Computergrafik Praktikum 4

Allgemeines zum Praktikum

Im Rahmen des Computergrafik-Praktikums lernen Sie, die aus der Vorlesung gewonnenen Erkenntnisse praktisch mit der Programmiersprache C/C++ umzusetzen.

Bei der Konzipierung der Praktika wurde darauf geachtet, dass die verwendeten Bibliotheken sowohl für Windows als auch für OS X erhältlich sind. Sofern Sie die Aufgaben im Medienlabor bearbeiten, steht Ihnen die Entwicklungsumgebung XCode zur Verfügung, mit der Sie Ihre C++-Programme entwickeln können. Wenn Sie die Aufgaben an Ihrem privaten Rechner unter Windows bearbeiten möchten, empfehle ich Ihnen die Entwicklungsumgebung Visual Studio 2013, die Sie als Student/-in über das Microsoft ELMS-Programm kostenlos herunterladen können.

Bitte beachten Sie bei der Bearbeitung der Praktikumsaufgaben, dass Ihre Lösungen für folgende Praktikumsaufgaben weiter genutzt werden müssen. Die Praktikumsarbeiten bauen also ineinander auf. Bearbeiten Sie deshalb die Lösungen sorgfältig und löschen Sie diese nicht, wenn die Aufgabe abgenommen wurde. Achten Sie bei der Entwicklung darauf, dass nur die Schnittstellen in den Header-Dateien stehen, während die Implementierung ausschließlich in den cpp-Dateien steht.

Zum Bestehen des kompletten Praktikums müssen Sie jede Praktikumsaufgabe erfolgreich abschließen und mindestens 80% der Gesamtpunkte für das komplette Praktikum erhalten. Wie viele Punkte Sie für eine einzelne Aufgabe erhalten, hängt von Ihrer Implementierung und Ihren Erläuterungen ab.

Um dieses Praktikum zu bestehen, müssen Ihre entwickelten Teillösungen lauffähig sein (5 Punkte) und Sie müssen die einzelnen Lösungen erläutern können (weitere 5 Punkte). Insgesamt können Sie bei diesem Aufgabenblatt 10 Punkte + 5 Zusatzpunkte erhalten.
Abnahmetermin: 17.05. (Di) & 20.05. (Fr)

Thema Praktikum 1

Bei diesen Praktikumsaufgaben geht es darum, dass Sie lernen, mit affinen und projektiven Transformationen praktisch zu arbeiten. Diese Transformationen werden in 3D-Darstellungssystemen i. d. R. durch 4x4-Matrizen realisiert. In der Vorlesung wurde Ihnen gezeigt, wie Matrizen konstruiert werden müssen, um mit ihnen Modelle zu verschieben (Translation), zu rotieren oder zu skalieren. Ferner wurde erläutert, wie durch Matrixmultiplikationen eine Hintereinander-Ausführung von mehreren Transformationen durchgeführt werden kann. Für die Bearbeitung der folgenden Praktikumsaufgaben wird Ihnen eine Klasse *Matrix* zur Verfügung gestellt, die bereits die gängigsten Transformationen unterstützt. Wollen Sie zum Beispiel eine Matrix erzeugen, die eine Verschiebung um 3 Einheiten entlang der X-Achse vornimmt, können Sie folgenden Aufruf nutzen:

Matrix TM; //erzeugt Matrix-Objekt
TM.translation(3, 0, 0); // initialisiert Matrix-Werte für Translation

