OpenSceneGraph jegyzetek

Valasek Gábor

2012. június

Memóriakezelés OSG-ben:

- allokáció
- deallokáció

Mindkettő smart pointereken keresztül történik.

osg::ref_ptr<>: smart pointer, #include <osg/ref_ptr>

- get(): public, visszaadja a hivatkozott memóriaterületet (pl. osg::Node* pointert)
- operator*(): dereferálás, a hivatkozott területet l-value-ként visszaadja (pl. osg::Node&-ot)
- operator->(): a hivatkozott objektum eljárásai hívhatóak rajta keresztül
- operator=(): kicseréli a managed ptr-t egy másikra
- operator==(), operator!=(): ugyanarra hivatkoznak-e
- operator!(): valid-e a pointer (nem valid, ha NULL-ra hivatkozik, vagy nem volt hozzárendelve még semmi)
- valid(): ld. fent
- release(): csökkenti a hivatkozásszámlálót ÉS visszaadjuk a referált memóriaterület (hasznos pl. fv-ből visszatérésre, ahol a függvény visszatérését egy ref_ptr_be fogjuk fel).

osg::Referenced: hivatkozásszámlált memóriaterületek, minden színtérgráfbeli elem ősosztálya, #include <osg/Referenced>

- ez tárolja a referenciaszámlálót
- ha a hivatkozásszámláló nullához ér, akkor megsemmisül az objektum:
  - ref(): növeli a hivatkozásszámlálót (ezt kézzel nagyon ritkán hívjuk)
  - unref(): csökkent a hivatkozásszámlálót (ezt kézzel nagyon ritkán hívjuk)
  - referenceCount(): hivatkozásszámláló értéke
- CSAK a heap-en hozhatóak létre ebből származó objektumok!
- a destruktora protected, szóval olyat lehet, hogy osg::ref_ptr<osg::Node> node = new osg::Node; de olyat nem, hogy osg::Node node; (mivel destruktor nem lesz)
- hivatkozások ne alkossanak köröket, mert azt nem tudja kezelni!

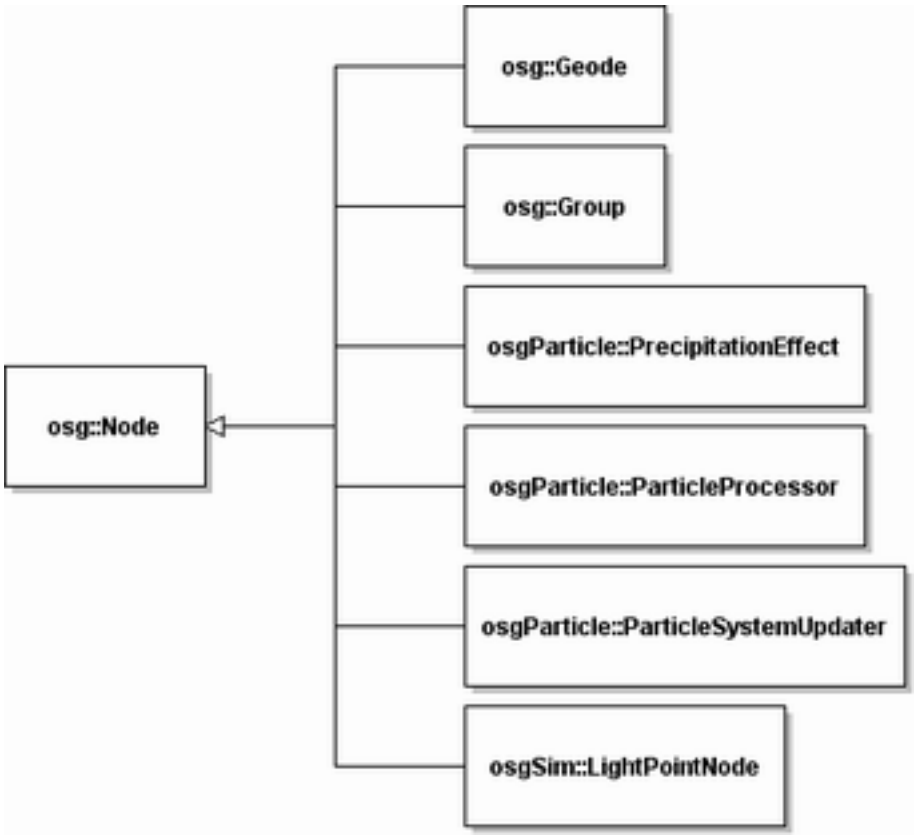
osg::DeleteHandler: törlő ütemező, ami szálbiztosan gondoskodik a nullára csökkent hivatkozásszámú objektumok eltakarításáról.

osg::Object: az összes olyan objektum ősinterfésze, amelyek IO-t, klónozást, referenciaszámlálást igényelnek. Az osg::Referenced-ből származik.

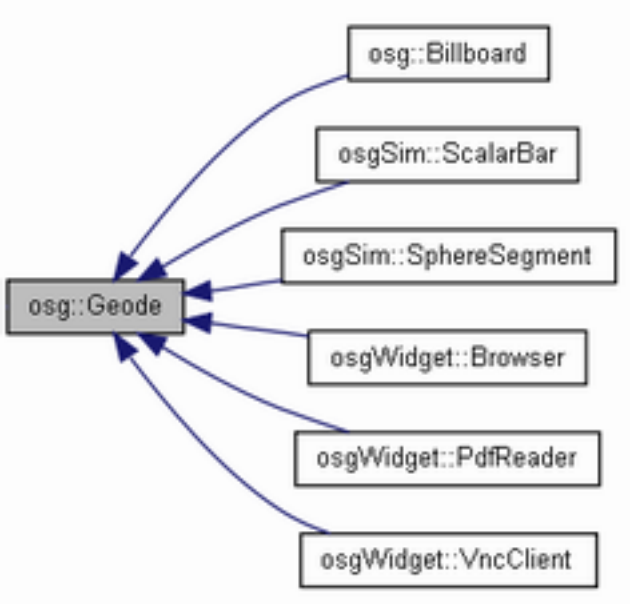
Notifier: egy beépített OSG mechanizmus debug infók közlésére és irányítására (std output, fájl stb.). osg::notify(NotifySeverity) << “uzenet”; NotifySeverity pl. osg::WARN osg::NotifyHandler leszármazottba át lehet irányítani a osg::notify(...) parancsokat, feltéve ha a osg::setNotifyHandler-nek be-new-oljuk egy példányát. Az osg::setNotifyLevel(osg::WARN és társai) beállítja, hogy milyen szintig jelenjenek meg a kiírások.

4. fejezet (62.o.)

Node a színtérgráf csúcsainak ősosztálya, interfésze. A leszármazottai: http://www.openscenegraph.org/documentation/OpenSceneGraphReferenceDocs/a00431.html



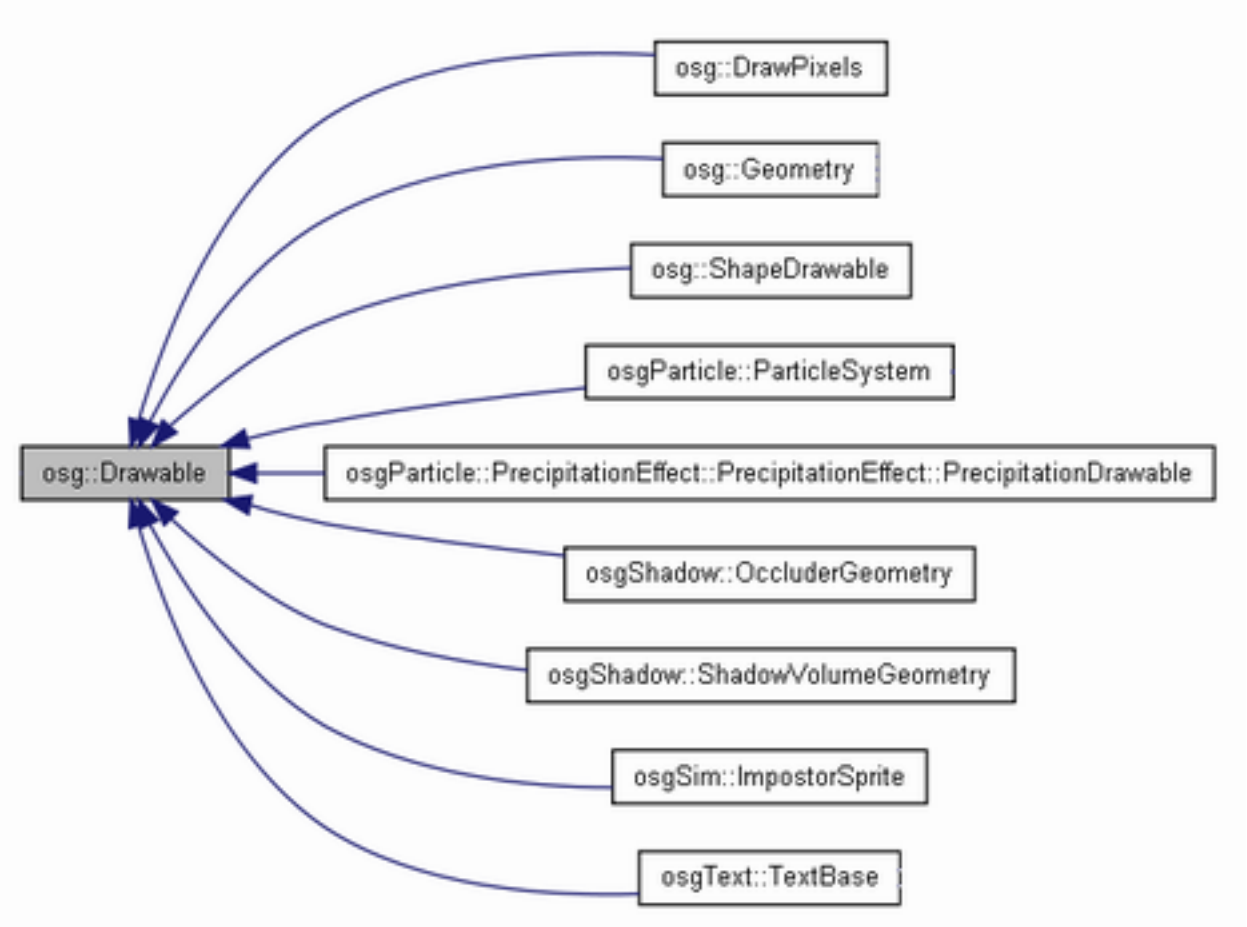
osg:Geode (#include <osg/Geode>): a színtérgráf levél elemeinek megfelelő osztály. Mindig kirajzolandó geometriát tartalmaz. A kirajzolásra kerülő geometriai adatokat a Geode által kezelt **osg::Drawable** objektumok tárolják.



