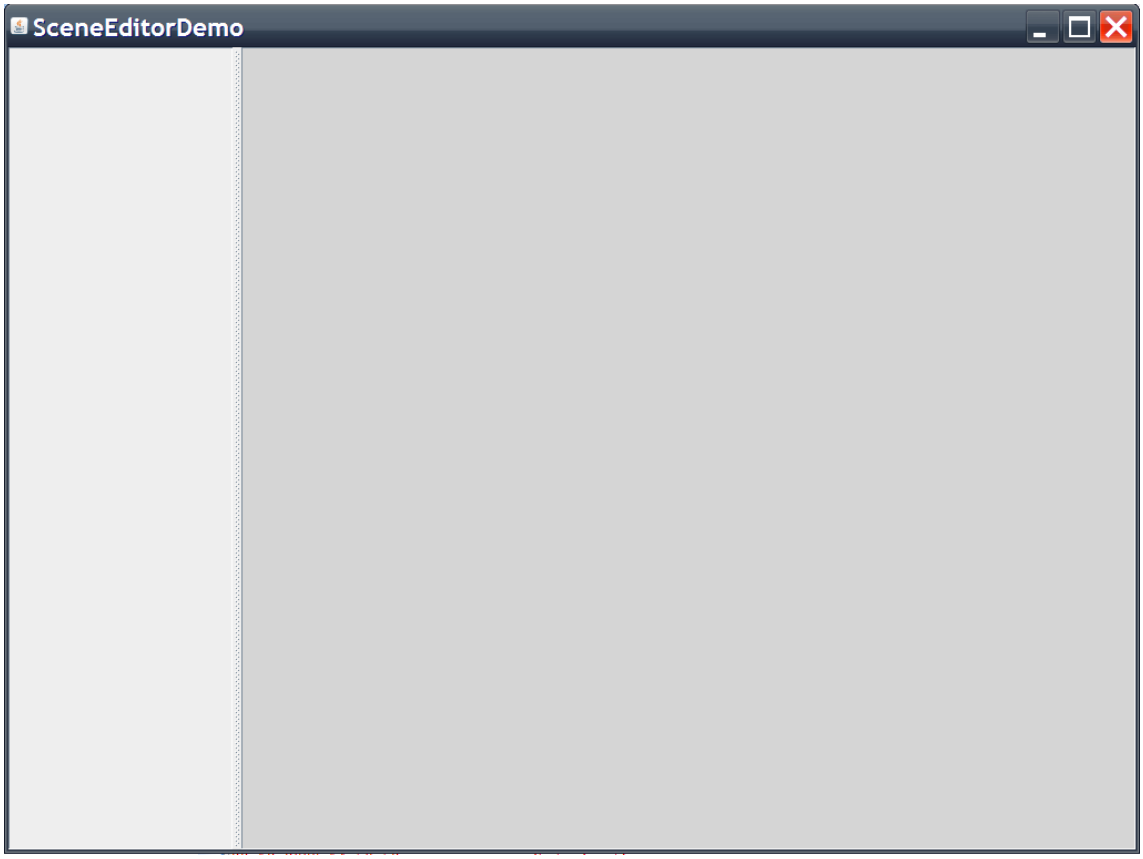
Lernkurs zum Erstellen eines WYSIWYG-Editors in einer 3D Game Engine

1 Starten der Engine in Swingumgebung



Ausgangspunkt der Kurses stellt eine Applikation dar, die einen OpenGL Bereich in der von Java verwendeten grafischen Oberfläche Swing implementiert hat. In diesem Kapitel lernen Sie wie eine jME Anwendung in einer Swing Umgebung eingebaut bzw. verwendet wird.

Als Grundlage für die Umsetzung unseres Editors verwenden wir den Sourcecode vom Partikeleditor (RenParticleEditor.java) [Sla07] im Package jmetest.effects. Ich habe diesen Code für den Kurs auf das notwendigste reduziert, damit die Übersicht erhalten bleibt und wir sinnvoll darauf aufbauend arbeiten können.
Der geplante Editor wird zunächst ähnlich wie eine gewöhnliche Swing Anwendung erstellt indem man die Klasse JFrame erweitert. Im Konstruktor des Programms erstellen wir einerseits die 2D Swing-Komponenten wie Menüleiste, Buttons, Panels usw. andererseits das notwendige OpenGL Fenster:

```
public SceneEditorDemo1()
{
    try
    {
        init();    // initialisiere Glidefenster und GUI
```

Wir unterteilen die graphische Oberfläche (GUI) grob in 2 Bereiche:

- Einen kleineren Bereich auf der linken Seite: dieser soll für Swingkomponenten wie Tabulatoren, Optionen, Buttons, Textfelder usw. zur Verfügung sein
- Einen größeren Bereich auf der rechten Seite: dieser soll das OpenGL Fenster anzeigen, in dem die Szene dargestellt wird.

```
// Panel fuer Glide Fenster
JPanel canvasPanel = new JPanel();
canvasPanel.setLayout(new BorderLayout());
canvasPanel.add(getGlCanvas(), BorderLayout.CENTER);

// Panel fuer Tabs und Eigenschaften
JPanel optionPanel = new JPanel();
optionPanel.setLayout(new GridLayout(2,1));

// Die Angabe einer Minimumsize bei SplitPanels ist Pflicht!
optionPanel.setMinimumSize(new Dimension(30,30));
optionPanel.setPreferredSize(new Dimension(200,10));
canvasPanel.setMinimumSize(new Dimension(100,50));

// Das SplitPanel wird in optionPanel (links)
// und canvasPanel (rechts) unterteilt
JSplitPane split = new JSplitPane();
split.setOrientation(JSplitPane.HORIZONTAL_SPLIT);
split.setLeftComponent(optionPanel);
split.setRightComponent(canvasPanel);
```

Zu beachten ist, dass beide Panels eine Mindestgröße benötigen, da sonst ein dynamisches Verschieben des Splitpanes nicht möglich ist. Anschließend legen wir die Gesamtgröße der Applikation auf feste 1024x768 Pixel fest.

Der OpenGL Bereich wird als zusätzlicher Thread ausgeführt, der laufend auf in Swing aktualisiert werden muss (glCanvas.repaint()).

```
new Thread()
{
      setDaemon(true);

      public void run()
      {
            try
            {
                  while(true)
                  {
                        if(isVisible())
                            glCanvas.repaint();
                        Thread.sleep(20);
                        yield();
                  }
            }
            catch (Exception e)
            {
                   logger.throwing(this.getClass().toString(),"run()",e);
            }
      }
}.start();
```

Der OpenGL Bereich selbst wird dabei mit einer eigens dafür vorgesehen Klasse, dem LWJGLAWTCanvasConstructor erstellt. Der Aufbau der Funktion befindet sich in Zeile 122 des Codes:

```
public Canvas getGlCanvas()
{
     if (glCanvas == null)
     {
          // Erzeugen eines Canvas fuer das 3D Glide Window
          DisplaySystem display = DisplaySystem.getDisplaySystem();
          display.registerCanvasConstructor("AWT",
LWJGLAWTCanvasConstructor.class);
          glCanvas = (Canvas)display.createCanvas(width, height);
          glCanvas.setMinimumSize(new Dimension(100, 100));
```

Zugegriffen kann auf das OpenGL-Fenster erst indem man für diesen Bereich einen Implementor erstellt. Der Implementor kümmert sich um die Update- und Renderlogik, die in einem Spiel ausgeführt werden muss:

```
impl = new MyImplementor(width, height);
((JMECanvas) glCanvas).setImplementor(impl);
```

Die Größe des verwendeten Fensters lässt sich wie in vielen andere Applikationen mit der Maus am Fensterrand vergrößern oder verkleinern. Die von Swing verwendete Oberfläche erlaubt dies zwar von Haus aus ohne weitere Eingriffe, für den 3D Bereich gilt dies jedoch nicht. Deswegen kümmern sich die folgenden beiden Funktionen um die Anpassung und Aktualisierung von dessen Größe:

```
public void doResize()
public void forceUpdateToSize()
```

Die Szene selbst wird im Implementor erstellt. In dieser wird die Kamera positioniert (ohne die überhaupt nichts sichtbar wäre), eine Hintergrundfarbe für den virtuellen 3D Raum erstellt, und die Wurzel unseres Scenegraphs unter dem Variablennamen „root“ global initialisiert.

```
/**Klasse zum Implementieren der Szene */
class MyImplementor extends SimpleCanvasImpl
{
     public void simpleSetup()
     {

     };
```

Diese Implementierungsmethode wirkt recht umständlich, ist aber notwendig um das Ganze in Swing am Laufen zu halten. Arbeitet man ohne Swing/AWT kann man sich das Erstellen des glCanvas und des Implementors sparen. Der zusätzliche Aufwand in Swing erklärt sich dadurch, dass die Eingaben auf Swingebene zuerst geparst und dann in die entsprechenden LWJGL Aufrufe umgesetzt werden müssen. Dies legt auch nahe, dass dadurch Geschwindigkeit verloren geht, selbst wenn der Verlust eher minimal ist.

In der interaktiven Spielumgebung selbst verwenden Spiele diese 2D-Oberfläche meist nicht mehr bzw. verwenden ein eigens dafür geschaffenes OpenGL Menüsystem. Ein solches wurde

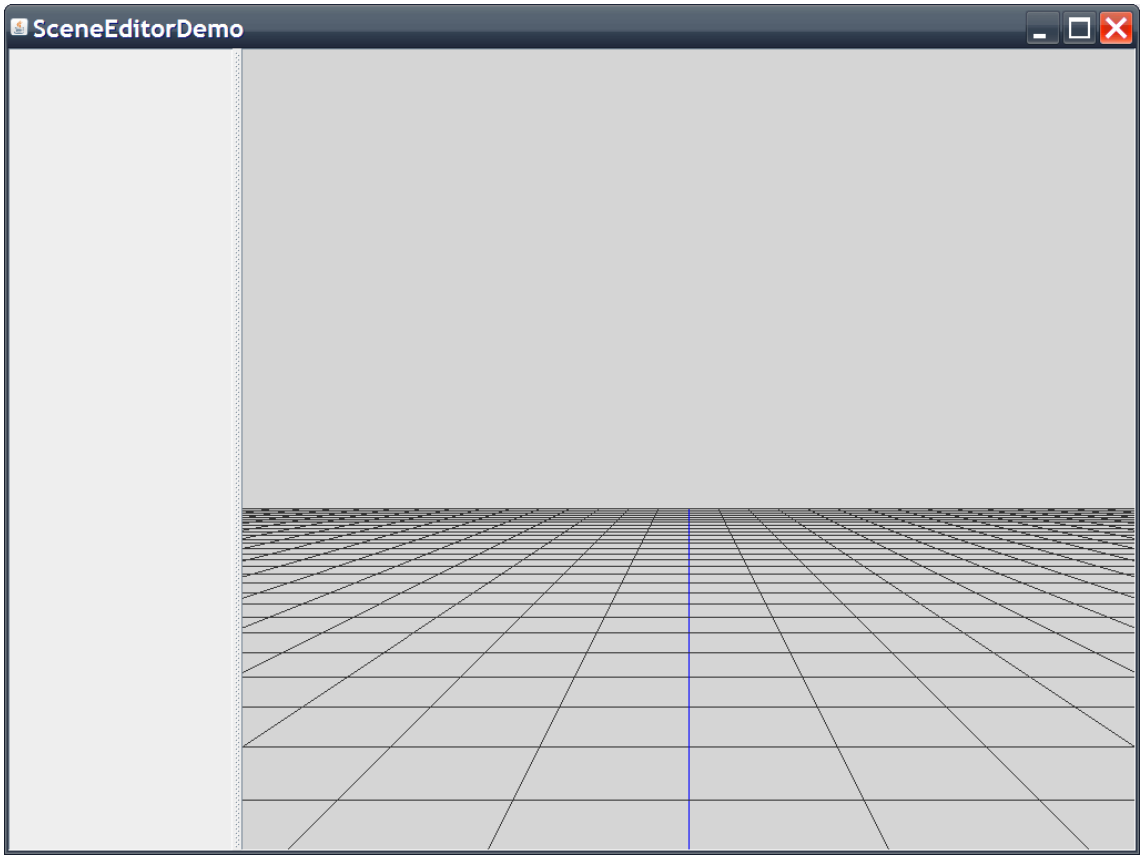
auch für jME entwickelt und ist unter dem Namen „FengGUI“ [Fen07] erhältlich. Es stellt sich natürlich die Frage, warum der Editor nicht gleich mit diesem OpenGL GUI implementiert worden ist:
Ursprünglich stand in meiner Absicht den Editor auf diese Weise umzusetzen; die direkten OpenGL-Aufrufe für Maus und Tastatur waren sehr angenehm und hätten einiges in der Implementierung der Steuerung mit Eingabegeräten vereinfacht. Das Fehlen von notwendigen Komponenten wie Toolbar, Slider, Splitpanes, Probleme beim Aufrufen von Lade- und Speicherdialogen sowie dem Mangel an Auswahl- und Formatierungsmöglichkeiten für das Layout hat mich aber eines besseren belehrt. Viele dieser Funktion sind unumgänglich beim Bau eines Editors. Ergo musste ich zu einer ausgereifteren Benutzeroberfläche greifen wie Swing.

Es folgen weiters die notwendigen Funktionen zum Update und Rendern der Szene. In der Game-Loop, die solange ausgeführt wird, bis das Programm beendet wird, können Objekte in Echtzeit verändert werden. Ein Update der Szene ist deswegen in verschiedenen Bereichen notwendig. Der Vollständigkeit halber schreiben wir gleich alle 3 bekannten Arten der Updates in die simpleUpdate() Methode.

```
public void simpleUpdate()
{
      rootNode.updateGeometricState(0,false);
      rootNode.updateWorldBound();
      rootNode.updateRenderState();
}

@Override
public void simpleRender()
{

}
```

updateGeometricState() kümmert sich um die Aktualisierung aller geometrischen Informationen der im Knoten vorhandenen Objekte (Position, Gesamtanzahl Dreiecke, Normalen usw.).
updateWorldBound() kümmert sich darum die Grenzen des entsprechenden Knotens an die Szene anzupassen, falls Objekte verschoben wurde. Diesen Bound kann man sich vorstellen wie einen riesigen Quader, der alle Objekte in der Szene bzw. dem entsprechenden Knoten umschließt.
updateRenderState() kümmert sich schlussendlich um die Aktualisierung der Beleuchtung der Szene oder der Veränderung von Oberflächenmaterialen wie z.B. die Farbe bei Objekten.

2 Orientieren im virtuellen Raum



Das im vorigen Kapitel erstellte Grundgerüst lässt noch einige Wünschen offen: beispielsweise wissen wir nicht, wohin die Kamera schaut, wo oben und unten ist, links und rechts. Das einzige was wir sehen ist ein leerer Raum. In diesem Kapitel lernen wir, wie man sich im virtuellen Raum orientiert, ein Gitternetz in Abhängigkeit vom verwendeten Koordinatensystem erstellt, und eine Sichtkamera oberhalb dieses positioniert.

Bevor wir ein Gitter (Grid) implementieren müssen wir uns Gedanken machen wo genau dieses in der Scenegraphstruktur positioniert werden soll. In diesem Kurs soll der Scenegraph so aufgespalten werden, dass ein Zweig davon für die Hilfsmittel der Szene zuständig ist, ein anderer für die tatsächliche Szene. Diese Herangehensweise soll es später ermöglichen den gesamten Hilfsknoten dynamisch an- und abzuschalten.
Wir erstellen also einen neuen Knoten den wir „helperNode“ nennen und fügen diesen an den „rootNode“ an. An diesem Knoten werden später noch andere Hilfsutensilien außer dem Gitter angehängt, weswegen wir für die Implementierung dieses noch eine Ebene tiefer steigen müssen.
Wir erstellen dann einen Knoten für das Gitter allein unter dem Namen „gridNode“ (siehe Funktion createGrid()) und fügen diesen dem helperNode hinzu.

```
root = rootNode;

// Knoten "helperNode" erzeugen (fuer Gizmos und Grid)
helperNode = new Node("helperNode");

createGrid();// Grid erzeugen
helperNode.attachChild(gridNode);//Grid an "helperNode" anhaengen

// eliminiere jeglichen Lichteinfluss
helperNode.setLightCombineMode(LightCombineMode.Off);
```

Es wird bereits im Voraus festgelegt, dass dieser Hilfsknoten nicht von Lichtquellen beeinflusst werden darf. Dazu deaktivieren wir den LightCombineMode. Falls das nicht gemacht wird, wird die Farbe der Achsen beim Hinzufügen einer Lichtquelle auf einen Grauwert zurückgesetzt und von hell bis dunkel schattiert. Eine Farbzuweisung wäre dann nur mehr möglich, indem man der Linie ein farbiges Material zuweisen würde. Das Problem der Schattierung bliebe trotzdem.

Bevor wir mit der Implementierung anfangen, müssen wir wissen um welche Art von Koordinatensystem es sich bei jME handelt. Für dreidimensionale Räumlichkeiten unterscheidet man zwei gebräuchliche Arten von Koordinatensystemen:

- Die OpenGL Technologie verwendet ein Right-handed Koordinatensystem
- Direct3D 9 von Microsoft verwendet ein Left-handed Koordinatensystem

Bei beiden Systemen zeigt die positive Y-Achse nach oben und die positive X-Achse nach rechts. Der Unterschied besteht lediglich in der Richtung der Z-Achse: Die positive Z-Achse befindet sich auf jener Seite, wo der Daumen hinzeigt, wobei die linke Hand für das Left-handed und die rechte Hand für das Right-handed Koordinatensystem verwendet wird [Mic09].

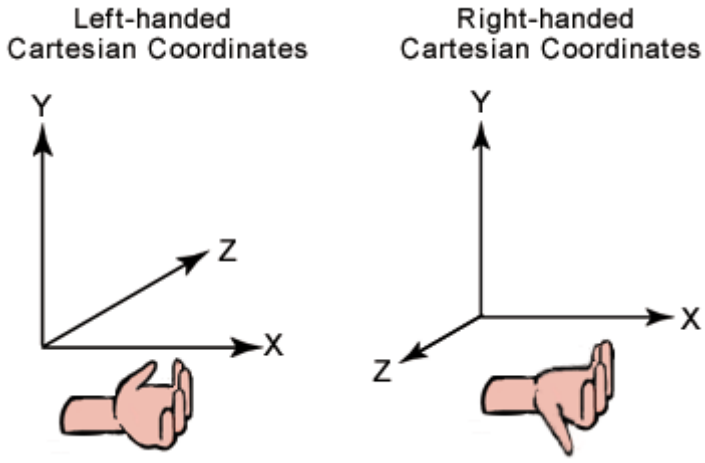


Abb. 4-3: Left-handed und Right-handed Koordinatensystem [Mic09]

Daneben gibt es noch weniger gebräuchliche Koordinatensysteme in denen die Z und Y-Achse vertauscht sind. Objekte die in einem solchen 3D Modellierungs-programm erstellt wurden,

sollten erst entsprechend angepasst (rotiert, gewendet) werden, bevor sie in die GameEngine eingefügt werden.
JME basiert auf OpenGL und verwendet daher das Right-handed Koordinaten-system. Nun können wir loslegen, ein Gitter für die horizontale Ebene zu erstellen.

Eine Linie besteht aus einer Geraden, die von einem Punkt in der horizontalen Ebene zu einem anderen Punkt der Ebene auf derselben Höhe verbunden wird. Um ein Gitter zu erstellen werden mehrere dieser Linien nebeneinander auf gleicher Höhe erstellt und anschließend im rechten Winkel mit anderen Linien gekreuzt, denen dasselbe Prinzip unterläuft.
In jME befindet sich auf der horizontalen Ebene die X und Z-Achse des Koordinatensystems, während Y die Vertikale darstellt. Ausgehend vom Nullpunkt des Koordinatensystems (0,0,0) erstellen wir eine Linie am besten so, indem wir 2 Punkte hernehmen die gleichweit in positiver und negativer Achsenrichtung von einander entfernt sind. In diesem Falle betrifft das +X/-X und +Z/-Z.
Wir erledigen die notwendige Implementierung in der Methode createGrid(); zuerst initialisieren wir für beide Achsen Linien von jeweils -300 bis +300. Wir verwenden dabei einen Abstand von 10 Einheiten im Gitter (Schleifenkopf). Als Gitterfarbe wird ein dunkelgrauer Farbton gewählt, der sich angenehm in den schwarzen Hintergrund einfügt.

```
// Gitternetz
for (int x = -300; x <= 300; x=x+10)
{
     Line l = new Line("xLine" + x, new Vector3f[]{
     new Vector3f((float) x, 0f, -300f),
         new Vector3f((float) x, 0f, 300f)
         }, null, null, null);
     l.setSolidColor(ColorRGBA.darkGray);
     l.setModelBound(new BoundingBox());
     l.updateModelBound();
     l.setCastsShadows(false);
     gridNode.attachChild(l);
}
```

Diese Linien erstellen wir jeweils für die X und für die Z-Achse. Wenn wir damit fertig sind wissen wir aber immer noch nicht, welches die X und Z-Richtung ist. Deswegen entschließen wir uns die Achsen jeweils farblich zu kennzeichnen. Wir ordnen den 3 Achsen des Koordinatensystems jeweils 3 Farben zu: X, Y, Z werden dabei der Reihe nach mit den Farben Rot, Grün, Blau assoziiert.
Das gesamte Gitter farblich zu markieren wäre ziemlich penetrant, weswegen wir lediglich eine einzige Linie – die die den Ursprung schneidet - farblich kennzeichnen. Wir erstellen dazu eine neue Linie mit der jeweiligen Achsenfarbe:

```
// rote x-Achse
final Line xAxis = new Line("xAxis", new Vector3f[]{
    new Vector3f(-300f, 0f, 0f),
    new Vector3f(300f, 0f, 0f)
    }, null, null, null);
xAxis.setModelBound(new BoundingBox());
xAxis.updateModelBound();
```

```
xAxis.setSolidColor(ColorRGBA.red);
xAxis.setCastsShadows(false);
gridNode.attachChild(xAxis);
```

Dasselbe wiederholen wir für die Z-Achse. Das Kennzeichnen der Y-Achse kann man sich allerdings sparen, da diese offensichtlich ist.

In diesem Zusammenhang möchte ich auch die erstmalige Verwendung ModelBounds, sowie einer nicht ganz offensichtlichen Stolperfalle beim Erstellen von Linien oder 3D Objekten erwähnen:

```
l.setModelBound(new BoundingBox());
l.updateModelBound();
```

Damit ein Objekt in der Szene gerendert werden soll oder nicht, muss erst geprüft werden welche Objekte bzw. welcher Teil eines Objekts noch im Sichtfeld enthalten ist. Bei komplizierten 3D Objekten (z.B. Mensch, Tier) würde die Berechnung einen längeren Aufwand benötigen als bei einem einfachen Würfel. Durch das Umschließen komplexer Objekte mit einer sehr einfachen Grundform (standardmäßig eine Kugel oder ein Quader) spart das Programm an Rechenzeit.

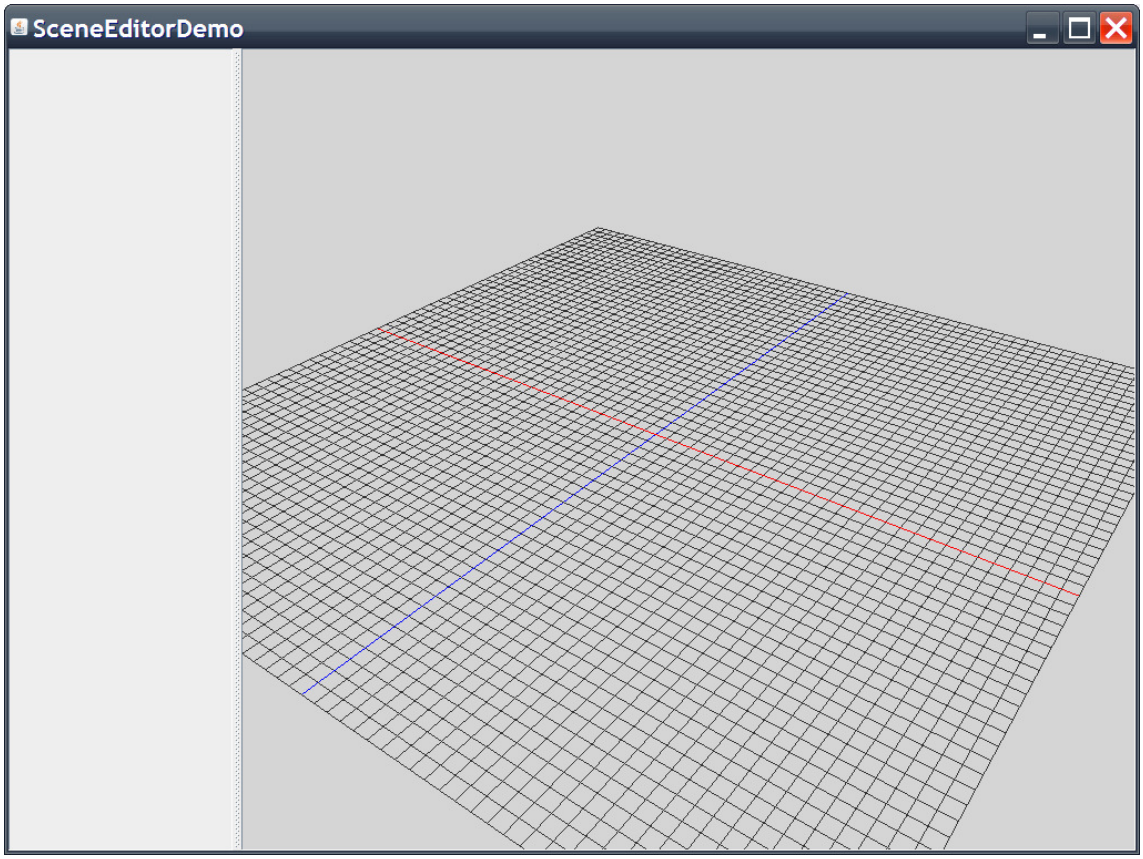
Wo liegt jetzt das Problem?
Durch das Mischen von Objekten mit ModelBounds und ohne ModelBounds in einer Szene kann ein unangenehmer Nebeneffekt passieren: Sobald ein neues Objekt z.B. eine Box in die Szene hinzugefügt wird, vererbt jede Geometrie, die keine eigenen ModelBounds besitzt – z.B. Linien bei denen man vergessen hat diese Bounds zu setzen – die ModelBounds der Box! Sobald diese Box dann außerhalb des Kamera-Sichtfelds gerät (und dabei nicht mehr gerendert wird), werden auch diese Linien bzw. das Gitter nicht mehr gerendert und das Gitter würde auf „magische Weise“ verschwinden. Es empfiehlt sich also immer darauf zu achten, dass entweder alle Objekte ModelBounds besitzen, oder überhaupt keines.

Wenn wir das Programm starten sehen wir lediglich einen horizontalen grauen Strich in der Mitte des OpenGL Bereichs. Der Grund dafür ist, dass die Kamera sich auf derselben Höhe wie das Gitter befindet und das Sichtfeld genau in der Waagrechten liegt. Um das zu ändern positionieren wir die Kamera 20 Einheiten weiter nach oben mittels folgender Zeile:

```
cam.getLocation().y = 20;
```

Wir erhalten nun einen Ausguck mit Sicht auf das Gitter und der blau markierten Z-Achse.

3 Maus- und Tastatureingaben



Obwohl wir uns jetzt in der Szene orientieren können, ist die Perspektive stark eingeschränkt. Wir sind nicht in der Lage die Sicht zu verändern, sie zu drehen, nach oben oder unten zu schauen. In diesem Kapitel lernen Sie, wie man Maus- und Tastatureingaben implementiert und damit die Kamera frei bewegen und steuern kann!

