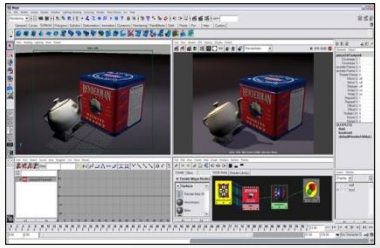
Szenengraphen

Virtual Reality
Sommer 2010

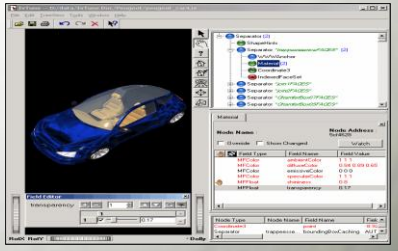
Rendering Systeme

- OpenGL und DirectX sind **Low-Level APIs**
  - OpenGL bietet nur Handle-basierte Schnittstellen
  - DirectX ist Objekt-orientiert, spezieller: Komponenten-orientiert via COM Interfaces bietet aber keine High-Level Objekte
- Low-Level APIs bieten eine geschlossene **Abstraktion zur Hardware**, jedoch keine Funktionale Abstraktion.
- Rendering Systeme kapseln **Applikationsfunktionalität:**

Cry Engine:
Realzeit Rendering
und Animation



Renderman:
Hochqualitatives
Offline-Rendering



Open Inventor:
Interaktive Visualisierung

Rendering Systeme

- System Erweiterbarkeit:
  ("Component Software. Beyond Object-Oriented Programming", von C. Szyperski,1998)
  ("A Generic Rendering System", von Döllner et al., 2002)

  White-Box Systeme (Generic 3D,Vision)
  - Erweiterbare Frameworks
  - Redesign möglich (Zugang zu Quelltext)

  Black-Box Systeme (Renderman,POV-Ray,Nvidia SceniX)
  - Verstecken die Softwarearchitektur über Interfaces
  - Nur sehr begrenzt erweiterbar

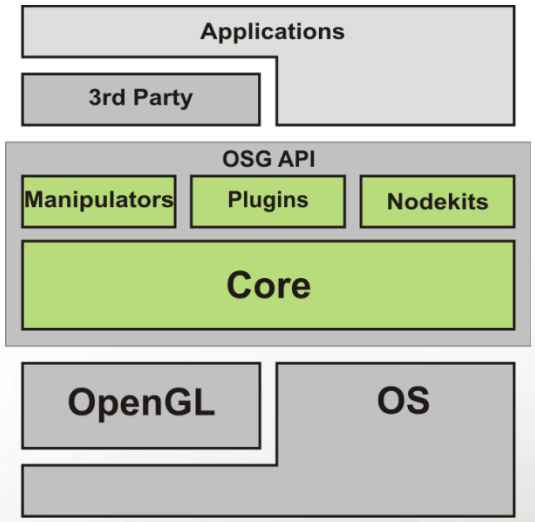
  Grey-Box Systeme (OpenSceneGraph, OpenInventor)
  - Sind applikations-spezifisch erweiterbar, erlauben jedoch kaum System-Änderungen.
  - Zum Teil Quelltext verfügbar

Szenenkomplexität

- Softwaretechnische Anforderungen von großen Szenen:
  - Abstraktion von der Rendering Pipeline
  - Ressourcen-Management
  - Hierarchische Beziehungen zwischen Objekten (Szenengraphstruktur)
  - Dynamische Struktur
  - Erweiterbarkeit und Import/Export

OpenSceneGraph (OSG)