osg::Drawable: pure virtual osztály mindenféle kirajzolható elem (geometria, meshek, szövegek, képek stb.) interfészének meghatározására. Az osg::Object-ből származik.

A geode drawable elemeinek kezelésére a következő eljárásai használhatóak:
- osg::Geode::addDrawable(...). Minden így hozzáadott osg::Drawable interfészt a geoda osztály egy osg::ref_ptr-rel kezeli.
- osg::Geode::removeDrawable(<ptr>), osg::Geode::removeDrawable(<drawable ptr-ek tombjének első indexe, amit törölni akarunk>, <törlendők elemek száma>)
- osg::Geode:getDrawable(<idx>): az <idx> indexű drawable ptr-ét adja vissza
- osg::Geode::getNumDrawables(): mennyi drawable van itt most

A Drawable leszármazottai: http://www.openscenegraph.org/documentation/OpenSceneGraphReferenceDocs/a00185.html



osg::ShapeDrawable (#include <osg/ShapeDrawable>): osg::Drawable-ből származó egyszerű geometriák. A kirajzolandó alakzat geometriáját és attribútumait egy osg::Shape objektumban tárolja.

shapeDrawable->setShape(new osg::Box(<x>, <y>, <z>, <w>, <h>, <d>))

Példák alap shape-ekre: osg::Box, osg::Capsule, osg::Cone, osg::Cylinder, osg::Sphere.

Létrehozás: baloldalt ref_ptr, jobboldalt new-olás, így:

```
osg::ref_ptr<osg::ShapeDrawable> shape1 = new ShapeDrawable();
shape1->setShape(...);
```

A színtérgráf felépítéséhez geodát használhatunk:

```
osg::ref_ptr<osg::Geode> root = new osg::Geode;
root->addDrawable( shape1.get() );

osgViewer::Viewer viewer;
```

```
viewer.setSceneData( root.get() );
return viewer.run();
```

Az osg::ShapeDrawable csak prototyping-ra javasolt, szuboptimális, a valós rajzolása az **osg::Geometry** osztály kell.

Vertex adatok tárolása

osg::Array: nem példányosítható, egy interfész, amin keresztül a puffer létrehozásokat és módosításokat lekommunikálja az OSG az OpenGL-lel. A leszármazottai (osg::Vec2Array, osg::Vec3Array, osg::UIntArray) STL vektor-barátak, azaz van push_back(...), pop_back() és size() eljárásuk is. **FONTOS**: az osg::Array-eknek a *heap-en kell allokálódniuk és smart pointereken keresztül piszkáljuk csak őket*. (DE: nyilván az osg::Vec3 stb-re ez nem vonatkozik).

osg::Geometry (#include <>): az OGL VBO-k elfedése, mind a 16 lehetséges OGL-es attribútom megadható vele. Ezeket array-ekben tárolja: (**67. o.**)

- setVertexArray(...)
- setNormalArray(...)
- setColorArray(...)
- setSecondaryColorArray(...)
- setFogCoordArray(...)
- setTexCoordArray(...)
- setVertexAttribArray(...)

Ez még a szokásosnál is kényelmesebb, osg::Geometry geom-ot feltételezve pl.::

- geom->setColorBinding(osg::Geometry::BIND_PER_VERTEX): minden pozícióhoz külön szín kapcsolódik
- geom->setColorBinding(osg::Geometry::BIND_OVERALL): minden pozícióhoz ugyanaz a szín tartozik

Haladóknak: ahhoz, hogy VBO-kat használjon az osg::Geometry több dolgot is kell tennünk:

1. `geometry->setUseVertexBufferObject(true)`
2. az attribútumok per-vertex legyenek (nincs per-primitív normális, szín stb.)
3. ne használjunk régi vertex index pufferes dolgokat (?)

Haladóknak: a VAO-k csatornáinak használatához az osg::Geometry setVertexAttribBinding(<idx>, osg::Geometry::AttributeBinding::BIND_PER_VERTEX) és a setVertexAttribArray(<idx>, array.get()); duó kell. Ha shadereket használunk, akkor az osg::Program addBindAttribLocation-jét használjuk (ha nem layout-ozunk).

Az osg::Geometry az osg::Drawable osztályból származik, így osg::Geode-okhoz hozzáadható (root->addDrawable(quad.get());).

Metában tehát a tennivalók:
- a használni kívánt attribútumokat belepakoljuk arraylistekben, megfelelően használva őket:
```
osg::ref_ptr<osg::Vec3Array> vertices = new osg::Vec3Array;
```
- ha ezek megvannak létrehozunk egy osg::Geomtry-t és rendeljük hozzá a különböző csatornákhoz az adatokat

A csatornák megadására van az “elavult” mód:
```
osg::ref_ptr<osg::Geometry> quad = new osg::Geometry;
quad->setVertexArray( vertices.get() );
quad->setNormalArray( normals.get() );
quad->setNormalBinding( osg::Geometry::BIND_OVERALL );
```
Ez a következő index hozzárendeléseket jelenti a vertex attribútum csatornáknál:

gl_Vertex	0
gl_Normal	2
gl_Color	3
gl_SecondaryColor	4
gl_FogCoord	5
gl_MultiTexCoord0	8
gl_MultiTexCoord1	9
gl_MultiTexCoord2	10
gl_MultiTexCoord3	11
gl_MultiTexCoord4	12
gl_MultiTexCoord5	13
gl_MultiTexCoord6	14
gl_MultiTexCoord7	15

(Forrás: http://www.opengl.org/sdk/docs/tutorials/ClockworkCoders/attributes.php)

Ami számunkra sokkal hasznosabb, hogy explicit is megadhatjuk a csatornákat:

setVertexAttribBinding(unsigned int csatorna, **AttributeBinding** binding), ahol az AttirbuteBinding a fentieknek megfelelő, azaz:

enum osg::Geometry::AttributeBinding

Enumerator:

BIND_OFF	
BIND_OVERALL	
BIND_PER_PRIMITIVE_SET	
BIND_PER_PRIMITIVE	
BIND_PER_VERTEX	

Ezután már csak a **setVertexAttribArray** (unsigned int index, **Array** *array) hívással kell beállítani a tényleges adatokat a csatornához.

Egy példával:

```
      quad->setVertexAttribBinding(0,
osg::Geometry::AttributeBinding::BIND_PER_VERTEX);
      quad->setVertexAttribArray(0, vertices.get() );
```

A csúcspontok segítségével kirajzolandó primitívek típusát kell már csak meghatározni. Erre szolgálnak az osg::Geometry-ben az osg::PrimitiveSet interfésszel kapcsolatos műveletek:
- addPrimitiveSet()
- removePrimitiveSet()
- getPrimitiveSet()
- getNumPrimitives()

Például:

```
quad->addPrimitiveSet( new osg::DrawArrays(GL_QUADS, 0, 4) );
```

Ezek a PrimitiveSet-ek a glDrawArrays és glDrawElements-et enkapszulálják. A vertexes és indexpufferes kirajzolás pedig a következőképpen különül el:
osg::DrawArrays: glDrawArrays
osg::DrawElements<UByte | UShort | UInt>(): glDrawElements

A szignatúra:

```
geom->addPrimitiveSet( new osg::DrawArrays(mode, first, count) )
```

ahol a `mode` az OpenGL primitívtípusokból kerül ki, a `first` az első index száma a vertex array-ben, ami ehhez a primitívhez hozzátartozik, a `count` pedig a primitív kirajzolása során használt vertexek számát jelöli.

osg::DrawElementsUInt példa:

Az eljárás ugyanaz, mint sima OpenGL-ben volt:

```
osg::ref_ptr<osg::DrawElemtsUInt> de =
                 new osg::DrawElementsUInt( GL_TRIANGLES );
de->push_back( 0 ); de->push_back(1); de->push_back(2);
```

Ezután ezt a de-t kell hozzáadni addPrimitiveSet-tel a geom-hoz. Az adatcsatornákat az vertex attrib csatornákhoz ugyanúgy kell hozzárendelni.

