

# 3. Computergraphik Übung: Einführung in komplexe Geometrien

Heni Ben Amor

Arbeitsgruppe Virtuelle Realität und Multimedia

TU Bergakademie Freiberg

# Aufgabe 1A: Texturierung

- Korrekte Abfrage ob Textur vorhanden

# Aufgabe 1A: Texturierung

- Korrekte Abfrage ob Textur vorhanden
- Fehlerbehandlung

# Aufgabe 1A: Texturierung

- Korrekte Abfrage ob Textur vorhanden
- Fehlerbehandlung
- Sichere Terminierung falls Textur fehlt!

# Texturierung

- Funktion mit Rückgabe: Pointer auf Textur

```
osg::Texture2D* load_texture(const char* textureName)
{
    osg::Texture2D* texture = new osg::Texture2D;
    texture->setDataVariance(osg::Object::DYNAMIC);

    osg::Image* image = osgDB::readImageFile(textureName);
    if (!image)
    {
        std::cerr << "Couldn't find texture!" << std::endl;
        return NULL;
    }
    texture->setImage(image);
    return texture;
}
```

# Texturierung

- Funktion mit Rückgabe: Pointer auf Textur

```
osg::Texture2D* load_texture(const char* textureName)
{
    osg::Texture2D* texture = new osg::Texture2D;
    texture->setDataVariance(osg::Object::DYNAMIC);

    osg::Image* image = osgDB::readImageFile(textureName);
    if (!image)
    {
        std::cerr << "Couldn't find texture!" << std::endl;
        return NULL;
    }
    texture->setImage(image);
    return texture;
}
```

# Texturierung

- Funktion mit Rückgabe: Pointer auf Textur

```
osg::Texture2D* load_texture(const char* textureName)
{
    osg::Texture2D* texture = new osg::Texture2D;
    texture->setDataVariance(osg::Object::DYNAMIC);

    osg::Image* image = osgDB::readImageFile(textureName);
    if (!image)
    {
        std::cerr << "Couldn't find texture!" << std::endl;
        return NULL;
    }
    texture->setImage(image);
    return texture;
}
```

# Aufgabe 1B: Animation mit Callbacks

- Knoten sollte wiederverwendbar sein

# Aufgabe 1B: Animation mit Callbacks

- Knoten sollte wiederverwendbar sein
- Knoten sollte Animationsparameter abkapseln

# Aufgabe 1B: Animation mit Callbacks

- Knoten sollte wiederverwendbar sein
- Knoten sollte Animationsparameter abkapseln
- Knoten sollte parametrisierbar sein

# Animation mit Callbacks

- Definition von Klasse und Parameter:

```
class planetNodeCallback : public osg::NodeCallback
{
protected:
    double rotAngle, incAngle;
public:
    planetNodeCallback(double angle){incAngle=angle; rotAngle = 0.0
    ...
}
```

# Animation mit Callbacks

- Definition von Klasse und Parameter:

```
class planetNodeCallback : public osg::NodeCallback
{
protected:
    double rotAngle, incAngle;
public:
    planetNodeCallback(double angle){incAngle=angle; rotAngle = 0.0
    ...
}
```

# Animation mit Callbacks

- Definition der Callback-Funktion:

```
virtual void operator()(osg::Node* node, osg::NodeVisitor* nv)
{
    // casten des pointers auf einen transformationsknoten
    osg::ref_ptr<osg::PositionAttitudeTransform>
        trans = dynamic_cast<osg::PositionAttitudeTransform*> (node);

    trans->setAttitude(
        osg::Quat(osg::DegreesToRadians(rotAngle), osg::Vec3(0,
rotAngle += incAngle;
    traverse(node, nv);
}
```

# Animation mit Callbacks

- Definition der Callback-Funktion:

```
virtual void operator()(osg::Node* node, osg::NodeVisitor* nv)
{
    // casten des pointers auf einen transformationsknoten
    osg::ref_ptr<osg::PositionAttitudeTransform>
        trans = dynamic_cast<osg::PositionAttitudeTransform*> (node);

    trans->setAttitude(
        osg::Quat(osg::DegreesToRadians(rotAngle), osg::Vec3(0,
rotAngle += incAngle;
    traverse(node, nv);
}
```