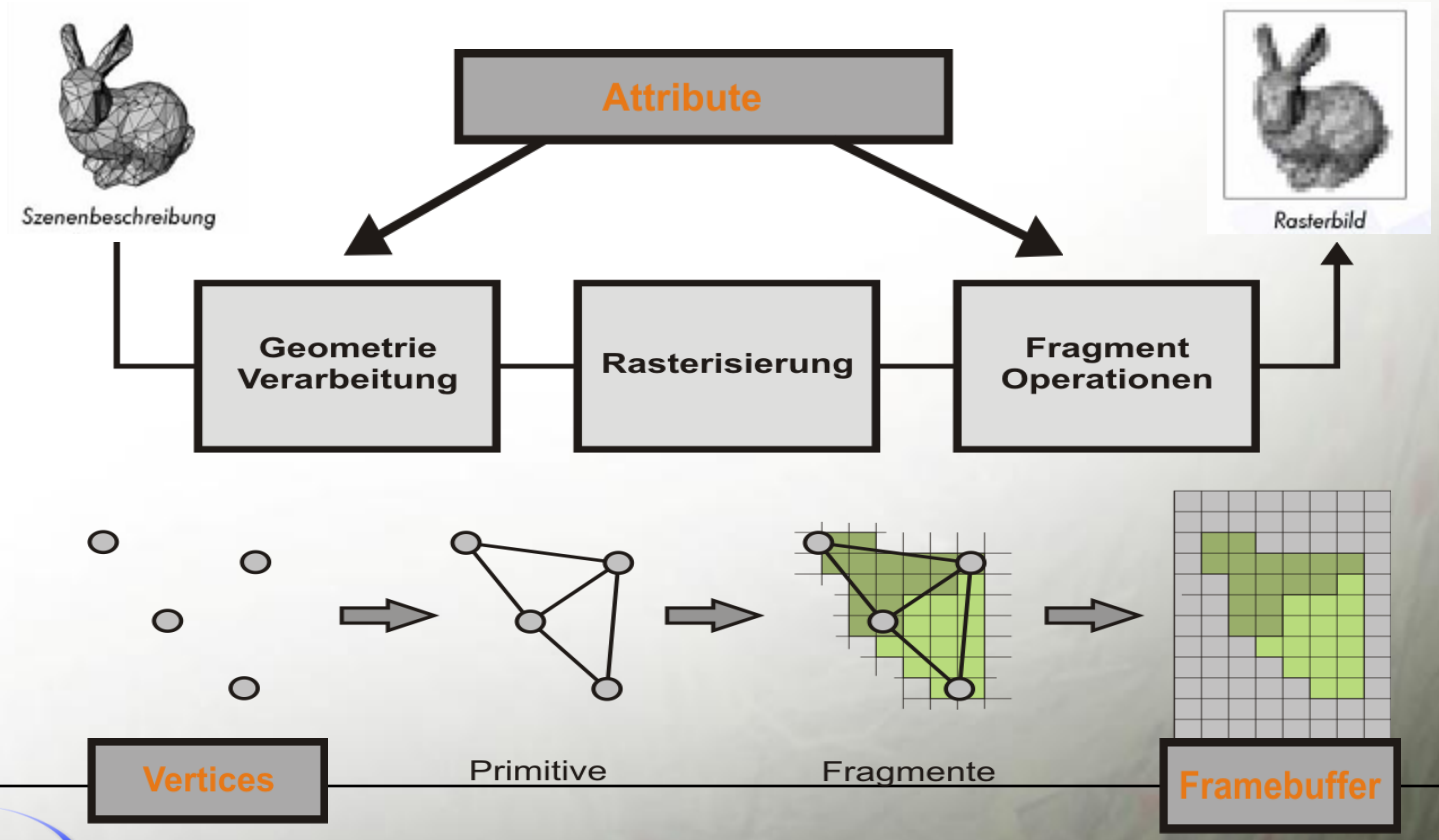
- Gray-Box System von Robert Osfield und Don Burns (Ursprünge bei Performer)
  ("OpenSceneGraph Quick Start Guide", von Paul Martz, 2007)
- Kleiner betriebsystem-unabhängiger Kern als **Facade** zur OpenGL-Funktionalität
  ("Design Patterns", von Gamma et al.,1993)




```
int main(int, char **)
{
    osg::Node* cow = osgDB::readNodeFile("cow.osg");

    osg::Group* root = new osg::Group;
    root->addChild( cow );

    osgViewer::Viewer viewer;
    viewer.getCamera()->setClearColor(osg::Vec4(1,1,1,1));
    viewer.setSceneData( root );

    return viewer.run();
}
```

Rendering Pipeline

- Rendering Pipeline erzeugt ein Rasterbild aus der Szenenbeschreibung
- Attribute (Matrizen, Texturen, Programe) und Vertices sind die Parameter der Rendering Pipeline. Ergebnisse landen im Framebuffer.



Rendering Pipeline

- Vertexdaten in OpenGL via Listen (CG 1) oder Buffer Objects unhandlich.

```
glBegin( GL_TRIANGLES );
    glColor4f(184.0f/255.0f,219.0f/255.0f,124.0f/255.0f,1.0f);
    glNormal3f(0.0f,-1.0f,0.0f);
    glVertex3f(-1.12056, -2.15188e-09, -0.840418);
    glVertex3f(-0.95165, -2.15188e-09, -0.840418);
    glVertex3f(-1.11644, 9.18133e-09, -0.716827);
    glVertex3f(-0.840418, 9.18133e-09, -0.778623);
    glVertex3f(-0.622074, 9.18133e-09, -0.613835);
    glVertex3f(-1.067, 9.18133e-09, -0.609715);
glEnd();
```

```
float data[] = {
-1.12056, -2.15188e-09, -0.840418, -0.95165,
-2.15188e-09, -0.840418, -1.11644, 9.18133e-09,
-0.716827, -0.840418, 9.18133e-09, -0.778623, -0.622074,
9.18133e-09, -0.613835, 0.0f, -1.0f, 0.0f, 0.0f, -1.0f,
0.0f, 0.0f, -1.0f, 0.0f, 0.0f, -1.0f, 0.0f, 0.0f, -1.0f, 0.0f,
0.0f, -1.0f, 0.0f, 184.0f/255.0f, 219.0f/255.0f, 124.0f/255.0f,
1.0f, 184.0f/255.0f, 219.0f/255.0f, 124.0f/255.0f, 1.0f,
184.0f/255.0f, 219.0f/255.0f, 124.0f/255.0f, 1.0f,
184.0f/255.0f, 219.0f/255.0f, 124.0f/255.0f, 1.0f, 184.0f/255.0f,
219.0f/255.0f, 124.0f/255.0f, 1.0f, 184.0f/255.0f, 219.0f/255.0f,
124.0f/255.0f, 1.0f );

GLuint vertexOffset = 0;
GLuint normalOffset = 6 * 3 * sizeof(float);
GLuint colorOffset = 6 * (3+3) * sizeof(float);

GLuint vbo;
glGenBuffers(1, &vbo);
glBindBuffer(GL_ARRAY_BUFFER_ARB, vbo);
glBufferData(GL_ARRAY_BUFFER_ARB, 6 * (3+3+4) * sizeof(float), data,
GL_STATIC_DRAW_ARB);

glEnableClientState(GL_VERTEX_ARRAY);
glVertexPointer( 3, GL_FLOAT, 3*sizeof(float), &vertexOffset );
glEnableClientState(GL_NORMAL_ARRAY);
glNormalPointer( GL_FLOAT, 3*sizeof(float), &normalOffset );
glEnableClientState(GL_COLOR_ARRAY);
glColorPointer( GL_FLOAT, 4*sizeof(float), &colorOffset );

glDrawArrays( GL_TRIANGLES, 0, 6 );
```

Rendering Pipeline (OSG)

- Entitäten, wie osg::Drawable, osg::Program, osg::Texture usw., abstrahieren OpenGL Funktionalität. (siehe QSG ab S.38)
- **osg::Geometry** eine mögliche Spezialisierung von osg::Drawable
- Methode kann variieren, jedoch Repräsentation bleibt gleich.