Poligon módosító utility-k:

Több osgUtil névtérbeli függvény segítségével módosíthatjuk a geometriákat, egy előfeldolgozás keretében. Sok meglehetősen számításigényes, alapvetően nem on-line jellegű használatra valók.

osgUtil::Simplifier: háromszögek számának csökkentése a simplify() eljáráshívás után
osgUtil::SmoothingVisitor: minden primitívnek kiszámítja a normálvektorát, átlagolva, a smooth() eljáráshívás hatására
osgUtil::TangentSpaceGenerator: a generate() hatására a getTangentArray(), getNormalArray() és getBinormalArray() tömbök feltöltődnek a megfelelő bázisvektorokkal (bázisvektor vajon? biztos egységhosszúak?)
osgUtil::Tessellator: GLU tesszelációs egység, retessellatePolygons() eljárás segítségével.
osgUtil::TriStripVisitor: a stripify() eljáráshívás hatására triangle strip-ekké alakítja a kirajzolandó felületet.

A fentiek osg::Geometry objektumokon működnek (illetve azokra mutató smart-pointerekkel):

```
osgUtil::TriStripVisitor tsv;
tsv.simplify( *geom );
```

Geometriai attribútumok újraolvasása

Nincs explicit topológiai adatszerkezet- és műveletkészlet az OSG-ben, de lehetőség van funktorok segítségével létező **drawable**-ekből visszaolvasni dolgokat. Funktorok:

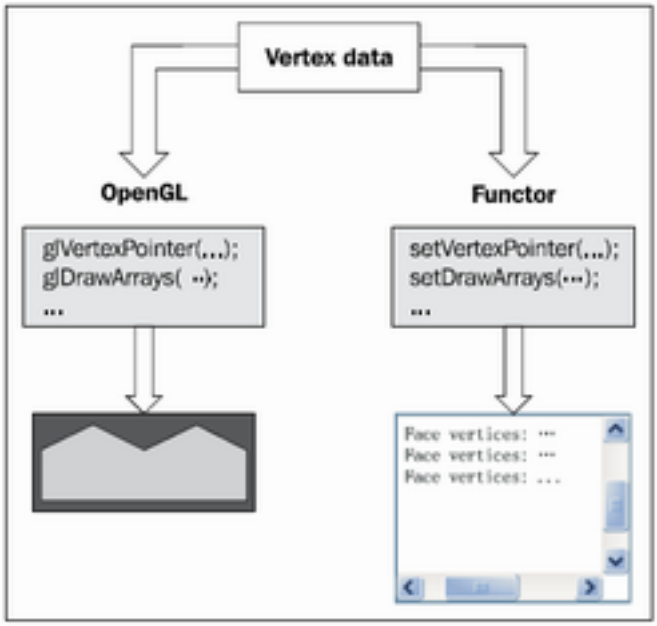
osg::Drawable::AttributeFunctor - az ebből származtatott funktor az apply() függvényt megvalósítva a geometria csúcspontjainak adatait tudja visszaolvasni
osg::Drawable::ConstAttributeFunctor - a fenti read-only verziója
osg::Drawable::PrimitiveFunctor - glDrawArrays() kirajzolás “emulálásával” végigiterál az osg::Geometry-ben meghatározott összes primitíven és primitívenként olvashatunk ki információkat
osg::Drawable::PrimitiveIndexFunctor - ez pedig a glDrawElements()-et emulálva olvas ki primitíveket

Példa egy háromszöges bejárásra

```
…
#include <osg/TriangleFunctor>
struct FaceCollector
{
    void operator()(
           const osg::Vec3& v1,
           const osg::Vec3& v2,
           const osg::Vec3& v3,
           bool )
    {
        std::cout     << "Face vertices: "
                      << v1 << "; "
                      << v2 << "; "
                      << v3 << std::endl;
    }
};

int main() {...
    osg::ref_ptr<osg::Geometry> geom = new osg::Geometry;
    geom->setVertexArray( vertices.get() );
    ...
    osg::TriangleFunctor<FaceCollector> functor;
    geom->accept( functor );
    ...
}
```

A geometriát az accept()-en keresztül kérjük meg a funktor kiértékelésére, ami a template paraméterben található struct operator()-ját fogja alkalmazni.



Saját Drawable leszármazottak készítése

Az osg::Drawable-ből leszármazó osztályoknak két fontos virtuális függvényt meg kell valósítaniuk:

- **computeBound() const**: bounding box kiszámítása (frustrum culling-hoz). Ehhez osg::BoundingBox objektumot kell használni (ez olyan mint az osg::Vec3, nincs smart pointer!)
- **drawImplementation()**: a tényleges geometriakirajzás. Ez a belső draw() függvény által kerül meghívásra, paraméterben kap egy osg::RenderInfo&, ami a konkrétan használt OSG rendering backend adatait tartalmazza. (Figyelem: ha rendes kirajzolás callback-et akarunk, akkor kapcsoljuk ki a displaylisteket: `myDrawable->setUseDisplayList( false );`)

Saját kirajzoláshoz szükség lehet az osg::GLBeginEndAdapter-re.

osg::DrawPixels: ne használjuk, elavult. **Helyette** textúrázott quad-ot csináljunk és **vagy** állítsunk be egy új, orto vetítéssel bíró kamerát (osg::Camera) és azt használjuk a kirajzolásakor, **vagy** pedig rakjuk be a quad-ot egy osg::AutoTransform vagy osg::Billboard (osg::Transform leszármazott) csúcsok gyerekeiként.

5. Színtérgráf kezelése (93.o.)

Színtérgráf felépítése
- osg::Geode - csak levél lehet
- osg::Group - belső csúcs (persze ha nincs gyereke, akkor levél). Ebből származik az **osg::Transform** is!

osg::Group (#include <osg/Group>, osg.lib) (94. o.): tetszőleges számú gyereke lehet, osg::Geode levelek, osg::Group csoportok stb. A különböző funkciójú node-ok, azaz **NodeKit**-ek alaposztálya.

Pedigré: osg::Node-ból származik, ezen keresztül az osg::Referenced-ből, azaz smart pointereken keresztül kezelendő és a heap-re allokálandó erőforrás.

A gyermekeit osg::ref_ptr<> listában tárolja. A gyermekek kezelésére a következő eljárások használhatóak:
- addChild(): a gyermeklista végére beszúr egy új gyereket.
- insertChild(): a gyermeklista egy megadott helyére szúrja be az új gyereket
- removeChild(), removeChildren(kezdőindex, törlendők száma)
- getChild(idx): visszaadja a gyerekre mutató osg::Node ref_ptr-t
- getNumChildren(): gyerkőcök száma

Megjegyzés: az **osg::Node** is képes részfákat reprezentálni, mint az osg::Group, vagy leveleket is. Vagy dynamic_cast<>-olni kell a megfelelőre az osg::Node változót, vagy pedig használni az osg::Node::asGeode(), osg::Node::asGroup() eljárásokat (utóbbiak hatékonyabbak is)!

Szülők kezelése
osg::Node-ból származó eljárások ezen a területen:
- getParent(idx): egy osg::Group (!) ref.pointerrel tér vissza, ami az adott csúcs, idx-edik szülőjét hivatott jelképezni
- getNumParents(): szülő csúcsok száma, a fenti getParent(idx)-ből az idx = 0, 1, …, getNumParents()-1
- getParentalNodePaths(): visszaadja a gyökérből ebben a csúcsba vezető összes utat a színtérgráfban egy osg::NodePath változóban. Az **osg::NodePath** (95.o.): egy std::vector-ba berakott osg::Node vektor.

Egy szülő kivételekor a színtérgráfból automatikusan törlődik a bejegyzés a leszármazottakból is.
A gyökér getNumParents() eljárása nullát ad vissza nyilván.

Csúcs típusának eldöntése
Ha például egy modellt töltünk be és tudni akarjuk, hogy osg::Group vagy csak osg::Geode, akkor lehet dynamic_cast-ot használni és ha a visszaadott ptr NULL, akkor NEM azt kaptunk, amire számítottunk.

```
osg::ref_ptr<osg::Group> model =
  dynamic_cast<osg::Group*>( osgDB::readNodeFile("cessna.osg") );
```

Vagy egyszerűen használhatjuk az asGroup/asGeode-ot is (98.o.):

```
// Assumes the Cessna's root node is a group node.
```

```
osg::ref_ptr<osg::Node> model = osgDB::readNodeFile("cessna.osg");
osg::Group* convModel1 = model->asGroup(); // OK!
osg::Geode* convModel2 = model->asGeode(); // Returns NULL.
```

Színtérgráf bejárása (98.o.)

Valamilyen, céljainknak megfelelő gráfbejárási stratégiát hogyan lehet megvalósítani az OSG színtérgráfban?

Többféle bejárástípus is lehetséges, a bejárás célját felhasználva kategorizálásra:
1. event traversal: a különböző felhasználói események (billentyű, egér stb.) lekezelése
2. update traversal: a színtérgráf módosítása (trafók, attribútumok változtatása), callback-ek hívása), node-ok egyéb függvényeinek meghívása stb. történik itt
3. cull traversal: a belső *renderelési lista* összeállítása, eldönti a csúcsokról, hogy láthatóak-e
4. draw traversal: a színtér tényleges kirajzolása az OpenGL-en keresztül. A cull traversal által létrehozott *render listát* használja!!

Többprocesszoros környezetben ezek a bejárások párhuzamosíthatóak! Az OSG ezt csinálja.

A bejárásnál jön elő a *látogató (visitor)* tervminta.