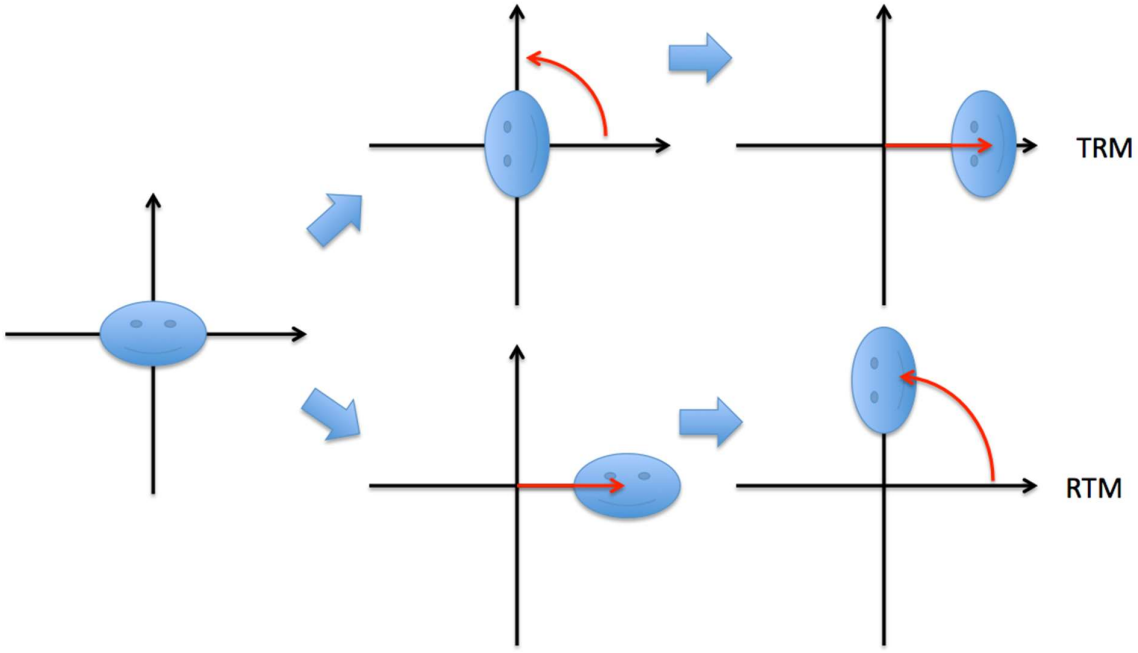
Die folgende Matrix rotiert um 90° um die Y-Achse:

Matrix RM; //erzeugt Matrix-Objekt
RM.rotationY(0.5f*M_PI); // initialisiert Matrix-Werte für Rotation

Möchte man beide Transformationen kombiniert auf ein Objekt anwenden, müssen die Matrizen im Vorhinein miteinander multipliziert werden, um dann die sich ergebene Matrix auf die Modellpunkte anzuwenden. Hierbei ist zu beachten, dass die Reihenfolge der Operanden bei der Multiplikation eine Rolle spielt:

Matrix TRM, RTM;
TRM = TM * RM; // zuerst wird rotiert, dann verschoben
RTM = RM * TM; // zuerst wird verschoben, dann wird rotiert.

Das nachfolgende Bild verdeutlicht den Unterschied:



Eine Matrix, die Sie auf ein Objekt anwenden wollen, muss vor dem Zeichnen der Primitiven dem Shader übergeben werden. Hierfür können Sie die Methode *transform(..)* der BaseModel-Klasse verwenden. Die Matrix, die Sie der *transform(..)*-Methode übergeben, wird als Welt- bzw. Modellmatrix an den Shader weitergeleitet. Dies passiert intern in der Klasse *BaseModel*. Die weiteren benötigten Matrizen (Kamera & Projektion) erhält die Model-Klasse über das Kamera-Objekt, welches bei der draw(..)-Methode übergeben wird.

```
pModel = new LinePlaneModel(10, 10, 10, 10);
Matrix MatRot;
MatRot.rotationZ(M_PI/2.0f);
pModel->shader(pConstShader, true);
pModel->transform(MatRot);
Models.push_back( pModel );
```

Schauen Sie sich die Schnittstelle der Matrix-Klasse an, um einen Überblick über deren Methoden zu erhalten.

Damit Sie die Praktikumsaufgaben bearbeiten können, benötigen Sie das Projekt aus dem Archiv cgprakt4.zip (OSCA-Lernraum).