Die starre Sicht resultiert daraus, dass die Kamera an einer fixen Position verankert ist. Wir müssen daher einen Weg finden, die Kamera zu drehen und zu verschieben. JME bietet für diese Möglichkeit bereits zwei vorgefertigter Klassen:

- First-Person Handler
- Third-Person Handler

Ein Third-Person Handler ist eine Kamera, die an ein Objekt angehängt wird (oftmals an ein steuerbares Spielermodell) und dabei leicht oberhalb und hinter dieses Objekt positioniert wird. Diese Kamera folgt dem Modell im immer gleichen Abstand, wie es sich fortbewegt.
Da wir die Kamera aber nicht an ein Objekt anhängen wollen, sondern diese direkt steuern wollen ist für uns nur der First-Person Handler (im Folgenden nur noch FPH genannt) interessant. Die meisten Ego-Shooter spielen sich aus der Ich-Perspektive des gesteuerten

Charaktermodells. Das entspricht gleichsam der Steuerung eines FPH. Eine solche Steuerung erlaubt einerseits das Umschauen in alle Richtungen mit der Maus andererseits ein Fortbewegen des Charaktermodells mit der Tastatur.
Genau dieses Prinzip wollen wir anwenden um die Kamera im Editor zu steuern!
Für die Tastatursteuerung hat sich WSAD als Fortbewegungsfunktion in heutigen Spielen etabliert. Dabei gilt folgendes:

W…nach vorne bewegen
S… nach hinten bewegen
A… seitlich nach links bewegen
D… seitlich nach rechts bewegen

Diese Steuerungselemente sind bereits vollständig in jME's FPH implementiert. Weiters gelten für jME:

Q... nach oben bewegen
Z... nach unten bewegen

Da wir mit einer Swing/AWT Oberfläche arbeiten müssen wir zuerst die entsprechenden Eingabemöglichkeiten für den 3D Bereich aktivieren. Dabei müssen die Tastatur- und Mauseingaben der AWT Oberfläche zu gleichwertigen Befehlen in LWJGL konvertiert werden. Wir fügen folgenden Code in der Funktion getGlCanvas() hinzu:

```
// Aktiviere Eingabemoeglichkeiten fuer Maus und Tastatur!
if (!KeyInput.isInited())
{
      KeyInput.setProvider(InputSystem.INPUT_SYSTEM_AWT);
      KeyListener kl = (KeyListener) KeyInput.get();
      glCanvas.addKeyListener(kl);
}

if (!MouseInput.isInited())
      MouseInput.setProvider(InputSystem.INPUT_SYSTEM_AWT);

((AWTMouseInput) MouseInput.get()).setDragOnly(true);
glCanvas.addMouseListener((MouseListener) MouseInput.get());
glCanvas.addMouseWheelListener(
      (MouseWheelListener) MouseInput.get());
glCanvas.addMouseMotionListener(
      (MouseMotionListener) MouseInput.get());
```

Im Game-Loop müssen wir diesen Input zur Laufzeit aktualisieren:

```
InputSystem.update();
```

Wir erstellen anschließend eine neue Funktion mit dem Namen createInput(). In dieser sollen alle Abschnitte implementiert werden, die mit der Eingabe von Maus und Tastatur zu tun haben. Das Prinzip funktioniert folgendermaßen:

Wir holen uns die bereits vorhandene Kamera vom Implementor und weisen diese einem Inputhandler zu. Als InputHandler wählen wir die Klasse FirstPersonHandler.

```
// hohle die gegebene Camera!
cam = impl.getCamera();

// weise dem FirstPersonHandler die cam zu
input = new FirstPersonHandler(cam,camSpeed, 1);
```

Danach müssen wir uns um die Handhabung von Mausklicks und Tastatureingaben kümmern. Dabei gibt es für die Tasten der Maus und Tastatur jeweils 2 wichtige Zustände:

1. Taste gedrückt
2. Taste losgelassen

Für jede dieser Tasten gilt auch folgendes:

a) Der Befehl kann wiederholt ausgeführt werden, solange die Taste gedrückt ist
b) Der Befehl wird nur 1x ausgeführt, trotz dass die Taste länger gedrückt wird

Die Grundeinstellung für Tastatur und Maus sieht vor, dass die Wiederholung bzw. der Repeat aktiviert ist. Will man diese Funktion abschalten muss man es erst für das entsprechende Eingabegerät deaktivieren.

Für den geplanten Editor müssen wir uns noch ein Konzept für die Steuerung der Maus überlegen. Dazu wägen wir ab, was wir brauchen. Wir wollen auf jeden Fall unsere Kamera mit WSAD fortbewegen und die Kamera mithilfe der Maus schwenken können.
Aufgrund der später noch zu implementierenden Funktionalitäten entscheiden wir uns die Schwenkmanöver nur bei gedrückter mittlerer Maustaste zu verwenden. Die linke und rechte Maustaste werden für andere Zwecke verwendet:
Soweit wir in diesem Kapitel vorwegnehmen können ist das Klicken mit der linken Maustaste dazu gedacht später Objekte in der Szene zu markieren. Das Gedrückthalten bzw. Bewegen der rechten Maustaste soll das Verschieben von Objekten in Echtzeit ermöglichen. Wir benötigen also einen Repeat für die rechte und mittlere Maustaste und keinen Repeat für die linke Maustaste.

Für die Tastatur ist uns der bereits aktivierte Repeat recht, da wir eine flüssige Fortbewegung der Kamera benötigen solang die entsprechende Taste gedrückt wird. Für die Maussteuerung hingegen haben wir Spezielles vor. In unserem Falle wollen wir eine 100% Kontrolle über das Verhalten der Maus erhalten. Wir definieren ein Skelett um den zukünftigen Zugriff auf gedrückte und losgelassene Maustasten zu definieren.

```
InputAction buttonAction = new InputAction()
{
     public void performAction( InputActionEvent iae )
     {
```

```
            if(iae.getTriggerPressed())// gedrueckte Maustasten
            {
                  if(iae.getTriggerIndex()==0)     // LMT gedrueckt
                  {}
                  if(iae.getTriggerIndex()==1)     // RMT gedrueckt
                  {}
                  if(iae.getTriggerIndex()==2)     // MMT gedrueckt
                  {}
            }
            else                              // losgelassene Maustasten
            {
                  if(iae.getTriggerIndex()==0)     // LMT losgelassen
                  {}
                  if(iae.getTriggerIndex()==1)     // RMT losgelassen
                  {}
                  if(iae.getTriggerIndex()==2)     // MMT losgelassen
                  {}
            }
      }
};
```

Damit auf gedrückter und losgelassener Maustaste reagiert wird, müssen wir erst allen Maustasten die beiden Funktionen Pressed/Released zuordnen. Um das zu erzwingen müssen wir zuerst für alle Maustasten den Repeat deaktivieren („false“ setzen am Ende der Zeile). Das erstmalige Einstellen auf „false“ deaktiviert dabei nicht nur den Repeat, sondern es weist auch jedem der Maustasten einen „Pressed“ und „Released“ Event hinzu! Diese nicht ganz offensichtliche Vorgangsweise ist leider nicht dokumentiert und war nur durch Trial & Error herauszufinden.

Da wir den Repeat aber noch für die mittlere und rechte Maustaste benötigen wie wir im Vorhinein festgestellt haben re-aktivieren wir diesen für die beiden Tasten indem wir den Repeat für diese wieder auf „true“ setzen.

Hinweis: falls man die erste Zeile aber weglässt funktioniert der Event "Released" bei all jenen Buttons nicht mehr die den Repeat auf "true" haben (in diesem Falle die mittlere und rechte Taste)!

```
// Allen Maustasten die Funktionen "Pressed" und "Released" zuordnen!
input.addAction(buttonAction,InputHandler.DEVICE_MOUSE,
      InputHandler.BUTTON_ALL,InputHandler.AXIS_NONE,false);

// Anschliessend erlauben wir MT 1 (Rechts) und 2 (Mitte) einen Repeat
input.addAction(buttonAction,InputHandler.DEVICE_MOUSE,
      1,InputHandler.AXIS_NONE,true);
input.addAction(buttonAction,InputHandler.DEVICE_MOUSE,
      2,InputHandler.AXIS_NONE,true);
```

Theoretisch ist es auch möglich Events durch die Mausbewegung allein zu aktivieren. Dazu müsste man nur in der entsprechenden Zeile InputHandler.AXIS_ALL statt AXIS_NONE verwenden. Man könnte dadurch realisieren, dass Objekte in der Szene jedesmal „highlighted“ oder markiert (z.B. mit einem Rahmen umgeben) werden, sobald sich die Maus über diese hinwegbewegt bzw. sie berührt.

Der sogenannte MouseLook, der für das Drehen der Kamera, rauf und runterschauen zuständig ist soll nur beim Drücken der mittleren Maustaste, und nicht bei der linken oder rechten Maustaste funktionieren. Um damit zu arbeiten muss man wissen, aus welchen Bestandteilen ein FPH besteht. Wie bereits zu erraten ist besteht dieser aus genau 2 Komponenten:

a) MouseLookHandler (Position 0 des FPH)
b) KeyboardLookHandler (Position 1 des FPH)

In unserem System lässt sich der Umgang mit der Steuerung also mit folgender Logik beheben: Beim Drücken der linken oder rechten Maustaste soll der MausLookHandler vom FPH entfernt werden, beim Drücken der mittleren Maustaste soll er wieder angefügt werden!

Wir speichern dazu jeden der beiden Handler erstmals in einer globalen Variable ab. Das erlaubt uns später diese dynamisch abgreifen zu können.

```
// speichere MouseLook- und KeyboardLookHandler zuerst global
mouseLook = (MouseLookHandler)input.getFromAttachedHandlers(0);
keyboardLook = (KeyboardLookHandler)input.getFromAttachedHandlers(1);
```

Beim Starten der Applikation ist der MouseLook noch am FPH angehängt. Beim Entfernen dieses sollte man darauf achten, dass nicht versucht wird diesen 100x mal in der Sekunde zu entfernen solang eine Taste gedrückt wird die auf Repeat gesetzt ist. Dasselbe gilt für das erneute Anhängen des MouseLookHandlers an den FPH.
Wir überprüfen dies anhand einer Boolean die beim erstmaligen Drücken bzw. Repeat der linken bzw. rechter Maustaste auf „false“ gesetzt wird und dabei den MouseLookHandler entfernt, während beim erstmaligen Drücken/Repeat der mittleren Maustaste der gespeicherte MouseLookHandler wieder an den FPH angehängt wird, und die Boolean auf „true“ gesetzt.

```
if(mouselookAttached)
{
	// entferne MouseLookHandler (wird nur 1x ausgefuehrt)

input.removeFromAttachedHandlers(input.getFromAttachedHandlers(1));
	mouselookAttached = false;
}
else
{
	// System.out.println("MouseLook ist inaktiv!");
}
```

Sobald der MouseLook entfernt ist, wird nur noch der else-Zweig ausgeführt.

Das erneute Anhängen funktioniert ähnlich:

```
glCanvas.requestFocus();

if(!mouselookAttached)
{
	input.addToAttachedHandlers(mouseLook);
	mouselookAttached = true;
}
```

```
else
{

}
```

Hinweis: der Fokus auf den 3D Bereich wird standardmäßig nur mit einem Klick der linken Maustaste aktiviert (z.B. wenn vorher auf Swing Buttons gedrückt wurde). Da wir jedoch die mittlere Maustaste verwenden, müssen wir den Fokus manuell erzwingen!

Wenn der Code mit diesen Veränderungen gestartet wird, rührt sich trotzdem nichts. Grund dafür ist, dass die Eingabe nicht aktualisiert wird im Game-Loop. Das kann man nachholen in der simpleUpdate() Methode:

```
InputSystem.update();
if(input!=null)
      input.update(tpf);
```

Von nun an kann sich die Kamera frei bewegen, wie in einem Ego-Shooter. Das einzig störende ist noch das Hoch- und Runterbewegen der Kamera. Das Hinunterbewegen ist auf die Taste Z belegt, die für deutsche Tastaturen unangenehm weit von den Tasten WSAD entfernt ist.
Wir lösen das Problem, indem wir dem KeyboardLook unseres FPH (ab Zeile 244 im Code nachzulesen) eine neue Taste für diese Funktion zuweisen. Dabei wählen wir die Taste C zum Runterbewegen und die Leertaste zum Raufbewegen der Kamera. Diese Steuerung ist nun gleich wie der sogenannte Spectatormodus im Spiel Unreal Tournament 2004. In herkömmlichen Spielen werden diese beiden Tasten oft verwendet um Spielaktionen wie Ducken und Springen auszuführen.

4 Strukturieren des Scenegraphs

Da die kommen Abschnitte im Kurs immer komplexer zu werden, wird es Zeit eine Ordnung in die Struktur des Scenegraphs zu bringen. In diesem Kapitel lernen Sie, was ein Scenegraph ist und wozu er gebraucht wird.

4.1 Scenegraph

Der Scenegraph erlaubt es Szeneelemente in einer Art Baumstruktur zu unterteilen. Je nach Art der Implementierung bestehen diese Szeneelemente nicht nur aus 3D-Objekten, sondern auch in Soundelementen oder Lichtquellen.
Ausgehend von einem Wurzelknoten kann sich der Scenegraph in viele weitere Unterbäume aufteilen. Elternknoten (parent node) können eine beliebige Anzahl Kinder (children) besitzen, Kinder können jedoch nur ein einziges Elternteil. Dieses Konzept erlaubt einerseits das Zusammenfassen von zusammengehörigen Objekten in einem Knoten, andererseits das Verarbeiten von ganzen Unterbäumen durch einen einzigen Zugriff. Dazu folgendes Beispiel:
Angenommen Sie erzeugen ein Gebäude. Das Gebäude besitzt mehrere Etagen. Jede Etage besitzt mehrere Zimmer. Jedes Zimmer beinhaltet Tische, Stühle oder andere Möbel. Wenn Sie jetzt dynamisch Zimmer oder ganze Stockwerke entfernen wollen inklusive aller darin befindlichen Gegenstände oder Räumlichkeiten brauchen Sie diese Zweigpunkte (Gebäude, Etage, Zimmer) nur als Knoten zu definieren. Diese Knoten können dann beliebig entfernt werden mitsamt den darin vorkommenden Gegenständen (z.B. Möbel). Ebenso können ganze Zweige wieder an bestimmte Knoten angefügt werden.
Der größte Vorteil eines Scenegraphs besteht aber in einer Technik genannt „Culling“. In einer 3D Szene werden all jene Objekte durch die Grafikpipeline gerendert, die im Sichtfeld der Kamera liegen. Bei einer Unterteilung in einer hierarchischen Struktur ist es jedoch möglich, ganze Sub-Bäume die außerhalb des Sichtfelds in einem Knoten liegen zu umgehen und damit enorm an Performance zu sparen [Bis98]. JME besitzt eine solche Implementierung.

Bis jetzt besteht unser aktueller Scenegraph aus lediglich 3 Knoten. Der erste Knoten stellt die Wurzel des Scenegraphs dar. An ihr hängt bis jetzt nur der Hilfsknoten.
Wie schon in den vorigen Kapiteln erwähnt soll dieser nur für Hilfsutensilien zuständig sein. Da wir für die zu erstellende Szene noch keinen Knoten definiert haben, ist jetzt die Zeit gekommen das umzusetzen. Um das Verständnis zu erleichtern skizziere ich den geplanten Aufbau des Baumes mit einem Diagramm:

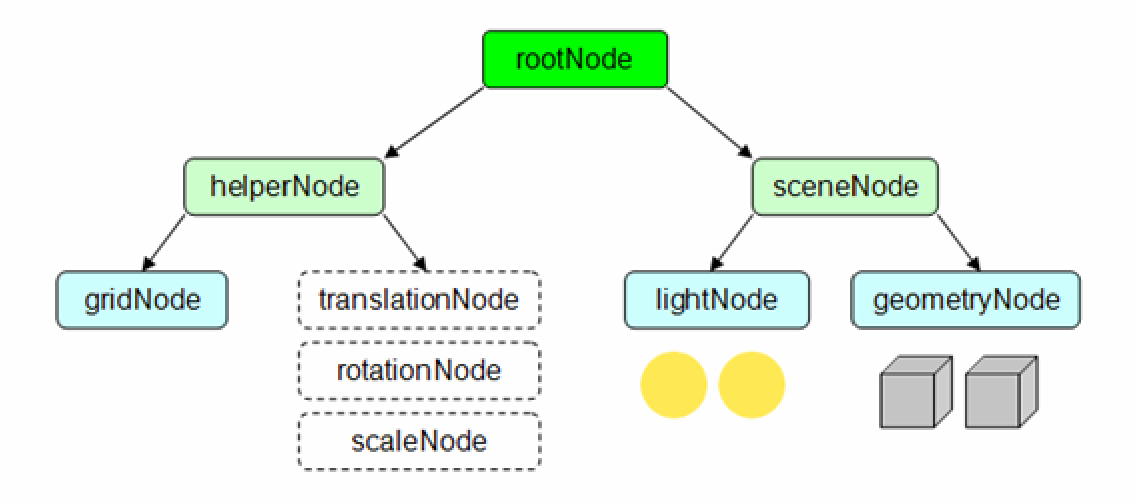


Abb. 4-4: Scenegraph für unseren Editor

Erklärung:
Die Szene wird im sceneNode dargestellt. Der Übersicht wegen vereinbaren wir, dass alle Lichter, die zur Szene hinzugefügt werden in einen gemeinsamen Lichtknoten (lightNode) gesetzt werden. Das soll später ein schnelles und direktes Zugreifen auf diese ermöglichen.
Alle geometrischen Objekte wie Quader, Kugeln, importierte 3D-Modelle werden hingegen in einen Geometrieknoten (geometryNode) gesetzt. Beide zusammen bilden die virtuelle Szene, die visualisiert werden sollen.

Im simpleSetup() fügen wir die folgenden Zeilen dazu:

```
sceneNode = new Node("sceneNode");

geometryNode = new Node("geometryNode");
lightNode = new Node("lightNode");

sceneNode.attachChild(geometryNode);
sceneNode.attachChild(lightNode);

root.attachChild(sceneNode);
```

Wir deklarieren die 3 neu hinzugekommen Nodes als statisch, weil sie immer im Scenegraph vorhanden sein sollen. Näheres dazu aber im Kapitel zum Laden und Speichern einer Szene.

Komplexe Szenen benötigen allerdings eine ausgefeiltere Scenegraphstruktur. Beispielsweise wollen wir der Szene eine Landschaft, ein Haus, sowie ein Auto hinzufügen das darauf fahren soll. Das Auto selbst soll 2 sichtbare Vorderlichter als Leuchtquelle besitzen. Diese Lichter müssen sich immer mit diesem Gefährt mitbewegen. Mögliche Darstellung:

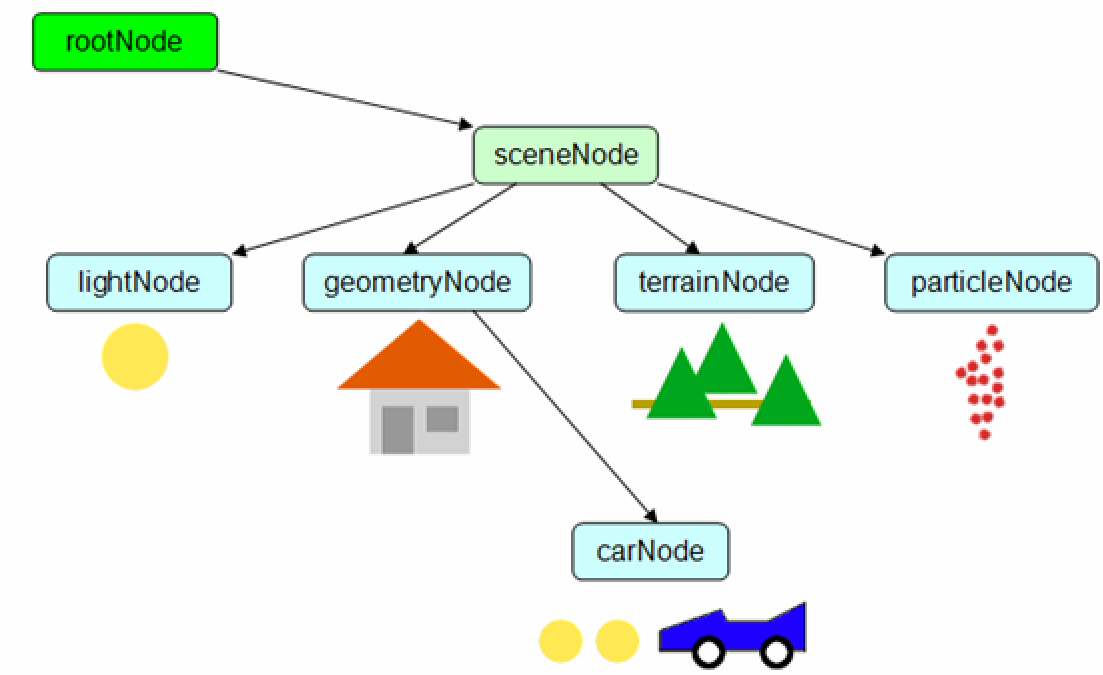


Abb. 4-5: Scenegraph mit komplexerer Unterteilung

In obigem Bild würden wir geometrische Objekte wie Häuser oder Autos im geometryNode erstellen. Da das Auto sich in mehrere Komponenten unterteilt müssen wir eine Ebene tiefer gehen und einen Knoten definieren, der diese beinhaltet. In diesem Falle wäre es auch umständlich die Fahrzeuglichter in den lightNode zu setzen, da diese sich ja mit demselben Knoten bewegen sollen, in dem sich das Auto befindet.

Die obige Struktur des lightNodes könnte dennoch beibehalten werden beispielsweise für das Verwenden von Lichtern, auf die während des Spiels gezielt und schnell zugegriffen werden muss, ohne dabei die gesamte Scenegraphstruktur abklappern zu müssen. Als Beispiel dafür wäre das Implementieren einer Sonne zu nennen, die beim Spielstart initialisiert werden soll und um die Szene kreisen soll.

Der carNode ist nur ein einfaches Beispiel, wie der Scenegraph erweitert werden könnte; will man mehrere Autos darstellen, müssten aber all diese carNodes an den geometryNode angehängen.

Man kann sich ausmalen, dass die Komplexität der Struktur je nach Anforderung der zu implementierenden Szene abhängt, und das Anordnen von Objekten mehrere Ebenen tief gehen kann.

Alternativ ist es natürlich auch möglich alle Objekte in der Szene stur dem rootNode anzuhängen. Dies mag für einige Projekte durchwegs sinnvoll und der schnellere Weg sein. Wie man den Scenegraph letztlich umsetzt liegt im Sinne des Erfinders.

Sollten Sie darauf bestehen neue Knoten im Editor dynamisch erstellen zu können, empfiehlt sich die Implementierung eines sichtbaren Baumes im GUI, der die Szene darstellt und in

welchem Objekte dynamisch per Drag&Drop an verschiedene Knoten angehängt werden können. Für unseren Kurs verwenden wir jedoch die Struktur aus der ersten Abbildung.

4.2 Renderreihenfolge

Die Hilfsutensilien, die in weiteren Kapiteln dazugekommen, verlangen nach einer besonderen Handhabung. Dabei spreche ich von der Renderreihenfolge. Wir legen dazu fest, dass alle Lichtobjekte, die wir später in den lightNode setzen, immer sichtbar gerendert werden sollen. Das erleichtert uns später das Auffinden von Lichtquellen in der Szene, selbst wenn diese von Objekten verdeckt sein sollten.
Dazu stellen wir zuerst ein, dass die gesamte Szene sichtbar gerendert werden soll. Das definieren wir an root. Daraufhin kümmern wir uns um die Renderreihenfolge:
Alle Objekte in jME besitzen standardmäßig einen ZBuffer von „LessThan“, was bedeutet, dass Objekte in „natürlicher“ Reihenfolge von hinten nach vorne gerendert werden, sprich: wenn Objekt, in einem größeren Abstand von der Kamera ist als ein anderes Objekt, das näher zur Kamera ist, wird zuerst das Hintere und danach das Vordere gerendert!
Für unsere Lichtobjekte stellen wir allerdings etwas ganz anderes ein: wir wollen dass die Lichter (die später durch farbige Hilfskugeln gekennzeichnet werden), immer sichtbar sind, selbst wenn sie sich hinter Objekten befinden sollten! Dazu stellen wir den Zbuffer des Lichtknotens auf „Always“.

```
root.setRenderQueueMode(Renderer.QUEUE_OPAQUE);

...

ZBufferState zbuffer = renderer.createZBufferState();
zbuffer.setEnabled(true);

// ersetzt jeden Pixel!
zbuffer.setFunction(ZBufferState.TestFunction.Always);
```

Damit sich Lichtquellen bzw. die Hilfskugeln nicht gegenseitig beleuchten, setzen wir diesen Zbuffer als Renderstate und stellen die Objekte darin sichtbar ein.

```
// LightSpheres im "lightNode" sollen immer sichtbar sein
lightNode.setRenderState(zbuffer);
lightNode.setRenderQueueMode(Renderer.QUEUE_OPAQUE);
```

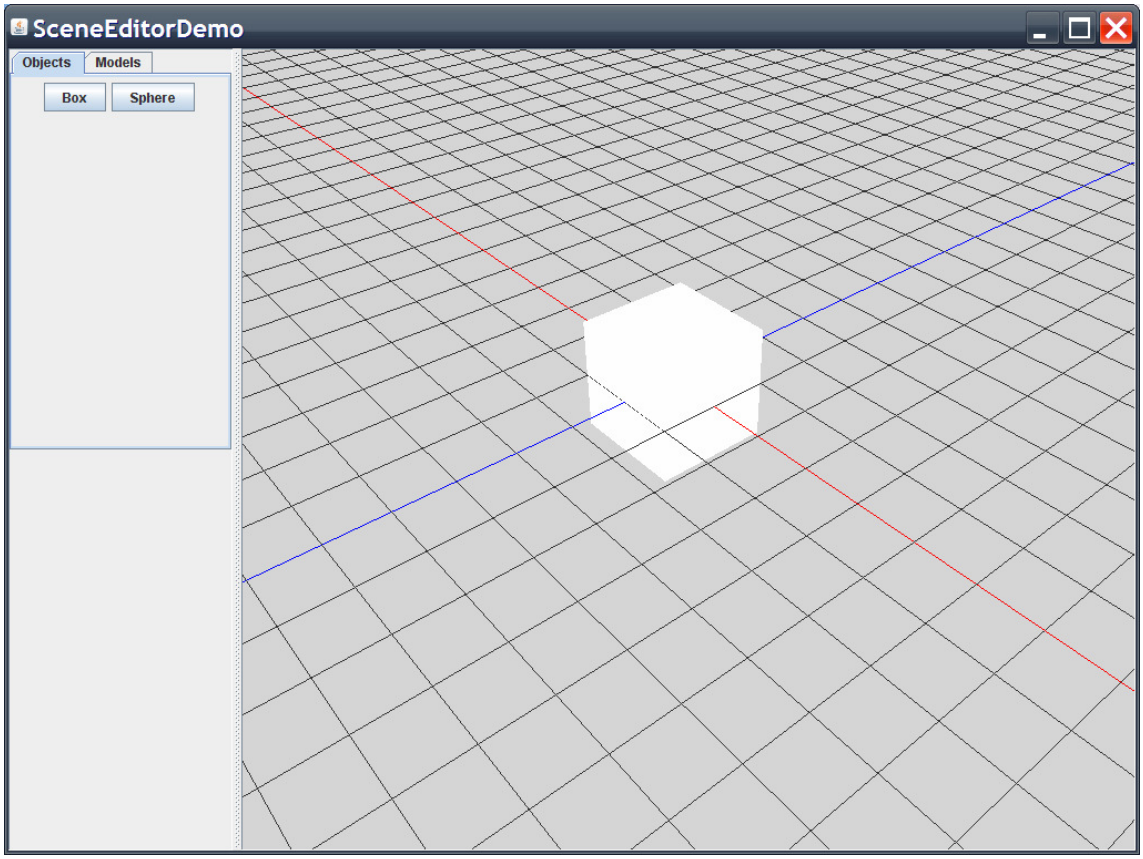
Ist es nicht so, dass dadurch die Lichtkugeln störenderweise unterhalb und durch andere Objekte hindurch sichtbar sind?
Beim späteren Abspeichern der Szene legen wir fest, dass nur das Licht selbst abgespeichert, nicht aber die Lichtkugel. Weiters wird die Renderreihenfolge nicht gespeichert. Wenn nichts weiter eingestellt wird folgt das Licht (nicht die Lichtkugel) selbst der Renderreihenfolge „LessThan“. Das Laden einer Szene in den Editor weist hingegen eine andere Reihenfolge auf, damit es möglich ist, die in der Szene vorhandenen Lichter zu bearbeiten.

Nicht zu verwechseln sind obige 2 Zeilen auch mit dieser:

```
// eliminiere jeglichen Lichteinfluss
helperNode.setLightCombineMode(LightCombineMode.Off);
```

Während beim lightNode lediglich die Hilfskugeln immer sichtbar und vom Licht unbeeinflusst gerendert werden sollen (ansonsten sieht man bei einer roten und gelben Lichtquelle einen gelben Lichtschein auf der roten Leuchtkugel und umgekehrt), soll der helperNode überhaupt keinen Lichteinfluss erhalten. Beim lightNode selbst sollen sich die Lichter aber gegenseitig überschneiden können, Farbmischungen ergeben usw., deswegen dürfen wir nicht auf dasselbe Prinzip (aka LightCombineMode.Off) setzen!

5 Hinzufügen von Objekten



Wir können uns nun im 3D Raum bewegen, haben eine angemessene Scenegraphstruktur organisiert und halten eine bestimmte Renderreihenfolge ein.
Von nun an konzentrieren wir uns auf die Implementierung von Szeneobjekten. In diesem Kapitel lernen Sie, wie man verschiedene Objekte erstellt, von externen 3D-Modellierungsprogrammen importiert und diese per Knopfdruck in die Szene hinzufügt!

5.1 Basisobjekte

Für nahezu jede GameEngine gibt es die Möglichkeit Grundformen in die Szene hinzuzufügen. Das umfasst die Erstellung von 3D Objekten wie Würfel, Zylinder, Kugel, Pyramide etc. als auch 2D Objekte wie Linie, Kreis, Quadrat.