Transzformációs csúcsok
osg::Transform (#include <osg/Transform>, osg.lib) (99.o.): az osg::Group-ból származó csúcs, ami közvetlenül nem példányosítható, hanem a különböző transzformációkat létrehozó függvényeken keresztül hozhatjuk létre. Bejáráskor az általa meghatározott trafó mátrixa hozzászorzódik az aktuális MVP mátrixhoz (matrix stack analógia).

osg::Matrix (100.o.) tárolja a trafókat, 4x4-es, konstruktorral explicit megadható, de a következő érdekesebb műveletei is elérhetőek:
- postMult() avagy operator*(): a mátrixunk után rak egy mátrixot (szorzás), a preMult() meg elé
- makeTranslate(), makeRotate(), makeScale(): a meglévő mátrixot lecserélik a nevüknek és paraméterezésnek megfelelő trafóra. A statikus verzióikkal (translate(), rotate(), scale()) is lehet új mátrixokat létrehozni.
- invert(): invertálja a mátrixot. Statikus verziója az inverse(), ami egy új, osg::Matrix-ban adja vissza az adott mátrix inverzét.

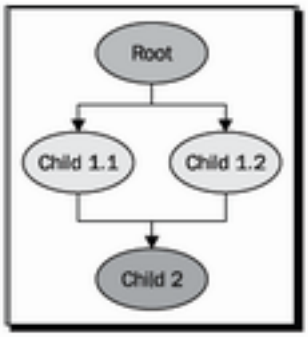
FONTOS: az OSG row-major (sorfolytonos) mátrixokat használ!
TEHÁT: az OSG-ben a vektorok SORvektorok, vagyis a mátrixok ELÉ kerülnek! Legbaloldalabbi mátrix lesz lesz a legelső trafó!

osg::MatrixTransform (#include <osg/MatrixTransform>) (101.o.): ez szúrandó be a színtérgráfba a geometria fölé szülőnek. Tartalmaz egy osg::Matrix változót és az ebben tárolt transzformációt alkalmazza a gyerekekre. Eljárás:
- setMatrix(mtx): a paramban lévő osg::Matrix típusú trafó mátrixát állítja be a node-hoz.

Példa használatra:

```
osg::ref_ptr<osg::MatrixTransform> trafo = new osg::MatrixTransform;
trafo->setMatrix( osg::Matrix::translate(1.0f, 0.0f, 0.0f) );
trafo->addChild( model.get() );
```

Fontos: ha ugyanazt a geomtriát rajzoljuk ki, akkor egyszerűen ugyanannak a Geode-nak TÖBB SZÜLŐJE lesz! Az OSG színtérgráf nem fa, így engedélyezett a több szülő, ezen keresztül pedig az object sharing.



Megjegyzés: osg::Quat a kvaternió osztály.

Switch node-ok (104.o.)

osg::Switch (#include <osg/Switch> (104.o.): az osg::Group-ból származik, a leszármazottak közül kiválasztható, hogy melyiket jelenítse meg. Eljárások:
- addChild(): a szokásos felülírtja, amivel megmondható plusz paraméterben, hogy a hozzáadott gyermek megjelenítendő-e vagy sem
- setValue(): az i-edik gyermek láthatóságát állítja be true vagy false-ra
- getValue(): i-edik gyermek láthatósága
- setNewChildDefaultValue(): alapértelmezésben mi legyen a láthatósága a bekerülő gyermeknek

```
osg::ref_ptr<osg::Node> model1= osgDB::readNodeFile("cessna.osg");
osg::ref_ptr<osg::Node> model2=
 osgDB::readNodeFile("cessnafire.osg");

osg::ref_ptr<osg::Switch> root = new osg::Switch;
root->addChild( model1.get(), false );
root->addChild( model2.get(), true );
```

Level-of-detail (107.o.)

osg::LOD (#include <osg/LOD> (107.o.): osg::Group-ból származik. Gyermekeinek ugyanazon modell egyre egyszerűbb verzióit szúrjuk be. Minden gyerekhez tartozik egy intervallum, amin belül megjelenik. Eljárásai:
- addChild(node, a, b): a node modell fog megjelenni, ha a modell (illetve a bounding boxának középpontja) [a,b] távolságban van a kamerától
- setRange(idx, a, b): az idx-edik gyermek megjelenítési távolság-tartományát átállítja [a, b]-re

Egy egyszerűsítő util-lal a következőképp használható:

Egy modell betöltése fájlból, aztán deep-copy készítése két példányban (az utóbbi kettőt fogjuk egyszerűsíteni). Ehhez a Node clone() eljárását kell használni:

```
osg::ref_ptr<osg::Node> modelL3 = osgDB::readNodeFile("cessna.osg");
osg::ref_ptr<osg::Node> modelL2 =
 dynamic_cast<osg::Node*>(modelL3->clone(osg::CopyOp::DEEP_COPY_ALL)
);
osg::ref_ptr<osg::Node> modelL1 =
 dynamic_cast<osg::Node*>(modelL3->clone(osg::CopyOp::DEEP_COPY_ALL)
);
```

A clone() a következő másolási módokat ismeri (a DEEP_COPY_ALL mindent deep copyzik, de lehet csak bizonyos dolgokat (Drawable-öket, primitíveket stb.), részletek itt: http://www.openscenegraph.org/documentation/OpenSceneGraphReferenceDocs/a00140.html

Ezután egyszerűsítsük (“újramintavételezzük”) a két másolatot, fele és tizedakkora “felbontásban”, mint az eredeti. Ehhez az osgUtil::Simplifier visitor-t használjuk:

```
osgUtil::Simplifier simplifier;
simplifier.setSampleRatio( 0.5 );
modelL2->accept( simplifier );
simplifier.setSampleRatio( 0.1 );
modelL1->accept( simplifier );
```

Most már csak a LOD gyermekeket kell beszúrni a range-ekkel:

```
osg::ref_ptr<osg::LOD> root = new osg::LOD;
root->addChild( modelL1.get(), 200.0f, FLT_MAX );
root->addChild( modelL2.get(), 50.0f, 200.0f );
root->addChild( modelL3.get(), 0.0f, 50.0f );
```

Proxy és page node-ok (110.o.)

osg::osgProxy (#include <osg/ProxyNode>) és **osg::PagedLOD**: Az alkalmazásunk az elindulása után tölti csak be a proxy-k által reprezentált geometriát egy külön szálon. Nem addChild()-ot, hanem setFileName()-et használ. Az osg::PagedLOD pedig egy tárolóról betöltött

LOD node.

```
osg::ref_ptr<osg::ProxyNode> root = new osg::ProxyNode;
root->setFileName( 0, "cow.osg" );
```

A tényleges betöltést az osgDB::DatabasePager végzi. Ha a rendering backend túl sok dolgot próbál megjeleníteni, akkor a nem használt PagedNode-okat kirakja a memóriából.

NodeKit-ek írása (112.o.)

Minden osg::Node-ból származó osztálynak van egy virtuális `traverse()` függvénye, amely egy osg::NodeVisitor leszármazottat vár. A traverse() függvényben kezelhetjük le a különböző típusú bejárókkal mit tegyünk. Alapból a következőt teszi (Node cpp fájljából):

```
        /** Traverse downwards : calls children's accept method with
NodeVisitor.*/
        virtual void traverse(NodeVisitor& /*nv*/) {}
```

A Node-okban az `accept()` eljárás (ez fogad el egy visitor-t) a látogató konkrét vendéglátásáért felelős.
Mj.: a traverse()-ben ne felejtsük el meghívni az ősosztály osg::<típus, pl. Switch>::traverse(<kapott osg::NodeVisitor>)-t, hogy továbbmenjen a bejárás!

```
virtual void traverse( osg::NodeVisitor& nv );
```

Ne felejtsük el a leszármazott visitornak a public részébe belerakni a következőt:

```
META_Node( osg,  <osztály neve>);
```

Ez a következőt teszi:

```
/** META_Node macro define the standard clone, isSameKindAs, className
  * and accept methods.  Use when subclassing from Node to make it
  * more convenient to define the required pure virtual methods.*/
#define META_Node(library,name) \
        virtual osg::Object* cloneType() const { return new name (); }
\
        virtual osg::Object* clone(const osg::CopyOp& copyop) const {
return new name (*this,copyop); } \
        virtual bool isSameKindAs(const osg::Object* obj) const {
return dynamic_cast<const name *>(obj)!=NULL; } \
        virtual const char* className() const { return #name; } \
        virtual const char* libraryName() const { return #library; } \
```

```
virtual void accept(osg::NodeVisitor& nv) { if
(nv.validNodeMask(*this)) { nv.pushOntoNodePath(this);
nv.apply(*this); nv.popFromNodePath(); } } \
```

A traverse()-ben ha tudni akarjuk, hogy milyen a bejárónk, akkor castoljunk:

```
      osgUtil::CullVisitor* cv =
dynamic_cast<osgUtil::CullVisitor*>(&nv);
      if ( cv ) ...
```

Tipp követéshez:
osg::computeLocalToWorld(<nodePath, pl. node->getParentalNodePaths()[0]>) segítségével.

osg::NodeVisitor (#include <osg/NodeVisitor>) (117.o.): Az apply() eljárásával alkalmazzuk a látogatót egy csúcsra. Többféle paraméterrel hívható:

```
- virtual void apply( osg::Node& );
- virtual void apply( osg::Geode& );
- virtual void apply( osg::Group& );
- virtual void apply( osg::Transform& );
```