# Aufgabe 2: Mathematische Grundlagen

- Rotation um z-Achse mit Winkel  $30^\circ$

$$T_1 = \begin{pmatrix} 0.5\sqrt{3} & -0.5 & 0 & 0 \\ 0.5 & 0.5\sqrt{3} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

# Aufgabe 2a: Mathematische Grundlagen

- Translation mit  $(1, 5, 4)^T$

$$T_2 = \begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 5 \\ 0 & 0 & 1 & 4 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

# Aufgabe 2b: Mathematische Grundlage

- Welche Transformationsmatrix ( $T_3$ ) muss angewendet werden, um von lokalen Koordinaten auf Weltkoordinaten zu kommen?

$$T_3 = T_1 \cdot T_2$$

$$\vec{v}' = T_3 \cdot \vec{v}$$

# Aufgabe 2C: Mathematische Grundlage

- Welche Transformationsmatrix ( $T_4$ ) muss angewendet werden, um von Weltkoordinaten auf lokale Koordinaten zu kommen?

$$T_4 = T_3^{-1}$$

$$T_4 = (T_1 \cdot T_2)^{-1}$$

$$T_4 = T_2^{-1} \cdot T_1^{-1}$$

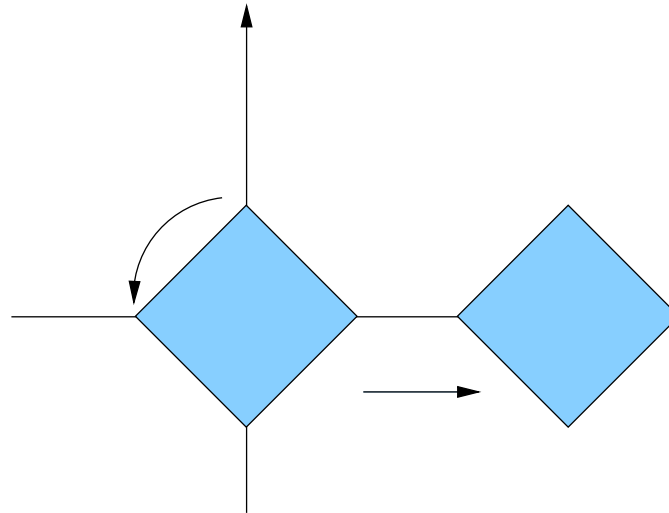
# Aufgabe 2C: Mathematische Grundlage

- Sind Transformationsmatrizen kommutativ bzgl. der Multiplikation?

Nein:  $A \cdot B \neq B \cdot A$

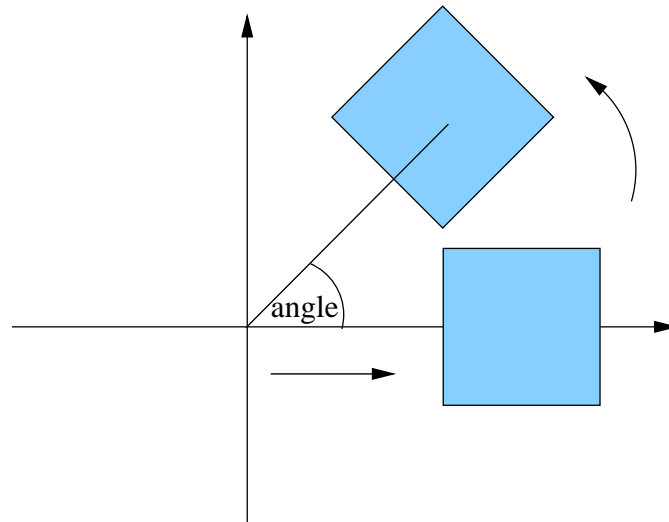
# Aufgabe 2c

- Zuerst Rotation dann Translation



# Aufgabe 2c

- Zuerst Translation dann Rotation



# Komplexe Geometrien

- Bisher: Ausschließlich Basis-Primitive (Box, Sphere, Cylinder, etc.)