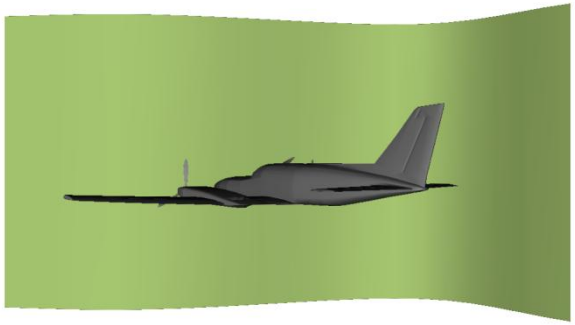
```
osg::Geometry* myGeom = new osg::Geometry();

osg::Vec3Array* vertices = new osg::Vec3Array(6);
(*vertices)[0].set(-1.12056, -2.15188e-09, -0.840418);
(*vertices)[1].set(-0.95165, -2.15188e-09, -0.840418);
(*vertices)[2].set(-1.11644, 9.18133e-09, -0.716827);
(*vertices)[3].set(-0.840418, 9.18133e-09, -0.778623);
(*vertices)[4].set(-0.622074, 9.18133e-09, -0.613835);
(*vertices)[5].set(-1.067, 9.18133e-09, -0.609715);
myGeom->setVertexArray(vertices);

osg::Vec4Array* colors = new osg::Vec4Array(1);
(*colors)[0].set(184.0f/255.0f,219.0f/255.0f,124.0f/255.0f,1.0f);
myGeom->setColorArray(colors);
myGeom->setColorBinding(osg::Geometry::BIND_OVERALL);

osg::Vec3Array* normals = new osg::Vec3Array(1);
(*normals)[0].set(0.0f,-1.0f,0.0f));
myGeom->setNormalArray(normals);
myGeom->setNormalBinding(osg::Geometry::BIND_OVERALL);

myGeom->addPrimitiveSet(
        new osg::DrawArrays(osg::PrimitiveSet::TRIANGLES,0,6));
```

Rendering Pipeline (OSG)

- Pipeline Attribute, wie Programme und Texturen werden über Objekte vom Typ **osg::StateSet** aggregiert (**State Pattern**). (”Design Patterns”, von Gamma et al.,1993)
- Abstraktion von “Low-Level” Pipeline Zuständen

```
osg::Texture2D* tex = new osg::Texture2D;
tex->setImage( osgDB::readImageFile( ''airplane.tif'' ) );
```

```
osg::Program* program = new osg::Program;
program->addShader( osg::Shader::readShaderFile( osg::Shader::VERTEX,
"MyVertexShader.glsl" ) );
program->addShader( osg::Shader::readShaderFile( osg::Shader::FRAGMENT,
"MyFragmentShader.glsl" ) );
```



```
osg::StateSet* ss = new osg::StateSet;
ss->setAttributeAndModes( program, osg::StateAttribute::ON );
ss->setTextureAttribute( 0, tex );
ss->addUniform(
    new osg::Uniform("MyTexture", 0) );
ss->addUniform(
    new osg::Uniform("MyColor",
        osg::Vec3(184.0f/255.0f,219.0f/255.0f,124.0f/255.0f,1.0f) );
```

Ressourcen-Management

- Smart Pointer:
  (“More Effective C++“, Scott Meyers, 1996)
  (siehe QSG ab S. 31)
  - **Template**-Objekte die wie Zeiger funktionieren.

```
{ // Stack push
 osg::ref_ptr<osg::Geometry> geomPtr = new osg::Geometry;
 geomPtr->setVertexArray( vertices );
} // Stack pop: Deletes geomPtr
```