In unserem Kurs implementieren wird lediglich 2 der Grundformen: einen Würfel (Box) und eine Kugel (Sphere). Dazu erweitern wir das GUI erstmals mit einem Tabulator und 2 neuen Buttons in der init() Methode. Diese Buttons fügen wir einem ActionListener (SpatialCreator) hinzu.

Betrachten wir nun den Code im ActionListener:

```
Box box = new Box("BOX",new Vector3f(0,0,0),10,10,10);
```

```
box.setLocalTranslation(0,0,0);
box.updateGeometricState(0,false);
box.setModelBound(new BoundingBox());
box.updateModelBound();
box.updateRenderState();
geometryNode.attachChild(box);

geometryNode.updateGeometricState(0,false);
geometryNode.updateRenderState();
```

Wir erstellen also eine Box mit dem Namen „BOX“. Der Ordnung halber vereinbaren wir, dass alle Objektnamen immer in Großbuchstaben geschrieben werden. Den Namen benötigen wir später, um den einzelnen Objekttyp zu identifizieren.

Die Größe des Quaders legen wir mit den Maßen Länge x Breite x Höhe = 20x20x20 fest; wenn wir 3 Längen hinschreiben wie die obigen „10,10,10“ bedeutet es, dass sich die Box jeweils 10 Einheiten in positiver und (!) negativer in X/Y/Z-Richtung erstreckt, was eine Gesamtlänge von 20 Einheiten pro Achse ergibt.

Nun zu einer beliebten Stolperfalle:

Der zweite Wert im Konstruktor der Box (new Vector3f(0,0,0)) stellt nicht die Position der Box dar, sondern den „Ursprung“ der Box!

Der Ursprung der Box befindet sich bei einer Vektorzuweisung von (0,0,0) genau im Zentrum der „Extends“. Die Box wird immer relativ zum Ursprung positioniert.

Theoretisch könnte man den Ursprung des Quaders auch in die „hintere untere Ecke“ versetzen, indem wir folgende Zeile verwenden:

```
Box box = new Box("BOX",new Vector3f(-10,-10,-10),10,10,10);
```

Jetzt befindet sich der Ursprung im äußersten hinteren Eck des Würfels. Wenn wir diesen Würfel jetzt zur Szene hinzufügen, sehen wir, dass der Würfel auf der Gitterfläche aufliegt und nur in die positive Achsenrichtungen hineinreicht.

Und ja – es ist sogar möglich den Ursprung der Box außerhalb der Box selbst festzusetzen!

Der Ursprung der Box ist nicht zu verwechseln mit der Position der Box! Die Position selbst wird erst mit setLocalTranslation() festgelegt. Dabei wird der Ursprung der Box (und die Box in deren Abhängigkeit) genau auf diese Position gesetzt!

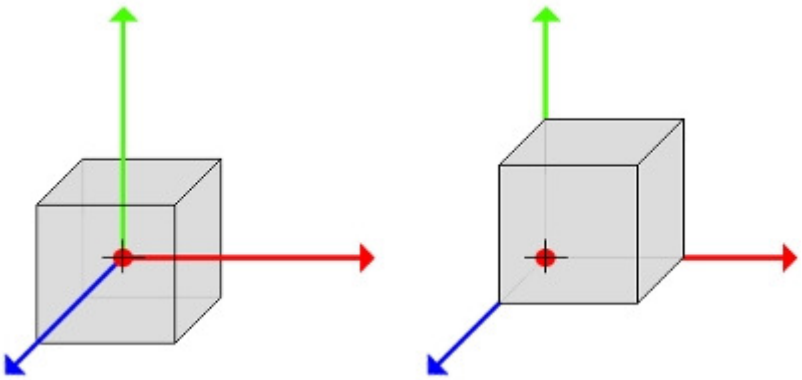


Abb. 4-6: Links: Ursprung in (0,0,0), Rechts: Ursprung in (-10,-10,-10)

Das Hinzufügen einer Sphere funktioniert sehr ähnlich und kann im Quellcode nachgelesen werden.
Im Übrigen verwenden wir für beide Objekte vorgegebene Größen, ohne diese dynamisch zu gestalten. Das hat einen ganz bestimmten Grund: ein Aufbau von Szenen wird grundsätzlich nicht aus den (modifizierten) Grundformen der GameEngine realisiert sondern aus Objekten, die in einem 3D Modellierungsprogramm erstellt und dann in die Szene der GameEngine eingefügt wurden. Näheres dazu aber später.

5.2 Import von externen 3D Modellen

Wir wollen nicht nur in der Lage sein primitive Objekte zu erstellen, sondern auch wissen, wie Objekte, die in 3D Modellierungsprogrammen erstellt wurden, in die GameEngine eingefügt werden können! Dazu stelle ich Ihnen ein eigens zu diesem Zwecke ein in „blender“ [Ble08] modelliertes Haus zur Verfügung. Dieses Objekt (house.obj) wurde als .obj Datei exportiert und ist im Kursordner des aktuellen Kapitels zu finden.

Um die Datei zu laden ohne von einem fixen Pfad abhängig zu sein müssen wir zuerst einen Dateiauswahldialog erstellen. Initialisiert wird der FileChooser bereits beim Start des Programms um diesen vorzuladen (dadurch ladet das Dialogfenster schneller, wenn es aufgerufen wird).
Um nur akzeptierte Dateiformate anzuzeigen benützen wir einen Parser, der uns entweder .jme oder .obj Dateien anzeigt. Das .jme Format benötigen wir später. Für uns interessant ist jedoch nur das .obj Format.

Springen wir zur Funktion in Zeile 345, die das 3D Modell in die Szene laden soll:

```
File objectfile = fc.getSelectedFile();
URI uri = objectfile.toURI();
URL url = uri.toURL();
```

JME verlangt explizit eine URL als Pfadangabe, die wir zuerst aus der URI erstellen müssen.

Daraufhin verwenden wir eine von jME implementierte Technik, die es ermöglicht ein 3D Modell in ein binäres Format umzuwandeln und auf diese Weise zu speichern.
Dazu erstellen wir einen Formatkonverter und konvertieren die .obj Datei während dem Ladevorgang in ein binäres Format um und behält dieses in der Szene:

```
// Erstelle einen Format-Konverter
FormatConverter converter = new ObjToJme();
// Point the converter to where it will find the .mtl file from
converter.setProperty("mtllib", url);

// Das importierte Objekt soll in einen binaere Stream umgewandelt
werden
ByteArrayOutputStream BO = new ByteArrayOutputStream();
```

```
try
{
	// konvertiere von .obj zu .jme
	converter.convert(url.openStream(), BO);

	Spatial obj = (Spatial)BinaryImporter.getInstance().load(new
ByteArrayInputStream(BO.toByteArray()));

	obj.setModelBound(new BoundingBox());
	obj.updateModelBound();

	obj.setName("OBJ");

	// Importiertes Model an "geometryNode" anhaengen
	geometryNode.attachChild(obj);

	// Dem Objekt ein Standardmaterial hinzufuegen wenn es keines
besitzt
	// (im Gegensatz zu Box, Sphere...), sonst Fehler bei der
Beleuchtung
	MaterialState material =
DisplaySystem.getDisplaySystem().getRenderer().createMaterialState();
	material.setDiffuse(ColorRGBA.white);
	obj.setRenderState(material);

	geometryNode.updateGeometricState(0,false);
	geometryNode.updateRenderState();
}
```

Wir geben dem Modell eine Box als ModelBound, die das Haus am einfachsten umschließt.

Unser Haus besitzt außerdem keine Texturen, deswegen müssen wir dem Objekt zumindest irgendein Material zuweisen, damit es korrekt in der Szene dargestellt wird. Wir weisen dem Objekt ein Standardmaterial hinzu, in diesem Fall eine weiße Farbe. Damit erhält es beim Hinzufügen einer Lichtquelle dieselbe Schattierung wie die Kugel und der Würfel.

Falls man texturierte Objekte verwendet, werden die Texturdaten im Falle von .obj Modellen in der Datei mit der Endung .mtl gespeichert. Um die Zuweisung einer Textur zu realisieren braucht man den vorhandenen Code nur ein wenig abändern.

5.3 Eigenschaften eines 3D Objekts

Da ich bestimmte Funktionalitäten in Abhängigkeit von 3D Objekten später im Kurs behandle, sollen hier ein paar Grundeigenschaften eines dreidimensionalen Objekts erklärt werden. Das umfasst vor allem die Definition ihres Aussehens. Werfen Sie dazu einen Blick auf diesen Würfel:

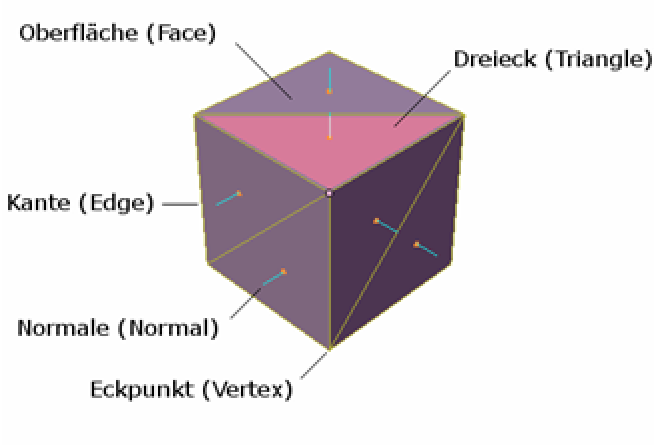


Abb. 4-7: Eigenschaften eines 3D Objekts

Was Kanten und Eckpunkte sind wird Ihnen vermutlich schon klar sein, deswegen werde ich sie hier nicht näher erläutern. Uns interessieren die Normalen, die Oberflächen und die Dreiecke.
Jedes 3D Objekt besitzt mehrere Oberflächen, auch „Faces“ genannt. Diese Oberflächen können einerseits in Dreiecke, als auch in Vierecke unterteilt werden. 3D Modellierungsprogramme wie Blender [Ble08] besitzen die Fähigkeit ein Objekt auf beide Arten zu unterteilen. In obiger Skizze würde ein Viereck also einer ganzen Seitenfläche des Würfels entsprechen. Viele GameEngines, darunter auch jME, unterstützen aber nur Dreiecke!
Diese Dreiecke werden später noch wichtig, wenn wir das Implementieren von Lichtquellen behandeln. Achten Sie beim Modellieren von 3D Modellen insbesondere darauf, dass die Normalen der Oberflächen nach außen zeigen und nicht nach innen! Wenn eine Normale nach innen zeigt, wird diese von einer Lichtquelle nicht gerendert und die Oberfläche mit dieser Normale erscheint daher schwarz!

5.4 Bekannte 3D Austauschformate

Die Entwicklung von verschiedensten 3D Modellierungsprogramme im Bereich des Computer Aided Design (CAD) hat schon seit den 70er Jahren stattgefunden. Unterschiedliche Konzepte zum Speichern eines 3D Objekts führten damals zu ebensovielen unterschiedlichen Speicherformaten. Das Fehlen von Standards zwischen diesen Variationen verhinderte einerseits das Weiterverwenden dieser Daten von Dritten, andererseits hielt es Entwicklungen im 3D Bereich zurück [Gee05].
Bedauerlicherweise gibt es aber kein einzelnes Format, das allen 3D Programmen gemein und damit allgemein akzeptiert und verwendbar ist. Alle 3D Programme haben mittlerweile ihr eigenes spezifisches Dateiformat entwickelt. Um Daten zwischen zwei 3D-Programmen A und B auszutauschen, muss erst ein Format F gewählt werden, das A exportieren kann und B importieren kann [Hav07].
Diese Formate beziehen diejenigen Eigenschaften von 3D Objekten mit ein, die allen gemeinsam und standardmäßig gegeben sind: diese umfassen beispielsweise die Anzahl

Eckpunkte, in welche Richtung die Normalen der Eckpunkte zeigen, die Koordinaten der Eckpunkte, die Anzahl Dreiecke, die Farbe usw. [Ros97]

Gängige 3D Modellierungsprogramme bieten zwar die Möglichkeit mithilfe spezieller Plugins 3D-Objekte im proprietären Format eines anderen 3D Programms abzuspeichern bzw. zu exportieren, jedoch sind diese manchmal verlustbehaftet, da nicht immer alle Spezifikationen des anderen Dateiformats preisgegeben wurden.

Diese Entwicklung im Bereich der Formate betrifft nicht nur 3D Modellierungsprogramme sondern auch 3D GameEngines. 3D Objekte für Spiele werden zuerst in einem 3D Modellierungsprogramm kreiert, in ein entsprechendes Dateiformat exportiert, und anschließend in die 3D GameEngine importiert. Damit 3D Modellierungsprogramme und 3D GameEngines untereinander kommunizieren können benötigen sie also ebenfalls diese Schnittstelle.

JME unterstützt eine Menge von Austauschformaten für 3D Modelle. Zu den bekanntesten und gebräuchlichsten zählen Formate mit folgender Endung:

- .3ds, ist ein proprietäres Format der Software 3ds Max von AutoDesk; die Spezifikationen für das Dateiformat sind leider nicht öffentlich erhältlich, jedoch gibt es Versuche dieses trotzdem zu verwenden [Lew98]
- .obj, auch Wavefront Object genannt, ist ein einfaches Datenformat um geometrische und polygonale Daten in ASCII Form darzustellen [Wav98]
- .ase, steht für ASCII Scene Exporter. Das Format wird von 3D Studio Max verwendet, stellt aber auch ein gebräuchliches Format für das Importieren von statischen Objekten in UnrealEd dar [Bey08]

Weiters werden Formate unterstützt, die ursprünglich in der Engine „idtech 2“ (Quake 2), „idtech 3“ (Quake 3 Arena) und „idtech4“ (Quake 4, Doom 3) angewendet wurden. Diese Formate ermöglichen auch das Speichern von Animationsdaten.

- .md2 [Hen04]
- .md3 [Pha06]
- .md5 [Hen05] , offiziell nicht von jME unterstützt

Ein universelles Austauschformat ist COLLADA: Collada ist eine Spezifikation in XML Format die den Austausch von digitalen Gütern in 3D Autorensystemen ohne Verlust ermöglicht. Die Spezifikationen zur Anwendung sind öffentlich zugänglich und können von jedem verwendet werden [Son08].

- .dae

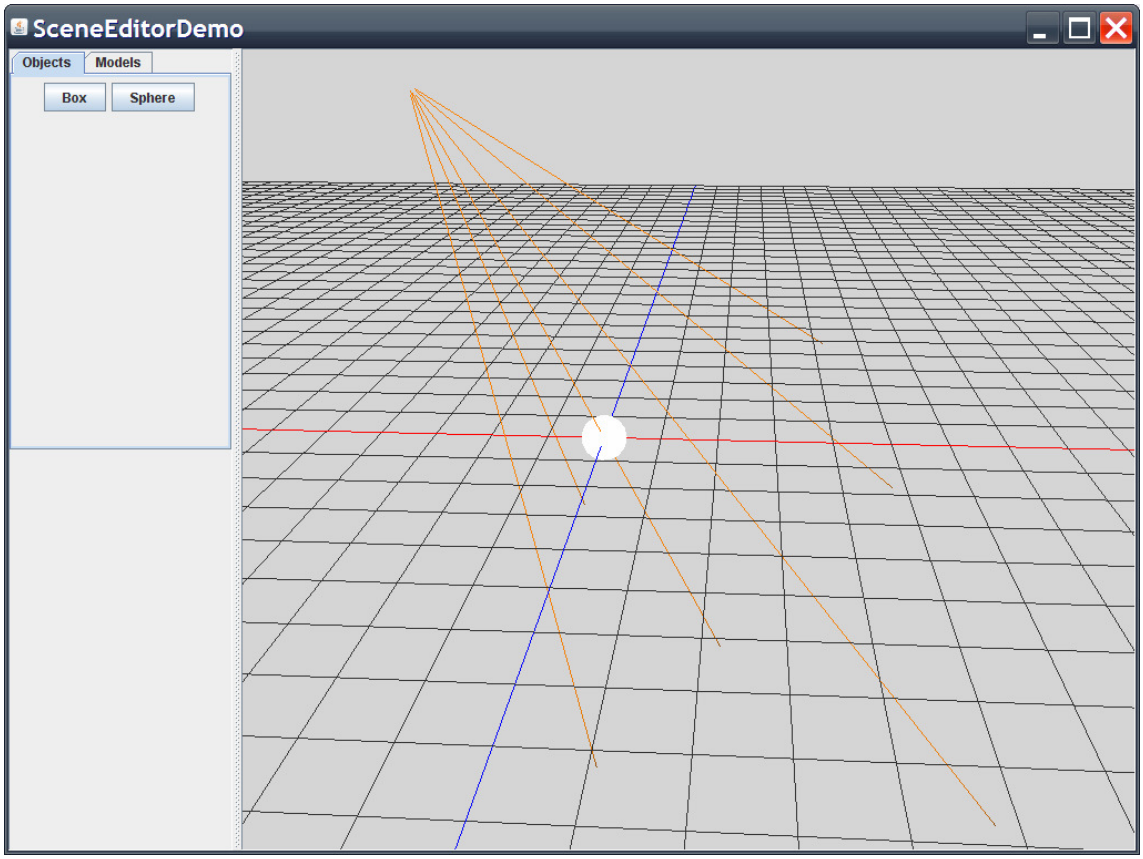
Weiters wäre das Format .lwo (Lightwave Object) von Lightwave zu nennen. Dieses wird von jME nicht unterstützt.

Beim Arbeiten mit GameEngines sollten Sie darauf achten, dass die GameEngine ein Format unterstützt, das in ihrem Lieblings-3D-Modellierungsprogramm exportiert werden kann. Tatsächlich gibt es einige GameEngines, die nur ein einziges 3D Austauschformat verwenden.

Wenn Ihr 3D Modellierungsprogramm die notwendigen Exportformate nicht kennt, bleibt nur noch das mehrmalige Exportieren und Importieren der Objekte in ein anderes 3D Modellierungsprogramm das diese kennt, oder es gibt einen entsprechenden Konverter um das Format umzuwandeln. Eine Verkettung verschiedener Programme mag aber recht umständlich auf Dauer sein und kann auch zu Datenverlust führen!
Von den oben genannten Formaten empfehle ich die Verwendung von .obj für statische und .md2 für animierte 3D Objekte.

Ein Modellieren von 3D Objekten wäre theoretisch zwar auch in einer GameEngine möglich, allerdings müssten dazu erst die notwendigen Tools entwickelt werden. Der Editor „UnrealEd“ [Unr05] bietet zwar grundlegende Optionen wie das Ausschneiden oder Hinzufügen von Objekten aus bzw. zu anderen Objekten an, aber diese Methode ist nur bedingt verwendbar, und unvergleichlich zum Komfort und Arbeitsvorgang ausgereifter 3D Modellierungsprogramme. Die erste Wahl bleibt das 3D Modellierungsprogramm.
Wenn Sie knapp bei Kasse sind, lege ich Ihnen das mächtige Open Source Modellierungsprogramm „blender“ ans Herz. Dieses wird zwar oft als schwierig angesehen, ist es aber nicht. Für einen schnellen Einstieg lege ich Ihnen das Tutorial „Your First Animation in 30 plus 30 Minutes“ [Ble08] ans Herz. Dieses lehrt Sie im ersten Teil Schritt für Schritt in die Basisfunktionen der 3D Modellierung ein indem Sie eine 3D Figur von Grund auf erstellen. Diese Figur wird im anschließenden zweiten Teil des Tutorials sogar animiert!
Andere bekannte 3D Programme sind Wings3D, Cinema4D, 3D Studio Max, Maya und Lightwave.

6 Mousepicking



Wir haben bis jetzt gelernt, wie man einen virtuellen Raum erstellt, wie man sich in diesem orientiert, Eingabemöglichkeiten implementiert, ihn strukturiert und Objekte einfügt. Trotzdem fehlt uns noch ein essentielles Feature um interaktiv mit der Szene agieren zu können. In diesem Abschnitt lernen Sie, wie Objekte mit der Maus angewählt werden können und der Name eines Objekts ausgegeben werden kann.

Die Objekte die wir im vorigen Kapitel hinzugefügt haben wurden alle am Nullpunkt des Koordinatensystems erzeugt und überlappen sich deswegen teilweise bzw. verschwinden manche Formen zur Gänze in oder hinter einer größeren Form.
Um diese Misere zu lösen beschließen wir, dass alle Objekte später mit der Maus angeklickt und verschoben werden können. Das Verschieben von Objekten ist allerdings erst Gegenstand der künftigen Kapitel. In diesem Kapitel behandeln wir erstmals wie Objekte im virtuellen Raum angewählt werden können.

Die Funktion, um die es sich handelt heißt „Mousepicking“. Wenn wir mit der Maus auf dem Bildschirm herumklicken werden diese lediglich auf der zweidimensionalen Oberfläche des Bildschirms abgegriffen. Wir arbeiten in einem GL-Bereich jedoch mit einem dreidimensionalen

Koordinatensystem. Diese 2D Koordinaten vom Bildschirm bzw. der Fensteroberfläche der Applikation müssen daher im virtuellen Raum auf die Z-Achse erweitert werden. Um eine Erweiterung auf den 3D Bereich zu ermöglichen, wird bei jedem Mausklick vom Zentrum der Kameraposition ausgehend (befindet sich schließlich in der Mitte des GL-Bereichs) ein Strahl in die Richtung des Mauszeigers geschossen.

Betrachten wir aber zunächst den Einstiegscode:
Wir haben in einem der vorigen Kapitel kurz erwähnt, dass wir die linke Maustaste dazu verwenden wollen Objekte anzuklicken. Wir schauen also in unserem Skelett des Inputhandlers nach und überlegen wo wir die Funktion zum Abschießen der Strahlen einbauen.
Wie Sie vielleicht aus manchen Strategiespielen kennen z.B. „Age of Empires“ [Ens97], werden Objekte nicht markiert, wenn die Maustaste hinuntergedrückt ist, sondern erst nachdem die Maustaste losgelassen wurde! Dadurch kann die Maus bei gedrückter Maustaste immer noch verschoben werden um beispielsweise ein anderes Objekt der Wahl zu markieren. Dieses Prinzip wollen wir ebenfalls in unserem Editor verwenden.

```
if(iae.getTriggerIndex()==0)   // Linke Maustaste losgelassen
{
      mousePicking();
```

Wir bauen eine zunächst leere Funktion mit dem Namen mousePicking() ein: in dieser Funktion wird ein solcher Mausstrahl abgeschossen. Alle Objekte, die von diesem Strahl quasi durchbohrt werden, werden in einem Array gespeichert: den sogenannten PickResults.
In der simpleSetup() Methode müssen wir zuvor allerdings den Behälter für die künftigen Ergebnisse der Mousepicks initialisieren, sonst erhalten wir bei jedem Mausklick eine NullPointerException.

```
pr = new TrianglePickResults();
```

Schauen wir uns jetzt die Funktion mousePicking() genauer an. Zuerst greifen wir die Position der Maus auf der 2D Oberfläche des OpenGL bereichs ab:

```
/**Funktion erzeugt Rays fuer Mausklicks im glCanvas */
public void mousePicking()
{
      mx = glCanvas.getMousePosition().x;
      my = glCanvas.getHeight()-glCanvas.getMousePosition().y;

      screenPos = new Vector2f(mx,my);
```

Was uns hier auffällt ist, dass die Position von Y offensichtlich invertiert dargestellt wird. Das hat einen ganz bestimmten Grund:
Der Ursprung des "normalen" OpenGL Fensters (ohne Swing) ist im glCanvas links unten. Swing allerdings zeichnet grafische Elemente (wie z.B. das GUI) immer beginnend von links oben [Top99]. Der Ursprung (0,0) in Swing befindet sich also links oben, im glCanvas von jME jedoch links unten. Ein Klicken im oberen Bereich des Fensters würde damit einen Strich im unteren Bereich des Fensters zeichnen! Da wir hier Swing als Schicht über dem OpenGL

Fenster verwenden, muss die Y-Position von Mausklicks entsprechend umgeändert werden, damit ein Mausklick korrekt übernommen werden kann!

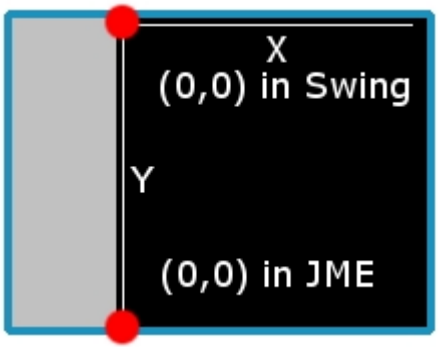


Abb. 4-8: Links oben der Ursprung in Swing, links unten der Ursprung in jME

Dazu ziehen wir die Mausposition in der Y-Achse von der Gesamthöhe des glCanvas ab und erhalten somit die korrekte Position. Arbeiten Sie allerdings mit einem reinen OpenGL Fenster wie in den Anfängertutorials, benötigen Sie diesen Umweg nicht!

Wir erzeugen anschließend einen Strahl (Ray), ausgehend vom Mittelpunkt der Camera in Richtung der Mausposition. Jedes Objekt, das von diesem Strahl getroffen wird, wird in einem Array gespeichert (den PickResults).
Hinweis: Mit der Zeile "subtractLocal()" wird die Genauigkeit der Mausklicks determiniert. Verzichtet man auf diese Zeile passieren Fehler wie beispielsweise dass ein Objekt angeklickt werden kann, obwohl man 5 oder mehr Pixel daneben klickt!

```
mouseRay = new Ray(startpoint,endpoint.subtractLocal(startpoint));

// Hilfslinien um MouseRays zu visualisieren
ColorRGBA[] col = new ColorRGBA[2];
col[0] = ColorRGBA.orange;
Line line = new Line("line",new Vector3f[]{
      startpoint,endpoint.subtractLocal(startpoint)
      },null,col,null);
line.setLightCombineMode(LightCombineMode.Off);
root.attachChild(line);
```

Des Lerneffekts wegen, visualisieren wir für dieses Kapitel auch die Darstellung der Strahlen. Wir erstellen für jeden Mausklick, der gemacht wird eine sichtbare orangefarbene Linie im dreidimensionalen Raum. Sie müssen nach dem Herumklicken allerdings die Kamera ein wenig zur Seite bewegen, um diese Strahlen zu sehen. Dieser Codeabschnitt zur Erstellung der Strahlen wird in den nachfolgenden Kapiteln wieder auskommentiert.

Für jeden neuen Mausklick setzen wir anschließend fest, dass das Ergebnis der vorigen PickResults gesäubert wird (pr.clear()). Dann setzen wir für die PickResults einen Distanz-Check. Wenn wir diesen einsetzen, befindet sich das erste getroffene Objekt immer an der ersten Position bzw. Stelle 0 der erhaltenen PickResults!

```
pr.clear();
```

```
// dadurch ist das erste getroffene Objekt immer an der Stelle 0
pr.setCheckDistance(true);
root.findPick(mouseRay,pr);
```

Zum Schluss setzen wir fest, dass die Rays alle Geometrie des Scenegraphs (vom rootNode) ausgehend in seinen Ergebnissen einbeziehen darf. Theoretisch könnte man das im jetzigen Stadium auch einschränken, indem wir nur die Ergebnisse für unseren sceneNode einbeziehen. Später brauchen wir jedoch auch PickResults aus anderen Zweigen des Scenegraphs.

Nun brauchen wir nur noch den Array der PickResults zu durchlaufen und schauen ob unser gewünschtes Objekt vorhanden ist. Das Objekt, das wir angeklickt haben wollen soll immer das sein, das der Kameraposition am nächsten ist bzw. zuerst vom Strahl getroffen wurde. Da wir den Distanz-Check eingestellt haben, kann man es sich sparen, eine Schleife zu erstellen, weil das von uns gewünschte Objekt befindet sich immer an der ersten Stelle (pr.getPickData(0)).

```
if(iae.getTriggerIndex()==0)  // Linke Maustaste losgelassen
{
	mousePicking();

	// Wir nehmen das Objekt, das als Erstes angeklickt
	// bzw. zuerst vom MouseRay getroffen wurde = pr.getPickData(0)
	if(pr.getNumber()>0)
	{
		spatial = pr.getPickData(0).getTargetMesh();

		// Objekte werden anhand ihres Namens identifiziert
		if(!spatial.getName().equals(null))
		{
			spatialCoords = spatial.getLocalTranslation();
			System.out.println("Spatial: "+spatial);
				}
```

Jetzt können wir auch bestimmen, dass wir jedes Mal einen Output in der Konsole als Feedback bekommen, wenn ein Objekt angeklickt wurde. Wir vereinbaren, dass jedes Objekt mit seinem Namen ausgegeben werden soll (im Übrigen ist das auch der Grund, warum wir den Grundformen und importierten Objekten im Kapitel zuvor einen Namen gegeben haben).
Wir erhalten folgende Ausgaben nun in der Konsole, je nach Objekt das wir hinzufügen und anklicken (OBJ ist ein 3D Modell im .obj Format, das wir importiert haben, hier: house.obj):

```
Spatial: OBJ (com.jme.scene.TriMesh)
Spatial: BOX (com.jme.scene.shape.Box)
Spatial: SPHERE (com.jme.scene.shape.Sphere)
```

6.1 Verwendung von Mauspicks in Spielen

Mousepicking ist essentiell für Computerspiele. Gerade bei Strategiespielen, die sehr mauslastig in der Bedienung sind, kommen diese oft zum Zug. Dort wird oftmals aus einer Vogelperspektive zuerst auf die zu steuernde Einheit geklickt und anschließend auf eine weitere Position in der dargestellten virtuellen Welt. Die Spielfigur oder mehrere Spielfiguren bewegen sich daraufhin von ihrem Startpunkt bis hin zum ausgewählten Endpunkt.