A használata így néz ki egy visitor-nak:

```
ExampleVisitor visitor;
visitor->setTraversalMode( osg::NodeVisitor::TRAVERSE_ALL_CHILDREN );
node->accept( visitor );
```

A következő bejárási módok adhatóak meg:
TRAVERSE_NONE,
TRAVERSE_PARENTS,
TRAVERSE_ALL_CHILDREN,
TRAVERSE_ACTIVE_CHILDREN

A következő bejárási konstansok vannak:
NODE_VISITOR,
UPDATE_VISITOR,
COLLECT_OCCLUDER_VISITOR,
CULL_VISITOR

Node-bejárási link: http://www.openscenegraph.org/projects/osg/wiki/Support/Tutorials/FindingNodes

6. fejezet (123. o.)
Effektek

osg::StateSet (#include <osg/StateSet>) (124.o): OGL állapotváltozók és értékeik részhalmaza, amiket egy node-hoz lehet hozzárendelni és a node kirajzolása előtt glEnable(),

glDisable() stb. utasításokkal érvényesíti őket az OSG. Így lehet egy osg::Node-hoz vagy osg::Drawable-höz hozzárendelni egy osg::StateSet-et:

```
osg::StateSet* stateset = new osg::StateSet;
node->setStateSet( stateset );
```

Ha biztosra akarunk menni, akkor a meglévőt módosítva használjuk, így mindig valid lesz a state set:

```
osg::StateSet* stateset = node->getOrCreateStateSet();
```

A state ref_ptr-rel kezelt a Node-okban, így ugyanazt a state-et több csúcs is megoszthatja.

A StateSet két csoportba osztja az attribútumokat: **textúra** és **nem-textúra**.

Eljárásai:
- setAttribute(): a paraméterben kapott osg::StateAttribute-ot hozzáadja a StateSet értékhalmazhoz
- setMode(): egy módhatározó enumot ad hozzá az állapothalmazhoz és az osg::StateAttribute::ON vagy osg::StateAttribute::OFF segítségével be/kikapcsolja.
- setAttributeAndModes(): rendering attrib + a hozzátartozó mód hozzáadása a StateSet-hez. Nem mindegyik attribútumhoz tartozik “mód”, de ettől lehet használni még mindenre ezt a függvényt.

osg::StateAttribute (#include <osg/StateAttribute>) (124. o.): egy virtuális ősosztály renderelési állapotváltozók (attribútumok) értékeinek tárolására, mint például megvilágítási adatok, anyagok stb.

Egy nem-textúra állapot beállítása a következőképp néz ki:

```
stateset->setAttributeAndModes( attr, osg::StateAttribute::ON );
```

A textúrákkal kapcsolatos állapotokhoz még hozzá kell fűzni, hogy melyik textúrázási egységet szeretnénk elérni:

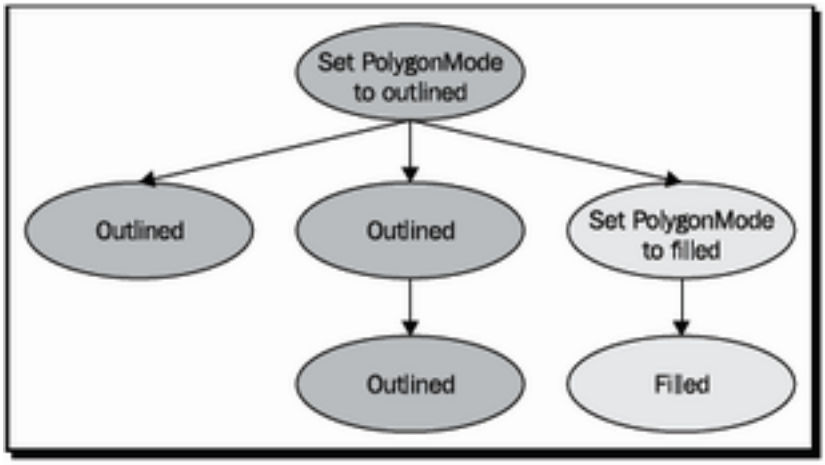
```
stateset->setTextureAttributeAndModes(0, texattr,
osg::StateAttribute::ON );
```

Pl. poligonmódú kirajzolásra:

```
osg::ref_ptr<osg::PolygonMode> pm = new osg::PolygonMode;
pm->setMode(osg::PolygonMode::FRONT_AND_BACK,
osg::PolygonMode::LINE);
transformation1->getOrCreateStateSet()->setAttribute( pm.get() );
```

Az osg::PolygonMode osztály segítségével lehet megadni, hogy milyen oldal rajzolási módját szeretnénk módosítani: FRONT, BACK, vagy FRONT_AND_BACK. Utána megadhatjuk, hogy miképp legyen rajzolva az a rész, POINT, LINE, vagy FILL. PolygonMode-hoz nem kell mode-ot kapcsolni (nincs ki/be kapcsolója).

A state-ek a színtérgráfbeli gyermekekre is érvényben vannak, hacsak valamelyik felül nem írja azt, ebben az esetben az ő gyermekeire (újabb felülírásig) az új értékek lesznek érvényben.



A osg::StateAttribute::OVERRIDE és társaik segítségével a leszármazottak attribútumfelülírási stratégiái korlátozhatóak: http://www.openscenegraph.org/documentation/OpenSceneGraphReferenceDocs/a00698.html#0ce0de58c1fd7c3d41486bc6f35ee44c (az enum **osg::StateAttribute::Values** doc page-e)
Pl. a *PROTECTED* nem engedi a felülírást felülről, az OVERRIDE pedig a lejjebb fekvő állapotváltozó értékadásokat felülbírálják (ezt védi ki a PROTECTED). Használatban:

```
stateset->setAttribute( attr, osg::StateAttribute::OVERRIDE );
stateset->setAttributeAndModes( attr, osg::StateAttribute::ON|
osg::StateAttribute::OVERRIDE );
```

Fixed function pipeline effektjei a következők segítségével érhetőek el:

Type ID	Class name	Associated mode	Related OpenGL functions
ALPHAFUNC	osg::AlphaFunc	GL_ALPHA_TEST	glAlphaFunc()
BLENDFUNC	osg::BlendFunc	GL_BLEND	glBlendFunc() and glBlendFuncSeparate()
CLIPPLANE	osg::ClipPlane	GL_CLIP_PLANEi (i ranges from 0 to 5)	glClipPlane()
COLORMASK	osg::ColorMask	-	glColorMask()

Type ID	Class name	Associated mode	Related OpenGL functions
CULLFACE	osg::CullFace	GL_CULLFACE	glCullFace()
DEPTH	osg::Depth	GL_DEPTH_TEST	glDepthFunc(), glDepthRange(), and glDepthMask()
FOG	osg::Fog	GL_FOG	glFog()
FRONTFACE	osg::FrontFace	-	glFrontFace()
LIGHT	osg::Light	GL_LIGHTi (i ranges from 0 to 7)	glLight()
LIGHTMODEL	osg::LightModel	-	glLightModel()
LINESTIPPLE	osg::LineStipple	GL_LINE_STIPPLE	glLineStipple()
LINEWIDTH	osg::LineWidth	-	glLineWidth()
LOGICOP	osg::LogicOp	GL_COLOR_LOGIC_OP	glLogicOp()
MATERIAL	osg::Material	-	glMaterial() and glColorMaterial()
POINT	osg::Point	GL_POINT_SMOOTH	glPointParameter()
POINTSPRITE	osg::PointSprite	GL_POINT_SPRITE_ARB	OpenGL point sprite functions
POLYGONMODE	osg::PolygonMode	-	glPolygonMode()
POLYGONOFFSET	osg::PolygonOffset	GL_POLYGON_OFFSET_POINT, and so on	glPolygonOffset()
POLYGONSTIPPLE	osg::PolygonStipple	GL_POLYGON_STIPPLE	glPolygonStipple()
SCISSOR	osg::Scissor	GL_SCISSOR_TEST	glScissor()

Type ID	Class name	Associated mode	Related OpenGL functions
SHADEMODEL	osg::ShadeModel	-	glShadeModel()
STENCIL	osg::Stencil	GL_STENCIL_TEST	glStencilFunc(), glStencilOp(), and glStencilMask()
TEXENV	osg::TexEnv	-	glTexEnv()
TEXGEN	osg::TexGen	GL_TEXTURE_GEN_S, and so on	glTexGen()

Mivel ezek elavultak, ezért nem részletezzük őket.

Ha beállítottunk valamilyen attribútumot (fent az első oszlopban láthatóak a lehetséges nevek), akkor lekérdezés így néz ki:

```
osg::PolygonMode* pm = dynamic_cast<osg::PolygonMode*>(
          stateset->getAttribute(osg::StateAttribute::POLYGONMODE));
```

A második oszlop az OSG-beli nevét tartalmazza az attribútumnak, amennyiben mód tartozik hozzá, ha a ki/bekapcsolt állapotára szeretnénk rákérdezni, akkor így lehet használni:

```
osg::StateAttribute::GLModeValue value = stateset->getMode(
GL_LIGHTING );
```

Köd beállítása példa (134. o.)