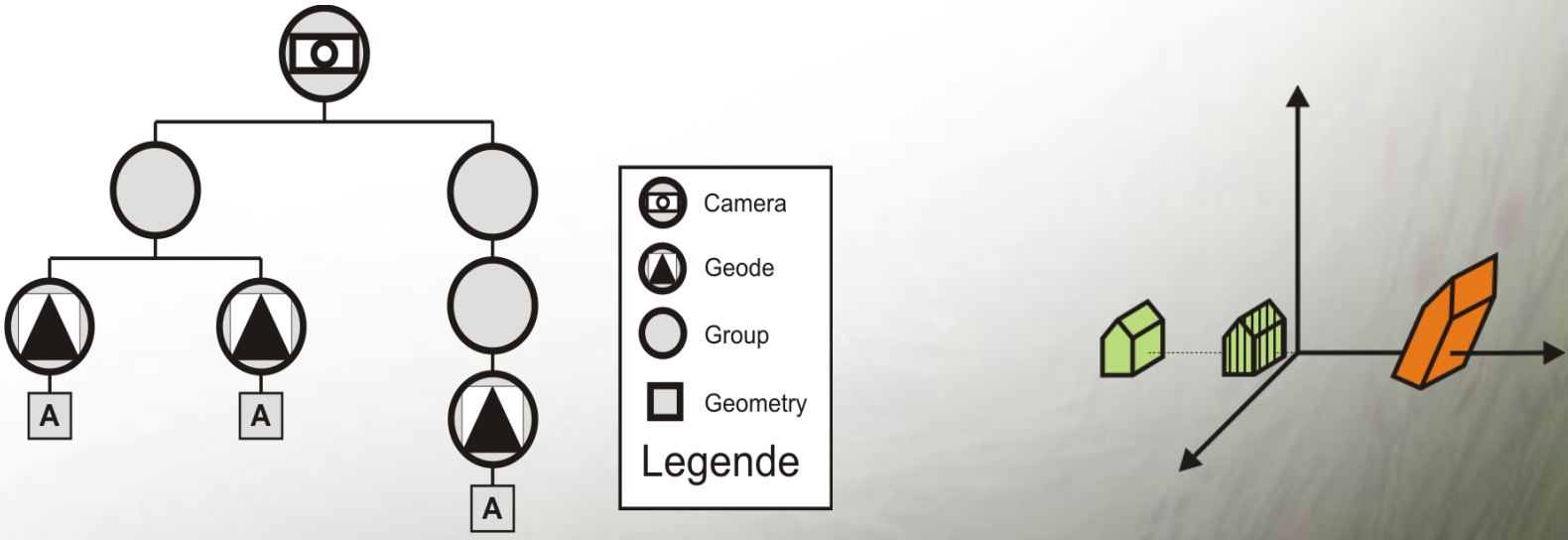
  - **Referenzsystem** klärt Besitzverhältnisse bei verteilten Objekten.

```
osg::ref_ptr<osg::Geometry> glPtr = NULL;
{ // Stack push
 osg::ref_ptr<osg::Geometry> geomPtr = new osg::Geometry;
 geomPtr->setVertexArray( vertices );

glPtr = geomPtr; // Increase reference count by 1
} // Stack pop: Does not delete geomPtr
```

  - Speicher wird erst on demand allokiert (**Lazy Evaluation**).

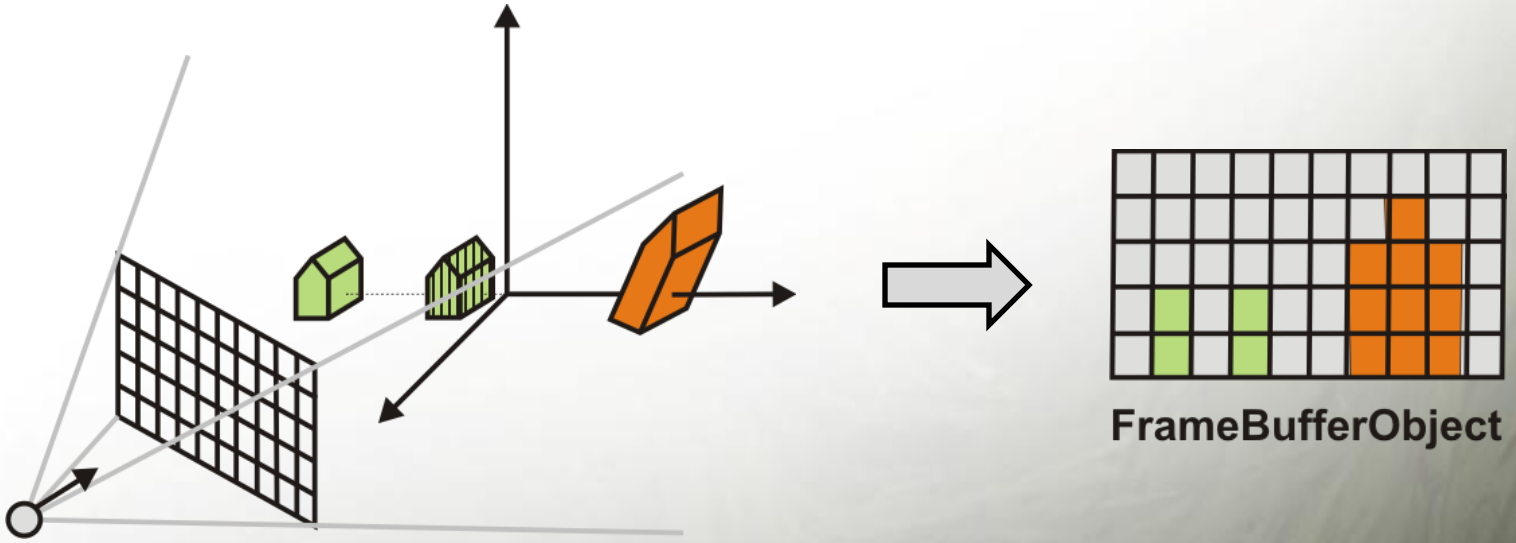
Szenengraphen

- Bisher nur elementare Grafik Entitäten
- Nächster Schritt: Aggregation dieser Objekte in einer Graphenstruktur (Szene).
  ("An Object-Oriented 3D Graphics Toolkit", Strauss et al., 1992)
- Geometry Nodes beinhaltet renderbare Objekte
- Group Nodes sind für die Hierarchie zuständig und gruppieren Nodes
- Camera Nodes definieren den Frame