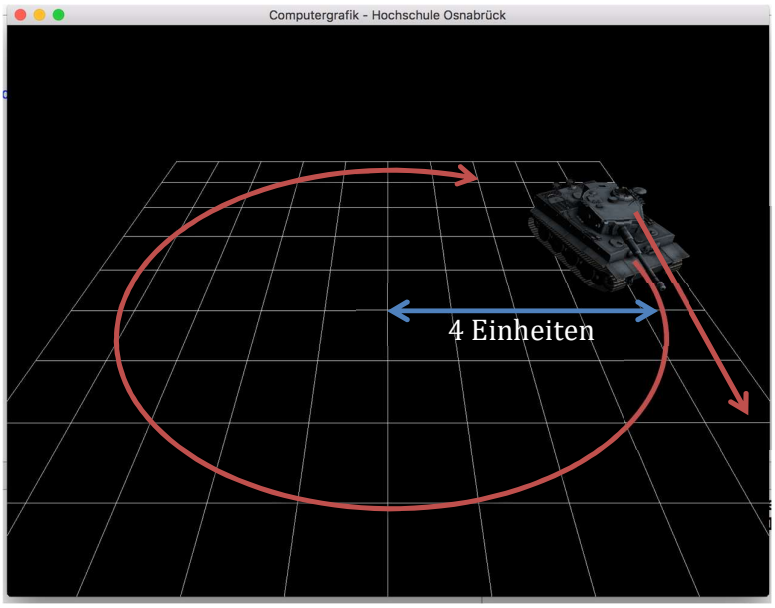
Darüber hinaus benötigen Sie die Klassen, die Sie in den vorherigen Praktikumsaufgaben selbst entwickelt haben:

- Vector.h & Vector.cpp
- Color.h & Color.cpp
- rgbimage.h & rgbimage.cpp
- Model.h & Model.cpp
- (LineBoxModel & TriangleBoxModel)

Die Shader- & Texture-Klassen wurden in diesem Projekt aktualisiert (für Alpha-Blending). Bitte kopieren Sie keine älteren Versionen dieser Dateien in das jeweilige Projekt.

Aufgabe 1 (4 Punkte)

Laden Sie die Modelle „tank_top.dae“ und „tank_bottom.dae“ in die Szene (FitSize=false) und transformieren Sie die Modelle zeitabhängig in der *update(..)*-Methode der *Application*-Klasse so, dass der Panzer im Uhrzeigersinn mit 4 Einheiten um das Zentrum fährt. Das Panzerrohr soll dabei immer nach unten, entlang der Z-Achse, zeigen:



Die Ausrichtung des Panzerrohrs soll immer gleich bleiben (entlang Z-Achse, Richtung Betrachter)..

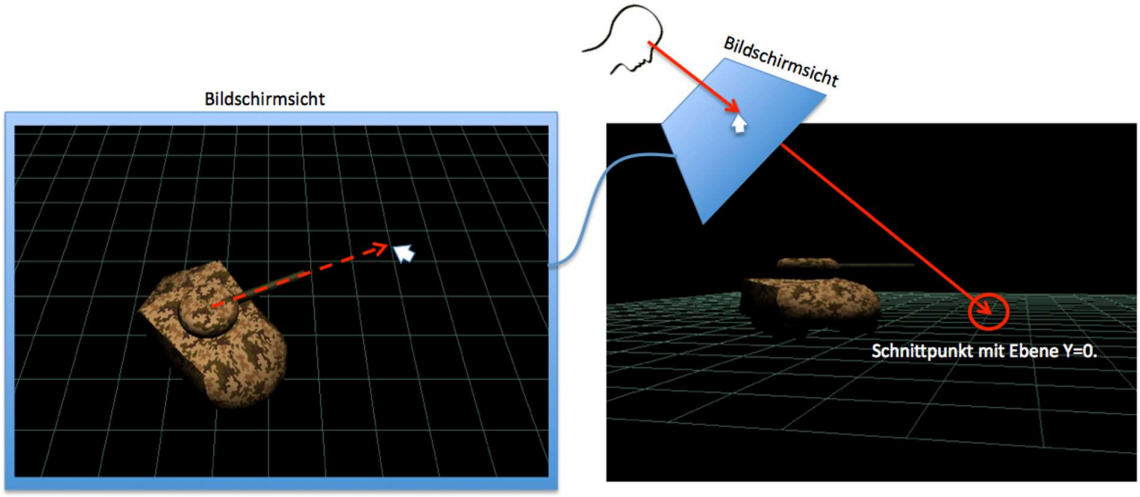
Der Panzer soll eine komplette Umdrehung in 6 Sekunden vollziehen. Erweitern Sie die update(..)-Methode um einen float-Parameter, der die Frametime (Deltatime) enthält und ändern Sie die Main-Funktion so ab, dass die Frametime der update(..)-Methode übergeben wird. Um die Frametime zu berechnen, können Sie die Methode glfwGetTime(..) benutzen.