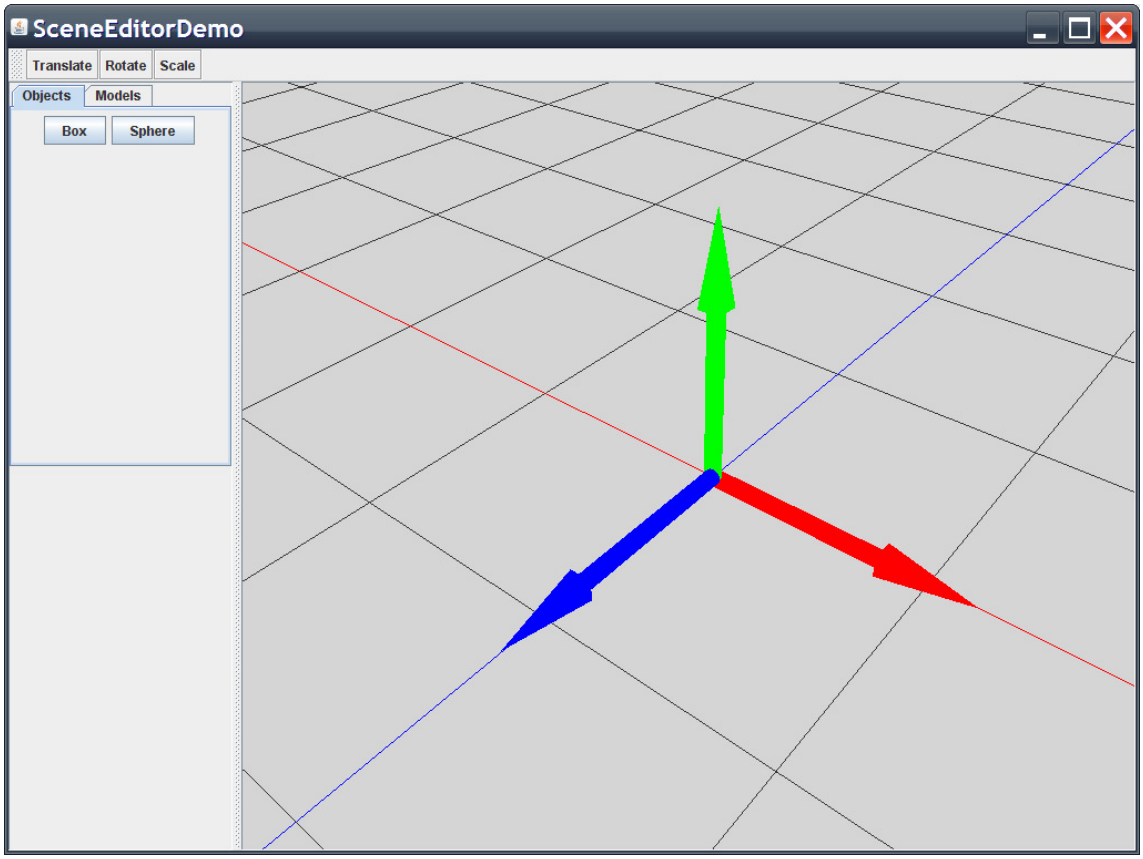
Zum Zuge kommt dieses System auch in Massive Multiplayer Online Role Playing Games (MMORPG) z.B. „Silkroad Online“ [Joy05]. Dabei kann ein Charakter entweder mit Maus zum Drehen der Kamera und Tastatur zum Bewegen der Einheut gesteuert werden oder einzig und allein mit der Maus.

In Shootern finden Mauspicks ebenfalls Verwendung. Dabei wird durch einen Mausklick meist ein Schuss aus einer Schnellfeuerwaffe getätigt, bei welcher die Kugelrichtung genau diesem MouseRay folgt (sofern keine anderen Dinge wie Bullet-Physics implementiert wurden).

Der Spiel „Half-Life 2“ [Val04] ermöglicht mithilfe von Mousepicks sogar ein strategisches Positionieren von Teammitgliedern: dabei wird mit dem vorhanden Fadenkreuz der Waffe aus der Ich-Perspektive eine Position in der Landschaft (z.B. Gebäude oder Baum) anvisiert und beim Drücken einer bestimmten Taste (Hinweis: diese Taste muss in der Tastaturbelegung des Spiels allerdings erst aktiviert werden) ein solcher (Maus)Strahl abgeschossen. Die vom Computer gesteuerten Teammitglieder versuchen daraufhin einen Weg zu diesem Ort zu finden um sich dort zu positionieren.

Beim Einsatz in Strategiespielen wäre eine Erweiterung von Mauspicks zu empfehlen, im Sinne davon dass es möglich ist mehrere Objekte zu selektieren. Dies wird am besten mit Hilfe eines Auswahlrechtecks getätigt. Die Implementierung eines solchen Systems ist allerdings nicht mehr Gegenstand des Kurses.

7 Konzept und Implementierung eines Gizmo



Mit denen im vorigen Kapitel behandelten Mauspicks ist es nun soweit Sie in das Konzept des geplanten Hilfstools einzuführen. In diesem Kapitel lernen Sie, wie ein solches Hilfstool konzipiert und erstellt wird.

Das Hilfstool soll es später ermöglichen Objekte im virtuellen Raum in Echtzeit zu verschieben. Derartige Hilfswerkzeuge werden manchmal auch als „Gizmo“ bezeichnet. Vielleicht kennen Sie so ein Gizmo bereits von früher her, beispielsweise aus 3D Modellierungsprogrammen. Ein solches Gizmo gehört zur notwendigen Standardausstattung eines Editors im virtuellen Raum insbesondere von 3D Modellierungsprogrammen. Ein Gizmo besteht meist aus:

- Farbigen Pfeilen zum Verschieben oder Skalieren eines Objekts
- Farbigen Ringen zum Rotieren eines Objekts

Dabei besitzt jeder dieser Pfeile oder Ringe die Farbe der entsprechenden Achse, auf bzw. um die es bewegt wird.

Zu beachten ist, dass nicht jeder GameEditor solch ein Gizmo beinhaltet. Der in vormaligen Kapiteln genannte und an sich recht umfangreiche Editor UnrealEd in der Version UnrealEngine2 besitzt kein solches Werkzeug: stattdessen werden Objekte anhand von 3

orthogonalen Perspektiven auf die jeweils zugehörige Achse mit Hilfe einer Mausbewegung und gleichzeitigem Drücken einer Maustaste sowie einer gedrückten Tastaturtaste gesteuert.

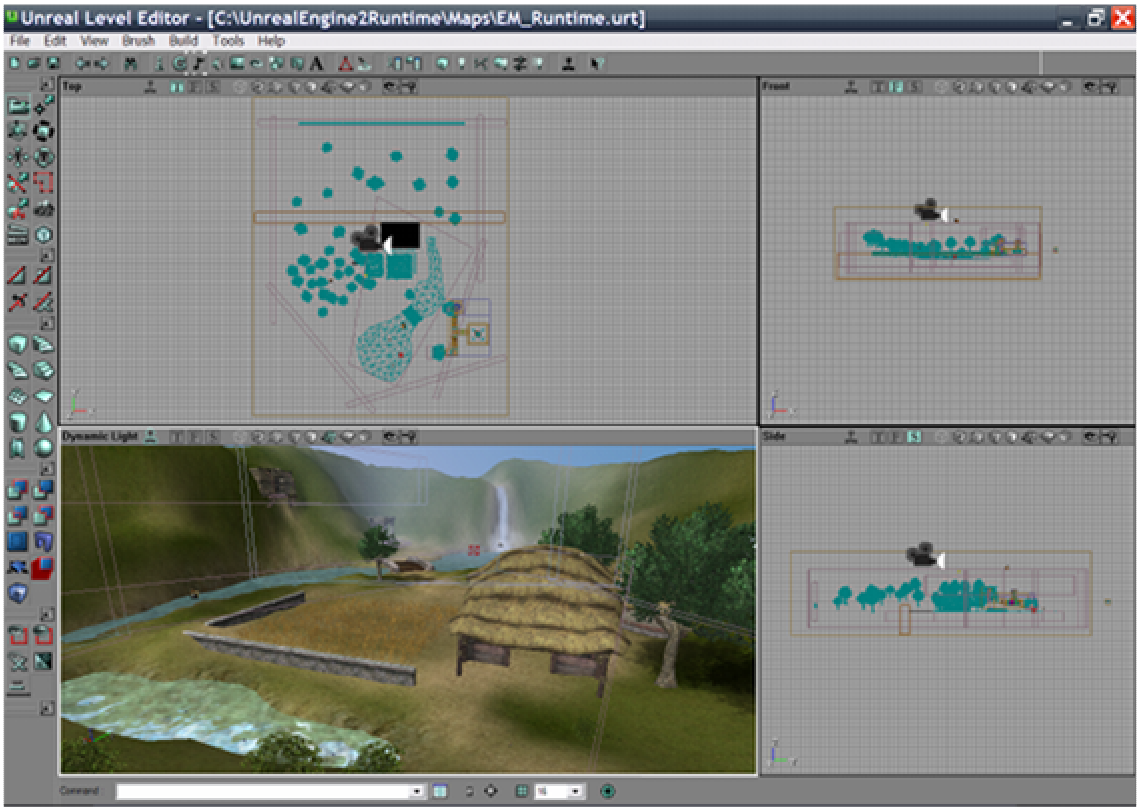


Abb. 4-9: Standardansicht von UnrealEd [Unr05] mit 4 unterschiedlichen Perspektiven

Da ich bereits Spielerkarten in diesem Editor erstellt habe und deswegen auf meine persönliche Erfahrung mit dem Umgang in diesem System zurückblicken kann, erlaube ich mir zu behaupten, dass das System relativ umständlich war; ich bin oft genug in Situationen gekommen, in welchen ich mir gewünscht hätte ein aktives Verschieben von Objekten im perspektivischen Gesichtsfeld mithilfe eines Gizmo zu bewerkstelligen.
In diesem Kurs zeige ich, dass wir als Anfänger dennoch in der Lage sind so ein Gizmo-System zu implementieren. Damit wir nicht gezwungen werden krampfhaft mehrere Tasten festzuhalten, verwenden wir ein Steuerungssystem, das wir bereits in einem der vorigen Kapitel angeschnitten haben:

- linke Maustaste zum Selektieren von Objekten
- rechte Maustaste zum Verschieben, Rotieren und Skalieren von Objekten

Das Verschieben, Rotieren und Skalieren von Objekten zählt zu den 3 Grundformen der Manipulation von jedem dreidimensionalen Objekt. Wenn auch diese nicht im Ausmaß wie in einem 3D Modellierungsprogramm implementiert werden sollen, können wir uns dennoch die Technik der jeweils verwendeten Gizmos zunutze machen. In diesem Abschnitt befassen wir uns allerdings nur mit der Erstellung der 3 Hilfswerkzeuge.

Zuerst erweitern wir unser Swing GUI mit einer Toolbar und 3 neuen Knöpfen mit den Namen Translate, Rotate, Scale (siehe Sourcecode). Per Klick soll nun ein Gizmo mit Pfeilen oder ein Gizmo mit Ringen der Szene hinzugefügt werden. Wenn dabei schon ein bestimmtes Gizmo sichtbar ist, soll dieses ausgeblendet werden und nur das gewünschte Gizmo dargestellt werden. Wir fügen den 3 neuen Buttons jeweils einen ActionListener hinzu und definieren das Verhalten wie folgt:

```
class GizmoListener implements ActionListener
{
    public void actionPerformed(ActionEvent ae)
    {
      if(ae.getActionCommand()=="Translate")
      {
            gizmo = 1;
            helperNode.detachChild(scaleNode);
            helperNode.detachChild(rotationNode);
            helperNode.attachChild(translationNode);
      }
```

Im obigen Ausschnitt des Codes kommt dabei unser Hilfsknoten (helperNode) wieder ins Spiel. An diesem wird nämlich immer das jeweils gewünschte Gizmo angehängt und sichtbar gemacht während die anderen zwei Gizmos entfernt werden. Wir weisen außerdem jedem Gizmotyp eine Variable zur Kennzeichnung des Modus hinzu:

1... Translationsmodus

2... Rotationsmodus

3... Skalierungsmodus

Diese drei Variablen benötigen wir aber erst im nächsten Kapitel.

Wie wir bereits vorweggenommen haben, erstellen wir für jedes Gizmo einen eigenen Node. Da wir bereits eine bildliche Vorstellung eines Gizmo haben, fangen wir an diesen zu implementieren. Betrachten wir den Code dazu:

```
public void buildTranslator()
{
      translationNode = new Node("translator");

      xArrow = new Arrow("xArrow",8.0f,0.5f);
      xArrow.setModelBound(new BoundingBox());
      xArrow.updateModelBound();
      xArrow.setSolidColor(ColorRGBA.red);
      Quaternion quatX = new Quaternion();

      quatX.fromAngleAxis((-1)*(90)*FastMath.DEG_TO_RAD,new
Vector3f(0,0,1));
      xArrow.setLocalRotation(quatX);
      xArrow.setLocalTranslation(new Vector3f(4,0,0));
      xArrow.updateGeometricState(0,false);

      translationNode.attachChild(xArrow);
```

Wir benötigen für jedes Gizmo einen eigenen Knoten. Dieser Knoten soll im oben angegebenen Falle einen roten, einen blauen und einen grünen Pfeil beinhalten.

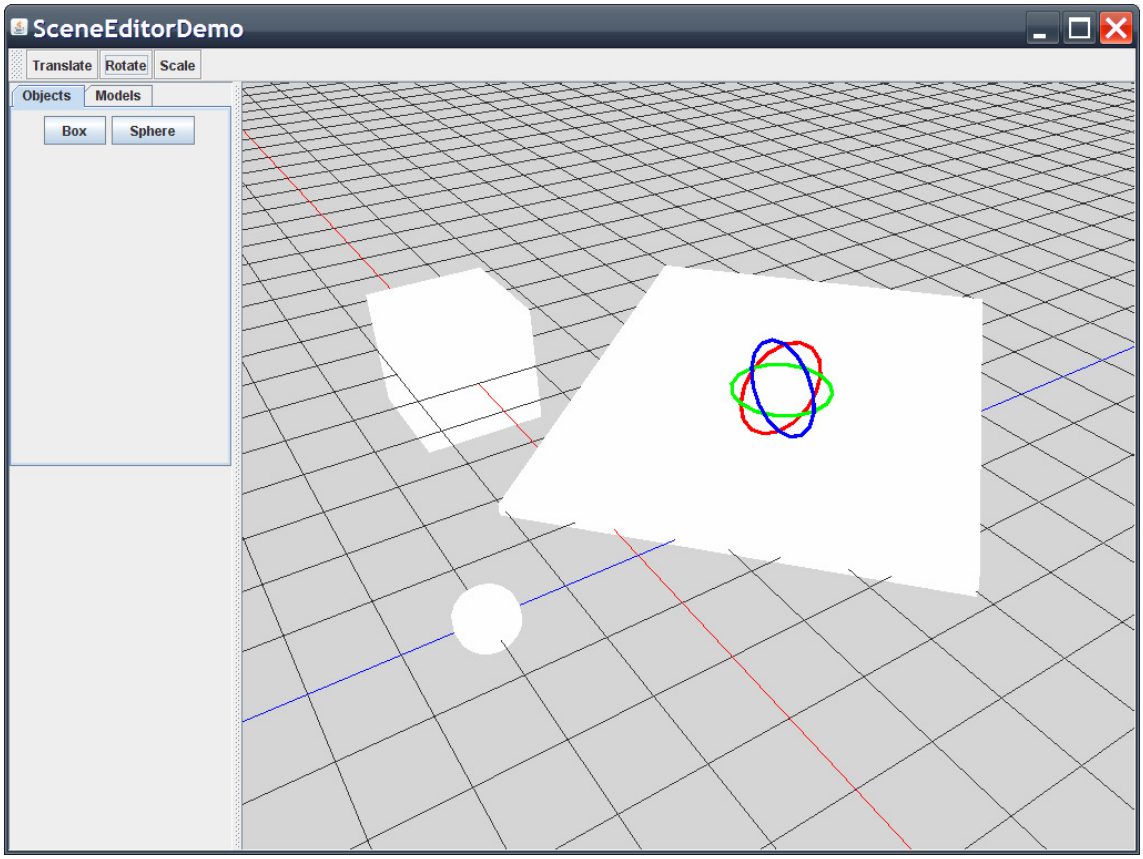
Wir vereinbaren, dass die Pfeilspitzen dieser 3 Pfeile, die gleichsam die entsprechenden Achsenrichtungen darstellen, jeweils in die positive X/Y/Z Richtungen schauen! Damit haben wir auch eine Vorstellung davon, wo der positive und negative Bereich des Koordinatensystems beginnt. Damit die Pfeile auch in die richtige Achsenrichtung zeigen werden sie mithilfe einer Quaternion in die entsprechende Richtung rotiert. Danach wird deren Position so festgesetzt, dass sich der Beginn des Pfeils genau am Nullpunkt des Koordinatensystems befindet.
Auf diese Weise erzeugen wir 3 Farbpfeile, die zusammen im translationNode gruppiert werden. Dasselbe Prinzip wenden wir an, um 3 Ringe für den rotationNode zu erstellen (siehe Sourcecode).

Die 3 Gizmos werden schon zu Beginn der Applikation in der simpleSetup() Methode initialisiert. Das erspart uns später ein mögliches verzögertes Anzeigen des Gizmos (obwohl dies höchstens bei sehr langsamen Computern auftreten würde).
Interessant für uns ist jetzt noch die Renderreihenfolge der Gizmos. Die Gizmos sollen – sofern sie erstmal angezeigt werden – immer und zur Gänze sichtbar sein! Dazu stellen wir ihre Rendereigenschaft und ZBuffer ein wie folgt:

```
//Gizmos werden schon zu Beginn erzeugt, jedoch erst bei Klick
//auf die entsprechenden Buttons an den Knoten "helperNode" angehaengt
buildTranslator();
buildRotator();
buildScalor();

...

ZBufferState zbuffer = renderer.createZBufferState();
zbuffer.setEnabled(true);

// ersetzt immer jeden Pixel!
zbuffer.setFunction(ZBufferState.TestFunction.Always);

translationNode.setRenderState(zbuffer);
rotationNode.setRenderState(zbuffer);
scaleNode.setRenderState(zbuffer);
translationNode.setRenderQueueMode(Renderer.QUEUE_OPAQUE);// sichtbar
rotationNode.setRenderQueueMode(Renderer.QUEUE_OPAQUE);
scaleNode.setRenderQueueMode(Renderer.QUEUE_OPAQUE);
```

Nun sind wir in der Lage Gizmos auf Knopfdruck in die Szene zu projizieren, können die Art des gewünschten Gizmo nach Belieben wechseln und haben auch die Renderreihenfolge unter Kontrolle. Damit Objekte mit diesen Werkzeugen manipuliert werden können, lesen Sie bitte im nächsten Kapitel weiter.

8 Manipulation von Objekten mithilfe des Gizmo



Dieses Kapitel ist relativ komplex und es ist unbedingt notwendig, dass Sie die vorhergehenden Kapitel, insbesondere die Kapitel mit dem Inputhandler der Maus sowie dem Mousepicking verstanden haben! In diesem Abschnitt lernen Sie, wie Sie Gizmos an Objekte anhängen können und anschließend das Objekt manipulieren können.

Um Objekte mithilfe der Gizmos zu manipulieren müssen wir sie zuerst auf irgendeine Art und Weise mit dem Gizmo in Verbindung setzen. Dazu konzipieren wir folgende Schritte zur Realisierung:

1. Objekt in den virtuellen Raum hinzufügen
2. Gizmo in den virtuellen Raum hinzufügen
3. Gizmo an ein beliebiges Objekt anfügen
4. Objektmanipulation mithilfe des Gizmo durchführen

Schritt 1 und 2 haben wir bereits in den vorigen Kapiteln behandelt. Wir konzentrieren uns jetzt auf Schritt 3 und 4.
Nachdem wir ein beliebiges Objekt unserer Wahl (z.B. eine Box) in die Szene hinzugefügt haben drücken wir die „Translate“ Taste. Dabei wird das Translations-Gizmo der Szene

hinzugefügt. Um das Gizmo der Box zuzuordnen klicken wir einfach einmal mit der linken Maustaste auf die Box! Fertig!

Der Code dazu findet sich ab Zeile 993:

```
if(gizmo == 1)
{
	// translationNode an das Objekt anhaengen
	translationNode.setLocalTranslation(spatialCoords);
}
```

Schreiten wir nun über zur Implementierung der Gizmo-Logik:

Da das Gizmo immer aus jeweils 3 Teilen besteht, darf nur jeweils die Aktion ausgeführt werden die für den entsprechenden Teil gilt z.B. roter Pfeile nur zum Verschieben auf der X-Achse, grüner Pfeil nur zum Verschieben auf der Y-Achse. Das können wir mithilfe von Booleans erzwingen. Ab Zeile 614 im Sourcecode findet sich folgender Abschnitt:

```
if(gizmo==1)
{
	// pick Parent
	gizmoSpatial = pr.getPickData(pd).getTargetMesh().getParent();

	if (gizmoSpatial.getName().equals("xArrow") &&
			!translate_z && !translate_y)
	{
		xArrow.setSolidColor(ColorRGBA.yellow);
		translate_x = true;
	}
	if (gizmoSpatial.getName().equals("zArrow") &&
			!translate_x && !translate_y)
	{
		zArrow.setSolidColor(ColorRGBA.yellow);
		translate_z = true;
	}
	if (gizmoSpatial.getName().equals("yArrow") &&
			!translate_z && !translate_x)
	{
		yArrow.setSolidColor(ColorRGBA.yellow);
		translate_y = true;
	}
}
```

Nach dem Mousepicking fragen wir ab, welcher Gizmo-Modus gerade aktiv ist.

Achtung: Wenn Gizmo-Modus 1 aktiviert ist (Translation) muss das Mousepicking mit der rechten Maustaste das Parent des entsprechenden Pfeils abgreifen und nicht das angeklickte Objekt selbst! In jME 1.0 war der Pfeil noch ein Objekt für sich allein, seit jME 2 wird der Pfeil jedoch in „tip“ und „base“ unterteilt! – d.h. wenn wir nur gizmoSpatial = pr.getPickData(pd).getTargetMesh() hinschreiben würden, würde die Funktion entweder nur die „tip“ oder die „base“ des Pfeils auswählen und beim weiteren Verschieben des Objekts würde sich der Pfeil spalten.

Wir haben in der Methode zur Erstellung des Pfeils den Namen „xArrow“ verwendet. Dieser umfasst bereits beide Komponenten des Pfeils. Wenn wir jetzt also den Pfeil mit der rechten Maustaste anklicken, wird der gesamte Pfeil einbezogen.

Für die Rotationsringe gilt diese Ausnahmeregelung nicht, da der Ring tatsächlich nur aus 1 Objekt besteht. Dort reicht es, nur das TargetMesh abzufragen.

Innerhalb des Translationsmodus gibt es weiters 3 Möglichkeiten. Entweder es wird der X, der Y, oder der Z-Pfeil gedrückt. Wir erstellen zu jeder dieser Mögichkeiten 3 passende Wahrheitswerte mit folgenden Namen:
translate_x
translate_y
translate_z

Diese Variablen werden standardmäßig auf „false“ gesetzt. Sobald einer der 3 Pfeile gedrückt wird, wird die entsprechende Boolean auf „true“ gesetzt. Es darf aber immer nur 1 Boolean auf „true“ sein!
Beim schnellen Bewegen der Maus kann es passieren, dass – obwohl nur ein Pfeil ausgewählt wurde - der Mauszeiger über einen zweiten Pfeil fährt und ihn wegen des permanenten Mousepicking selektieren würde. Um dieses Problem zu lösen benötigen wir für jede if-Bedingung einen weiteren Prüfvorgang. Die Funktionsweise dieses zeige ich Ihnen in folgendem Fallbeispiel:

1. Wir klicken mit der rechten Maustaste (und halten diese gedrückt) auf den roten X-Pfeil
2. Die if-Bedingung erkennt, dass ein Pfeil mit dem Namen „xArrow“ angeklickt wurde und setzt die Variable „translate_x“ auf „true“.
3. Sobald die Maus jetzt beispielsweise durch die schnelle Mausbewegung auch den Z-Pfeil berührt, prüft sie ob schon einer der anderen 3 Wahrheitswerte auf true ist:

```
if (gizmoSpatial.getName().equals("zArrow") && !translate_x &&
!translate_y)
```

Falls einer der Wahrheitswerte bereits auf „true“ ist, dann ist der Gesamtwahrheitswert für die anderen if-Bedingungen immer negativ und der if-Zweig von diesen wird niemals ausgeführt. Gleichzeitig verbleibt diejenige if-Bedingung positiv, die zuerst aktiviert wurde.
Sobald einer der Pfeile angeklickt bzw. aktiviert ist geben wir dem entsprechenden Teil des Gizmos als Feedback für den Benützer und als Zeichen, dass dieser Teil aktiv ist, eine gelbe Farbe. Diese gelbe Farbe muss wieder auf die ursprüngliche Farbe des Pfeils zurückgesetzt werden, sobald die rechte Maustaste losgelassen wird (ab Zeile 1015).

```
if(iae.getTriggerIndex()==1)  // Rechte Maustaste losgelassen
{
      // Der gelbe Pfeil oder Ring des Gizmos muss wieder
      // auf seine urspruengliche Farbe zurueckgesetzt werden
      xArrow.setSolidColor(ColorRGBA.red);
      translate_x = false;
      zArrow.setSolidColor(ColorRGBA.blue);
      translate_z = false;
      yArrow.setSolidColor(ColorRGBA.green);
      translate_y = false;
```

Beim Loslassen der rechten Maustaste werden müssen außerdem alle booleschen Werte wieder auf „false“ zurückgesetzt werden. Dies erlaubt das Anklicken eines anderen Teils des Gizmo.

Die 3 booleschen Werte pro Gizmo sind nicht nur dafür zuständig welcher Teil eines Gizmos angeklickt wurde, sondern auch um jeweils eine ganz bestimmte Aktion auszuführen. Diese Aktionen betreffen das Verschieben, Rotieren und Skalieren eines Objekts. Sie sind relativ komplex und werden deswegen in den nachfolgenden 3 Unterkapiteln näher erläutert.

8.1 Translation

Die grundlegende Funktionsweise der Positionsänderung eines Objekts mithilfe eines Gizmos funktioniert folgendermaßen (unter der Voraussetzung dass das Gizmo wie im vorigen Abschnitt beschrieben bereits einem Objekt zugeordnet wurde):

1. Anklicken (und gedrückt halten) eines Pfeils im Gizmo mit rechter Maustaste
2. Bewegen der Maus in die entsprechende Achsenrichtung
3. Aktualisieren des zuvor selektierten Objekts an die Position des Gizmos

Wie wir im Kapitel zuvor bereits besprochen haben soll die rechte Maustaste das Interagieren mit den Gizmos ermöglichen. Im Falle des Translations-Gizmos bedeutet dies, dass einer der drei Richtungspfeile im Knoten mit der rechten Maustaste angeklickt werden kann und durch Bewegen der Maus in seiner Achse nach vor oder nach hinten bewegt werden kann. Dabei soll das zuvor angeklickte Objekt (Box, Sphere) dieser Bewegung folgen.
Kommen wir nun darauf zurück, wieso wir manchen Maustasten einen Repeat beim Werfen eines Events erlauben. Dazu zählt die rechte Maustaste.
Wie man bereits erraten kann benötigen wir zum Verwenden eines Gizmo wieder die Funktion des Mousepicking. Dabei muss das Mousepicking nicht nur einmal, sondern wiederholt ausgeführt werden. Nur so kann eine flüssige Bewegung im Raum entstehen!
In weiser Voraussicht haben wir bereits in den vorhergehenden Kapiteln der rechten Maustaste diesen Repeat erlaubt. Dabei werden, solange die rechte Maustaste gedrückt ist, Mouse-Strahlen geworfen.