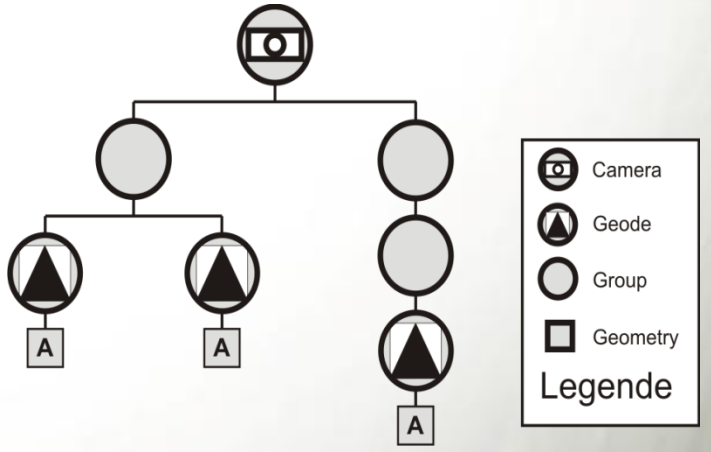
Szenengraphen

- Kameras definieren die Projektion:
  - Viewing-Matrix,
  - Projection-Matrix
  - Viewport
- Ergebniss-Pixel können in Textur geschrieben werden (FrameBufferObjects).
  (Siehe osgPreRender Beispiel im SDK)



Gerichtete Azyklische Graphen

- Gerichteter azyklischer Graph (DAG):
  - Viele spezialisierte Nodes (Geode, Group, Switch usw.)
  - Mehrfach Instanzierung
  - Keine Zyklen
  - Bestimmte Verarbeitungs-Reihenfolge

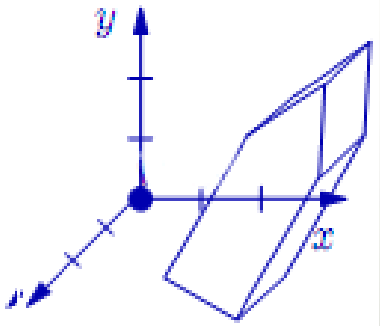


- Konkrete Aufgaben der Softwarestruktur:
  - Bewegung von Objekten (Verschieben, Drehen, Skalieren)
  - Eigenschaftszuweisung an Gruppen von Objekten
  - Flexible Zusammenstellung von Modellen. (Ein Koordinatensystem zu unhandlich)

Hierarchien

- Kumulierung affiner Transformationen (siehe CG 1):

$$P^{T}_{WC} = P^{T}_{MC}\, S_{(1,2,1)}\, R_{(z,-30.0^{\circ})}\, T_{(2,0,0)}$$

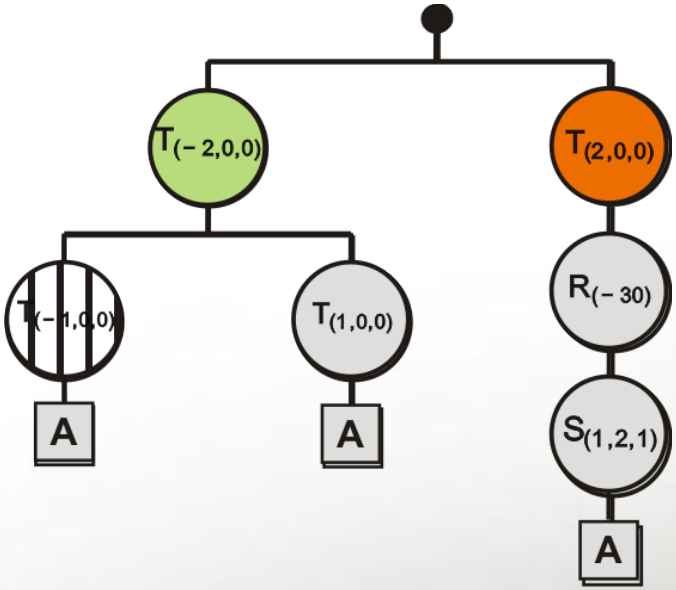


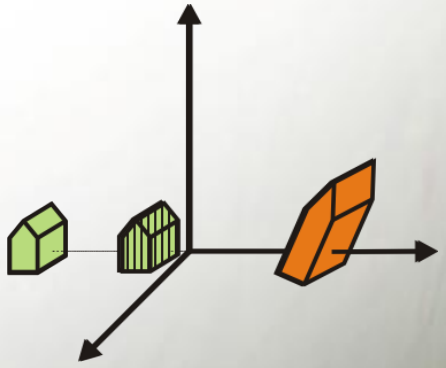
OSG verwendet standardmäßig diese Transponierte Form
(siehe QSG ab S.50)

```
osg::Matrix S = osg::Matrix::scale(osg::Vec3f(1,2,1));
osg::Matrix R = osg::Matrix::rotate(osg::Math::inRadians(-30),osg::Vec3f(0,0,1));
osg::Matrix T = osg::Matrix::translate(osg::Vec3f());

osg::Matrix LtoW = S * R * T;
```

Hierarchien

- Transformationshierachie in Graphennotation
- Nodes akkumulieren Pipeline Zustände (Für OSG: StateSet)
  Vererbung von zusätzlichen Eigenschaften (Für OSG: Texture,Material)