Aufgabe 2 (6 Punkte)

Bei dieser Aufgabe sollen Sie eine kleine Klasse zur Steuerung eines Panzers programmieren. Diese Art der Steuerung findet man sehr häufig in digitalen Spielen und sie ist eine gute Übung, um ein tiefergehendes Verständnis über affine Transformationen und über die Transformations-Pipeline zu erhalten.
Die Steuerung des Panzers soll durch eine Kombination von Tastatur und Maus erfolgen. Über die Tastatur wird durch die Pfeiltasten der Panzer gedreht sowie nach vorne und hinten bewegt:

- Pfeil-Oben-Taste: der Panzer fährt nach vorne.
- Pfeil-Unten-Taste: der Panzer fährt nach hinten.
- Pfeil-Links-Taste: der Panzer dreht gegen den Uhrzeigersinn (nach links).
- Pfeil-Rechts-Taste: der Panzer dreht mit dem Uhrzeigersinn (nach rechts).

Die Drehung des Panzers kann im Stand oder während der Vorwärts- bzw. Rückwärtsbewegung erfolgen (wie richtiger Panzer).
Die Steuerung der Haubitze des Panzers erfolgt über die Maus. Die Haubitze soll immer in Richtung des Mauszeigers zeigen. Hierfür muss die 2D-Position des Mauszeigers in Pixelkoordinaten in einen 3D-Strahl in Welt-Koordinaten umgerechnet werden. Der Schnittpunkt dieses Strahls mit der Ebene Y=0 liefert die Position für die Ausrichtung der Haubitze. Das folgende Bild verdeutlicht das Prinzip:



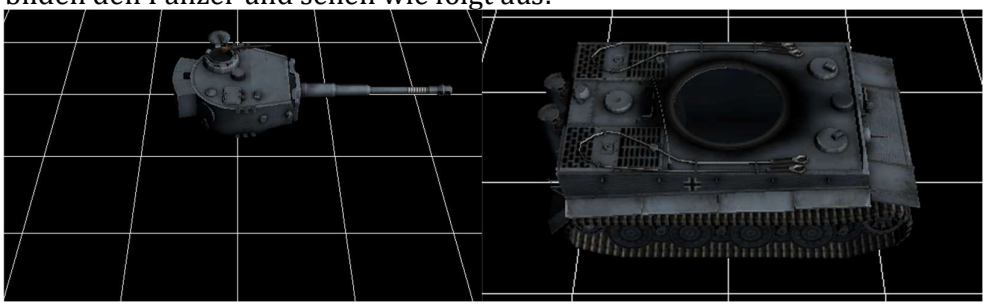
(Das Panzer-Modell in der Abbildung ist noch veraltet.)
Die Neigung der Haubitze nach oben oder unten muss nicht abgebildet werden. Die Panzer (engl. Tank)-Klasse soll die folgende Schnittstelle besitzen:

```
class Tank : public BaseModel
{
public:
    Tank();
    virtual ~Tank();
    bool loadModels(const char* ChassisFile, const char* CannonFile);
    void steer( float ForwardBackward, float LeftRight);
    void aim( const Vector& Target );
    void update(float dtime);
    virtual void draw(const BaseCamera& Cam);
```

Erläuterungen:

- Die *load*(..)-Methode nimmt zwei Modell-Dateinamen. Dem Parameter ChassisFile übergeben Sie bitte die „tank_bottom.dae“-Datei und dem Parameter

CannonFile die „tank_top.dae“ Datei (aus Assets-Ordner). Die beiden Modelle bilden den Panzer und sehen wie folgt aus:

- 
- (links tank_top.dae, rechts tank_bottom.dae)
- Die *steer*(..)-Methode soll zur Steuerung des Panzers mit der Tastatur aufgerufen werden. Ihr werden zwei float-Parameter übergeben, die jeweils steuern, ob der Panzer nach vorne (*ForwardBackward*=1.0f) oder nach hinten (*ForwardBackward*=-1.0f) fährt bzw. ob er nach links (*LeftRight*=1.0f) oder nach rechts (*LeftRight*=-1.0f) drehen soll. Das Einlesen der Tastatureingaben und die Umrechnung auf die beiden Parameter müssen Sie selbst in der *Application-Klasse* implementieren. Um festzustellen, ob eine Taste gedrückt bzw. losgelassen wurde, steht Ihnen die Funktion *glfwGetKey(..)* zur Verfügung.
- Die *aim*(..)-Methode soll zur Steuerung der Haubitze genutzt werden. Der Methode wird die 3D-Position des Mauszeigers auf der Ebene Y=0 im Parameter Target übergeben. Die Position des Mauszeigers können Sie mit der Funktion *glfwGetCursorPos(..)* in Fenster-Pixel-Koordinaten erhalten. Implementieren Sie die Methode *Application::calc3DRay(..)*, um aus den 2D-Mauskoordinaten einen 3D-Strahl zu berechnen (siehe Tipps). Für diesen Strahl berechnen Sie dann den Schnittpunkt mit der Ebene Y=0. Den Schnittpunkt übergeben Sie der *aim(..)*-Methode.
- Die *update*(..)-Methode soll die Animation des Panzers steuern. In dieser Methode werden die Transformationsmatrizen für die Panzerteile konstruiert. Hierbei ist darauf zu achten, dass die Animation zeitbasiert abläuft! Eine Veränderung der Framerate darf nicht dazu führen, dass der Panzer sich schneller bewegt.
- Die *draw*(..)-Methode wird aufgerufen, um Modelle zu zeichnen.

Sie müssen von Ihrer Panzer-Klasse ein entsprechendes Objekt in der Application-Klasse erzeugen und es der *Application::Models*-Liste hinzufügen.

Tipps:
Implementieren Sie erst die Steuerung des Panzers durch die Tastatur und danach die Steuerung der Haubitze. Um die Haubitze auszurichten, müssen Sie die Transformations-Pipeline rückwärts durchlaufen. Das bedeutet,

1. Sie rechnen die Pixelkoordinaten des Mauszeigers in normalisierte Bildkoordinaten um (x in[-1,1] und y in[-1,1]). Dies entspricht der inversen Viewport-Matrix.
2. Wenn Sie die Mauszeigerkoordinaten zwischen -1 und 1 normalisiert haben, können Sie diese Koordinaten nutzen, um einen Richtungsvektor zu erzeugen, indem Sie die Projektionsmatrix (diese erhalten Sie von dem Kamera-Objekt, welches in *Application::Cam* gespeichert ist) invers auf die neuen Mauszeigerkoordinaten anwenden. Die fehlende Z-Koordinate können Sie mit 0 annehmen. Das Ergebnis der Transformation ist ein Richtungsvektor im Kameraraum (View-Coordinates).

3. Sie besitzen nun einen Richtungsvektor in Kamera-Koordinaten und müssen diesen noch in Weltkoordinaten umrechnen. Für die Umrechnung sollten Sie bedenken, dass Sie nur die Richtung des Vektors anpassen müssen und nicht dessen Ursprung (Matrix::transformVec3x3(..)). Wenn Sie die Richtung des Strahls in Weltkoordinaten überführt haben, müssen Sie nur noch den Ursprung des Strahls bestimmen. Der Ursprung des Strahls ist die Kameraposition, diese ist in der Kameramatrix kodiert (sie erhalten die Kameramatrix über Cam.getViewMatrix()).
4. Wenn Sie vom Strahl die Richtung und den Ursprung in Weltkoordinaten kennen, müssen Sie noch den Schnittpunkt mit der Ebene Y=0 berechnen. Die Berechnung des Schnittpunkts zwischen Strahl und Ebene wurde bereits beim Raytracing-Verfahren erläutert.
5. Schlussendlich nutzen Sie die Position des Schnittpunkts, um die Haubitze des Panzers mit homogenen Matrix-Transformationen auszurichten. Hierbei kann die Tatsache hilfreich sein, dass eine affine Matrix auch als Koordinatensystem verstanden werden kann.