Wir implementieren nun die Funktionsweise für den Fall, dass der translationNode angehängt wurde und beispielsweise translate_x auf „true“ ist. Betrachten wir dazu den Codeabschnitt ab Zeile 865:

```
if (translate_x)
{
      float planeY = translationNode.getLocalTranslation().y;
      float startY = mouseRay.origin.y;
      float endY = mouseRay.direction.y;
      float coef = (planeY - startY) / endY;
```

```
      float planeX = mouseRay.origin.x + (coef *
mouseRay.direction.x);
```

Man erstellt zuerst einen Koeffizient aus einer Koordinatenposition des Gizmos im aktuellen Zustand und der Richtung, in welche der aktuelle Mausstrahl in derselben Richtung deutet. Dieser Koeffizient wird anschließend mit der Richtung des Maus-Strahls in X-Richtung multipliziert und an die (im obigen Falle) X-Position des Gizmos hinzugefügt! Diese Vorgehensweise bewirkt, dass sich das Gizmo in passender Relation zur Mausbewegung verhalten kann!

Im dem Moment, wo man jetzt mit der rechten Maustaste den Pfeil erstmals an einer bestimmten Stelle anklickt macht der gesamte translationNode einen Sprung und positioniert sich selber genau an diese Position. Dieser „Hüpfer“ wird dadurch ausgelöst, dass der „Nullpunkt“ des translationNode an diese neue Position gesetzt wird, und damit auch der gesamte translationNode; allerdings ist das nicht das was wir geplant haben, deswegen müssen wir eine Logik überlegen, bei der dieser Sprung nicht gemacht wird.
Wenn man den Pfeil anklickt an einer bestimmten Stelle, soll der translationNode-Nullpunkt nicht an die neue Position springen sondern der Pfeil soll genau dort angeklickt und vom Mauszeiger "festgehalten" werden, wo die Position mit der angeklickten x-Achse übereinstimmt!

```
if(bClicked == false)
{
      distance0 = planeX-translationNode.getLocalTranslation().x;
      bClicked = true;
}
```

Beim ersten (Rechts)Klick soll immer die Position der Maus in Entfernung vom "Nullpunkt" des translationNodes gespeichert werden (distance0); diese ist wichtig weil wir nachher programmieren, dass der Abstand immer gleich sein soll zwischen dem Mauszeiger und dem "Nullpunkt" des TranslationNodes. Das Speichern der Ausgangsabstandes wird nur 1x ausgeführt!

Da sich der Abstand von dem Mauszeiger logischerweise vergrößert/verkleinert durch die Mausbewegungen, müssen wir den Abstand zum "Nullpunkt" des translationNodes wieder richtig setzen, bzw. den translationNode "nachziehen" - und zwar solange nachziehen oder "auf Abstand halten" bis der Abstand wieder komplett gleich ist wie der Wert "distance0" (der Wert, den wir am Anfang abgespeichert haben um festzulegen an welcher Stelle der Mauszeiger den Pfeil "festhält").

```
translationNode.getLocalTranslation().x = planeX-distance0;
translationNode.setLocalTranslation(
      translationNode.getLocalTranslation().x,planeY,planeZ);
```

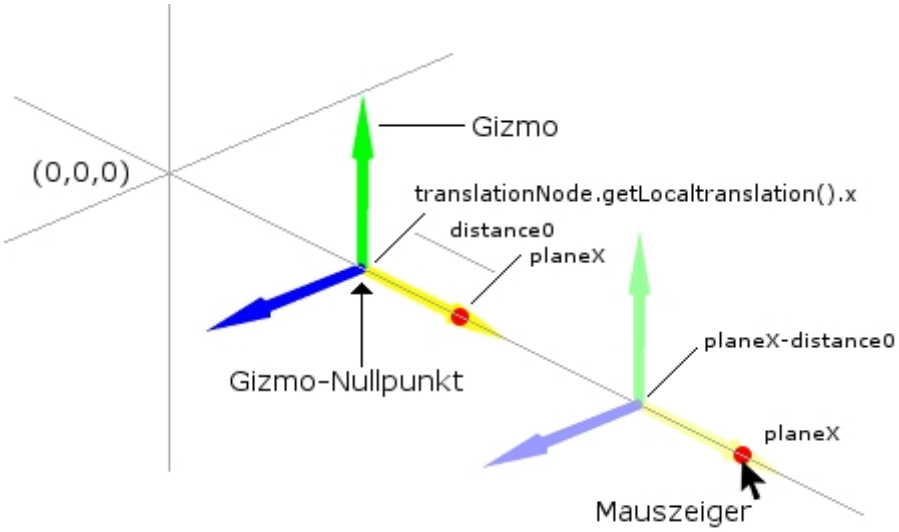


Abb. 4-10: beide planeX-Punkte sind Positionen an denen die Mouserays die X-Ebene schneiden. planeX kann einerseits nur als Punkt, oder auch als Strecke gesehen werden (vom Nullpunkt des Koordinatensystems bis zum entsprechenden Wert auf der X-Achse). Beim Bewegen der Maus wird immer der einmalig gespeicherte Abstand distance0 von planeX abgezogen um den Gizmo-Nullpunkt an diese Stelle festzusetzen. Dadurch befindet sich die Maus immer im selben Abstand zum Gizmo-Nullpunkt!

Die ursprünglichen Y und Z Positionen des Objekts speichern wir während diesem Vorgang in den Variablen planeY, planeZ. Diese sollen ja gleich bleiben wenn ein Objekt auf der X-Achse verschoben wird. Anschließend setzen wir das Spatial (die Box) an die Position des Gizmos.

```
// setze das Spatial an die Position wo die Pfeile sind!
spatial.setLocalTranslation(translationNode.getLocalTranslation());
```

Dies ist ein laufender Vorgang, der in jedem Frame ausgeführt wird. Wir erhalten eine flüssige Bewegung beim Verschieben des Objekts in irgendeine Richtung!
Die Implementierung für die anderen 2 Pfeile funktioniert analog.

8.2 Rotation

Die Rotation eines Objekts funktioniert ein wenig anders. Hier braucht man die Umsetzung nicht von einem Koeffizienten abhängig zu machen. Stattdessen machen wir die Rotation eines Objekts abhängig von der Anzahl Pixel, die auf der 2D Oberfläche des OpenGL-Fensters getätigt wird. Sobald sich die Maus nach dem Anklicken eines der Rotationsringe in eine Richtung bewegt (entweder X oder Y Achse) wird das Objekt dabei um einen bestimmten Winkel rotiert. 1 Pixel Entfernung vom Anklickpunkt soll dabei der Änderung um 1 Grad im Ring von 0°-360° entsprechen!

```
if (rotate_z)
{
      if(bClicked == false)
      {
            // Speichern des ersten y-Werts in "y0"
            y0 = my;
            bClicked = true;

            // hole die aktuelle Rotation des Objekts
            quatDefault = spatial.getLocalRotation();
      }
```

Für eine Rotation um die Z-Achse lege ich fest, dass diese Bewegung anhand der Y-Bewegung der Maus festgelegt wird. Wie im vorigen Abschnitt müssen wir zuerst einen Anfangswert speichern, von dem wir die weitere Berechnung aus durchführen. Wir speichern die Koordinaten der aktuellen Maus-Y-Position (my) auf der 2D Oberfläche in „y0“ sobald die rechte Maustaste gedrückt wird. Gleichzeitig holen wir uns die aktuelle Drehung des angewählten Objekts! Diese brauchen wir später noch.

Wenn sich die Maus nun bewegt, entfernt sie sich von dem gespeicherten Punkt. Da wir nur im 2D Koordinatensystem arbeiten, können wir die Distanz zwischen diesen Punkten zeitgleich aktualisieren, indem wir die Differenz zwischen den beiden Werten berechnen.

```
// Berechnen des Abstandes zwischen y0 und aktueller Mausposition (my)
yDistance = (y0-my);

// Bei einem Abstand von mehr als 360 Pixel (Grad) wieder bei 0
beginnen
if(yDistance>360)
      yDistance = yDistance-360;
if(yDistance<0)
      yDistance = yDistance+360;
```

Sollte der Abstand mehr als 360 Pixel ergeben bzw. in diesem Falle mehr als 360°, wurde bereits eine volle Drehung getätigt und wir lassen die Rotation wieder von 0° beginnen. Analog behandeln wir den umgekehrten Fall.

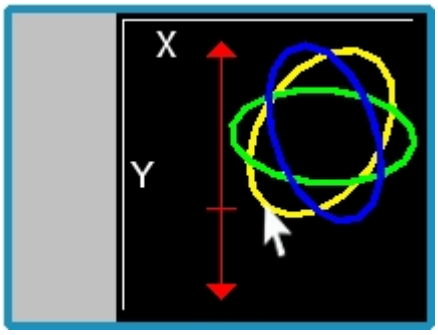


Abb. 4-11: Mit der rechten Maustaste ist gerade der Ring an einer bestimmten Position angeklickt. Dieser Ausgangspunkt wird in der 2-dimensionalen Ebene in y0 gespeichert. Von diesem Punkt ausgehend wird die Distanz des Mauszeigers auf der Y-Achse gemessen, sobald sich die Maus bewegt.

Damit das Objekt (Box) entsprechend gedreht wird, benötigen wir noch die Hilfe einer Quaternion. Eine Quaternion ist ein recht komplexes Objekt (das zu erklären hier zu lange dauern würde), mit dem wir Rotationen in alle Dimensionen tätigen können. Wir haben sie bereits beim Erstellen der Gizmos kurz kennengelernt um die Pfeile oder Ringe in die richtige Achsenlage zu kippen.

```
quatTemp = new Quaternion();

quatTemp.fromAngleAxis(yDistance*FastMath.DEG_TO_RAD,Vector3f.UNIT_Z);
```

In obigem Falle erstellen wir für jeden Grad, der geändert wird eine neue Quaternion aus der Achse (hier die Z-Achse) und dem Winkel (Abstand von aktueller my zu y0 in Pixeln).

Um nun eine Drehung zu bewerkstelligen, die von der ursprünglichen Drehung des Objekts (der Box) ausgeht, muss man die gegebene Quaternion der Box mit der neuen Quaternion, die wir berechnet haben, multiplizieren!

```
quatNew = quatDefault.mult(quatTemp);

if(yDistance!=0)
      spatial.setLocalRotation(quatNew);
```

Danach setzen wir für die Box die neue Quaternion als Rotation fest.

Hinweis: sollten Sie darauf verzichten die vorige Drehung eines Objekts (quatDefault) zu berücksichtigen, dreht sich ein Objekt beim Anklicken sofort in eine Nullrotation (0° um X/Y/Z) zurück!

8.3 Skalierung

Obwohl man glauben könnte die Skalierung eines Objekts funktioniert ähnlich wie die Rotation, liegt die technische Umsetzung wieder näher an der Translation. Grund dafür ist folgender:

Die Drehung eines Objekts findet in einem begrenzten Ausmaße von ~360 Pixel auf der 2D Oberfläche statt. Diese Abstände in Pixel können wir aber nicht in einem anständigen Verhältnis auf die 3D Welt übertragen. Bei einem Skalierungswert von 1 Pixel (2D) auf 1 Einheit (3D) wäre die Skalierung zu schwach und langsam. Wir müssten daher mehrmals die rechte Maustaste neu am Gizmo einsetzen um Objekte größer zu skalieren – verwenden wir hingegen einen Faktor von 1 Pixel (2D) auf 10 Einheiten (3D), wäre die Aktualisierung sprunghaft und ungenau!

Es bleibt nichts anderes übrig als wieder das System mit Koeffizienten wie bei der Translation zu verwenden (siehe ab Zeile 795). Die ersten Zeilen gleichen dabei denen bei der Translation, deswegen überspringe ich diesen Schritt hier und wir können gleich zum nächsten schreiten.

```
if(bClicked == false)
{
      // distance0 wieviel die Maus vom Nullpunkt des scaleNodes
entfernt
      distance0 = planeX-scaleNode.getLocalTranslation().x;
      defaultScale = spatial.getLocalScale().x;
      bClicked = true;
}
```

Wie schon bei der Translation berechnen wir den Abstand der angeklickten Mausposition am Gizmopfeil vom Nullpunkt des Gizmo. Diesen Wert speichern wir ebenso wie zuvor in „distance0“. Gleichzeitig speichern wir auch den aktuellen Skalierungswert des Objekts (oben in X-Richtung) einmalig ab.

Wichtig ist: dieser erste Abstand wird mit dem aktuellen Skalierungswert assoziiert!

Danach berechnen wir einen zweiten Abstand aus, der laufend aktualisiert wird. Hier ergibt sich auch der Unterschied zur Translation: Das Skalierung-Gizmo bewegt sich nicht vom Platz im Gegensatz zum Translations-Gizmo.

Der Abstand, um wie viel sich die Maus auf der Achse vom Objekt entfernt, errechnet sich aus dem Abstand der Mausstrahlen vom Nullpunkt des Gizmos.

```
// Abstand, wieviel sich die Maus auf der x-Achse vom Objekt entfernt
distance = planeX-scaleNode.getLocalTranslation().x;
```

Dieser Wert ändert sich laufen in Abhängigkeit von den Mausbewegungen.

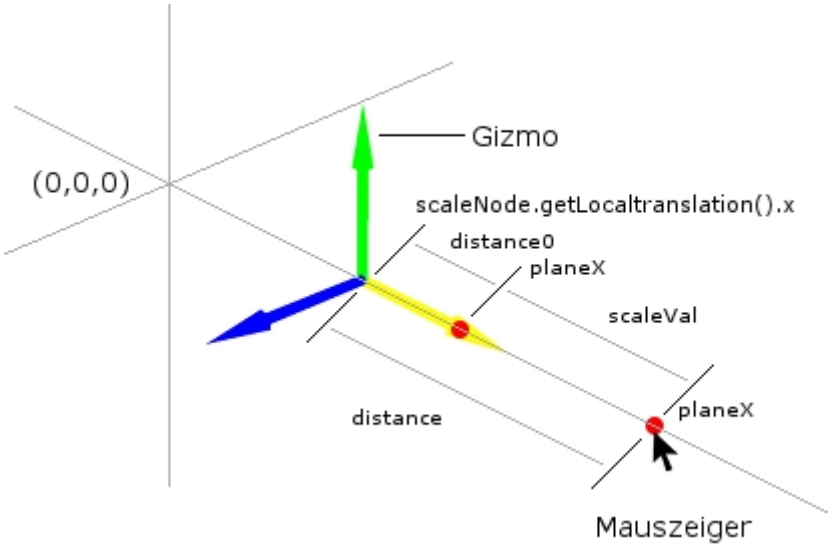


Abb. 4-12: Im Gegensatz zum translationNode wird hier immer der Abstand des Mauszeigers vom Gizmo-Nullpunkt berechnet, die Differenz von planeX-scaleNode.getLocalTranslation().x. Der Skalierungswert ergibt sich durch die Differenz von distance-distance0.

Wir müssen den ersten Abstand (der sich aus dem Abstand von der am Pfeil angeklickten Stelle zum Nullpunkt des Gizmos ergab) vom „großen“ Abstand abziehen um den echten Abstandswert zu ermitteln (distance-distance0).

```
// Berechnen der Distanz zwischen Anklickpunkt am Pfeil und Node-
// "Nullpunkt"; damit nicht zu schnell skaliert wird dividieren wir
// den Wert durch 10
float scaleVal = (distance-distance0)/10;
```

Aus der Differenz ergibt sich der echte Abstand, den wir als Skalierungswert heranziehen. Da dieser Skalierungswert relativ hoch und damit zu schwer handhabbaren Resultaten führt (bei geringer Mausbewegung große Skalierung), dividieren wir das Ergebnis noch durch 10. Anschließend addieren wir diesen Skalierungswert dem Skalierungswert der Box hinzu:

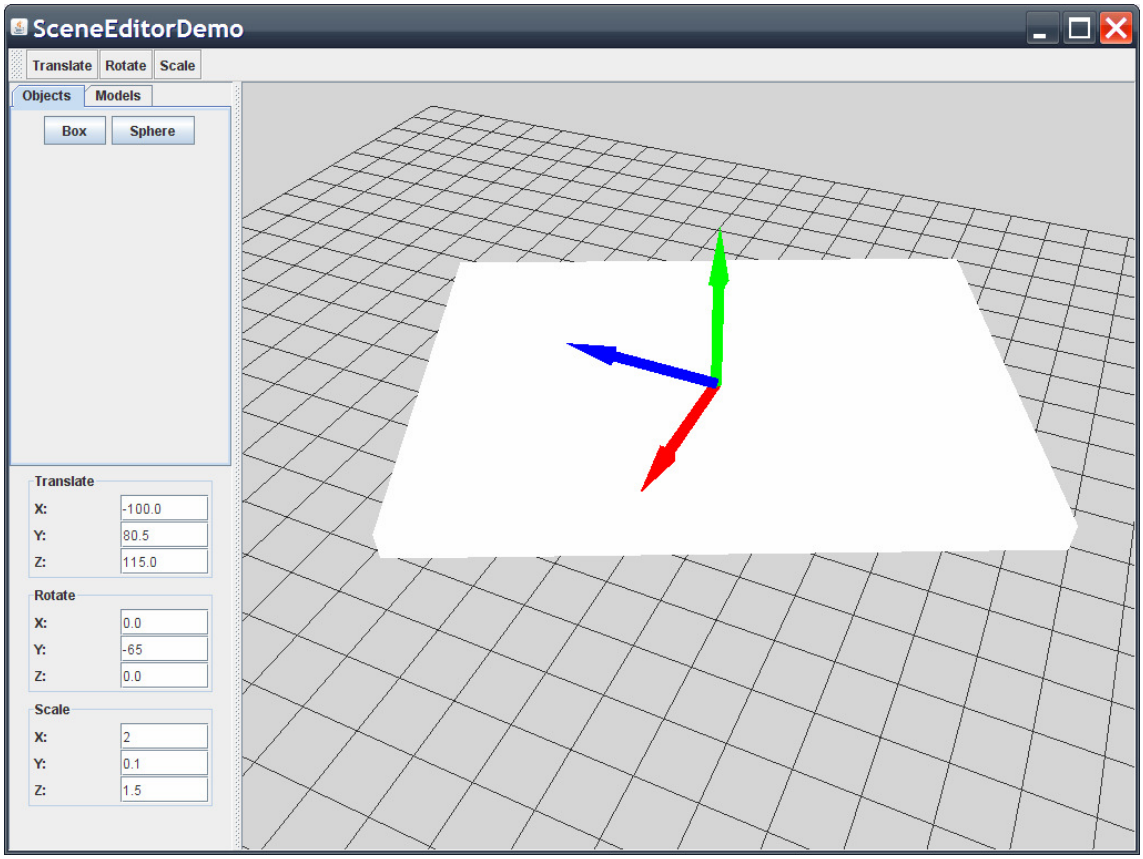
```
spatial.getLocalScale().x = defaultScale+scaleVal;
```

Die Vorgangsweise für die anderen Teile des Gizmos funktioniert analog. Mit Ende dieses Kapitels ist auch eins der schwersten Kapitel des Kurses abgeschlossen. Es gibt sicher andere Methoden zur Implementierung eines Gizmos, aber mit dieser Implementierung sind die Grundlagen für zumindest eine Art der Umsetzung gefestigt. Theoretisch könnte man die Gizmos erweitern um Objekte auf einer 2-dimensionalen Ebene zu verschieben, sprich: auf der XY, YZ, XZ Achse. Das mag aber jeder für sich selber entscheiden.

Wenn Sie bis jetzt alles (halbwegs) verstanden haben, haben Sie einen großen Schritt zur Realisierung von Inhalten in virtuellen Welten geschafft. Durch das Steuern eines Gizmo im virtuellen Raum haben Sie bereits den Anfang gemacht, Spieleinheiten in Echtzeit zu steuern.

In einigen Weltraumspielen (darunter z.B. „Eve Online“ [CCP03]) werden Objekte dadurch gesteuert, indem man ein Raumschiff mit der linken Maustaste anklickt und die man mit einem weiteren Klick im leeren Raum an eine Position bzw. in eine Richtung fliegen lässt. Der einzige Unterschied der im Vergleich zu unserem Editor noch besteht ist folgender:
In derartigen Spielen werden meist nur eine Anfangs- und eine Endposition für die zu steuernde Einheit gewählt. Alle verbleibende Schritte dazwischen bzw. die Bewegung damit das Objekt von Punkt A zu Punkt B läuft/fährt/fliegt müssen noch implementiert bzw. mit einer entsprechenden Geschwindigkeit versehen werden.
Für jME gibt es beispielsweise die Funktion der Interpolation zwischen 2 Vektorpositionen im virtuellen Raum. Nachteil dieser Methode ist allerdings, dass die Geschwindigkeit nicht konstant ist, und je näher das Objekt dem Ziel kommt, diese abnimmt. Diese Methode ist daher nicht empfehlenswert.
Stattdessen bietet sich an, diesen Weg konstant umzusetzen mit der vorhandenen Funktion addLocal(). Die Durchführung derartiger Spielaktionen erfolgt in jME mit Hilfe von Controllern. Controller erlauben ein Pausieren animierter oder bewegter Objekte, sowie ein Fortsetzen an dem Punkt wo angehalten wurde. Diese sind allerdings nicht mehr Gegenstand des Kurses.

9 Daten von Objekten in Echtzeit aktualisieren



Wir haben im vorigen Kapitel gelernt, wie man Objekte im virtuellen Raum in Echtzeit manipuliert und sind bereits in der Lage eine einfache Szene zusammenzustellen. Es fällt jedoch folgendes auf: Das Verschieben, Rotieren, Skalieren von Objekten kann man lediglich „Pi mal Daumen“ abschätzen und durchführen – kurz gesagt: es fehlt die Möglichkeit Objekte geometrisch genau zu positionieren! In diesem Kapitel lernen Sie, wie man die geometrischen Daten eines Objekts abfragen und zielgenau verändern kann!

9.1 Objektdaten abfragen

Wie man ein Objekt selektiert und dessen Namen ausgeben kann haben wir schon im Kapitel zum Mousepicking behandelt. Nun sollte es an sich keine allzu große Schwierigkeit bereiten, von diesem Objekt die Koordinaten, den Rotationswinkel, die Skalierungswerte abzufragen. JME bietet entsprechende Methoden für jede dieser 3 Möglichkeiten an.

Damit wir diese Daten auch sehen können, benötigen wir zuerst Textfelder in unserem GUI. In diese sollen die Daten eines angeklickten Objekts eingetragen werden. Wir erstellen in der Swingoberfläche also für jede der 3 Grundmanipulationsformen einen entsprechenden Bereich und ein Textfeld pro Achsenrichtung (ab Zeile 210):

```
// Panels fuer die gemeinsamen Grundeigenschaften von Objekten
// Translation
JPanel t_properties = new JPanel();

t_properties.setBorder(BorderFactory.createTitledBorder("Translate"));
t_properties.setLayout(new GridLayout(3,2));
xTranslation = new JTextField();
xTranslation.setPreferredSize(new Dimension(80,24));
yTranslation = new JTextField();
zTranslation = new JTextField();
t_properties.add(new JLabel(" X:"));
t_properties.add(xTranslation);
t_properties.add(new JLabel(" Y:"));
t_properties.add(yTranslation);
t_properties.add(new JLabel(" Z:"));
t_properties.add(zTranslation);
```

Die Daten sollen zur Laufzeit aktualisiert werden, d.h. falls ein Objekt gerade verschoben wird, sollen die Textfelder immer die aktuellen Koordinaten des Objekts ausgeben! Da die Manipulationsvorgänge immer bei gedrückter rechter Maustaste stattfinden, können wir die entsprechende Methode zum Abgreifen der Daten direkt unten anfügen (Zeile 1130):

```
// Daten des angeklickten Objekts in die Textfelder eintragen
retrieveData();
```

In dieser Methode werden die Daten in die entsprechenden Textfelder eingetragen:

```
public void retrieveData()
{
	// Hole Position
	xTranslation.setText(String.valueOf(
			translationNode.getLocalTranslation().x));
	yTranslation.setText(String.valueOf(
			translationNode.getLocalTranslation().y));
	zTranslation.setText(String.valueOf(
			translationNode.getLocalTranslation().z));

	// Hole Rotation
	float[] getEulerAngles = new float[3];
	spatial.getLocalRotation().toAngles(getEulerAngles);

	xRotation.setText(String.valueOf(
			getEulerAngles[0]*FastMath.RAD_TO_DEG));
	yRotation.setText(String.valueOf(
			getEulerAngles[1]*FastMath.RAD_TO_DEG));
	zRotation.setText(String.valueOf(
			getEulerAngles[2]*FastMath.RAD_TO_DEG));

	// Hole Skalierung
	xScale.setText(String.valueOf(spatial.getLocalScale().x));
	yScale.setText(String.valueOf(spatial.getLocalScale().y));
	zScale.setText(String.valueOf(spatial.getLocalScale().z));
```

Hinweis zur Rotation: in unserem Editor arbeiten wir mit Euler-Winkeln. Das erlaubt uns Winkel in ähnlichem Maße abzufragen, wie man es vom Mathematikunterricht in der Schule gewöhnt ist. Mehr dazu aber im nächsten Abschnitt.

9.2 Objektdaten verändern

Optimal wäre es, wenn man die Daten manuell in diese Textfelder eingeben könnte, und diese Änderungen am besten gleichzeitig auf die Szene übertragen würden.
Um das zu bewerkstelligen müssen wir die Textfelder lediglich einer Art ActionListener hinzufügen. Tatsächlich soll das Textfeld auf Tastatureingaben reagieren, deswegen verwenden wir einen KeyListener. In welcher Methode im KeyLlistener diese Eingaben vollzogen werden soll ist jedem seine Sache. Ich habe es allerdings als angenehm empfunden, dass beim Loslassen einer Taste der gesamte Wert des Textfelds gleich auf das entsprechende Objekt übertragen wird.