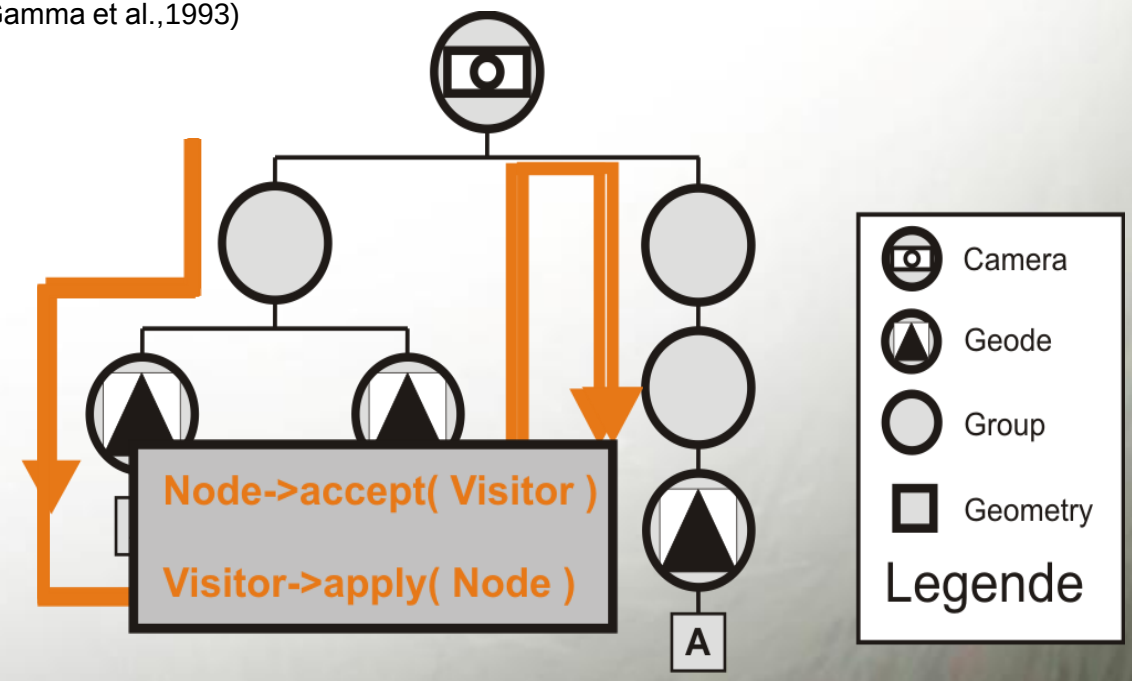




Traversierung

- Abarbeitung des Graphen in festgelegter Reihenfolge:
  ("A Flexible and Extensible Traversal Framework for Scenegraph Systems", Reiners, 2002)

  Pre-Order, In-Order, Post-Order oder Breadth-First Traversierung
- Traversierung mittels gegenseitigem Aufruf (**Double-Dispatch**) von Nodes und Traversierungsobjekten (z.B. **Visitor Pattern**)
  ("Design Patterns", von Gamma et al.,1993)



Traversierung (OSG)

- Dynamischer Graph erfordert Verarbeitung in mehreren Zyklen (pro Frame):

EventHandling

(1) Translate OS Events to osg::Events

(2) For each event:
EventVisitor->apply(RootNode)

(3) Call EventHandlers

Update

(4) UpdateVisitor->apply(RootNode)

(5) Run update operations

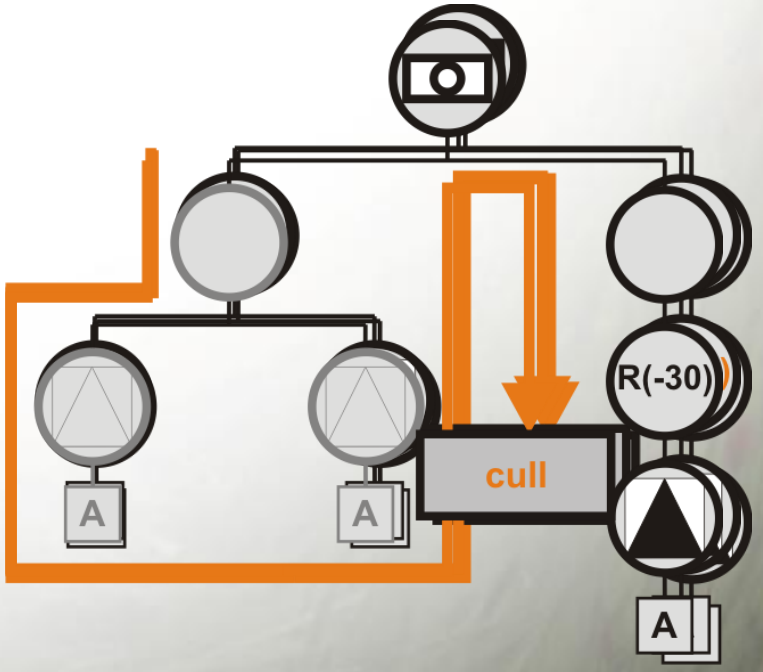
Cull

(5) Reset osg::State

(6) CullVisitor->apply(RootNode)