# Komplexe Geometrien

- Bisher: Ausschließlich Basis-Primitive (Box, Sphere, Cylinder, etc.)
- Nachteil: Beschränkte Modellierungsmächtigkeit

# Komplexe Geometrien

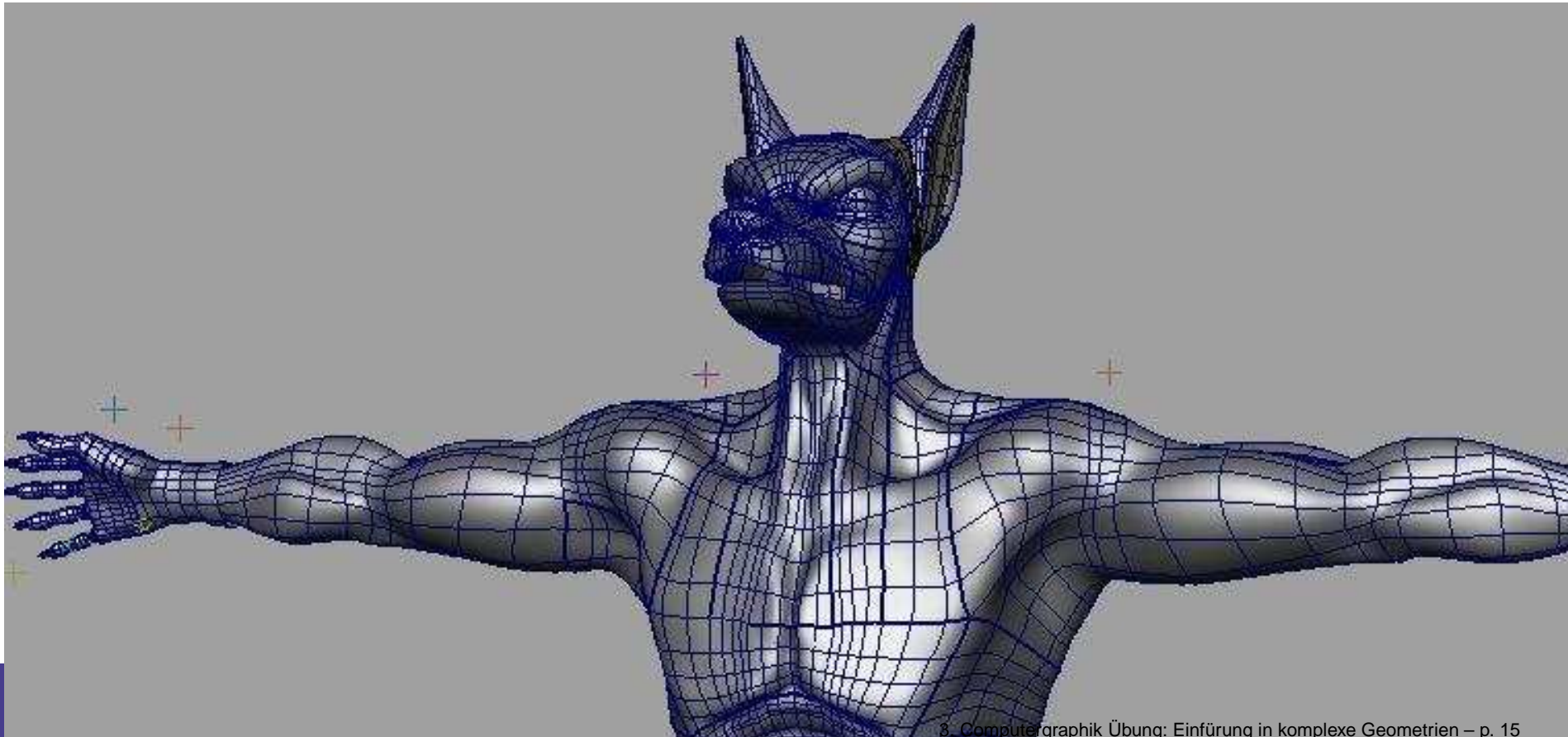
- Bisher: Ausschließlich Basis-Primitive (Box, Sphere, Cylinder, etc.)
- Nachteil: Beschränkte Modellierungsmächtigkeit
- Beschreibung komplexerer Objekte durch Polygon-Netze