Die Translation und Skalierung funktionieren einfach nach folgenden Zeilen:

```
// Translation
if(xTranslation.isFocusOwner())
{
      translationNode.getLocalTranslation().x = Float.parseFloat(
            xTranslation.getText());
}

...

// Skalierung
if(xScale.isFocusOwner())
{
      spatial.getLocalScale().x = Float.parseFloat(xScale.getText());
}
```

Dabei weisen wir dem „spatial“ (das angewählte Objekt) einfach die Werte zu, die wir manuell eingeben haben.
Die Rotation ist nicht so einfach umzusetzen. Damit beim eingegebenen Winkel die bereits vorhandenen und in andere Achsenrichtungen getätigten Rotationen berücksichtigt werden, müssen wir auch die anderen zwei Textfelder der Rotationsachsen abfragen!
Dabei erstellt man die notwendigen Variablen als Platzhalter und speichert die eventuell vorhandenen Werte darin ab. Die Rotation wird daraufhin anhand des neuen gesetzten Werts sowie der 2 bereits vorhandenen durchgeführt!

```
// Rotation
if(xRotation.isFocusOwner())
{
      // hole mir die bereits getätigten Drehungen des Objekts
      // und fülle damit den Array "rotation"
      rotation[1] =
Float.parseFloat(yRotation.getText())*FastMath.DEG_TO_RAD;
      rotation[2] =
Float.parseFloat(zRotation.getText())*FastMath.DEG_TO_RAD;

      //hole mir die vormals getätigte x-Drehung des Objekts
      // und setze sie neu
      rotation[0] =
Float.parseFloat(xRotation.getText())*FastMath.DEG_TO_RAD;

      // führe Drehung durch anhand aller 3 Rotationswerte
```

```
        spatial.getLocalRotation().fromAngles(rotation);
}
```

Weitere Erörterung zur Rotation:
In unserem Szene-Editor arbeiten wir mit Euler-Rotationen. Um diese besser zu verstehen dient folgende Darstellung:

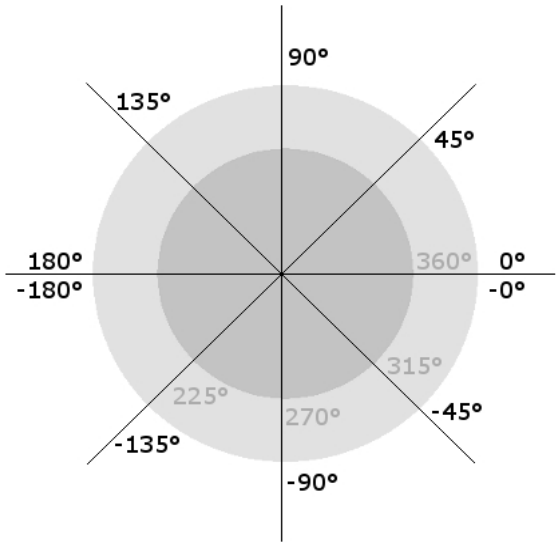
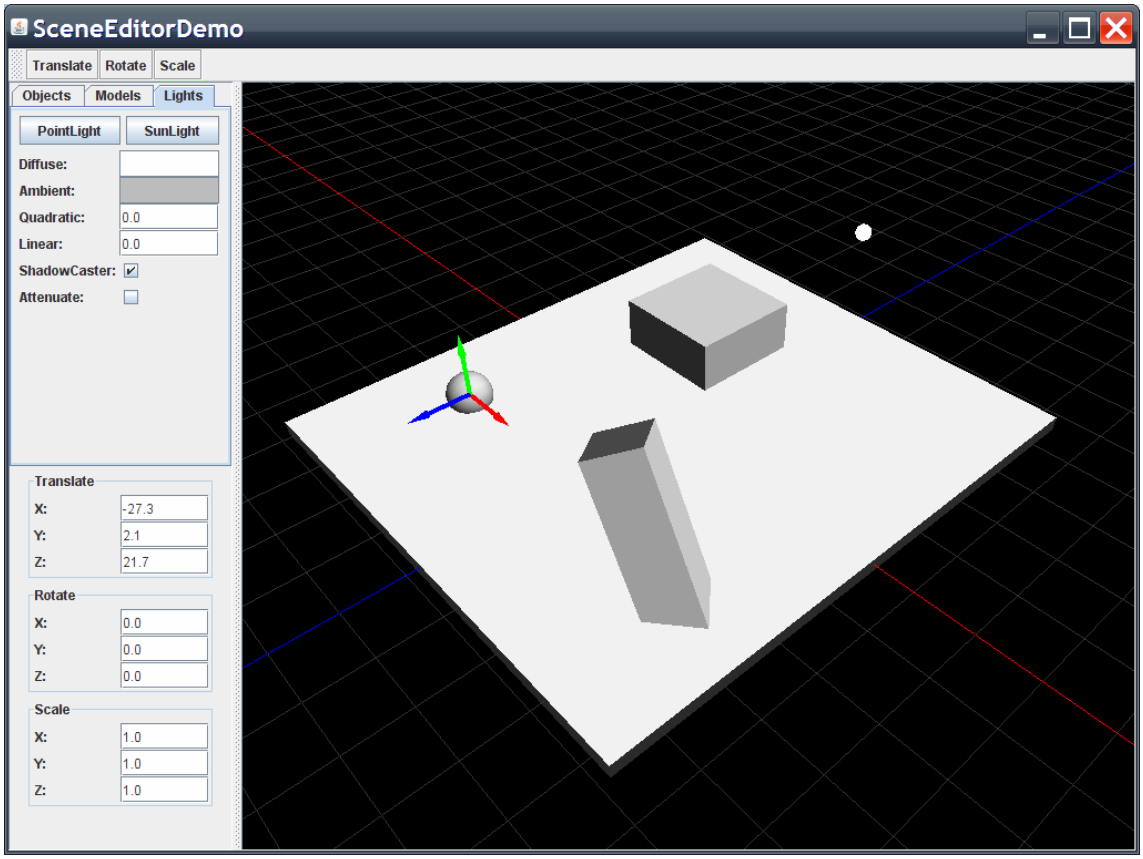


Abb. 4-13: Rotationen in jME nach Eulerwinkeln

Die hellen Werte beschreiben den Kreis von 0 bis 360 Grad. Die dunklen Werte entsprechen den Eulerwinkeln. Beides sind verschiedene Konzepte zum Rechnen mit Winkelfunktionen im Kreis.
Wie wir anhand der Skizze sehen sind von 0 bis 180 Grad beide Konzepte identisch. Erst bei 180° unterscheiden sie sich. Im Winkelmaß von Euler wird dabei der Winkel invertiert und weiter von -180° auf 0° heraufgezählt.
Dementsprechend ergeben sich andere Zahlen, z.B. 270° entsprechen -90° im Eulermaß, 315° entsprechen -45° usw. JME beherrscht allerdings die Umrechnung zwischen diesen beiden Konzepten. Wenn wir einen Wert im Textfeld eingeben wie 225, wird diese Zahl automatisch in -135° umgewandelt, geben wir die Zahl -135 ein, wird sie beibehalten. Es steht uns frei welche Art der Eingabe wir vorziehen.
Alternativ zur Rotation mit den Eulerwinkeln gäbe es die Möglichkeit eine Rotation mithilfe einer Quaternion durchzuführen. Ich habe diese Methode allerdings nicht sehr brauchbar gefunden, weil sie mit normierten Werten von 0-1 arbeitet. Es ist sehr umständlich aus diesen normierten Werten wieder den Winkel zurückzuberechnen.

10 Erstellen von Lichtobjekten



Wir haben im vorigen Kapitel gelernt, wie man Objekte in Echtzeit mit Hilfe eines Gizmo sowie mit manueller Eingabe von Werten geometrisch genau manipulieren können. Wenn wir jetzt versuchen 2 Objekte Wand an Wand nebeneinander zu positionieren fällt allerdings auf, dass wir nur schwer in der Lage sind die Seiten eines Objekts auszumachen, da jede Seite in strahlendem Weiß dargestellt ist. Was uns fehlt ist eine entsprechende Schattierung eines Objekts. Diese tritt erst dann auf, wenn wir mindestens 1 Lichtquelle in die Szene setzen.

Dazu erweitern wir das Swing GUI mit einem neuen Tabulator „Lights“ und setzen zu Beginn mehrere Textfelder, Buttons und Checkboxen fest. Ziel davon ist, dass wir bei diesen Lichter genau wie bei den geometrischen Grundobjekten die Daten aktiv verändern können.

In unserem Kurs behandeln wir zuerst das Punktlicht. Ein Punktlicht strahlt nach allen Seiten gleichmäßig Licht aus und die Lichtstärke nimmt in der Entfernung ab.
Beim Erweitern der graphischen Oberfläche (Zeile 254) legen wir zuerst eine Farbe für das Licht fest. Diese Farbe wird in der Klasse Color des AWT Packages erstellt. Sie kann allerdings nicht direkt mit dem von jME verwendeten ColorRGBA Format verwendet werden und muss zuerst umgewandelt werden. Der Farbwert in ColorRGBA ist normiert, d.h. wir müssen zuerst

jeden RGB Wert der Klasse Color durch 255 dividieren. Betrachten wir dazu den notwendigen Code ab Zeile 822.

```
ColorRGBA diffuseRGBA = new ColorRGBA();
diffuseRGBA.r = (float)diffuse.getRed()/255;
diffuseRGBA.g = (float)diffuse.getGreen()/255;
diffuseRGBA.b = (float)diffuse.getBlue()/255;
diffuseRGBA.a = 1;

ColorRGBA ambientRGBA = new ColorRGBA();
ambientRGBA.r = (float)ambient.getRed()/255;
ambientRGBA.g = (float)ambient.getGreen()/255;
ambientRGBA.b = (float)ambient.getBlue()/255;
ambientRGBA.a = 1;
```

Dasselbe Prinzip führen wir mit dem „ambient“ Wert aus. Die Farbe des Ambientwerts stellt die Helligkeit für Oberflächen dar, die nicht von der Lichtquelle beschienen werden. Wie wir im nächsten Kapitel sehen werden, beeinflusst dieser Wert auch die Helligkeit des Schattens. Für diesen wählen wir Schwarz als Ausgangsfarbe.
Anschließend erzeugen wir ein Punktlicht mit folgenden Eigenschaften:

```
// Erzeuge neues PointLight
PointLight pl = new PointLight();
pl.setDiffuse(diffuseRGBA);// Farbe des Lichts
pl.setQuadratic(0.001f);// Quadratische Abnahme
pl.setAttenuate(true);// Licht nimmt ab in der Distanz
pl.setEnabled(true);// aktiviert
pl.setAmbient(ambientRGBA);// Helligkeit des Schattens
pl.setShadowCaster(true);// wirft Schatten
```

Damit Lichter sichtbar sind, bzw. effektiv in einer Szene umgesetzt werden, ist es essentiell notwendig ein LightState für die Szene bzw. den entsprechenden Knoten zu erstellen.
In Zeile 1875 in der simpleSetup() Methode erstellen wir ein LightState und weisen diesen dem rootNode des Scenegraphs als RenderState zu.

```
lightState =
DisplaySystem.getDisplaySystem().getRenderer().createLightState();
lightState.detachAll();
lightState.setEnabled(true);
root.setRenderState(lightState);
root.updateRenderState();
```

Jetzt können wir wieder zurückspringen in die vorige Funktion und die folgende Zeile hinzufügen:

```
lightState.attach(pl);
```

Wenn wir das Programm jetzt ausführen, funktioniert das Licht bereits (z.B. PointLight hinzufügen, Box hinzufügen und ein wenig verschieben). Dummerweise sitzt das Licht am Nullpunkt des Koordinatensystems fest und wir sind nicht in der Lage dieses Licht anzuklicken oder zu verschieben. Am besten wäre es also, wenn man die Lichtquellen so ähnlich manipulieren könnte wie die normalen geometrischen Objekte.
Diese Notwendigkeit erinnert an den Editor „UnrealEd“ und die Art und Weise, wie dort das Verschieben von Lichtern in einer Szene umgesetzt wurde. Dabei wurde im Editor jede

Lichtquelle mit einem zweidimensionalen Glühbirne-Symbol dargestellt. Diese Glühbirnen konnte man im Editor wie gewöhnliche Objekte verschieben, waren im Spiel selbst aber nicht sichtbar!

Abb. 4-14: Die Lichtquellen im Editor „UnrealEd“ [Unr05] besitzen immer die Farbe des Lichts (hier orange-braun) und sind als 2D Billboards in Glühbirnensymbol dargestellt. Beachten Sie auch, dass die Fackeln selbst kein Licht abstrahlen, sondern nur die Glühbirnen! (vgl. hinteren Raum, in welchem ich die Lichtquellen entfernt habe).

Dieses Konzept wollen wir für unseren Editor ebenfalls anwenden. In jME sind wir zwar nicht in der Lage einem Licht direkt ein anderes Objekt anzuhängen, aber wir können mithilfe von Lichtknoten dieselbe Funktionsweise erzwingen. Statt den in UnrealEd verwendeten Billboard-Symbolen verwenden wir außerdem eine Farbkugel als geometrisches Objekt. Wir benötigen also folgende 4 Bausteine:

- Lichtknoten (LightNode)
- Licht (Light)
- Sphere (Farbkugel)
- Übergeordneter Knoten

Das Licht wird dabei in einen LightNode gepackt, die Sphere und der LightNode dem übergeordneten Knoten angehängt. Es ist zwar möglich die Sphere ebenfalls dem LightNode anzuhängen, aber das führt zu Problemen beim späteren Verschieben des Objekts. Das gesamte Lichtobjekt (wir benennen den obersten Knoten mit „lightObject“ sieht folgendermaßen aus:

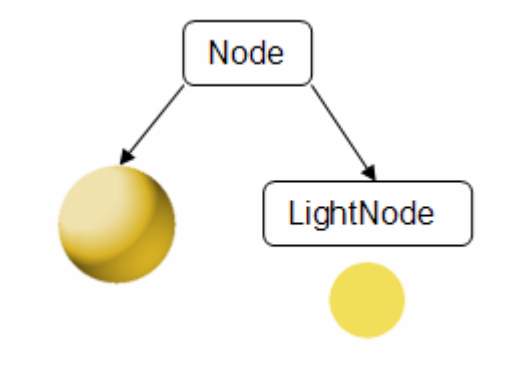


Abb. 4-15: Strukur eines „lightObject“: links die geometrische Kugel, rechts ein Lichtknoten, der das Licht enhält

Der linke Kreis stellt hier die Farbkugel dar und entspricht dem geometrischen Objekt, das angeklickt werden kann. Wie in der Zeichnung bereits farblich skizziert achten wir auch im Code darauf, dass die Kugel stets dieselbe Farbe besitzt wie das Licht!

```
Sphere ls = new Sphere("LIGHT", 10, 10, 1.0f);
ls.setModelBound(new BoundingBox());
ls.updateModelBound();
ls.setLightCombineMode(Spatial.LightCombineMode.Off);
ls.setSolidColor(diffuseRGBA);

LightNode ln = new LightNode("ln");
ln.setLight(pl);

// Erstelle einen uebergeordneten Knoten fuer das Licht und die
Leuchtkugel!
Node lightObject = new Node("lightObject");
lightObject.attachChild(ln);
lightObject.attachChild(ls);
ln.setLocalTranslation(ls.getLocalTranslation());

lightNode.attachChild(lightObject);
```

Da wir vereinbarten jeder Geometrie in der Szene einen Namen gegeben haben, geben wir auch der Sphere einen passenden: wir nennen sie „LIGHT“, da sie für eine Lichtquelle stehen soll. Weiters muss sich die Kugel an Position 0 im übergeordneten Knoten befinden, ansonsten findet der MouseRay das Objekt nicht und wirft eine Exception, weil er den LightNode findet und nicht das geometrisches Objekt. Bei Knoten gilt das LIFO (Last In First Out) Prinzip, d.h. das Objekt das als Letztes angehängt wird, befindet sich an erster Stelle. In diesem Fall hängen wir die Kugel erst am Schluss an. Anschließend legen wir fest, dass sich der LightNode „ln“ immer an jener Position befindet, an der die Leuchtkugel ist! Damit stellen wir sicher, dass beim Verschieben der Leuchtkugel auch die Lichtquelle ortgleich verschoben wird.

Wenn wir jetzt eine Farbkugel in der Szene anklicken, gelten bereits alle 3 Grundfunktionen der Manipulation geometrischer Objekte auch für das Licht und wir erhalten alle Daten darüber in den entsprechenden Textfeldern. Da das Licht jedoch noch über weitere Eigenschaften (Lichtfarbe, Reichweite etc.) verfügt, müssen wir auch diese berücksichtigen. Wir erstellen dazu

eine Reihe neuer Buttons und Checkboxen auf der Swing Oberfläche (bitte im Code nachlesen). In der Funktion retrieveData() (Zeile 1066) müssen wir danach folgende Dinge für die Zuordnung bewerkstelligen:

- Die Farbe des Lichts im Button anzeigen
- Das Ambient des Lichts im Button anzeigen
- Den Wert für die quadratische Abnahme des Lichts im Textfeld eintragen
- Den Wert für die lineare Abnahme des Lichts im Textfeld eintragen
- Die Checkbox auf true/false setzen, je nachdem ob dem Licht erlaubt sein soll Schatten zu werfen
- Die Checkbox auf true/false setzen, in Abhängigkeit davon, ob die Leuchtkraft des Lichts überhaupt abnehmen soll in der Distanz, oder konstant verbleiben soll.

```
// Hole die Eigenschaften des Lichts heraus
if(spatial.getName().equals("LIGHT"))
{
	LightNode ln = (LightNode) spatial.getParent().getChild(0);

	if(ln.getLight() instanceof PointLight)
	{
		PointLight pl = (PointLight) ln.getLight();

		// Hole RGB Werte des Lichts
		int diffR = (int) (pl.getDiffuse().r*255);
		int diffG = (int) (pl.getDiffuse().g*255);
		int diffB = (int) (pl.getDiffuse().b*255);

		// Button soll Farbe der Lichtquelle anzeigen
		diffuseButton.setBackground(new Color(diffR,diffG,diffB));

		// Hole RGB Werte der Schattenfarbe
		int ambR = (int) (pl.getAmbient().r*255);
		int ambG = (int) (pl.getAmbient().g*255);
		int ambB = (int) (pl.getAmbient().b*255);


		// Buttons soll die Ambientfarbe der Lichtquelle anzeigen
		ambientButton.setBackground(new Color(ambR,ambG,ambB));

		// Hole die Abnahmewerte des Lichts
		quadratic.setText(String.valueOf(pl.getQuadratic()));
		linear.setText(String.valueOf(pl.getLinear()));

		// Hole CheckBox-Werte
		if(pl.isShadowCaster())
			shadowCaster.setSelected(true);
		else
			shadowCaster.setSelected(false);

		if(pl.isAttenuate())
			attenuate.setSelected(true);
		else
			attenuate.setSelected(false);
	}
```

Hinweis: die Funktion pl.getDiffuse() liefert einen normierten ColorRGBA Wert zurück. Diese RGB Werte können natürlich wieder in die für uns gebräuchlichen (und vor allem von Swing

erkannten) Werte von jeweils 0-255 umgewandelt werden, indem man sie jeweils mit 255 multipliziert. Das Ergebnis ist meist mit Nachkommastellen behaftet, weswegen wir das Ergebnis auf einen ganzzahligen Wert casten.

Schreiten wir nun über zum aktiven Verändern all jener Werte, beginnend bei den zwei Buttons für die aktive Farbzuweisung eines Objekts.
Die einfachste Möglichkeit zum Implementieren einer Farbzuweisung für Lichter wäre aus Sicht des Programmierers 4 Textfelder zu erstellen für die Zahlenwerte in Rot/Grün/Blau/Alpha. Auf Dauer mag diese Lösung aber für den Anwender des Editors unbequem sein, da wir von Programmen wie Photoshop oder Gimp bereits wissen, dass es eine Art Auswahlfenster für Farbwerte geben sollte. Solch ein Farbauswahlfenster gibt es auch in Swing: den so genannten JColorChooser.
In unserem Editor benötigen wir 2 Farbwerte: Farbwert des Lichts, Helligkeitswert des Ambient. Für diese 2 Werte haben wir im GUI 2 Buttons erstellt und mit der Hintergrundfarbe des entsprechenden Lichts versehen. Nun wollen wir, dass ein Klick mit der Maus auf diesen Button das Farbauswahlfenster öffnet, wir eine bestimmte Farbe auswählen können, und diese Farbe sowohl den Button färbt, als auch die Leuchtkugel, sowie dem Licht diese Farbe zuweist!
Für die Buttons benötigen wir einen ActionListener, den wir unter dem Namen ColorListener implementieren (Zeile 542).

```
/**Klasse ermoeglicht das Veraendern der Farbe von Lichtobjekten */
class ColorListener implements ActionListener,ChangeListener
{
	boolean bDiffuse = false;
	boolean bAmbient = false;

	public void actionPerformed(ActionEvent ae)
	{
		String ert = ae.getActionCommand();
		if(ert.equals("Diffuse"))
		{
			JDialog dialog = new JDialog();
			dialog.setSize(400,400);

			cc = new JColorChooser();
			dialog.add(cc);
			dialog.setVisible(true);
			cc.getSelectionModel().addChangeListener(this);

			bDiffuse = true;
		}
```

Der Instanz des ColorChooser müssen wir einen ChangeListener hinzufügen. Dieser bewirkt, dass wir im Farbfeld des ColorChooser mit gedrückter Maustaste herumfahren können und die Farbe zeitgleich mit jedem abgegriffenen Wert in Echtzeit aktualisiert wird. Würde der ChangeListener nicht implementiert werden, würde erst beim Schließen des Dialogfelds der zuletzt angeklickte Farbwert angezeigt werden.

```
public void stateChanged(ChangeEvent ce)
```

```
{
	if(bDiffuse == true)
	{
		diffuse = cc.getColor();
		diffuseButton.setBackground(diffuse);

		if(spatial.getName().equals("LIGHT"))
		{
			Sphere sp = (Sphere) spatial;
			LightNode ln = (LightNode)
sp.getParent().getChild(0);
			if(ln.getLight() instanceof PointLight)
			{
				// Umwandeln der Farbe ins RGBA Format
				ColorRGBA diffRGBA = new ColorRGBA();
				diffRGBA.r = (float)diffuse.getRed()/255;
				diffRGBA.g = (float)diffuse.getGreen()/255;
				diffRGBA.b = (float)diffuse.getBlue()/255;
				diffRGBA.a = 1;

				// setze die Farbe fuer die Sphere
				sp.setSolidColor(diffRGBA);

				// setze die neue Farbe
				ln.getLight().setDiffuse(diffRGBA);

				sceneNode.updateRenderState();
			}
```

Wir weisen damit dem Button (diffuseButton), der Sphere (sp.setSolidColor(diffRGBA)) und dem Licht (ln.getLight().setDiffuse(diffRGBA)) diesen Farbwert zu. Der Farbwert, den wir aus dem ColorChooser übernehmen, muss allerdings auch hier erst normiert werden.

Wir kümmern uns anschließend um die beiden Textfelder mit dem Abnahmewert des Lichts im KeyListener (Zeile 738):

```
if(spatial.getName().equals("LIGHT"))
{
	Sphere sp = (Sphere) spatial;
	ln = (LightNode) sp.getParent().getChild(0);

	if(ln.getLight() instanceof PointLight)
	{
		// setze neue Werte
		ln.getLight().setQuadratic(Float.parseFloat(
quadratic.getText()));

	ln.getLight().setLinear(Float.parseFloat(linear.getText()));
	}
```

Anschließend müssen auch die beiden Checkboxen ensprechend behandelt werden (ab Zeile 758):

```
// Handling fuer CheckBoxes
public void itemStateChanged(ItemEvent ie)
{
	JCheckBox cb = (JCheckBox) ie.getItem();
```

```
if(cb==shadowCaster)
{
	if(cb.isSelected())
	{
		if(spatial.getName().equals("LIGHT"))
		{
			Sphere sp = (Sphere) spatial;

			ln = (LightNode) sp.getParent().getChild(0);
			ln.getLight().setShadowCaster(true);
		}
	}
	else
	{
		if(spatial.getName().equals("LIGHT"))
		{
			Sphere sp = (Sphere) spatial;

			ln = (LightNode) sp.getParent().getChild(0);
			ln.getLight().setShadowCaster(false);
		}
	}
	root.updateRenderState();
}
```

Hier wird je nachdem, ob der Benützer die Checkbox ankreuzt oder nicht der Wert auf „true“ oder „false“ gesetzt. Angekreuzt bedeutet „true“, nicht angekreuzt bedeutet „false“. Die Implementierung für die zweite Checkbox funktioniert analog dem oberen Codeabschnitt.
Ein Hinweis noch zur quadratischen oder linearen Abnahme des Lichts: Wenn Sie hier die Zahl 0 eingeben entspricht es einem konstanten Licht, d.h. das Licht ist in jeder Entfernung gleich stark. Tippen Sie die Zahl 1 ein, schaltet sich die Lichtquelle quasi aus. Je höher die Zahl, desto schwächer ist das Licht. Die lineare Abnahme unterscheidet sich von der quadratischen lediglich durch Stärke der Abnahme (in diesem Falle ist die Abnahme des Lichts in der Distanz nicht so stark).

Damit wäre die Implementierung für ein Punktlicht erledigt. Wie Sie bereits im Code sehen können, fordert die Implementierung eines anderen Lichttyps, z.B. eines sehr weit entfernten gerichteten Lichts (so wie beispielsweise die Sonne von der Erde aus gesehen) eine eigene, wenn auch analoge Behandlung. Die Richtung des Lichts wird dabei so festgestellt, indem von der Position der Lichtquelle in zum Nullpunkt des Koordinatensystems eine Gerade gezogen wird:

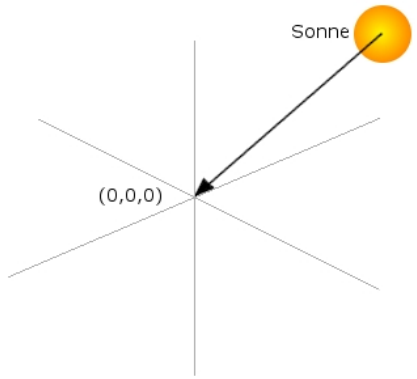


Abb. 4-16: Gerichtetes Licht

Neben diesen 2 Lichttypen gibt es noch einen dritten in jME: das Spotlight. Dieses funktioniert ähnlich wie das gerichtete Sonnenlicht, jedoch gibt es noch die zusätzliche Eigenschaft des Leuchtkegels. Zum Implementieren muss auch auf die Rotation des Lichtkegels Rücksicht genommen werden. Allerdings ist das nicht mehr Gegenstand des Kurses, es sollte Ihnen aber nicht allzu viele Schwierigkeiten bereiten, den letzten Typ selber zu implementieren.

Wir sind nun in der Lage all möglichen Eigenschaften eines Objekts zentral per Mausklick abzufragen und gleichzeitig auch zu verändern. Dies deckt sich auch mit der Funktionsweise von UnrealEd: dort kann man die erweiterten Eigenschaften (Advanced Options) eines Objekts ebenfalls zentral aufrufen:

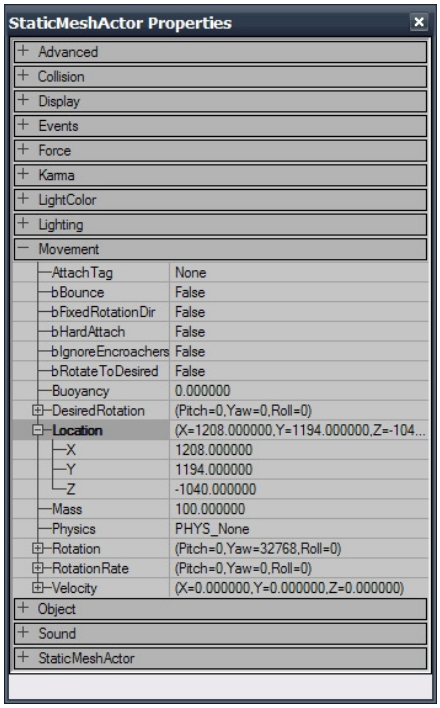


Abb. 4-17: Dialogfeld zum Anzeigen der Eigenschaften eines Objekts in UnrealEd [Unr05]: viele Eigenschaften bestimmter Objekte decken sich mit den Eigenschaften anderer Objekte wie z.B. hier das „Movement“, welches die Positions- und Rotationswerte beinhaltet. Skalierungs-eigenschaften befinden sich im Menü „Display“.

Für Benützer des Editors mag diese technisch anmutende Darstellung von Objekteigenschaften in UnrealEd relativ abschreckend sein. Wie wir jetzt allerdings herausgefunden haben, ist dies eine brauchbare Methode um spezielle oder gemeinsame Eigenschaften von Objekten abzufragen und zu verändern. Im Grunde genommen haben wir dasselbe in unserem Editor gemacht, nur ein wenig anders im GUI aufgeteilt. Wie in unserem Editor bewirken auch bei UnrealEd bestimmte Änderungen an manchen Optionen nichts für das ausgewählte Objekt (z.B. Farbänderung des Lichts, wenn ein Quader angeklickt wurde), sondern die Änderungen finden nur bei den Objekten statt, für die sie vorgesehen sind.

10.1 OpenGl und die Anzahl Lichter

Zum Schluss sollten Sie nicht vergessen, dass in jME mit OpenGL gearbeitet wird. OpenGL unterstützt standardmäßig nur 8 Lichter in der Szene. Je nach Art der Implementierung kann das auch erweitert werden. [Woo97]
Stellen Sie sich vor, dass sie eine Stadt modellieren wollen, mit Ampeln und Straßenlaternen... mit obiger Regelung wären Sie aufgeschmissen. Würden Sie mehr als 8 Lichtquellen hinzufügen, wäre das neunte Licht nicht mehr sichtbar!
In jME Version 1.0 ist man bedauerlicherweise noch einer alten Implementierung unterworfen, die nur 8 Lichter darstellen kann, deswegen arbeiten wir mit der inoffiziellen jME Version 2. In dieser wird das Manko automatisch gelöst, indem man folgende Zeile in der simpleUpdate() Methode einfügt:

```
// Funktion die mehr als 8 Lichter in einer Szene erlaubt
rootNode.sortLights();
```

Von nun an ist möglich, eine unendliche Anzahl Lichter in die Szene hinzuzufügen.

10.2 Vertex-Lighting

In jME wird Vertex-Lighting als Beleuchtungsmodell unterstützt
Beim Vertex-Lighting wird zuerst für jeden Eckpunkt einer Oberfläche (Face) berechnet wie viel Licht diese erhalten. Anschließend wird ein Gradient gezogen in Abhängigkeit von den berechneten Helligkeitswerten von jeweils einer Ecke zu den anderen. Jede Oberfläche besteht aus Dreiecken; es werden also immer Gradienten zwischen den 3 Eckpunkten berechnet.

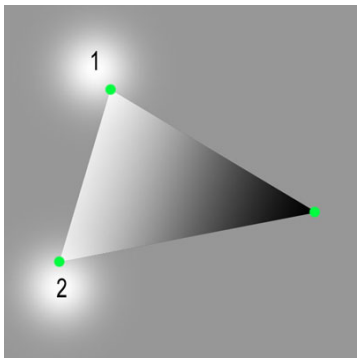


Abb. 4-18: Vertex Lighting

Wie die RGB-Farben werden die Helligkeitswerte in 0-255 unterteilt. Je näher der Wert an 0 ist, desto mehr verschwindet das Licht, es verblasst.
Die Zahlen 1 und 2 im Bild stellen zwei Lichtquellen dar. Der Helligkeitswert auf der rechten Ecke der Oberfläche stellt 0 dar und wird quasi nicht beleuchtet. Die Helligkeitswerte der oberen und unteren Ecke befinden sich beide beispielsweise auf dem Wert 240. Der Fläche zwischen den beiden linken Eckpunkten und dem einen rechten Eckpunkt wird jeweils in 0 bis 240 interpoliert [DeJ06].
Das Problem an dieser Methode ist, dass – sobald ein Objekt eine große Oberfläche hat – die Interpolation zwischen den Eckpunkten folgendes Resultat präsentiert:

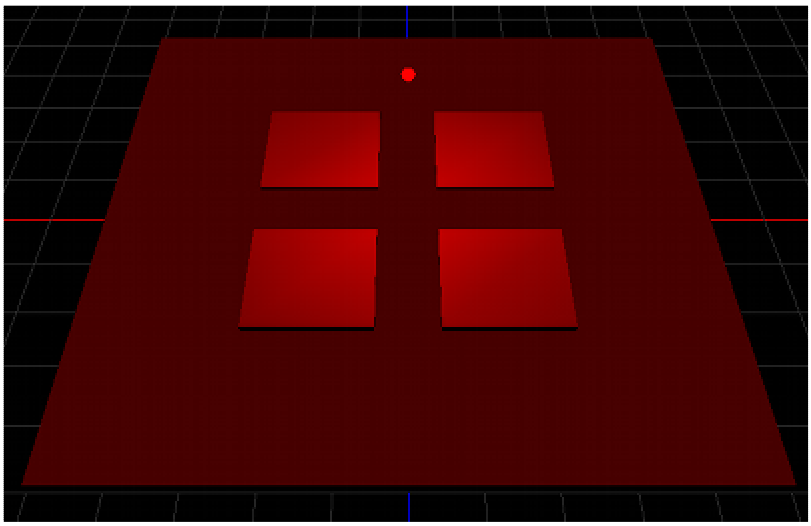


Abb. 4-19: Die untere große Box (oder Fläche) wird kaum beleuchtet, obwohl sie sich in nahezu gleichem Abstand zum Licht befindet wie die anderen 4 Boxen.