Aufgabe 3 (5 Zusatzpunkte)

Affine Transformationen werden auch genutzt, um Szenen hierarchisch anzuordnen (Szenengraph, Szenen-Management). Dies erlaubt eine Darstellung kompletter Umwelten bestehend aus wenigen, meistens einfachen, (Teil-)Modellen. Bei dieser Aufgabe sollen Sie ein eigenes kleines Szenen-Management-System implementieren, das hierarchische Szenendaten aus einer Datei ausliest und diese korrekt transformiert darstellt. Die hierarchischen Szenendaten, die Sie zum Testen Ihres Szenen-Systems nutzen sollen, finden Sie in der Datei scene.osh (Assets-Ordner). Der Inhalt der Datei sieht z. B. wie folgt aus (Sie müssen möglicherweise die Pfade im Tag FILE=... für ihr Projekt anpassen):

```
MODEL ID=cylinder FILE=cylinder.obj
MODEL ID=cone FILE=cone.obj
MODEL ID=plane FILE=plane.obj
NODE ID=P0 PARENTID=NULL MODELID=plane TRANSLATION=-0.000000 0.000000 0.000000 ROTATIONAXIS=0.000000 1.000000 0.000000 ROTATIONANGLE=0.000000 SCALE=50.00000 50.00000 50.00000
NODE ID=T0_0 PARENTID=NULL MODELID=cylinder TRANSLATION=-24.999609 0.000000 -18.423111 ROTATIONAXIS=0.000000 1.000000 0.000000 ROTATIONANGLE=0.000000 SCALE=0.153277 0.429325 0.153277
NODE ID=T0_1 PARENTID=T0_0 MODELID=cone TRANSLATION=0.000000 0.386393 0.000000 ROTATIONAXIS=0.706254 0.000000 0.707959 ROTATIONANGLE=0.081567 SCALE=0.523522 1.828439 0.523522
NODE ID=T0_2 PARENTID=T0_1 MODELID=cone TRANSLATION=0.000000 0.914219 0.000000 ROTATIONAXIS=0.579517 0.000000 -0.814960 ROTATIONANGLE=0.004665 SCALE=0.420851 1.420149 0.420851
NODE ID=T0_3 PARENTID=T0_2 MODELID=cone TRANSLATION=0.000000 0.710075 0.000000 ROTATIONAXIS=-0.973086 0.000000 -0.230441 ROTATIONANGLE=0.005833 SCALE=0.351086 1.144555 0.351086
```

Der Aufbau des Dateiformats ist sehr einfach. Die Datei unterscheidet zwei Objektdefinitionen (Eine Definition pro Zeile):

- Model: Definiert ein OBJ-Modell, dieses besteht aus:
  - ID: ID des Modells innerhalb der Datei
  - FILE: Der Name der OBJ-Datei.
- Node: Definiert einen Szenen-Knoten, dieser besteht aus:
  - ID: ID des Knotens innerhalb der Datei
  - PARENTID: Die ID des Vaterknotens (hierarchischer Aufbau, NULL bedeutet kein Vaterknoten vorhanden)
  - MODELID: Die ID des Modells zum Knoten
  - TRANSLATION: Die Position des Knotens im Raum des Vaterknotens
  - ROTATIONAXIS: Rotationsachse im Raum des Vaterknotens
  - ROTATIONANGLE: Rotationswinkel im Raum des Vaterknotens
  - SCALE: Skalierung des Knotens. Die Skalierung gilt individuell für den Knoten und wird nicht hierarchisch weitergegeben!

Die Schnittstelle der Klasse, die eine komplette Szene lädt und anzeigt, soll wie folgt aussehen:

```
class Scene : public BaseModel
{
public:
    Scene();
    virtual ~Scene();
    bool addSceneFile( const char* Scenefile);
    virtual void draw(const BaseCamera& Cam);
protected:
    void draw( SceneNode* pNode);
    SceneNode m_Root;
    std::map<std::string, Model*> m_Models;
};
```

Erläuterungen:

- Die Methode addSceneFile(...) soll die .osh-Datei laden. Modell-OBJ-Dateien werden nur einmalig geladen und in der map *m_Models* hinterlegt (hier müssen Sie wieder Ihre Model-Klassen nutzen). Die Szenenknoten erhalten Referenzen zu den Model-Objekten.
- Die Methode *draw(const BaseCamera& Cam)* zeichnet eine komplette Szene, während die Methode *draw(SceneNode* pNode)* einen einzelnen Szenenknoten zeichnet.
- Alle Knoten einer Szene, die keinen Vaterknoten besitzen, erhalten *m_Root* als Vaterknoten. Das bedeutet, dass *m_Root* einen oder mehrere Kindknoten besitzt. *m_Root* ist also die Wurzel des Szenen-Baums (Hierarchie).

