight->setDataVariance(osg::Object::STATIC);
    …
```


NodeVisitors

- NodeVisitor는 scene graph를 traverse 하고 각 visited node를 위한 함수를 부른다.
- OSG osgUtil::UpdateVisitor (derived from NodeVisitor)를 사용해서 update traversal을 수행한다 – 그리고 update traversal에서 NodeCallback::operator()()를 부른다.
- NodeVisitor는 대부분의 OSG 노드 타입에 맞게 overload되는 여러 가지 apply() 함수를 가지고 있다.
- Articulated robot arm 모델 예제
  - 조인트에 Transform 있는 로봇 모델을 로딩한 후, NodeVisitor를 사용하여 Transform 노드에 애니메이션을 활성화하려면,
  - 특별 제작한 NodeVisitor를 사용하고 apply(osg::Transform &) 함수를 override하여 사용한다.
  - 그러면, 이 NodeVisitor가 scene graph를 방문하면서 Transform에서 파생된 노드에 apply()를 실행시킨다.

NodeVisitors – FindNamedNode

```
// Derive a class from NodeVisitor to find a node with a
//   specific name.
class FindNamedNode : public osg::NodeVisitor
{
public:
    FindNamedNode( const std::string& name )
      : osg::NodeVisitor( // Traverse all children.
                osg::NodeVisitor::TRAVERSE_ALL_CHILDREN ),
        _name( name ) {}

    // This method gets called for every node in the scene
    //   graph. Check each node to see if its name matches
    //   our target. If so, save the node's address.
    virtual void apply( osg::Node& node )
    {
        if (node.getName() == _name)
            _node = &node;

        // Keep traversing the rest of the scene graph.
        traverse( node );
    }

    osg::Node* getNode() { return _node.get(); }

protected:
    std::string _name;
    osg::ref_ptr<osg::Node> _node;
};
```


NodeVisitor

- NodeVisitor가 traverse할 때 사용하는 규칙
  - 평범한 NodeVisitor 를 사용하여 TRAVERSE_ALL_CHILDREN 한다.
  - 특별 제작한 NodeVisitor 클래스로 NodeVisitor::traverse() 를 불러서 노드의 children을 traverse 하는 하나 또는 그 이상의 apply() 함수를 override 한다. 이렇게 하기 위하여, 특별 제작한 NodeVisitor가 pre- 또는 post-traversal operations을 수행하고 필요 시 traversal을 멈추게 한다.
  - NodeVisitor 클래스로 실행된 callback을 사용할 때 (예를 들어, update callback), NodeVisitor는 노드의 children을 callback없이 traverse한다. NodeVisitor는 callback이 attach된 노드의 children을 traverse하지 않는다. 그 대신, operator()가 NodeCallback::traverse() 를 불러서 callback 함수가 children를 traverse하는 것을 책임진다.

NodeVisitor

- NodeVisitor 로 scene graph를 traverse 하기 위해, NodeVisitor 를 파라메터로 Node::accept()로 보낸다.
- 어떤 노드에서든 accept()를 부를 수 있고, NodeVisitor는 그 노드에서부터 scene graph를 traverse 한다.
- 따라서, 전체 scene graph를 search하려면, root 노드에서 accept()를 부른다.

Picking

- 대부분의 3차원 응용 프로그램은 화면의 일부를 선택하는 picking 기능이 필요하다.
- Picking은 사용자가 화면에 마우스 버튼을 선택하면, 내부적으로 2차원 마우스 위치를 scene graph에 위치로 변경시켜주는 transformation 을 수행한다.
- Picking 연산
  - 마우스 이벤트를 받는다. (osgGA 라이브러리 사용)
  - 마우스 커서가 scene graph의 어느 부분으로 대응되는지 결정한다.

Capturing Mouse Events

- osgGA::TrackballManipulator는 마우스 이벤트를 받아서 카메라의 view matrix를 변환할 수 있는 것으로 osgGA::GUIEventHandler에서 파생되었다.
- osgGA::GUIEventHandler는 abstract base class로 GUI 이벤트와 관련된 기능을 수행하려면 이 클래스로부터 파생된 클래스를 사용한다.
- 마우스와 관련된 picking을 수행하려면, GUIEventHandler::handle() 함수를 override 해야 한다.