# Komplexe Geometrien

- Bisher: Ausschließlich Basis-Primitive (Box, Sphere, Cylinder, etc.)
- Nachteil: Beschränkte Modellierungsmächtigkeit
- Beschreibung komplexerer Objekte durch Polygon-Netze
- Koordinaten aller Eckpunkte müssen angegeben werden

# Komplexe Geometrien

- Bisher: Ausschließlich Basis-Primitive (Box, Sphere, Cylinder, etc.)



# Klassenhierarchie

- Geode abgeleitet von *Node* und can mehrere *drawables* haben

# Klassenhierarchie

- Geode abgeleitet von *Node* und can mehrere *drawables* haben
- Drawable kann ein *Shape* oder ein Geometry Objekt sein

# Klassenhierarchie

- Geode abgeleitet von *Node* und can mehrere *drawables* haben
- Drawable kann ein *Shape* oder ein Geometry Objekt sein
- Geometry kann mehrere *PrimitiveSets* haben

# Klassenhierarchie

- Geode abgeleitet von *Node* und can mehrere *drawables* haben
- Drawable kann ein *Shape* oder ein Geometry Objekt sein
- Geometry kann mehrere *PrimitiveSets* haben
- Die PrimitiveSet-Klasse kapselt OgenGl Primitive: POINTS, LINES, LINE\_STRIP, QUADS, POLYS, etc..

# Erstellen von Geometrie

- Ein Geode- und ein Geometry-Pointer muss erstellt werden

```
osg::Geode* pyramidGeode = new osg::Geode();  
osg::Geometry* pyramidGeometry = new osg::Geometry();
```

# Erstellen von Geometrie

- Geometry als *Drawable* von Geode setzen

```
pyramidGeode->addDrawable(pyramidGeometry);  
root->addChild(pyramidGeode);
```

# Erstellen von Geometrie

- Eckpunkte der Geometrie erstellen, speichern und zuordnen

```
osg::Vec3Array* pyramidVertices = new osg::Vec3Array;
pyramidVertices->push_back( osg::Vec3( 0, 0, 0) ); // front left
pyramidVertices->push_back( osg::Vec3(10, 0, 0) ); // front right
pyramidVertices->push_back( osg::Vec3(10,10, 0) ); // back right
pyramidVertices->push_back( osg::Vec3( 0,10, 0) ); // back left
pyramidVertices->push_back( osg::Vec3( 5, 5,10) ); // peak

pyramidGeometry->setVertexArray(pyramidVertices);
```

# Erstellen von Geometrie

- Aus einzelnen Punkten müssen jetzt Flächen erstellt werden
- Indexe the Eckpunkte für Unterseite angeben (gegen UZS)

```
osg::DrawElementsUInt* pyramidBase =  
    new osg::DrawElementsUInt(osg::PrimitiveSet::QUADS, 0);  
pyramidBase->push_back(3);  
pyramidBase->push_back(2);  
pyramidBase->push_back(1);  
pyramidBase->push_back(0);  
pyramidGeometry->addPrimitiveSet(pyramidBase);
```

# Erstellen von Geometrie

- Flächendefinition muss für alle Seiten der Pyramide wiederholt werden
- Achtung diesmal PrimitiveSet::Triangles (Polygon)

```
osg::DrawElementsUInt* pyramidFaceOne =  
    new osg::DrawElementsUInt(osg::PrimitiveSet::TRIANGLES, 0);  
pyramidFaceOne->push_back(0);  
pyramidFaceOne->push_back(1);  
pyramidFaceOne->push_back(4);  
pyramidGeometry->addPrimitiveSet(pyramidFaceOne);
```