Die *Scene*-Klasse verwendet die Klasse *SceneNode*. Die Schnittstelle und die Member-Variablen der Klasse *SceneNode* sollen wie folgt aussehen:

```
class SceneNode
{
public:
    SceneNode();
    SceneNode( const std::string& Name, const Vector& Translation, const Vector&
        RotationAxis, const float RotationAngle, const Vector& Scale, SceneNode* pParent,
        Model* pModel);

    //getter
    const Matrix& getLocalTransform() const;
    Matrix getGlobalTransform() const;
    const SceneNode* getParent() const;
    const Model* getModel() const;
    const std::string& getName() const;
    const Vector& getScaling() const;

    //setter
    void setLocalTransform( const Vector& Translation, const Vector& RotationAxis, const
        float RotationAngle );
    void setLocalTransform( const Matrix& LocalTransform);
    void setParent( SceneNode* pNode);
    const std::set<SceneNode*>& getChildren() const;
    void addChild(SceneNode* pChild);
    void removeChild(SceneNode* pChild);
    void setModel( Model* pModel);
    void setName( const std::string& Name);
    void setScaling( const Vector& Scaling);
    void draw(const BaseCamera& Cam);
protected:
    std::string m_Name;
    Model* m_pModel;
    SceneNode* m_pParent;
    std::set<SceneNode*> m_Children;
    Matrix m_LocalTransform;
    Vector m_Scaling;
};
```

Erläuterungen:

- Ein *SceneNode*-Objekt besitzt die folgenden Daten: Name (*m_Name*), einen Zeiger auf sein Model (*m_pModel*, dieser darf auch NULL sein, z. B. beim Root-Node), einen Zeiger auf seinen Vaterknoten (*m_pParent*), ein Set von Kindknoten (*m_Children*), einen Skalierungsvektor (*m_Scaling*) und eine lokale Transformation (*m_localTransform*, diese enthält keine Skalierung). Die lokale Transformation beschreibt die Orientierung und die Position des Knotens in Bezug auf den Vaterknoten. Basierend auf den gespeicherten Daten im Knoten kann die globale (Welt-)Transformation (*getGlobalTransformation*()) berechnet werden. Für die Berechnung muss die Transformation des Vaterknotens berücksichtigt werden.
- Die meisten Methoden der Klasse sind einfache Setter- und Getter-Methoden. Zum Setzen der lokalen Transformation (*setLocalTransform*) stehen zwei Methoden zur Verfügung: eine, die eine fertige Matrix entgegennimmt und eine, der man einen Translationsvektor sowie eine Rotationsachse mit dazugehörigen Rotationswinkel übergeben kann. Die Methode muss die übergebenen Daten in eine Matrix umwandeln und diese in der Variable m_LocalTransform abspeichern. Die lokale Transformation beinhaltet nicht die Skalierung. Diese wird separat übergeben mit *setScaling*(..).
- Die Methode *getGlobalTransform*() liefert die absolute Welt-Transformation für einen Knoten in der Szene. Um die absolute Welt-Transformation zu bestimmen, müssen nacheinander die folgenden Transformationen angewendet werden (Hintereinanderausführung):
    1. Anwendung der Skalierungsmatrix.
    2. Anwendung der eigenen lokalen Transformation.
    3. Anwendung der globalen Transformation des Vaterknotens.

Um nun einen Szenenknoten zu zeichnen, muss dessen globale Transformation der BaseModel::transform(..)-Methode übergeben werden, bevor das Modell gezeichnet wird.