Wenn Sie riesige Objekte darstellen wollen z.B. eine Stadt visualisieren, die aus Hochhäusern besteht welche als einfache große Quader umgesetzt wurden, kann es sein, dass die Szene sehr dunkel bzw. quasi unbeleuchtet erscheint.
Abhilfe schafft bei diesem Problem leider nur folgende Taktik: man muss jede große Oberfläche in mehrere Abschnitte unterteilen (tesselieren) so wie hier:

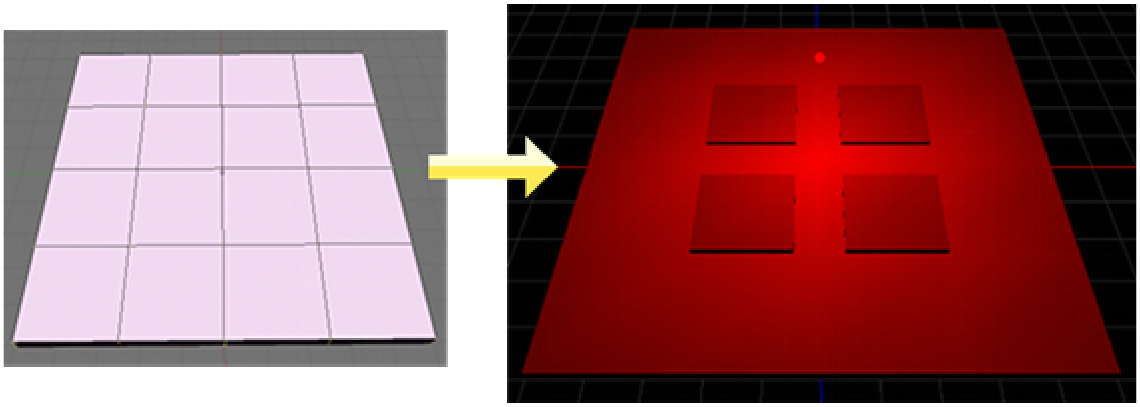


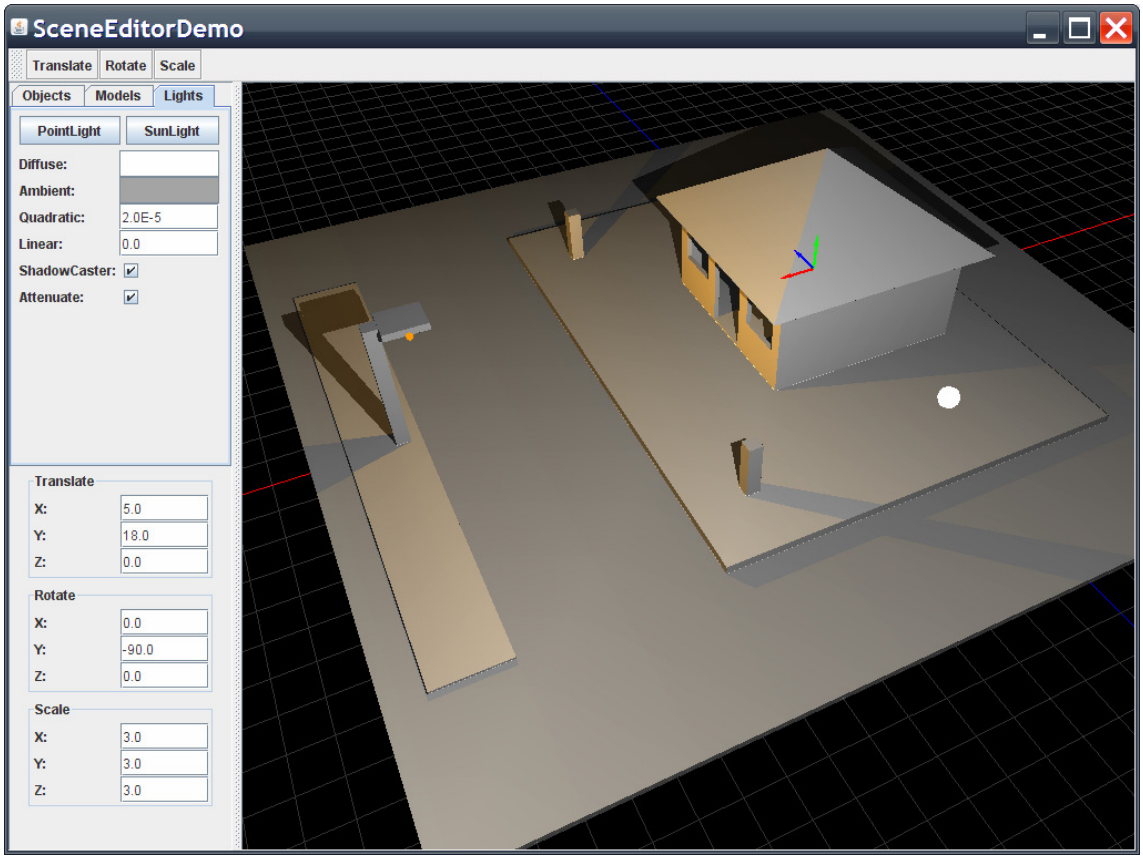
Abb. 4-20: Eine große Fläche unterteilt in mehrere kleine Flächen (in Blender3D erstellt) führt zu einem besseren Ergebnis. In diesem Falle sind es sogar die kleineren Boxen aus jME, die dunkler erscheinen.

Diese Umsetzung erfordert allerdings zusätzliche Arbeit im 3D-Modellierungsprogramm. Je mehr Unterteilungen, desto genauer und besser sieht das Ergebnis zwar aus, desto mehr hat aber auch die Performance darunter zu leiden aufgrund der zusätzlichen Polygone! Ein weiterer Nachteil ist auf dem rechten Bild zu sehen: dadurch, dass die verschiedene Flächen nicht genau gleich groß sind, ergeben sich Unregelmäßigkeiten in der Stärke der Beleuchtung, die diese Flächen erhalten!

10.3 Per-Pixel-Lighting

Eine Lösung und Alternative zum Problem des Vertex-Lighting ist das Per-Pixel-Lighting. Das Per-Pixel Lighting wurde erstmals von Nvidia entwickelt unter dem sogenannten NVIDIA Shading Rasterizer [Nvi00]. Bei dieser Methode wird für jeden einzelnen Pixel ein Beleuchtungswert errechnet. Damit ist man einerseits nicht mehr davon abhängig, die Oberflächen eines 3D Modells zu tessellieren, andererseits erspart man sich auch die Gefahr einer inkonsistenten Beleuchtung. Selbst bei weit gestreckten Flächen erzielt man so einen realistischen und genauen Beleuchtungseffekt erzielen.
Neben der Tatsache, dass das Ergebnis realistischer und besser aussieht ist das Per-Pixel Lighting auch deswegen empfehlenswert, weil sie von der Hardware bereits ab der Geforce2 GTS unterstützt wird. Leider ist diese Beleuchtungsmethode nicht in der aktuellen Version von jME implementiert.

11 Spezialeffekt: Schatten



Wo Licht ist, fällt auch Schatten. Wie Ihnen vielleicht aufgefallen ist, wird kann man den Lichtfluss nicht blockieren, indem man beispielsweise einen Quader zwischen eine Lichtquelle und eine Fläche schiebt. Der Grund dafür ist, dass die Oberflächen der Objekte in der Szene lediglich schattiert werden in Abhängigkeit von den Lichtquellen, nicht aber Schatten werfen. Eine Szene verliert dadurch merklich an Realismus. In diesem lernen Sie, wie man eine bekannte Art eines dynamischen und realistischen Schattentyps implementiert.

Bevor wir loslegen noch ein paar Worte zur Schattierung von Objekten: In der Computergrafik gibt es verschiedene Schattierungsmodelle für 3D Objekte. JME verwendet die Methode des Gouraud Shading. Diese wurde erstmals von Henri Gouraud revolutioniert. Dabei werden für jeden Eckpunkt einer Oberfläche bzw. eines Polygons die Normalen berechnet, und die erhaltenen Farbwerte von diesen Punkten auf der Oberfläche interpoliert. Durch den linearen Farbverlauf erscheinen die facettenartigen Oberflächen eines gewölbten 3D Objekts nun weicher [Gou71].

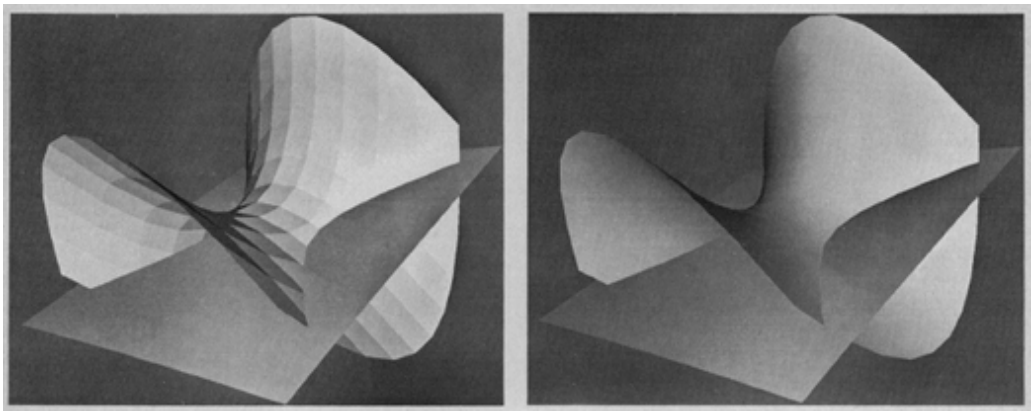


Abb. 4-21: Flat shading (Watkins) vs. Algorithmus von Gouraud rechts [Gou71]

Im Gegensatz dazu wird beim Flat-Shading von Watkins nur jeweils ein einziger Helligkeitswert für eine Oberfläche berechnet. Die Methode von Gouraud ist sehr schnell und gebräuchlich in der Computergrafik. Obwohl das Gouraud-Shading ähnlich wie das Vertex-Lighting ist, das wir im vorigen Kapitel besprochen haben, muss es nicht zwingend mit diesem Beleuchtungsmodell umgesetzt werden.
Eine alternative Schattierungsmethode stellt das Phong-Shading dar. Ihr Vorteil besteht vor allem darin, dass Lichter auf glänzenden Oberflächen besser dargestellt werden können. Sie interpoliert dazu die Oberflächennormalen und evaluiert einen Helligkeitswert für jeden einzelnen Pixel auf dem Polygon [Lyo93]. Diese Methode ist sehr rechenaufwändig, liefert aber ein schöneres Ergebnis. Sie ist nicht in jME implementiert.

Im Gegensatz zu den Schattierungsverfahren ist die Berechnung eines richtigen Schattenwurfs noch ein Stück aufwändiger. Bevor wir uns den Code für den Editor anschauen werde ich Ihnen zuerst 3 gebräuchliche Arten von Schatten in Spielen vorstellen:

- Lightmapping
- Shadow Volumes
- Shadowmaps

11.1 Lightmaps

„A 2D light map is a texture map applied to the surfaces of a scene, modulating the intensity of the surfaces to simulate the effects of a local light. If the surface is already textured, then applying the light map becomes a multipass operation, modulating the intensity of a surface detail texture.“ [McR98]
Dieses Zitat beschreibt die Vorgangsweise der angewendeten Technik: in dieser Methode wird ein Abbild der Beleuchtung, die auf ein Objekt fällt, auf die entsprechenden Oberflächen des Objekts gesetzt. Das Abbild kann aus Helligkeits- oder Farbwerten bestehen. Wenn die Oberfläche bereits eine Textur besitzt, wird dieses Abbild (genannt „Light Map“) mithilfe einer

Multipass Operation auf die vorhandene Textur gesetzt. Der Effekt, der durch eine Kombination dieser Art hervorgerufen wird nennt man auch „Multitexturing“.

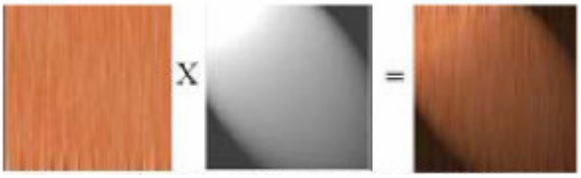


Abb. 4-22: Erzeugen von Lightmaps [Maz05]

Dieser Prozess erzeugt Licht und Schatten, die beständig und unveränderbar auf Objekten dargestellt werden. Durch das nur einmalige Berechnen von Licht und Schatten spart man sich in der aktiven Szene viel Rechenzeit. Somit können auch komplexe Szenen in Spielen mit realistischer Licht- und Schattendarstellung eine sehr gute Performance verzeichnen.
Lightmaps wurde erstmals in Spielen wie z.B. Quake eingesetzt [Maz05]. Der Editor von Unreal Tournament 2004 verwendet diese Methode ebenfalls zum Erstellen von Licht- und Schatten. Dabei wird eine Spielkarte mitsamt diesen Lightmaps gespeichert und beim Starten einer Spiels in UT2004 wird diese Spielkarte mitsamt den fertig berechneten Lightmaps geladen.
In Hinblick auf die Verwendung von UnrealEd möchte ich auf ein Problem in der Anwendung dieser Technik aufmerksam machen: gerade beim Erstellen großer Szenarien mit vielen darin vorkommenden Objekten und Lichtquellen ist das Designen von manchen Spielkarten sehr zeitaufwändig. Stellen Sie sich vor, sie setzen Lichtquellen an bestimmt Positionen im virtuellen Raum und wollen wissen, wie der Schattenwurf bzw. die fertige Lightmap dazu aussieht. Damit der Licht- und Schattenwurf sichtbar werden muss zuerst das gesamte Szenario nach diesem Algorithmus gerendert werden. Je nach Größe von Szenario dauert das Rendern von einigen Sekunden bis hin zu mehreren Minuten! Sollten Sie dann mit dem Ergebnis der Beleuchtung nicht zufrieden sein, muss nach dem erneuten Verschieben der Lichtquellen die gesamte Szene wiederum neu gerendert werden. Gerade bei großen und komplexen Spielkarten ist dieser Prozess sehr langwierig und Sie müssen wegen dem ständigen Warten auf die Lightmaps eine Menge Geduld aufbringen. Ein weiterer Nachteil der Lightmaps ist, dass sie nur für statische Objekte verwendet werden kann. Es damit nicht möglich eine dynamische Beleuchtung in Echtzeit darzustellen. Spiele, die über einen Tag-Nacht Zyklus verfügen müssen sich also eine Alternative suchen.
Die gebräuchlichsten Techniken, um Echtzeitschatten zu visualisieren sind Shadow Volumes und Shadow mapping.

11.2 Shadow Volumes

Das Schattenvolumen wurde erstmals von Frank Crow 1977 [Cro77] erfunden. Ein Schattenvolumen ist ein Raum zwischen der Silhouette eines 3D Objekts, das Schatten werfen soll, und der Fläche auf die der Schatten fällt bzw. genauer:
„…an object’s shadow volume encloses the region of space for which light is blocked by the object. This volume is constructed by finding the edges in the object’s triangle mesh

representing the boundary between lit triangles and unlit triangles and extruding those edges away from the light source.“ [Len02]
Diese Silhouette wird also bestimmt, indem man die Grenzen zwischen den beleuchteten und unbeleuchteten Bereicken in den Polygonen eines Objekts berechnet, mit den Kanten eine Schablone erstellt, und von dieser Schablone ausgehend ein Volumen extrudiert bis (im Idealfall) in die Unendlichkeit. Alle darin befindliche Objekte und Oberflächen werden schattiert.

Abb. 4-23: Screenshot von „Doom 3“ [Ids04]

Die Berechnung ist allerdings sehr aufwändig und benötigt eine hohe Texturfüllrate. Diese wird in den heutigen Grafikkarten immer mehr erweitert. Die größten Vorteile dieser Technik bestehen darin, dass sie pixelgenaue Resultate liefert, Selbstschattierung erzeugt, sowie für Omni-Lichter [Eve02] (Lichter die nach allen Seiten abstrahlen) verwendet werden kann. Das Computerspiel Doom 3 verwendete diese Technik zur Darstellung von Echtzeitschatten.

11.3 Shadowmaps

Shadow mapping ist eine Technik, die bereits 1978 von Lance Williams entwickelt wurde [Wil78]. Das Prinzip lässt sich so erklären: eine Szene wird zunächst aus der Sicht einer Lichtquelle gerendert: alles was von dort aus sichtbar ist, befindet sich im Licht, alles was nicht sichtbar ist, befindet sich im Schatten. Von allen Oberflächen, die aus der Sicht des Lichts sichtbar sind, werden nun die Tiefenwerte in einem Buffer gespeichert (Shadow Map). Danach wird die Gesamtszene aus Sicht der Kamera gerendert. Dabei werden die Tiefenwerte von jedem Punkt, der gezeichnet wird, mit den Werten im Buffer vergleicht und ihn in Abhängigkeit davon in Licht oder Schatten darstellt [Wim04].
Diese Technik eignet sich insbesondere dafür hardwaremäßig implementiert zu werden. Ein Standardproblem bei Shadowmaps ist allerdings die Stufenbildung.

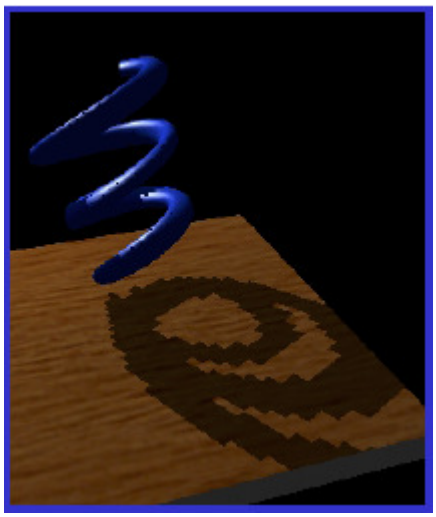


Abb. 4-24: Stufenbildung bei Shadowmaps [Eve05]

Um dieses Problem zu lösen wurde die Technik „Percentage Closer Filtering“ (PCF) entwickelt. Für ATI Grafikkarten wurde erst ab der Radeon X1300 Reihe eine entsprechende Technik entwickelt, die diese Berechnungen hardwaremäßig unterstützt, genannt „Fetch 4“ [Isi06]. Grafikkarten von Nvidia unterstützen diese Technik schon seit der Geforce3 Reihe.
Die Implementierung von Shadow mapping in jME ist aufwändiger als die von Shadow Volumes. Shadow mapping wurde erst gegen Ende Sommer 2008 von einem Forumsmitglied (Kevin Glass) für jME implementiert [Gla08] und war bis dato nicht vorhanden. Da ich die Programmierarbeit auf einem Computer mit einer Ati9600 Grafikkarte erstellte, die PCF leider nicht unterstützte war das Resultat für die Schatten nicht sehr ansehnlich, und das Resultat waren die befürchteten Treppeneffekte. Dazu erlebte ich eine extrem schlechte Performance beim Ausprobieren von dem mitgelieferten Samplecodes. Aus diesem Grund habe ich mich entschieden Shadow Volumes als Schatteneffekt zu verwenden.

Wir erweitern das GUI mit einer neuen Funktion um die Schatten ein- oder auszuschalten. Wir wollen diesmal keinen neuen Button erstellen, sondern die Aktion auf eine ganz bestimmte Taste der Tastatur zuweisen. Dabei soll ein einmaliges Drücken die Schatten ausschalten, ein weiteres Drücken die Schatten wieder einschalten. Das Toggeln des Zustandes wird mit einer Boolean umgesetzt, die ihren Wahrheitswert bei jedem erneuten Drücken der Taste invertiert.
Das Drücken der Taste soll weiters nicht davon abhängig sein, ob der glCanvas im Fokus steht oder nicht (sowie beim Steuern der Kamera), sondern „global“ in der Applikation zugänglich sein. Ab Zeile 414 findet sich der notwendige Code:

```
Toolkit.getDefaultToolkit().getSystemEventQueue().push(new
EventQueue()
{
      public void dispatchEvent(AWTEvent awte)
      {
            if (awte instanceof KeyEvent)
            {
                  KeyEvent keyEvent = (KeyEvent) awte;
```

```
                    if (keyEvent.getID() == KeyEvent.KEY_RELEASED)
                    {
                        if(keyEvent.getKeyCode() == KeyEvent.VK_H)
                        {
                            if(!showShadows)
                                showShadows = true;
                            else
                                showShadows = false;
                            }
                        }
                    }
                    super.dispatchEvent(awte);
                }
            });
    }
```

Die EventQueue ist eine plattform-unabhängiges-Klasse, die Ereignisse von zugrunde liegenden Peer-Klassen und sicheren Anwendungen aufnimmt. Sie kapselt asynchronen Ereignisse aus der Warteschlange und versendet sie durch den Aufruf von dispatchEvent(AWTEvent). In unserem Beispiel erstellen wir ein KeyEvent als Ereignis und reihen dieses in die aktuelle Ereignis-Warteschlange ein! Wir wählen die Taste H zum Toggeln des Schattens.

Um die Schatten zu erstellen, benötigen wir einen ShadowedRenderPass, sowie einen PassManager. Wir erstellen für unsere Szene gleich zwei von ihnen. Einen, für diejenigen Knoten in der Szene, die vom Schatten beeinflusst sein sollen, und einen anderen, die nicht davon betroffen sind. Wir initialisieren diese in Zeile 168.

```
BasicPassManager passManager;
static ShadowedRenderPass sPass1 = new ShadowedRenderPass();
static ShadowedRenderPass sPass2 = new ShadowedRenderPass();
```

Weiters verwenden wir in unserem Editor additive Schatten. Grundsätzlich gibt es neben diesen auch die modulativen Schatten.
Die modulativen Schatten zeichnen sich darin aus, dass alle Schatten mit einer vordefinierten festen Farbe gezeichnet werden z.B. Schwarz. Zuerst wird die Szene gerendert mit allen Objekten, die einen Schattenbereich fallen, dann wird ein modulativer Pass erzeugt, der diese Bereiche im Schatten verdunkelt. Anschließend werden die Objekte gerendert, die keinen Schatten erhalten sollen [Ogr08].
Wenn Sie verschiedene Lichtquellen in der Szene besitzen und diese Schatten sich überschneiden besitzen alle dieselbe Farbe. In der Realität besitzen Schatten jedoch oft eine unterschiedliche Stärke und können zusammen mit anderen Lichtern Farbkombinationen erzeugen. Diese werden bei den additiven Schatten berücksichtigt.
Bei den additiven Schatten wird eine Szene zunächst mehrmals gerendert – jedes Mal aber nur mit einem einzigen Licht. Die Anzahl der Rendervorgänge ist dabei von der Anzahl Lichter in der Szene abhängig. Bei jedem dieser Durchgänge wird ein Pass mit jeweils nur einem Licht erzeugt und anschließend zum vorherigen Pass (bzw. zu den vorhergehenden) hinzugefügt und

erzeugt damit diese Farbkombinationen. Diese Technik ist sehr effizient und erzeugt realistisch aussehende Beleuchtung [Ogr08].

Wir nehmen für unseren Scenegraph (ab root) den ersten RenderPass und fügen diesem alle Zweige hinzu, die vom Schattenwurf betroffen sein sollen. Dazu reicht es, wenn wir ihm die root als Occluder mitgeben. Der Schattenwurf betrifft automatisch auch alle darunter vorkommenden Zweige bzw. Blätter. Hier aktivieren wir das Rendern von Schatten.
Anschließend müssen wir noch diejenigen Knoten in einen anderen Pass setzen, die unbeeinflusst vom Schattenwurf sein sollen (helperNode, lightNode). Wir nehmen also den zweiten RenderPass für unseren Scenegraph (ab root) und fügen diesem all jene Zweige hinzu, die nicht vom Schattenwurf beeinflusst werden sollen. Dort deaktivieren wir das Rendern von Schatten.

```
// Zuerst alle Nodes einem gemeinsamen RenderPass hinzufügen
sPass1.add(root);
sPass1.addOccluder(root);
sPass1.setRenderShadows(true);
sPass1.setLightingMethod(ShadowedRenderPass.LightingMethod.Additive);
sPass1.setEnabled(true);
passManager = new BasicPassManager();
passManager.add(sPass1);

// Zweiter RenderPass für Nodes, die unbeeinflusst sein sollen von
Schatten
sPass2.add(root);
sPass2.addOccluder(helperNode);
sPass2.addOccluder(lightNode);
sPass2.setRenderShadows(false);
sPass2.setEnabled(true);
passManager.add(sPass2);
```

Damit die Schatten auch effektiv angezeigt bzw. gerendert werden, brauchen wir zum ersten Mal die Funktion simpleRender() in Zeile 2002.

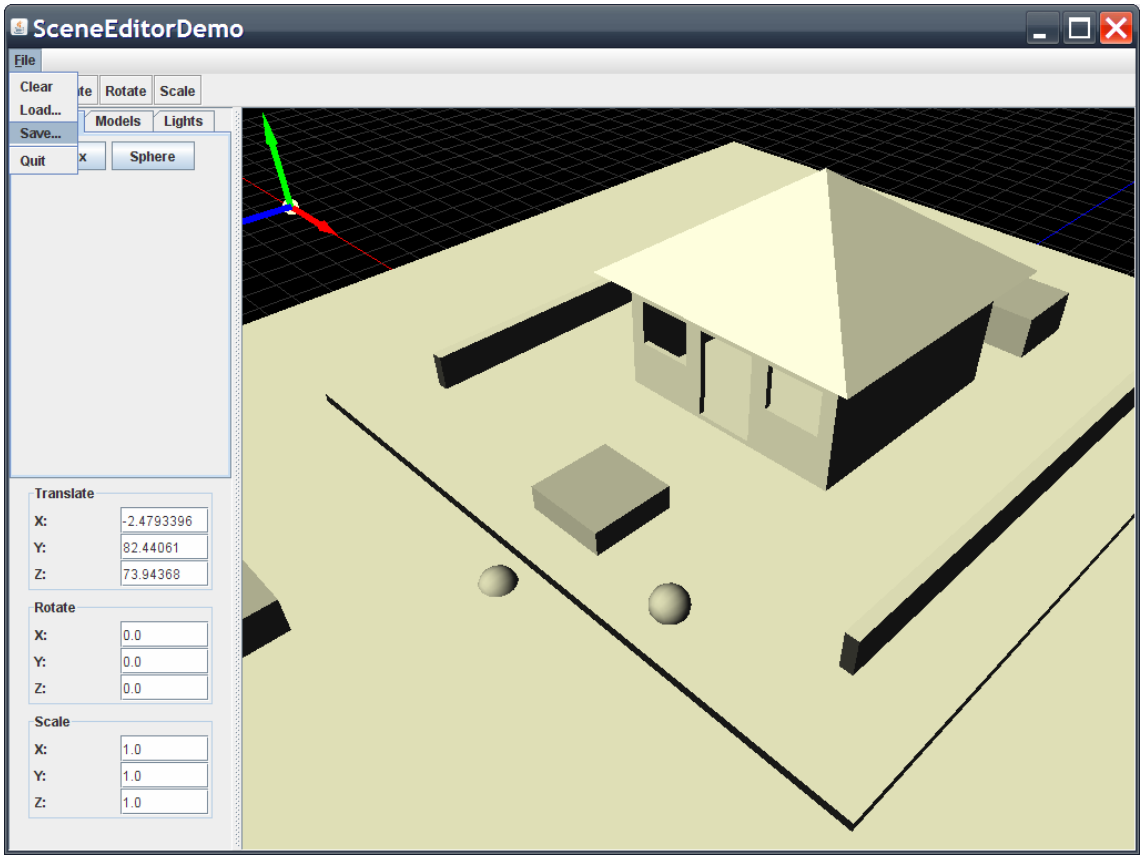
```
if(showShadows)
{
      passManager.renderPasses(
      DisplaySystem.getDisplaySystem().getRenderer());
            ShadowedRenderPass.blended.setSourceFunction(
      BlendState.SourceFunction.One);
            ShadowedRenderPass.blended.setDestinationFunction(
      BlendState.DestinationFunction.One);
            ShadowedRenderPass.blendTex.setSourceFunction(
      BlendState.SourceFunction.Zero);
            ShadowedRenderPass.blendTex.setDestinationFunction(
      BlendState.DestinationFunction.One);
}
```

Mit den 4 unterschiedlichen Werten, die hier zu sehen sind, lassen sich Helligkeit und Verhalten von Schatten festlegen. Obige Einstellung ist von mir nach mehrmaligem Herumprobieren festgesetzt worden weil sie meines Erachtens nach das schönste Ergebnis liefert. Sollen Sie dennoch unzufrieden sein, gibt es die Hilfsklasse „ShadowTweaker“ in jME (in Zeile 1964), die Sie mit Start der Applikation aktivieren können. Dabei öffnet sich ein kleines Dialogfeld, in

welchem Sie die 4 verschiedenen Werte anpassen können. Wenn Ihnen eine besondere Einstellung gefällt vergessen Sie nicht diese in der simpleRender() Methode entsprechend zu implementieren!

```
// new ShadowTweaker(sPass1).setVisible(true);// zum Tweaken von
Schatten
```

12 Speichern und Laden von Szenen



Wie uns bereits aufgefallen ist, ist es etwas umständlich jedes Mal die Applikation neu zu starten um eine neue Szene zu erschaffen. Ebenso vermissen wir eine Funktion, die es erlaubt eine schön zusammengesetzte Szene abzuspeichern und bei Bedarf neu zu laden. In diesem Kapitel lernen Sie, wie man Szenen löschen, abspeichern und laden kann!

