

2

Grundlagen

2.1	Projekte ohne das Starter Kit erstellen	30
2.2	Ein Mesh laden und zeichnen	34
2.3	BasicEffects	40
2.4	Textur	40
2.5	Material und Licht	44
2.6	Das GraphicsDevice	49
2.7	Eingabegeräte	56
2.8	Kamera	66
2.9	Grundlagen der 3D-Mathematik	80
2.10	Bewegte Objekte	103
2.11	Physik	111
2.12	Sound	116
2.13	Sprites	120
2.14	Textausgabe	126

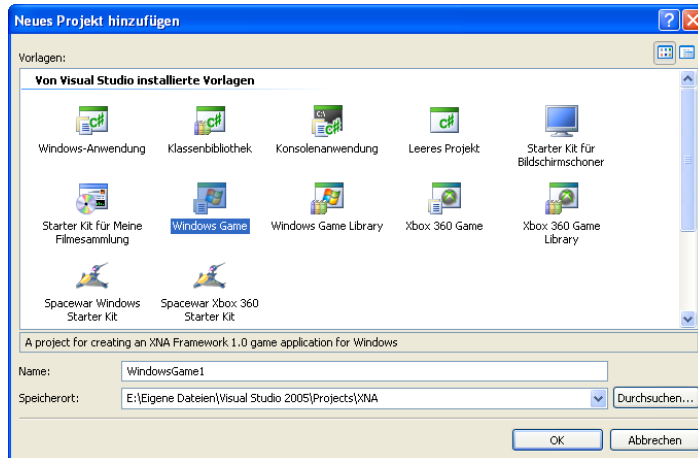
2.1 Projekte ohne das Starter Kit erstellen

Während das Starter Kit einen schönen Überblick über die Fähigkeiten von XNA gibt, ist es in seiner Komplexität doch weniger für den Einstieg geeignet. Für erste eigene Versuche ist der Projekttyp `WINDOWS GAME` die bessere Wahl.

Abbildung 2.1
Projekttyp
`WINDOWS GAME`

Beispiel

Das Beispiel zu diesem Abschnitt finden Sie auf der Begleit-CD im Ordner `KAP2\KAPO2_01`.



Der vom Assistenten erstellte Code

Die Klasse Program

Die Startklasse `Program` sieht immer gleich aus (und zwar so wie auch schon beim Starter Kit) und enthält lediglich das Hauptprogramm `Main`:

Listing 2.1
Die Klasse `Program`

```
static class Program
{
    /// <summary>
    /// The main entry point for the application.
    /// </summary>
    static void Main(string[] args)
    {
        using (Game1 game = new Game1())
        {
            game.Run();
        }
    }
}
```

Hier wird eine Instanz der zweiten vom Assistenten erstellten Klasse, `Game1`, erzeugt und deren `Run`-Methode aufgerufen. (Sie werden allerdings in `Game1` vergeblich nach dieser `Run`-Methode suchen: Sie ist von der Basisklasse `Game` geerbt.)

Die Klasse `Game1`

Das eigentliche Spiel wird durch die Klasse `Game1` dargestellt. Wenn Sie möchten, können Sie diese Klasse natürlich auch umbenennen, z.B. in `MeinCoolerEgoShooter` oder was auch immer. Wenn Sie das mit Hilfe der Funktion `UMGESTALTEN` von Visual C# Express durchführen, werden auch alle Bezüge auf diesen Namen, etwa in der Klasse `Program`, automatisch mit angepasst. (Klicken Sie im Quelltext-Editor mit der rechten Maustaste auf den Namen und wählen Sie aus dem Kontextmenü den Befehl `UMGESTALTEN|UMBENENNEN`.)

Der Assistent hat in die Klasse `Game1` bereits eine Reihe von leeren oder fast leeren Funktionsrümpfen eingefügt, die Sie »nur« noch ergänzen müssen, um Ihr Spiel zu implementieren.

GraphicsDevice und ContentManager

Die Klasse `Game1` enthält bereits zwei Membervariablen: einen `GraphicsDeviceManager` und ein `spriteBatch`-Objekt. Ersterer stellt unserem Spiel Grafikdienste zur Verfügung. Beispielsweise können Sie hierüber auf das `GraphicsDevice`-Objekt, das die `Game`-Klasse ja auch in einer Eigenschaft bereitstellt (und über dieses auf alles, was mit der Grafikkarte zu tun hat), zugreifen. Das `spriteBatch`-Objekt wird sich später beim Zeichnen von Bildern als hilfreich erweisen. Für unsere ersten Beispiele benötigen wir allerdings beide Objekte noch nicht.

```
GraphicsDeviceManager graphics;
SpriteBatch spriteBatch;
```

Im Konstruktor wird der `GraphicsDeviceManager` initialisiert:

```
public Game1()
{
    graphics = new GraphicsDeviceManager(this);
    Content.RootDirectory = "Content";
}
```

Listing 2.2

Der Konstruktor von `Game1`

Von der Basisklasse `Game` haben wir außerdem einen `ContentManager` geerbt. Dieser ist für das Laden von Content (Mediendateien) zustän-

dig. Im Konstruktor der `Game`-Klasse wird das Wurzelverzeichnis für das Auffinden solcher Mediendateien festgelegt.

Initialize

Die Methode `Initialize` ist zunächst leer bis auf den Aufruf der Basisklassen-Methode. Hier können Sie eigene Initialisierungen (mit Ausnahme des Ladens von Ressourcen) vornehmen.