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

osg::ref_ptr<osg::Fog> fog = new osg::Fog;
fog->setMode( osg::Fog::LINEAR );
fog->setStart( 500.0f );
fog->setEnd( 2500.0f );
fog->setColor( osg::Vec4(1.0f, 1.0f, 0.0f, 1.0f) );

osg::ref_ptr<osg::Node> model = osgDB::readNodeFile( "lz.osg" );
model->getOrCreateStateSet()->setAttributeAndModes( fog.get() );

osgViewer::Viewer viewer;
viewer.setSceneData( model.get() );
return viewer.run();
```

FFP fények (136.o.)

osg::Light (136.o.):

SKIP, mi úgyis shaderezni fogunk.

Textúrázás (140.o.)

osg::Image (#include <osg/Image>) (140.o.): kép tárolásáért felelős. Műveletek/eljárások:
- betöltés: `osg::ref_ptr<osg::Image> img = osgDB::readImageFile( "pix.bmp" );`
- s(), t(), r(): a kép szélessége, magassága és mélysége
- data()?
- allocateImage(): ezzel lehet kézzel létrehozni egy képet:

```
osg::ref_ptr<osg::Image> image = new osg::Image;
image->allocateImage( s, t, r, GL_RGB, GL_UNSIGNED_BYTE );
unsigned char* ptr = image->data();
... // Operate on the ptr variable directly!
```

A textúrázás ugyanúgy működik, mint OGL-ben: a vertexeknek meg kell adni a textúrakoordinátáit és be kell utána állítania textúrákat.

osg::Texture (#include <osg/Texture>) (142.o.): a textúrák ősosztálya, leszármazottai az osg::Texture1D, osg::Texture2D, osg::Texture3D, osg::TextureCubeMap. Az ősosztály eljárásai:
- setImage(): beállítja az adott textúrához tartozó képet:

```
osg::ref_ptr<osg::Image> image = osgDB::readImageFile("a.bmp");
osg::ref_ptr<osg::Texture2D> texture = new osg::Texture2D;
texture->setImage( image.get() );
```

- a konstruktorral is meg lehet ám ezt csinálni:

```
osg::ref_ptr<osg::Image> image = osgDB::readImageFile("a.bmp");
osg::ref_ptr<osg::Texture2D> texture =
            new osg::Texture2D( image.get() );
```

- getImage(): a textúra visszaolvása egy osg::Image-be (illetve egy smart pointerrel védettbe)

Az osg::Geometry-ben ne feledjük megadni egy (pl.) osg::Vec2Array-ben a textúrakoordinátákat és ezeket “bindolni” a geometriában a setTexCoordArray() paranccsal, a megfelelő mintavételező egységnek szánt koordinátákkal. Pl. ha a 0-ás egységet akarjuk használni, akkor

```
geom->setTexCoordArray( 0, texcoord.get() );
```

Azután már csak a mintavételezőt kell beállítanunk:

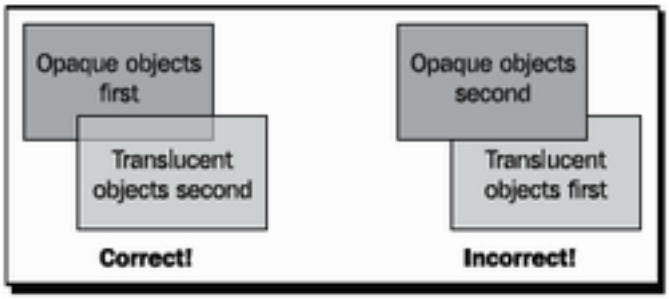
```
geom->getOrCreateStateSet()->setTextureAttributeAndModes(
      texture.get() );
```

Ha az Image-et csak kirajzoláskor használjuk textúraként, akkor a rendszermemóriabeli másolatot eltakaríttattathatjuk a következőképpen:

```
texture->setUnRefImageDataAfterApply( true );
```

Így a GPU-ra feltöltött textúra marad a kép egyetlen példánya, nem lesz rendszermemóriabeli másolata.

Átlátszóság kezeléséhez:



A sorrendmeghatározást az osg::State setRenderingHint() eljárásval tudjuk befolyásolni. A következőképpen mondhatjuk meg, hogy egy node átlátszatlan (ez a default):

```
node->getOrCreateStateSet()->setRenderingHint(
           osg::StateSet::OPAQUE_BIN );
```

És hogy átlátszó:

```
node->getOrCreateStateSet()->setRenderingHint(
           osg::StateSet::TRANSPARENT_BIN );
```

Átlátszó elemek keverésének beállítása:

```
osg::ref_ptr<osg::BlendFunc> blendFunc = new osg::BlendFunc;
blendFunc->setFunction( GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA );

osg::StateSet* stateset = geode->getOrCreateStateSet();
stateset->setTextureAttributeAndModes( 0, texture.get() );
stateset->setAttributeAndModes( blendFunc );
stateset->setRenderingHint( osg::StateSet::TRANSPARENT_BIN );
```

Shaderek (152.o.)

osg::Shader (#include <osg/Shader>) (152.o.): shader object wrapper, eljárásai:
- konstruktor: ha a vertText std::string-ben van a szöveg, akkor egy vertexshader létrehozása:

```
osg::ref_ptr<osg::Shader> vertShader =
     new osg::Shader( osg::Shader::VERTEX, vertText );
```

- setShaderSource(): egy std::string-ből készít egy shadert
- loadShaderSourceFromFile(): a paraméterben kapott fájlból betölti a shader szöveges fájlját

osgDB-t is használhatunk a betöltésre:

osg::Shader* fragShader = osgDB::readShaderFile(“source.frag”);

osg::Program (#include <osg/Program>) (153.o): a shaderek összelinkelése után kapott program kezelőosztálya. A shader-t a node-hoz attribútumként kell hozzárendelni:

```
osg::ref_ptr<osg::Program> pr = new osg::Program;
pr->addShader( vertShader.get() );
pr->addShader( fragShader.get() );
pr->addShader( geomShader.get() );
node->getOrCreateStateSet()->setAttributeAndModes(pr.get());
```

A uniform változók kezelésére több lehetőségünk van. Az **osg::StateSet**, amibe belerakjuk a programunkat rendelkezik a következő eljárásokkal is:
- addUniform(“uniform neve”, <érték>)
- getOrCreateUniform(“uniform neve”, <uniform típusa, osg::Uniform::Type-ból>), ami visszaad egy **osg::Uniform**-ot.

Példákkal:

```
stateset->addUniform(
 new osg::Uniform("color1",
                  osg::Vec4(1.0f, 0.5f, 0.5f, 1.0f)) );

osg::Uniform* color1 = stateset->
 getOrCreateUniform("color1",osg::Uniform::FLOAT_VEC4)->
          set( osg::Vec4(0, 1, 1, 1) );
color1->set( osg::Vec4(1, 0, 1, 1) ); // értékadás
```

Fontos, hogy az OSG-t megkérhetjük a régi GLSL attribútumok kitüntetett nevű uniform változókba való átküldését a shader programunkba. Így a shaderünkbe nem nekünk kell kézzel eljuttatni a transzformációs mátrixokat, vertex pozíciókat, színeket stb., hanem csak deklarálnunk kell a megfelelő nevű változókat a shader elejében és bejáráskor majd az OSG ezekbe a nevekbe tölt be értékeket. Példával:

C++ oldalon:

```
// assert: a viewer.realize() már megtörtént;
// különben nem lenne érvényes camera state!
```

```
    osg::State* state =
           viewer.getCamera()->getGraphicsContext()->getState();
    state->setUseModelViewAndProjectionUniforms(true);
    state->setUseVertexAttributeAliasing(true);
```

shaderben:

```
#version 130

in vec4 osg_Vertex;

uniform mat4 osg_ModelViewProjectionMatrix;

void main()
{
     gl_Position = osg_ModelViewProjectionMatrix * osg_Vertex;
}
```

A megfeleltetésekhez tehát egyszerűen az OpenGL GLSL régi beépített változóinak gl_ prefixét kell osg_ prefixre lecserélni:

Táblázatosan (WIP):

	Típus	**GLSL név**	**OSG név**	**Megjegyzés**
Attribútumok	vec4	gl_Vertex	osg_Vertex	
	vec3	gl_Normal	osg_Normal	
	vec4	gl_Color	osg_Color	
	vec4	gl_SecondaryColor	osg_SecondaryColor	
	vec4	gl_MultiTexCoord0	osg_MultiTexCoord0	
	vec4	gl_MultiTexCoord1	osg_MultiTexCoord1	
	vec4	gl_MultiTexCoord2	osg_MultiTexCoord2	
	vec4	gl_MultiTexCoord3	osg_MultiTexCoord3	
	vec4	gl_MultiTexCoord4	osg_MultiTexCoord4	
	vec4	gl_MultiTexCoord5	osg_MultiTexCoord5	
	vec4	gl_MultiTexCoord6	osg_MultiTexCoord6	

	vec4	gl_MultiTexCoord7	osg_MultiTexCoord7	
	float	gl_FogCoord	osg_FogCoord	
Mátrixok	mat4	`gl_ModelViewMatrix`	osg_ModelViewMatrix	
	mat4	gl_ProjectionMatrix	osg_ProjectionMatrix	
	mat4	gl_ModelViewProjectionMatrix	`osg_ModelViewProjectionMatrix`	
	mat4	gl_TextureMatrix[n]		n = gl_MaxTextureCoords
	mat3	gl_NormalMatrix	osg_NormalMatrix	transpose(inverse(world))
	mat4	gl_ModelViewMatrixInverse		
	mat4	gl_ProjectionMatrixInverse		
	mat4	gl_ModelViewProjectionMatrixInverse		
	mat4	gl_TextureMatrixInverse[n]		
	mat4	gl_ModelViewMatrixTranspose		
	mat4	gl_ProjectionMatrixTranspose		
	mat4	gl_ModelViewProjectionMatrixTranspose		
	mat4	gl_TextureMatrixTranspose[n]		
	mat4	gl_ModelViewMatrixInverseTranspose		
	mat4	gl_ProjectionMatrixInverseTranspose		

	mat4	gl_ModelViewProjection MatrixInverseTranspose		
	mat4	gl_TextureMatrixInverse Transpose[n]		
	mat4		osg_ViewMatrixInverse	

Az osg80-osgUtil.dll-ből még ezeket sikerült kibányászni:
osg_ViewMatrix, osg_ViewMatrixInverse, osg_DeltaSimulationTime, osg_SimulationTime, osg_DeltaFrameTime, osg_DeltaFrameTime, osg_FrameTime, osg_FrameNumber

7. fejezet (163.o.)
Nézőpontok

osg::Camera (165.o): az osg::Transform-ból származik, ami a színtérgráfban is használható group node-ként. Négy fő feladata van:
1. a view és projection mátrixok kezelését végzi. Ehhez a következő függvényeket biztosítja:
- setViewMatrix(), setViewMatrixAsLookAt(): vagy egy osg::Matrix, vagy pedig a szokásos nézőpont-nézett_pont-felfelé_mutató_vektor segítségével történő view mátrix megadás
- setProjectionMatrix() (osg::Matrix-ból), illetve setProjectionMatrixAsOrtho(), setProjectionMatrixAsOrtho2D(), setProjectionMatrixAsPerspective() a projection mátrix megadására (ugyanúgy paraméterezendően, mint a glOrtho(), gluPerspective() stb.)
- setViewport() egy osg::Viewport alapján beállítja az ablakon belüli rajzterületet
- példa (166.o.):