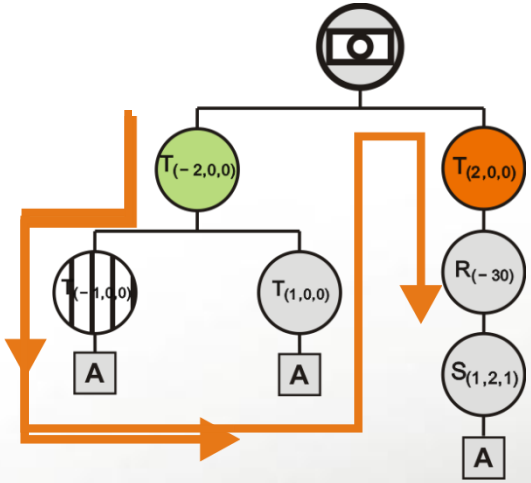
Render

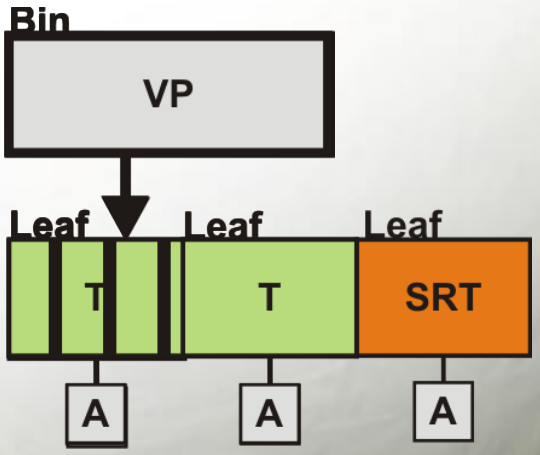
(7) Traverse RenderBins



Rendering (OSG)

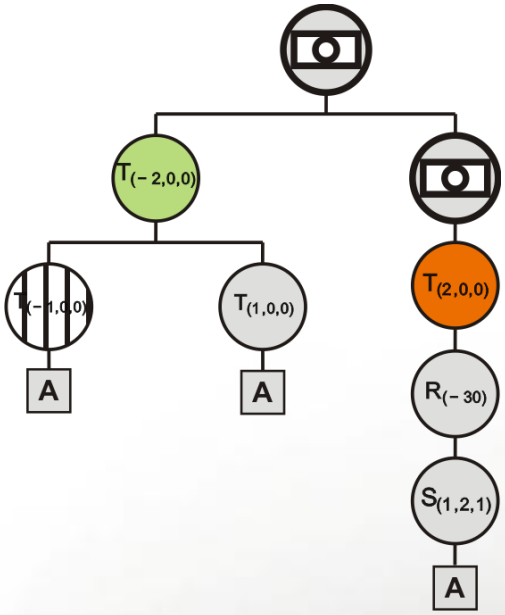
- Was ist nun mit der Entscheidenden Traversierung, dem Rendering?
- CullVisitor baut optimierte Rendering-Struktur aus RenderBins und RenderLeafs auf.
- RenderLeafs speichern eigenen Pipeline Zustand
- Die RenderBins können beliebig sortiert werden (z.B. nach Tiefe)

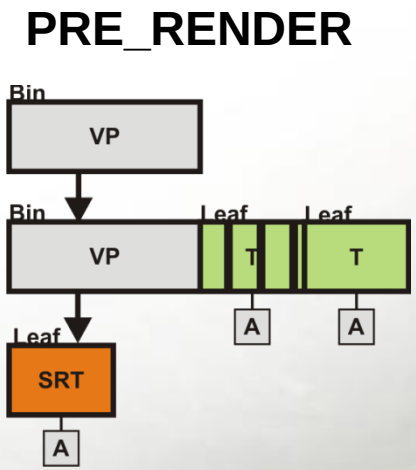


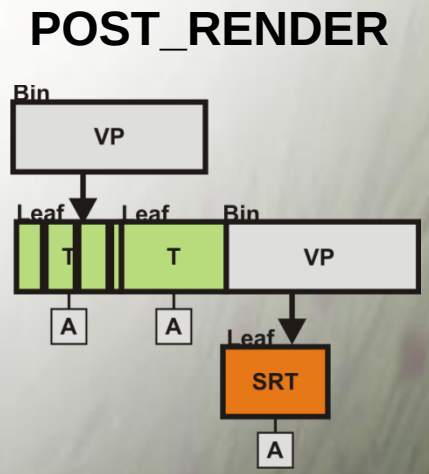


Rendering (OSG)

- Auch eine Anordnung mehrerer hierachischer Bins ist möglich:
  - PRE_RENDER und POST_RENDER Flags bestimmen die Ausführungsreihenfolge
- Ausführung in **PreOrder** Reihenfolge