Tipp: Das Datei-Format für die Szene ist sehr einfach gehalten. Sie können eine komplette Zeile (gespeichert in char Line[512]) einfach mit sscanf parsen:

Model-Zeile:

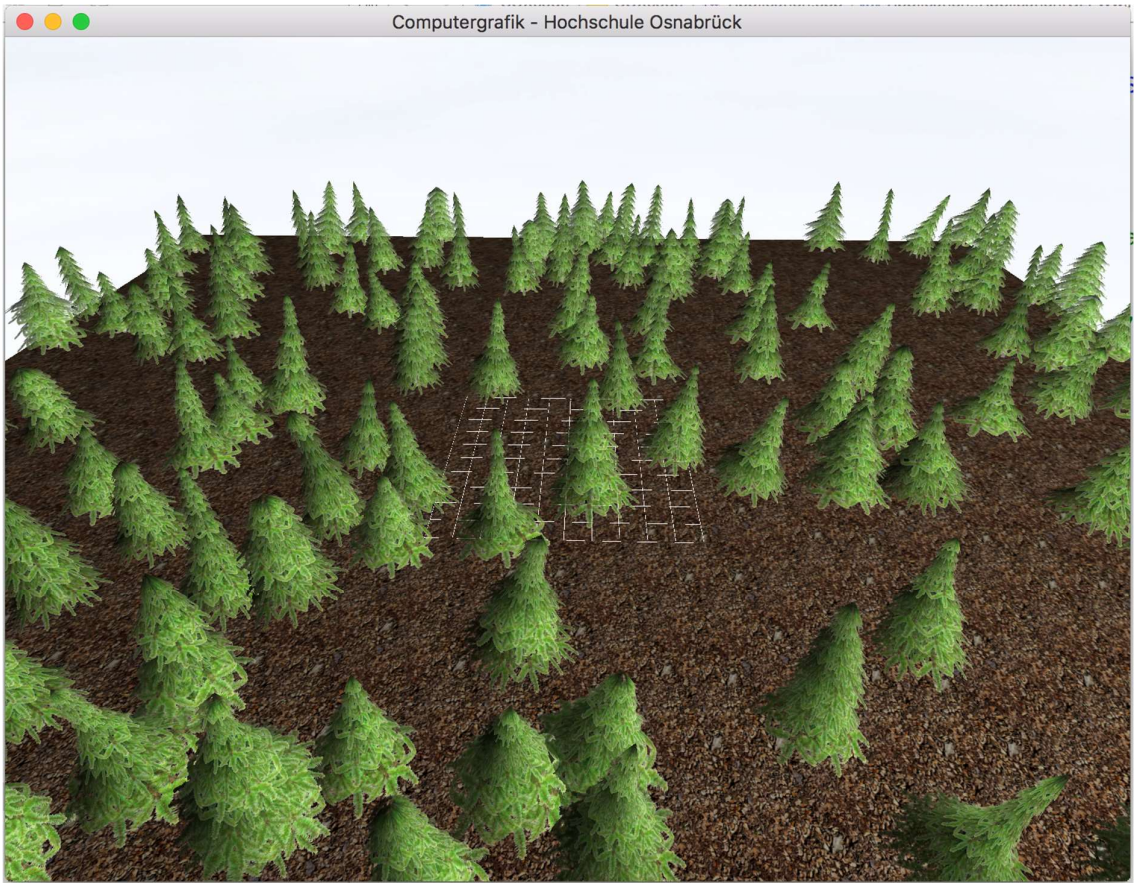
```
char Modelfile[256];
char ModelID[256];
sscanf( Line, "MODEL ID=%s FILE=%s", ModelID, Modelfile );
```

Node-Zeile:

```
Vector Pos, Scale, RotAxis;
float Angle;
char NodeID[256];
char ParentID[256];
char ModelID[256];

sscanf( Line, "NODE ID=%s PARENTID=%s MODELID=%s TRANSLATION=%f %f %f ROTATIONAXIS=%f %f %f
ROTATIONANGLE=%f SCALE=%f %f %f",
        NodeID, ParentID, ModelID,
        &Pos.X, &Pos.Y, &Pos.Z,
        &RotAxis.X, &RotAxis.Y, &RotAxis.Z,
        &Angle,
        &Scale.X, &Scale.Y, &Scale.Z);
```

Wenn Sie die Scene- und SceneNode-Klasse richtig implementiert haben, sollte das Ergebnis wie folgt aussehen:



Viel Erfolg!