```
virtual bool GUIEventHandler::handle (
                    const osgGA::GUIEventAdapter& ea,
                    osgGA::GUIActionAdapter& aa);
```


Capturing Mouse Events

- Overriding하는 GUIEventHandler::handle() 함수는 GUIEventAdapter에서 마우스 이벤트를 포함한 GUI 이벤트를 받는다.
- GUIEventAdapter header file은 EventType을 선언하여 프로그램에서 관심있는 이벤트를 시험한다.
- GUIActionAdapter는 GUI 시스템에 대한 응용 프로그램 interface이다. GUIEventHandler가 attach되면 그 결과로 GUIActionAdapter가 viewer 클래스이다.
- 렌더링하기 전에, GUIEventHandler의 instance를 생성하고 그것을 Viewer::addEventHandler()에 viewer로 attach한다.
- Viewer는 여러 개의 event handler를 추가할 수 있다. 그리고 각 GUI event마다 handle()을 부른다.

Intersections

- Mouse picking intersection tests
  - Mouse picking을 마우스 위치에서부터 scene으로 향하는 ray로 생각하고 scene의 일부가 ray와의 intersection을 찾는다.
  - OSG에서는 ray보다는 **polytope**라는 피라미드 볼륨을 사용하여 intersection을 계산한다.
- **osgUtil::IntersectionVisitor**
  - NodeVisitor의 파생 클래스
  - 각 노드의 bounding volume 와 intersection volume을 테스트한다.
  - 만약 subgraph가 child intersection이 없을 것 같으면 완전히 subgraph traversal을 skip한다.
  - IntersectionVisitor는 여러가지 다양한 geometry (planes과 line segments를 포함)와의 교차 계산을 한다.
  - 생성자에서 osgUtil::Intersector (geometry 정의하고 실제 intersection testing을 수행하는)를 파라메터로 받는다. osgUtil::PolytopeIntersector가 마우스 기반의 picking에서 주로 사용된다.

Intersections

- OSG에서 마우스 기반의 picking을 수행하려면, 다음과 같이 한다.
  - GUIEventAdapter에 저장된 normalized 마우스 위치를 사용하여 **PolyetopeIntersector**를 생성한다.
  - **IntersectionVisitor**를 생성하고, 이것을 PolytopeIntersector 생성자에 파라메터로 넘겨준다.
  - 다음과 같이 IntersectionVisotor를 scene graph의 root node에서 실행 (일반적으로 Viewer Camera를 통하여) 한다.

```
// 'iv' is an IntersectionVisitor
viewer->getCamera()->accept(iv);
```

  - PolytopeIntersector 가 intersection을 가지고 있으면, NodePath를 얻는데 이것을 가지고 관심있는 node를 찾는다.

PickHandler

```
class PickHandler : public osgGA::GUIEventHandler {
public:
    PickHandler() : _mX(0.), mY(0.) {}
    bool handle (const osgGA::GUIEventAdapter& ea,
                        osgGA::GUIActionAdapter& aa) {
        osgViewer::Viewer* viewer =
                dynamic_cast<osgViewer::Viewer*>(&aa);
        if (!viewer) return false;
        switch(ea.getEventType()) {
            case osgGA::GUIEventAdapter::PUSH:
            case osgGA::GUIEventAdapter::MOVE:
              { // record mouse location for button press & move events
                _mX = ea.getX(); _mY = ea.getY();
                return false;
              }
```


PickHandler

```
            case osgGA::GUIEventAdapter::RELEASE:
              { // if mouse hasn't moved, perform a pick
                if (_mX == ea.getX() && _mY == ea.getY()) {
                    if (pick(ea.getXnormalized(), ea.getYnormalized(),
                             viewer))
                        return true;
                }
                return false;
              }
            default:
                return false;
        }
    }
```

PickHandler

```
protected:
    float _mX, _mY; // mouse x, y
    bool pick(const double x, const double y,
                                    osgViewer::Viewer* viewer) {
        if (!viewer->getSceneData()) return false;
        double w(.05), h(.05);
        osgUtil::PolytopeIntersector* picker =
            new osgUtil::PolytopeIntersector(
                osgUtil::Intersector::PROJECTION, x-w, y-h, x+w, y+h);
        osgUtil::IntersectionVisitor iv(picker);
        viewer->getCamera()->accept(iv);
        if (picker->containsIntersections()) {
            const osg::NodePath& nodePath =
                picker->getFirstIntersection().nodePath;
```


PickHandler

```
            unsigned int idx = nodePath.size();
            while (idx--) {
                // find the last MatrixTransform in the node path
                osg::MatrixTransform* mt =
                    dynamic_cast<osg::MatrixTransform*>(nodePath[idx]);
                if (mt == NULL) continue;
                // if we there, we just found a MatrixTransform in nodePath
                // clear the previous selected node's callback
                if (_selectedNode.valid())
                    _selectedNode->setUpdateCallback(NULL);
                _selectedNode = mt;
                _selectedNode->setUpdateCallback(new RotateCB);
                break;
            }
```


PickHandler

```
            }
            if (!_selectedNode.valid())
                osg::notify() << "Pick failed" << std::endl;
        }
        else if (_selectedNode.valid()) {
            _selectedNode->setUpdateCallback(NULL);
            _selectedNode = NULL;
        }
        return _selectedNode.valid();
    }
};
int main(int argc, char **argv) {
…
    viewer.addEventHandler(new PickHandler);
…
}
```