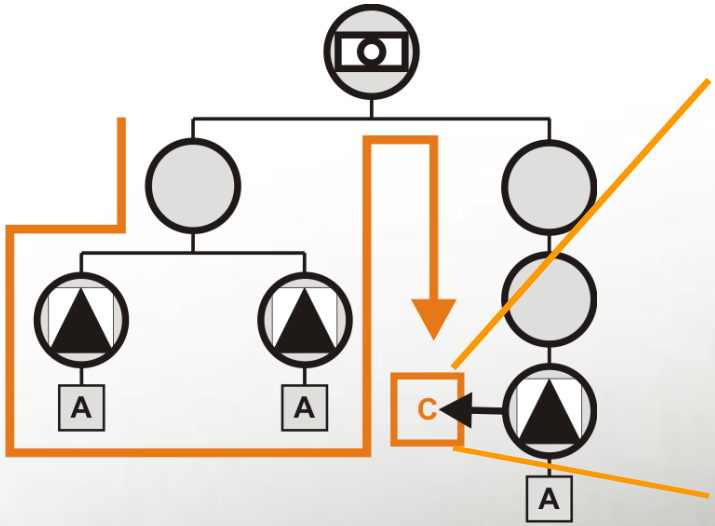






Callbacks

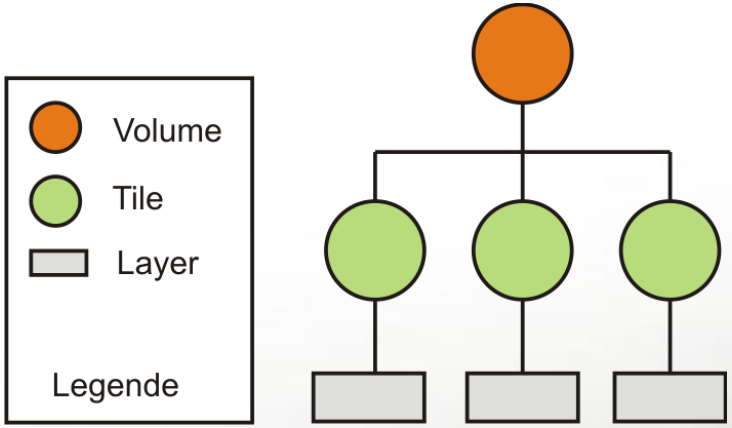
- Callbacks (**Strategy Pattern**) zur Ausführung eigener Logik zum Zeitpunkt der Traversierung:
  ("Design Patterns", von Gamma et al.,1993)
  - Node-spezifische Eventverarbeitung
  - Anpassung der Graphstruktur
  - Applikationsspezifisches Renderingverhalten
- Lokalitätsprinzip:
  Wird der Node entfernt oder deaktiviert ist die entsprechende Verarbeitungslogik ausgeschaltet.

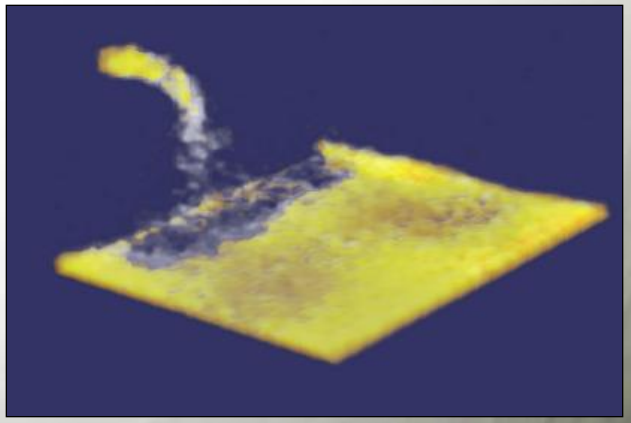



```
virtual void MyCallback::operator()
        (osg::Node* node,osg::NodeVisitor* nv)
{
    osgGA::EventVisitor* ev =
        dynamic_cast<osgGA::EventVisitor*>( nv );
    if( !ev || !ev->getActionAdapter() )
        return;

    osgGA::GUIActionAdapter* ea = ev->getActionAdapter();
    if( ea->getEventType() == osgGA::GUIEventAdapter::KEYUP
            && ea->getKey() == 'p' )
        osg::notify(osg::INFO)<< node->getName() << std::endl;
}
```

Nodekits

- Ein Nodekit erlaubt den Satz von existierenden Nodes zu erweitern.
  ("An Object-Oriented 3D Graphics Toolkit", Strauss et al., 1992)
- Geben neue strukturelle Richtlinien/Merkmale für Erweiterungen vor.
- Kapseln Graphenstrukturen. (Bsp. osgAnimation, osgVolume, etc.)

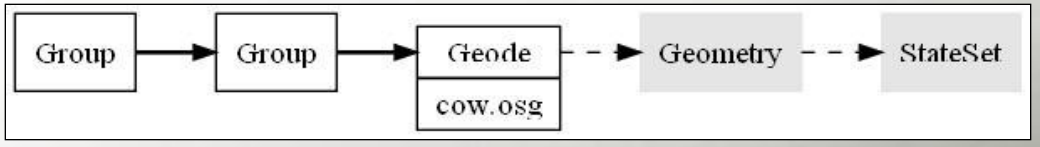




Import/Export

- Kommunikation mit anderen Tools ist bedeutend für die Softwareentwicklung: Große Szenen enstehen in Modellierungstools wie Maya.
- In OSG via Plugins für osgDB (OSG Database library)
- Endung entscheidet zur Laufzeit über zu ladende Bibliothek (**Dynamische Bibliotheken**)

```
int main( int, char** )
{
    osg::ref_ptr<osg::Node> root = createSceneGraph();
    bool result = osgDB::writeNodeFile(*root,"Simple.dot");

    if ( !result )
        osg::notify(osg::FATAL)
            << "Failed in osgDB::writeNodeFile()." << endl;
}
```



Zusammenfassung

- Anforderungen an Szenengraphen:
  - Abstraktion von der Graphikpipeline (osg::Geometry usw.)
  - Resourcenmanagement wird implizit gewährleistet via Referenzzählung
  - Gerichtete azyklische Graphenstruktur
  - Traversierung per Frame erlaubt dynamische Struktur
  - Kommunikation mit externenTools mittels Plugins und Erweiterung mittels Nodekits
- Was nicht erwähnt wurde:
  - User Interfaces zur Interaktion mit dem Graphen:
    Picking und Manipulatoren zum verschieben von Szenenobjekten (**osgManipulator Beispiel im SDK**)
  - osgCompute Framework für CUDA Integration und andere 3rd Party libaries