```
camera->setViewMatrix( viewMatrix );
camera->setProjectionMatrix( projectionMatrix );
camera->setViewport( new osg::Viewport(x, y, w, h) );
```

- természetesen a megfelelő get*() fv-k is élnek a mátrixok elkérésére:

```
osg::Matrix viewMatrix = camera->getViewMatrix();
```

2. a glClear, glClearColor, glClearDepth parancsokat is az osg::Camera fedi el, a következő eljárásokkal:
- void osg::Camera::setClearMask(GLbitfield mask): a default mask a GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT
- void osg::Camera::setClearStencil(int stencil)
- void osg::Camera::setClearColor(const osg::Vec4& color)
- void osg::Camera::setClearDepth(double depth): default 1
- void osg::Camera::setClearAccum(const osg::Vec4 &color)

3. Magát az OpenGL context-et is az osg::Camera kezeli!
4. A textúrába renderelést is a kamera kezeli

Minden színtérgráfban legalább egy osg::Camera node szerepel, a főkamera. Ha osgViewer::Viewer-t használunk, akkor a neki beállított root node-ot automatikusan a kamera

gyerekévé teszi.

Ha több kamera van a színtérben, akkor köztük egy renderelési sorrendet kell felállítani. Erre való az

osg::Camera::setRenderOrder(RenderOrder order**, int** orderNum=0**)**: függvény, ahol a RenderOrder-ben a PRE_RENDER, NESTED_RENDER és POST_RENDER segítségével azt mondhatjuk meg, hogy a frame tényleges kirajzolása előtt, közben, vagy után történjen a kamera képének elkészítése. A második paraméter az adott rajzolási csoporton belüli sorrendet határozza meg.

A PRE_RENDER elsősorban a render-to-texture-re szokott előfordulni, a PRE_RENDER képeit elfedik a későbbiek.

A POST_RENDER-hez praktikus setClearMask()-ot használni, hogy ne írjuk felül a kép eredményét mindenhol, HUD/GUI kirajzolás egy tipikus alkalmazás POST_RENDER-hez.

HUD kamerára példa:

```
osg::ref_ptr<osg::Camera> camera = new osg::Camera;
camera->setClearMask( GL_DEPTH_BUFFER_BIT );
camera->setRenderOrder( osg::Camera::POST_RENDER );

camera->setReferenceFrame( osg::Camera::ABSOLUTE_RF );
camera->setViewMatrixAsLookAt(
           osg::Vec3(0.0f,-5.0f,5.0f),
           osg::Vec3(),
           osg::Vec3(0.0f,1.0f,1.0f) );
camera->addChild( hud_model.get() );

osg::ref_ptr<osg::Group> root = new osg::Group;
root->addChild( model.get() );
root->addChild( camera.get() );
```

Ekkor a hud_model-be betöltött modellre nem fog hatni a viewer-ben történő egerészés.

setReferenceFrame(**ReferenceFrame** rf): az osg::Camera egyik őséből, az osg::Transform-ból származó eljárás, amivel beállíthatjuk, hogy az adott csúcs milyen KR-ben legyen:
- RELATIVE_RF (default): a szülő koordinátarendszerében él ez a node
- ABSOLUTE_RF: abszolút KR-ben értendőek a node transzformációi (ez egyúttal beállítja a CullingActive flaget a trafós node-on)
- ABSOLUTE_RF_INHERIT_VIEWPOINT: mint a sima abszolút, de a szülő csúcs KR-jének origója (nézőpont) is elérhető

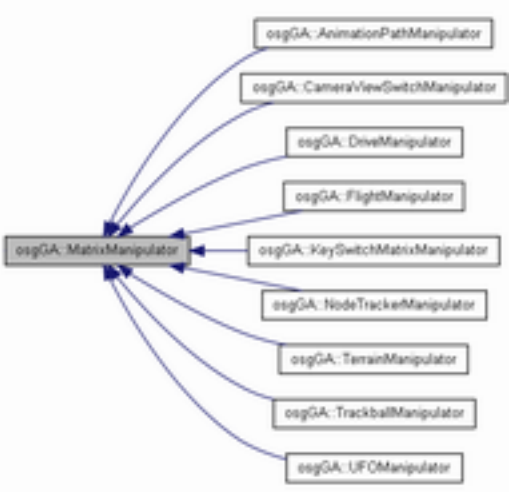
osgViewer::Viewer (170. o)

Belső adattagként rendelkezik egy főkamerával, amely a színtérgráfjának gyökere lesz. A **setSceneData()**-val átadott részfát ennek a főkamerának szúrja be gyermekként. A kamerát a Viewer osgGA::CameraManipulator

A **run()** eljárással indul el az ún. *szimulációs ciklus*, amelyben három fő feladata van a Viewernek:
- az osgGA::CameraManipulator beállítását
- elkészíti a szükséges grafikai context-eket
- és ezután egy végtelen ciklusban elkezdi kirajzolni a színteret

Az osgGA::CameraManipulator a felhasználói billentyűzet illetve egéreseményeken keresztül módosítja a kamera nézeti mátrixát. Beállítására szolgál az

osgViewer::View::setCameraManipulator(**osgGA::MatrixManipulator*** manipulator) (171.o): ahol a manipulátorok a következők és utána rövide leírás is jön:



Manipulator class	Description	Basic usage
`DriveManipulator`	Drive-like simulator	Key space: reset the viewer position Mouse moving: changes the viewer's orientation Mouse dragging: the left button accelerates, the right decelerates, and the middle stops the navigation
`FlightManipulator`	Flight simulator	Key space: reset the viewer position Mouse moving: changes the viewer's position and orientation
`KeySwitchMatrixManipulator`	A decorator allowing different manipulators to be switched	Use `addMatrixManipulator()` to add a manipulator and switch to it by pressing the specified key on the fly, for instance: `addMatrixManipulator('1', "trackball", new osgGA::TrackballManipulator);`
`NodeTrackerManipulator`	A manipulator tracking a node	Use `setTrackNode()` to select a node to track before starting
`SphericalManipulator`	A manipulator for browsing spherical objects	Key space: reset the viewer position Mouse dragging: the left mouse button rotates the viewer, the middle mouse button pans the world, and the right mouse button scales the world
`TerrainManipulator`	An enhanced trackball-like manipulator for viewing terrains	Key space: reset the viewer position. Mouse dragging: the left mouse button rotates the viewer, the middle mouse button pans the world, and the right mouse button scales the world
`TrackballManipulator`	The default trackball manipulator	Key space: reset the viewer position. Mouse dragging: the left mouse button rotates the viewer, the middle mouse button pans the world, and the right mouse button scales the world

A **realize()** metódus inicializálja a context-et, a megjelenítéshez szükséges erőforrásokat és a szálakat. A **frame()** az aktuális frame kirajzolását végzi, a monitor vsync-jéhez igazodva (ha úgy akarjuk). A **done()** függvény visszatérési értéke azt jelöli, hogy véget kell-e érjen a renderelési ciklus (mert pl. a user ki akar lépni, vagy mi magunk állítottuk be valahol ezt true-ra a **setDone()** függvénnyel). Tehát a kirajzolási ciklus így írható fel:

```
while ( !viewer.done() )
{
      viewer.frame();
}
```

void setRunFrameScheme(FrameScheme fs): a kirajzolás sűrűségét adhatjuk meg vele. A ViewerBase-ben deklarált, de az online dokumentációból egyelőre hiányzik. A FramScheme egy enum, aminek a következő két értke lehet:
- ON_DEMAND: csak akkor történik rajzolás, ha módosul a színtérgráf
- CONTINUOUS: folyamatosan újrarajzoljuk a színteret

setRunMaxFrameRate(double frameRate): meghatározhatjuk vele, hogy mekkora legyen a maximális frame-rate.

getFrameStamp(): visszaadja az aktuális frame-t leíró statisztikákat egy osg::FrameStamp-ben. A számunkra érdekes get függvények:
- **getFrameNumber**()
- **getReferenceTime**()

Több megjelenítő használata (175.o.)

osg::View-öket kell létrehozni és egy osgViewer::CompositeViewer-be belepakolni:

```
...
osg::ref_ptr<osgViewer::View> view1 = new osgViewer::View;
view1->setSceneData( root );
view1->setUpViewInWindow( 50, 50, 400, 400 );

osg::ref_ptr<osgViewer::View> view2 = new osgViewer::View;
view2->setSceneData( root );
view2->setUpViewInWindow( 500, 50, 400, 400 );

osgViewer::CompositeViewer viewer;
viewer.addView( view1 );
viewer.addView( view2 );

return viewer.run();
```

osg::DisplaySettings (179.o.): singleton, a renderelés globális attribútumainak beállítására, pl.:

```
osg::DisplaySettings::instance()->setNumMultiSamples( 4 );
```

8. fejezet - animáció (193.o.)

Az **update** bejárást fogjuk felhasználni arra, hogy animáljuk a színterünk objektumait, illetve még az **event** bejárást használjuk a felhasználói interkaciók kezelésénél. Ezen, és a többi (pl. cull) bejárás egy fontos ősosztálya a következő:

osg::NodeCallBack (194.o.): csak node-okhoz csatlakoztatható. A virtuális **operator()**-t kell felülírnia a felhasználónak amikor származtat ebből az osztályból, hogy a saját update függvényét definiálja a node-nak, amihez hozzárendeli ezt a callback-et (amit vagy a node setUpdateCallback vagy addUpdateCallback eljárásával tehetünk meg).

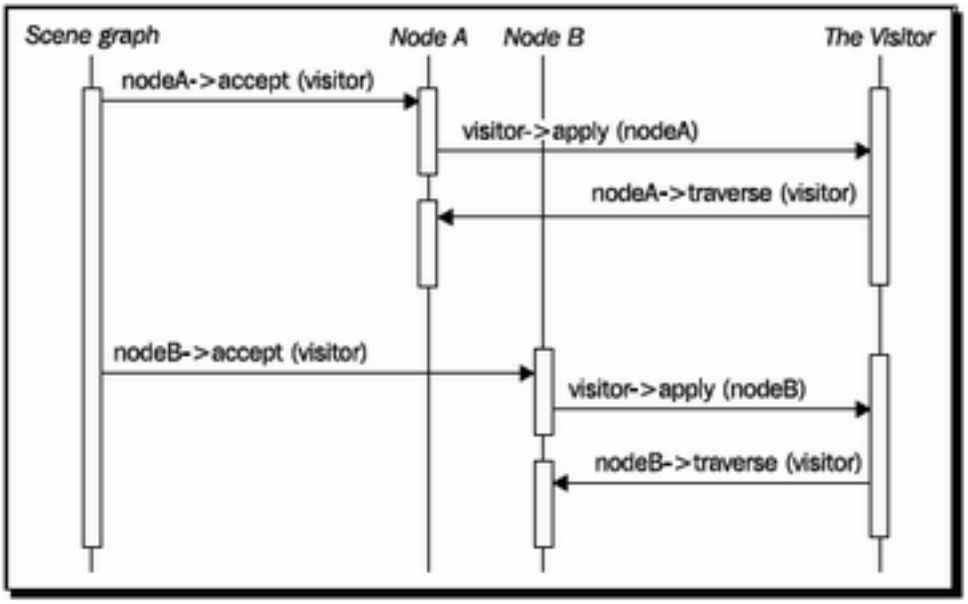
osg::Drawable::UpdateCallback, **osg::Drawable::EventCallback**, **osg::Drawable::CullCallback**: az osg::Drawable-ök callbackjei.

Az, hogy milyen a callback szignatúra (azaz: mi az a virtuális függvény, amit felül kell definiálnunk) függ a konkrét típustól is. Ez táblázatosan a könyvből:

Name	Callback functor	Virtual method	Attached to
Update callback	`osg::NodeCallback`	`operator()`	`osg::Node::setUpdateCallback()`
Event callback	`osg::NodeCallback`	`operator()`	`osg::Node::setEventCallback()`
Cull callback	`osg::NodeCallback`	`operator()`	`osg::Node::setCullCallback()`
Drawable update callback	`osg::Drawable::UpdateCallback`	`update()`	`osg::Drawable::setUpdateCallback()`

Name	Callback functor	Virtual method	Attached to
Drawable event callback	`osg::Drawable::EventCallback`	`event()`	`osg::Drawable::setEventCallback()`
Drawable cull callback	`osg::Drawable::CullCallback`	`cull()`	`osg::Drawable::setCullCallback()`
State attribute update callback	`osg::StateAttributeCallback`	`operator()`	`osg::StateAttribute::setUpdateCallback()`
State attribute event callback	`osg::StateAttributeCallback`	`operator()`	`osg::StateAttribute::setEventCallback()`
Uniform update callback	`osg::Uniform::Callback`	`operator()`	`osg::Uniform::setUpdateCallback`
Uniform event callback	`osg::Uniform::Callback`	`operator()`	`osg::Uniform::setEventCallback`
Camera callback before drawing the sub-graph	`osg::Camera::DrawCallback`	`operator()`	`osg::Camera::setPreDrawCallback()`
Camera callback after drawing the sub-graph	`osg::Camera::DrawCallback`	`operator()`	`osg::Camera::setPostDrawCallback()`

És ami nekünk még nagyon fontos:
osg::Uniform::Uniform::Callback: aminek a felülírandó művelete:
- **operator()** (**Uniform** *, **NodeVisitor** *)



Van ilyen.

9. fejezet (231. o.)

osgGA: OSG GUI absztrakciós rétege.

osgGA::GUIEventHandler (232.o.): eseménykezelő ősosztály, a felhasználói inputot a scene viewer-hez addható hozzá az addEventHandler() utasítással, illetve vehető ki a removeEventHandler()-rel. Az **eseménybejárás** (event traversal) során mindig meghívásra kerül. A handle() eljárását felül kell írni!

```
bool handle(
      const osgGA::GUIEventAdapter& ea,
      osgGA::GUIActionAdapter& aa )
{
… // concrete operations
}
```

osgGA::GUIActionAdapter: az esemény a GUI felé küldhet kéréseket. Legtöbbször ez a Viewer, szóval ilyet látni fogunk:

```
osgViewer::Viewer* viewer = dynamic_cast<osgViewer::Viewer*>(&aa);
```

osgGA::GUIEventAdapter (233. o.): az OSG által támogatott összes esemény ezen keresztül kezelhető, scrollozás, billentyűleütések, egerészés stb.

osgGA::EventQueue (239.o.): az eseménysor, amit FIFO alapon dolgoz fel a rendszer és sorban meghívja a várakozók handle() eljárását, megfelelően felparaméterezve őket. A viewer üzenetsorára mutató pointer a viewer.getEventQueue() segítségével érhető el.

Új esemény hozzáadása pl.:

```
viewer.getEventQueue()->userEvent( data );
```

Matematikai osztályok

osg::Vec<2,3,4> (ezek float-ok), osg::Vec<2,3,4>d (double)

Egyéb megjegyzések
Visual Studio:
- Debugging/Environment beállításoknál **NINCS szóköz, vessző, semmi** a PATH, =
és %PATH% között! Tehát így néz ki: PATH=%PATH%

Fontos OSG-hez:
PATH=%PATH%;D:\Dev\SDKs\OpenSceneGraph-3.0.1-x86-dnr\bin_debug
(vagy release)
OSG_FILE_PATH=D:\Dev\SDKs\OpenSceneGraph-3.0.1-x86-dnr\data (ez a Viewer-hez!)

Wizard: http://www.openscenegraph.org/projects/osg/wiki/Support/TipsAndTricks
Statikus linkeléshez információk: http://www.openscenegraph.org/projects/osg/wiki/Community/Tasks/Win32StaticLink
osgvertexattributes
here is an example of how to setup your custom geometry (even for morphing) using VBOs
http://www.openscenegraph.org/projects/osg/wiki/Support/Tutorials
Look at the External examples, for osgGPUMorph.3.zip
http://www.openscenegraph.org/projects/osg/attachment/wiki/Support/Tutorials/osgGPUMorph.3.zip
https://groups.google.com/group/osg-users/tree/browse_frm/month/2012-4/df93ae3082da2181?rnum=301&start=250&_done=/group/osg-users/browse_frm/month/2012-4?start%3D250%26sa%3DN%26&pli=1
osgsimplegl3: http://webcache.googleusercontent.com/search?q=cache:j1QESOkMHi4J:www.openscenegraph.org/projects/osg/browser/OpenSceneGraph/trunk/examples/osgsimplegl3/osgsimplegl3.cpp%3Frev%3D12973+&cd=3&hl=hu&ct=clnk&gl=hu

realize(): ő inicializálja a viewer context-jeit és erőforrásait

osgconv: modell fájl konvertáló alkalmazás, pl. 3ds fájlokat is képes osg formátumra konvertálni
www.dependencywalker: egy alkalmazáshoz szükséges egyéb fájlok kiderítésére használható util
osg::ref_ptr<>: smart pointer
osg::Node: a színtérgráf egy csúcsa