Wir erweitern unser GUI mit einer Menüleiste. In dieser legen wir ein Menü an für die folgenden Funktionen:

- clear(): Szene löschen
- load(): Szene laden
- save(): Szene speichern

Behandeln wir zuerst das Löschen der vorhandenen Szene (ab Zeile 603):
Damit alle Objekte von der Bildfläche verschwinden müssen wir sie nur vom entsprechenden Knoten entfernen. Da sich unsere Objekte alle im geometryNode befinden, können wir also folgende Funktion anwenden:

```
geometryNode.detachAllChildren();
```

Weitere Objekte befinden sich im lightNode (die Farbkugeln, die wir verwendeten um Lichter zu positionieren). Dasselbe hier:

```
lightNode.detachAllChildren();
```

Würde man jetzt eine neue Szene erschaffen (bzw. neue Objekte hinzufügen) würde uns etwas auffallen: all diese Objekte würden schon von irgendwelchen Lichtquellen beleuchtet werden. Nach näherer Untersuchung würden wir feststellen, dass es dieselben Lichtquellen wären wie die in der vorigen Szene, obgleich wir alle Farbkugeln entfernt haben! Der Grund für dieses Verhalten liegt darin, dass die Szene immer noch den alten LightState besitzt (und wie wir uns erinnern, haben wir dem „lightState“ alle Lichter hinzugefügt). Wir müssen also auch den vorhandenen LightState leeren:

```
lightState.detachAll();// entfernt alle Lichter
```

Gehen wir nun über zum Speichern einer Szene:
Nehmen wir an Sie haben eine Szene teilweise zusammengesetzt, und diese befindet sich einem ähnlichen „baustellenartigen“ Zustand wie das Bild am Anfang des Kapitels zeigt. Plötzlich werden Sie durch irgendein Ereignis abgelenkt und müssen aufhören weiter an der Szene zu basteln. Um später weiterarbeiten zu können, und um nicht wieder von neu beginnen zu müssen, wollen Sie das bereits vorhandene Werk abspeichern.
Um eine Szene zu speichern verwendet jME den Ansatz einen Objektbaum zu serialisieren, d.h. alle Objekte und deren Zustände und Beziehungen eines bestimmten Zweiges im Scenegraph können damit komplett automatisch abgespeichert werden. Die Zustände aller betroffenen Objekte werden in eine Bytefolge umgewandelt und können daraus auch wieder hergestellt werden.
Dabei ist es nur notwendig den entsprechenden Knoten zu speichern! Alle sich darunter befindlichen weiteren Knoten, Zweige, Blätter werden automatisch mitgespeichert.
Bei unserer Szene ist darauf zu achten, dass nur die relevanten Teile der Szene abgespeichert werden, nur die Dinge, die auch in einem Spiel sichtbar sein sollen! In unserem Falle setzt sich die Szene nur aus dem de lightNode und dem geometryNode zusammen. Diese beiden Knoten besitzen den übergeordneten Knoten sceneNode. Genau diesen Knoten müssen wir also speichern (Zeile 690). Der Grid oder das Gizmo sind nur Hilfsmittel der Applikation und sollen nicht in die abgespeicherte Szene miteinbezogen werden.

```
BinaryExporter.getInstance().save(sceneNode,fc.getSelectedFile());
```

Sie können jede beliebige Endung für den Dateinamen wählen, es empfiehlt sich allerdings die Endung .jme, da ich bereits einen Filter für das Dialogfenster erstellt habe.

Das Laden einer Szene gestaltet sich allerdings nicht ganz so einfach wie das Speichern (ab Zeile 614):
Sobald wir eine Szene laden, muss zuerst eine eventuell vorher vorhandene Szene gelöscht werden. Dazu rufen wir einfach die clear() Funktion noch einmal auf. Anschließend müssen wir

die Knoten lightNode und geometryNode vom aktuellen sceneNode im Editor entfernen. Wie wir uns erinnern, wurden diese beiden Knoten gespeichert. Würden wir sie nicht aus der aktuellen Szene entfernen, hätten wir nach dem Ladevorgang zwei lightNodes und zwei geometryNodes. Das würde zu Komplikationen führen. Anschließend importieren wir unsere abgespeicherte Datei, die wir mit dem Dialogfenster anwählen können:

```
// Laden der abgespeicherte Szene als Node
Node loadedNode = (Node) BinaryImporter.getInstance().load(
      fc.getSelectedFile());
```

Die geladene Szene bzw. der „Hauptknoten“ der geladenen Szene (der abgespeicherte sceneNode) wird zuerst in einen Knoten konvertiert. Danach müssen die in diesem sceneNode vorhandenen Objekte in die aktuelle Szene hinzugefügt werden. Dazu erstellen wir eine ArrayList und speichern alle Kinder des geladenen Knotens in dieser.

```
ArrayList<Spatial> allspat = new ArrayList<Spatial>();
allspat.addAll(loadedNode.getChildren());

for(Spatial s:allspat)
{
      sceneNode.attachChild(s);
}
```

Wir durchlaufen diese ArrayList also und hängen jedes Objekt, das wir darin finden, dem sceneNode der aktuellen Szene an! Wie wir wissen, sollte es sich bei den Objekten nur um 2 Kinder handeln:

1. der Knoten „lightNode“
2. der Knoten „geometryNode

Damit die vormalige Scenegraphstruktur im Editor wieder gültig ist müssen diese zwei Knoten in der aktuellen Szene den statischen Referenzen entsprechen; wir weisen also jeden der geladenen Knoten dem entsprechenden statischen Pendant der aktuellen Szene hinzu:

```
// setze die beiden geladenen Knoten gleich den statischen Referenzen
geometryNode = (Node)sceneNode.getChild(0);
lightNode = (Node)sceneNode.getChild(1);
```

Hinweis: Achten Sie darauf in welcher Reihenfolge die Knoten in der Szene angeordnet sind, damit Sie sie auch wieder in richtiger Reihenfolge anhängen können!

Würde man mit obigen Zeilen den Ladevorgang ausführen, würde man bemerken, dass zwar alle Objekte und 3D Modelle geladen würden, die Szene jedoch dunkel bleibt. Die geladenen Lichter schalten sich leider nicht von selbst ein und müssen manuell aktiviert werden. Dazu müssen erst alle Lichtquellen aus dem vorhandenen Knoten (in diesem Falle lightNode) extrahiert werden, und dann an den aktuellen LightState der Szene angehängt werden. Dadurch dass wir alle Lichter zentral in den lightNode verfrachtet haben, bewahren wir den Überblick und können diesen Knoten direkt abgreifen:

```
for(int i=0;i<lightNode.getQuantity();i++)
{
      // der uebergeordnete Knoten fuer Lichter ist bei uns
      // das "lightObject"
```

```
        if(lightNode.getChild(i).getName().equals("lightObject"))
        {
                Node node = (Node) lightNode.getChild(i);
                LightNode ln = (LightNode) node.getChild(0);

                Light l = ln.getLight();
                lightState.attach(l);
        }
}
```

12.1 Weitere Speichermöglichkeiten

Neben oben erwähnter binärer Speicherart, gibt es auch andere Methoden zur Speicherung von Szenen. Da nicht alles immer binär gespeichert werden kann, sollte man sich eine Art Struktur überlegen für das Abspeichern von Szenen z.B. Ordner für Texturen, Ordner für Musik zur Untermalung der Szene, Ordner für verwendete 3D Modelle usw.
Wenn Sie eine Szene abspeichern, benötigen Sie theoretisch nur eine Beschreibung des Pfads der verwendeten Dateien (z.B. den verwendeten 3D Modellen) und die Szene holt sich diese entsprechend der Szenebeschreibung aus einer festen Pfadangabe. Diese Pfadangabe resultiert meist aus einer Ordnerstruktur, die beim Installieren eines Spiels feststeht und die auf jedem Computer gleich aussieht.
Eine Ordnerstruktur für ein Spiel kann beispielsweise so aussehen:

```
Game
        |-Game.exe
        |-Game.ini
        |-Game.conf
        |- …
        |-Maps
                |- Scenario1.map
                |- …
        |-Models
                |-Player1.obj
                |- …
        |-Sounds
                |-Bird1.wav
                |- …
        |-Textures
```

So wissen Sie als Spielentwickler immer, wo bzw. in welcher Ebene sich die Texturen, 3D Modelle, Sounds etc. befinden und können beim Zusammensetzen einer Szene direkt auf diese Komponenten zugreifen. Die UnrealEngine verwendet ein Ähnliches System wie oben [Unr05].
Nicht alle Spiele verwenden so ein offenes System. Manche speichern verschiedene Komponenten einer Szene in archivierte Dateien ab. Damit das Spiel auf diese zugreifen kann, muss die GameEngine in der Lage sein, diese Dateien zu öffnen und wieder zu schließen. Ob sich jetzt die gesamte Szene (mitsamt den notwendigen Texturen, Modellen etc.) in autonomer Form in diesem Archiv befindet, ob dieses Archiv nur eine Beschreibung der Szene beinhaltet, oder ob diese Archivierung gar so funktioniert dass jede Komponente einer Szene sich in einer eigenen Archivdatei befindet, hängt wieder von der GameEngine ab.

Oft werden Archiv-Formate verwendet, die eher unbekannt sind bzw. verschlüsselt und nicht mit gängigen Komprimierungsprogrammen wie „7Zip“ oder „Winrar“ geöffnet werden können. Ein Grund dafür ist, dass es sich um geschützte Inhalte handelt auf die nicht jeder zugreifen soll und darf.

Um eine Szene in Form einer Beschreibung zu erstellen, bietet sich an, ein XML Format zum Abspeichern einer Szene zu verwenden. Diese XML Datei könnten Sie sogar außerhalb des Szeneeditors in einem gewöhnlichen Texteditor öffnen, und darin vorkommende Werte ändern. Das rundenbasierte Rollenspiel „Heroes of Might & Magic V“ [Niv05] speichert Spielkarten, die in dem hauseigenen Editor des Spiels erstellt wurden, in einem archivierten Format ab. Dabei besteht dieses Archiv aus mehreren Dateien z.B.: eine Spielerkarte aus der Community mit dem Namen Crepusculo.h5m [Woo08] ist im Grunde nichts anderes als eine archivierte und auf die Endung .h5m umbenannte Datei im .zip Format, die mehrere Dateien zur Beschreibung der Szene beinhaltet. Nach dem Öffnen der Zip-Datei finden wir folgende Struktur:

```
Crepusculo.h5m
        |-Effects
        |       |- …
        |-MapObjects
        |       |- …
        |-Maps
                |-Multiplayer
                        |-Crepusculo
                                |- map.xdb
                                |- map-tag.xdb
                                |- …
```

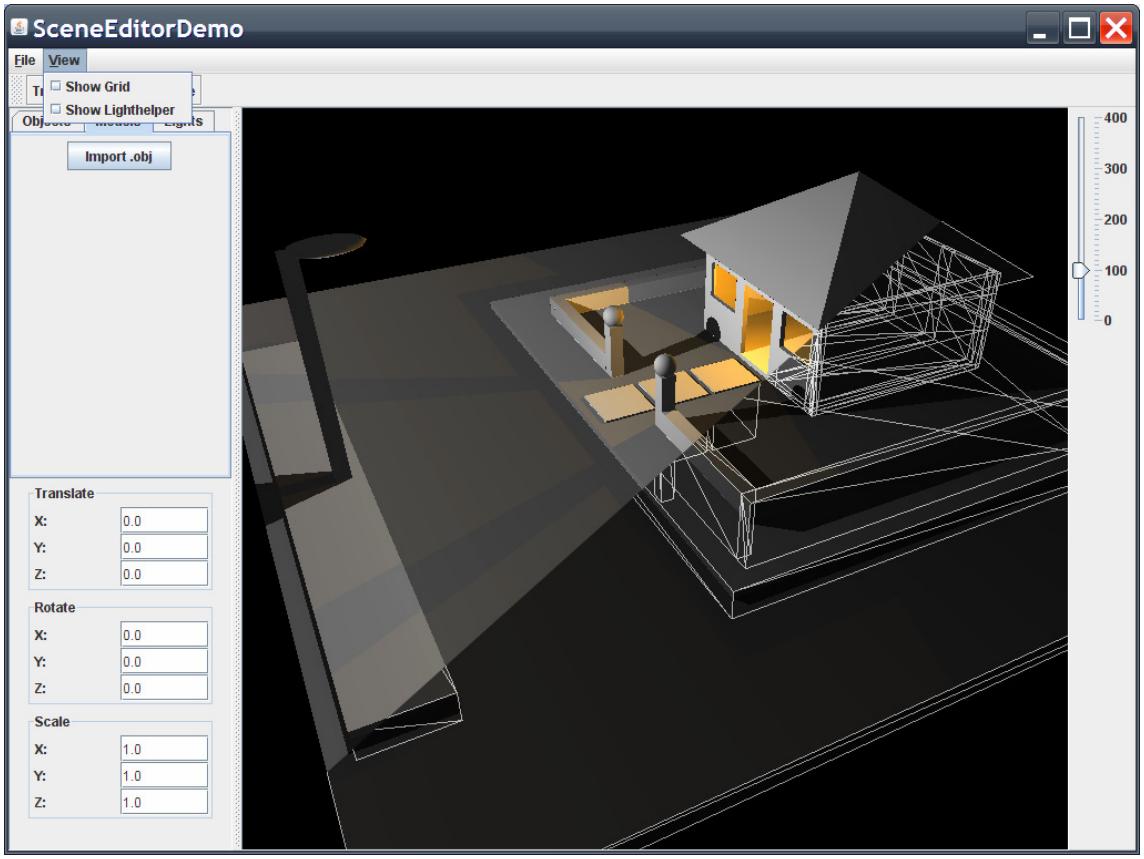
Wenn man jetzt die Datei map.xdb in einem gewöhnlichen Texteditor öffnet, sieht man, dass es sich um eine gewöhnliche XML Datei handelt.

```
<?xml version="1.0" encoding="UTF-8"?>
<AdvMapDesc>
        <CustomGameMap>true</CustomGameMap>
        <Version>3</Version>
        <TileX>176</TileX>
        <TileY>176</TileY>
        <HasUnderground>true</HasUnderground>
        <HasSurface>true</HasSurface>
        <InitialFloor>0</InitialFloor>
        <objects>
                <Item href="#n:inline(AdvMapStatic)" id="item_7B9CEE3B-2872-4F56-BDDB-
203C1E1E75FC">
                        <AdvMapStatic>
                                <Pos>
                                        <x>67</x>
                                        <y>115</y>
                                        <z>0</z>
                                </Pos>
                                <Rot>0</Rot>
                                <Floor>0</Floor>
                                <Name/>
```

```
                                    <CombatScript/>
                                    <pointLights/>
                                    <Shared
href="/MapObjects/Grass/Bridges/Bridges.(AdvMapStaticShared).xdb#xpointer(/AdvMapStaticS
hared)"/>
                                    <IsRemovable>false</IsRemovable>
                                    <TerrainAligned>false</TerrainAligned>
                                    <ScalePercent>100</ScalePercent>
                            </AdvMapStatic>
                    </Item>
                    …
```

In der vollständigen XML-Datei lassen sich viele Eigenschaften von Objekten manuell ändern. Es ist sogar möglich die Spielerteams komplett neu festzulegen. Damit diese Änderungen auch wirksam werden, muss man natürlich darauf achten, die veränderten Dateien wieder entsprechend in das Archiv einzupacken.

13 Extras



Sie haben gelernt, wie man mit Benutzereingaben implementiert, eine Szene zusammensetzt, Licht und Schatten verwendet, sowie vorhandene Szenen sichert. Die Implementierung des Editors im Kurs ist damit fertig. In diesem Kapitel lernen Sie ein paar nützliche Extras, die das Arbeiten im Editor erleichtern können.

Wie Sie anhand obigen Screenshots sehen können, ist auf der rechten Seite der Applikation eine neue Toolbar mit einem Slider hinzugekommen. Dieser Slider ist für das Verändern der Kamerageschwindigkeit zuständig und ermöglicht ihnen ein schnelleres oder langsameres Herumfliegen mit der Kamera in der Szene.

```
/**Klasse verwaltet den Slider fuer die Kameraschwindigkeit */
class SliderListener implements ChangeListener
{
	public void stateChanged(ChangeEvent ce)
	{
		JSlider source = (JSlider)ce.getSource();
		if (!source.getValueIsAdjusting())
		{
			int slidervalue = (int)source.getValue();
			keyboardLook.setActionSpeed(slidervalue);
		}
	}
```

```
}
```

Da wir jedes Mal, wenn wir in den Kameramodus übergehen, den FPH in einen keyboardLook und mouseLook zerlegen, können wir ihm dadurch auch die Geschwindigkeit mit der Funktion setActionSpeed() zuteilen.

Was uns sonst noch auffällt ist, dass wir die Szene nie so betrachten können, wie wenn sie in ein Spiel geladen werden würde. Grund dafür ist, dass uns der Grid, das Gizmo und die Farbkugeln stören. Wir entschließen uns daher eine Funktion einzubauen, die es ermöglicht diese abzuschalten, und bei Bedarf wieder anzuschalten.
In der Menüleiste oben erstellen wir also ein neues Menü mit 2 Unterpunkten:

1. ShowGrid: toggelt Grid und Gizmo
2. ShowLighthelper: toggelt Farbkugeln

Ab Zeile 296 folgt die Implementierung:

```
ActionListener aListener = new ActionListener()
{
	public void actionPerformed(ActionEvent ae)
	{
		if(ae.getActionCommand().equals("showgrid"))
		{
			AbstractButton showGrid = (AbstractButton)
ae.getSource();
			boolean selected = showGrid.getModel().isSelected();

			if(selected)
			{
				root.attachChild(helperNode);
			}
			else
			{
				root.detachChild(helperNode);
			}
			root.updateRenderState();// Update
		}
```

Die Vorgangsweise bedarf vermutlich keiner allzu großen Erklärung: wir hängen den gesamten helperNode je nach Option dem rootNode an oder entfernen ihn zur Gänze.

Nun benötigen wir nur noch eine Methode zum Ausschalten der Farbkugeln. Leider gibt es keine Methode, diese per Knopfdruck zu erledigen, und wir müssen an die uns bereits bekannte Methodik gehen diese von ihrem übergeordneten Knoten einzeln zu entfernen. In der Methode disableLightHelpers() gehen wir in einer Schleife alle Knoten mit dem Namen „lightObject“ durch (dies sind nämlich die übergeordneten Knoten der Farbkugel und des Lichts) und entfernen die Kugel.

```
// Auslesen aller Lichter
for(int i=0;i<lightNode.getQuantity();i++)
{
	// der uebergeordnete Knoten fuer Lichter heisst "lightObject"
	if(lightNode.getChild(i).getName().equals("lightObject"))
```

```
	{
		Node node = (Node) lightNode.getChild(i);
		node.detachChildAt(1); // entferne die Leuchtkugel
		}
	}
	root.updateGeometricState(0,false);
	root.updateRenderState();
```

Wie bereits beim Laden von Lichtern im vorigen Kapitel müssen wir auch hier darauf achten, an welcher Stelle sich die Farbkugel im Lichtobjekt befindet.

Das erneute Anfügen von Farbkugeln an die Lichter gestaltet sich dabei relativ umständlich in folgenden Schritten:

1. Zuerst alle Lichtobjekte mit dem Namen „lightObject“ in einer Schleife auslesen
2. Die Art des vorhandenen Lichttyps aus dem LightNode herausfiltern
3. Die Farbe des darin vorkommenden Lichts im LightNode herausfiltern
4. Eine neue Kugel erstellen mit derselben Größe und Farbe wie die früheren Farbkugeln
5. Diese Kugel an das entsprechende übergeordnete lightObject anhängen
6. Festlegen, dass das der LightNode immer der Position des Kugel folgt (wie einst beim erstmaligen Erstellen des Lichts)

```
// Auslesen aller Lichter
for(int i=0;i<lightNode.getQuantity();i++)
{
// Der uebergeordnete Knoten fuer Lichter heisst "lightObject"
if(lightNode.getChild(i).getName().equals("lightObject"))
{
	Node node = (Node) lightNode.getChild(i);
	LightNode ln = (LightNode) node.getChild(0);

	if(ln.getLight() instanceof PointLight)
	{
		PointLight pl = (PointLight) ln.getLight();
		// Hole mir die Farbe vom Licht
		ColorRGBA sphereColor = pl.getDiffuse();
		// Erstelle eine Sphere mit dem Namen "LIGHT"
			Sphere ls = new Sphere("LIGHT", 10, 10, 1.0f);
		ls.setModelBound(new BoundingBox());
		ls.updateModelBound();
		ls.setLightCombineMode(Spatial.LightCombineMode.Off);
		ls.setSolidColor(sphereColor);

		// Fuege die LightSphere dem entsprechenden Knoten hinzu
		node.attachChild(ls);

		// Setze die LightSphere an die Position vom PointLight
		Vector3f location = new Vector3f(pl.getLocation());
		ls.setLocalTranslation(location);

		// Lightnode muss immer an derselben Stelle wie Sphere
sein
		ln.setLocalTranslation(ls.getLocalTranslation());
	}
```

Weiters habe ich 3 Tasten für Funktionen zugewiesen, die jME von Haus aus unterstützt bzw. zum Debuggen verwendet werden können.

Eine davon ermöglicht es, die Objekte eines Knotens (hier für die relevante Szene) als Wireframe bzw. deren Unterteilung in Dreiecke anzuzeigen, eine andere macht die geometrischen Grenzen (Bounds) von Knoten sichtbar, die letzte ermöglicht es alle Lichter in der Szene auszuschalten. Jeder dieser Funktionen wird ab Zeile 547 ein ToggleButton zugewiesen, gleich wie beim Aktivieren von Schatten.

Lediglich der WireframeState benötigt eine längere Implementierung (Zeile 2304):

```
// Initialisiere WireframeState
wireState =
DisplaySystem.getDisplaySystem().getRenderer().createWireframeState();
wireState.setEnabled(false);// zu Beginn deaktivieren
```

Wir holen uns hier zuerst der WireFrame der Szene. In der simpleRender() Methode wird der entsprechende RenderState dann dem sceneNode zugewiesen, d.h. der Wireframe soll nur bei jenen Objekten angezeigt werden, die sich unter dem Knoten sceneNode (bzw. Unterknoten davon) befinden.

```
if(showTriangles)
{
	wireState.setEnabled(true);
	sceneNode.setRenderState(wireState);
	sceneNode.updateRenderState();
}
```

Wie Ihnen vielleicht aufgefallen ist, gibt es im Code ein paar Zeilen, die noch auskommentiert sind:

```
// SceneMonitor.getMonitor().registerNode(root, "Root Node");
// SceneMonitor.getMonitor().showViewer(true);
```

„SceneMonitor“ ist ein von Andrew K. Carter [Car08] für die Community zur Verfügung gestellte Erweiterung für jME Applikationen, die es erlaubt die Struktur des Scenegraph der gesamten Szene visuell darzustellen! Dieser eignet sich hervorragend zum Debuggen des Scenegraphs.

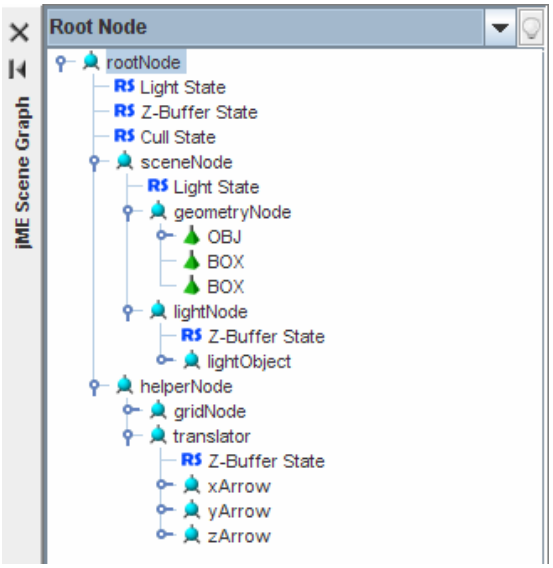


Abb. 4-25: Ausschnitt aus „SceneMonitor“ für eine Szene in unserem Editor

Ein Archiv mit den notwendigen JAR Dateien, das mir auf Anfrage hin persönlich von Andrew Carter erstellt und zugeschickt wurde, befindet sich im Ordner „Tools“ der CD.

14 Anwendung und Erweiterungsmöglichkeiten

In diesem Kapitel stellt sich die Frage der Anwendungsmöglichkeit dieses Editors, sowie um welche Komponenten dieser erweiterbar ist.

Sinn und Zweck eines Editors wurde bereits zu Beginn des Kurses besprochen: dadurch, dass man eine Szene flexibel verändern kann, spart man sich den Aufwand alles zu „hardcoden“. Um eine in einem Editor erstellte Szene in einem interaktiven Spiel zu verwenden, benötigt es noch folgende Dinge:

1. Im Hauptprogramm der Anwendung müssen Sie eine Scenegraphstruktur implementieren, die jener vom Editor (ab dem Knoten sceneNode) ähnelt!
2. Das Laden einer Karte muss nach einem ähnlichem System erfolgen wie das Laden einer Szene im Editor
3. Damit Sie eine Spielfigur auf der erstellten Szene bewegen können, müssen Sie noch eine Art Kollisionsabfrage implementieren

Mit der gewonnen Erfahrung beim Implementieren des Editors, sollten Ihnen zumindest die ersten zwei Punkte keine Probleme mehr verursachen. Der dritte Punkt wäre eine mögliche Erweiterung für den Editor. Wie bereits in mehreren Kapiteln angeschnitten wurde, bieten sich neben dieser noch viele andere Erweiterungsmöglichkeiten für einen umfassenden Szene-Editor an:

- Erweitern der Importmöglichkeiten mit allen von jME unterstützten 3D Formaten (nicht nur .obj)
- Import der 3D Modelle soll auch die zugehörigen Texturen integrieren können sowie im fortgeschrittenem Stadium die Implementierung von Bump- und Normalmaps zu berücksichtigen.
- Möglichkeit mit der Maus mehrere Objekte auszuwählen und zu verschieben
- Erweitern des Gizmo z.B. Verschieben von Objekten auf gleichzeitig 2 Ebenen z.B. XZ-Achse. Diese Funktion hat Parallelen zum Positionieren von Gebäuden in Aufbaustrategiespielen.
- Neue Funktionen wie „Rückgängig“, „Löschen“, „Duplizieren“ einbauen
- Spezialeffekte, die optional zuschaltbar und konfigurierbar sind (z.B. Bloom)
- Erstellen von Terrain von Grund auf sowie aus Grayscale Bildern (Heightmaps). Notwendig sind dabei auch folgende Punkte:
    - Malen (Terrain mit Texturen von Gras, Erde, Gesteinsuntergrund überziehen)

    - Formen (Geometrie des Terrains anheben, senken etc. z.B. auf einem zerklüfteten Gebirge ein kleines Plateau erstellen, auf die ein Gebäude platzieren werden soll)
    - In fortschrittlichen Stadien könnte man an die Implementierung von Deco-Layern sprich: „echtem“ Gras, Bäume, Büsche denken
- Skybox (Hintergrundkulissen für die Szene)
- Wasserbereiche (Flüsse, Meere)
- Partikel (z.B. Feuer, Rauch, glitzerndes Wasser etc.)
- Physik einsetzen
- Kollisionsabfrage