LoadContent

Für das Laden von Ressourcen ist eine eigene Methode vorgesehen, `LoadContent`:

Listing 2.3
`LoadContent`

```
protected override void LoadContent()
{
    // Create a new SpriteBatch, which can be used to draw textures.
    spriteBatch = new SpriteBatch(GraphicsDevice);

    // TODO: use this.Content to load your game content here
}
```

Draw

Die `Draw`-Methode wird vom Framework mehrere Male in der Sekunde aufgerufen, um die aktuelle Szene zu *rendern* (darzustellen). Hier müssen Sie also Ihren eigenen Code zum Zeichnen der 3D-Modelle einfügen. Der vom Assistenten bereitgestellte Funktionsrumpf enthält lediglich das Löschen des Fensterinhalts mit der `Clear`-Methode des `GraphicsDevice`-Objekts. Probieren Sie einmal andere Farben, z.B. `Color.Black`.

Listing 2.4
`Draw`

```
protected override void Draw(GameTime gameTime)
{
    graphics.GraphicsDevice.Clear(Color.CornflowerBlue);

    // TODO: Add your drawing code here

    base.Draw(gameTime);
}
```

Näheres zum `GraphicsDevice` finden Sie in Abschnitt 2.6.

Update

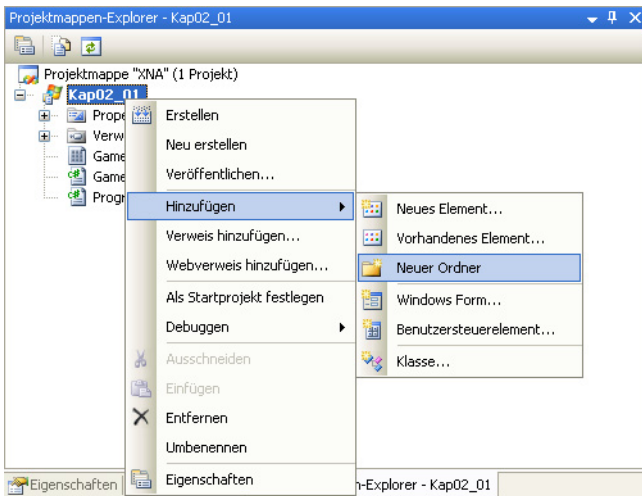
Update wird vom Framework jedes Mal *vor* dem Zeichnen aufgerufen: Hier wird Code zum Aktualisieren der Szene eingefügt (z.B. wenn sich Objekte bewegen sollen).

UnloadContent

Das Entladen von Ressourcen und die zugehörigen Aufräumarbeiten werden in der Regel automatisch durch den ContentManager erledigt. Wenn Sie bestimmte Ressourcen ausnahmsweise »von Hand« entladen wollen (oder müssen), sollten Sie dies in UnloadContent tun.

Content-Ordner

Um über die verschiedenen für das Spiel verwendeten Ressourcen (3D-Modelle, Texturen, Sounds usw.) den Überblick zu behalten, empfiehlt es sich, innerhalb des Projekts hierfür einen eigenen Ordner anzulegen. Klicken Sie dazu im Projektmappen-Explorer mit der rechten Maustaste auf das Projekt und wählen Sie aus dem Kontextmenü den Befehl HINZUFÜGEN|NEUER ORDNER.



Tipp

Im Allgemeinen ist es eine gute Idee, beim Testen als Hintergrundfarbe *nicht* Schwarz zu verwenden, da Sie bei einem schwarzen Hintergrund nicht feststellen können, ob Objekte gar nicht gerendert werden oder nur dunkel erscheinen, weil etwas mit der Beleuchtung der Szene nicht stimmt.

Abbildung 2.2

Anlegen des CONTENT-Ordners

Wir wollen dem Ordner wie im Starter Kit den Namen CONTENT geben.

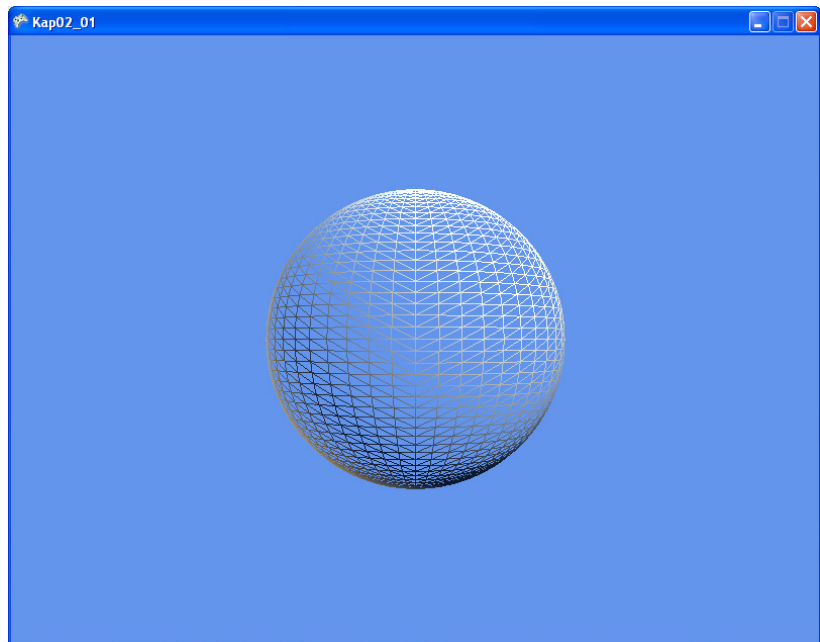
2.2 Ein Mesh laden und zeichnen

3D-Modelle (*Models*) werden bei XNA standardmäßig in *.x*-Dateien gespeichert. Dabei handelt es sich um ein DirectX-spezifisches Format, das die Daten wahlweise in Klartext oder in einem effizienteren Binärformat enthalten kann.

Letztendlich ist jedes Modell zusammengesetzt aus Polygonen (meist Dreiecken), deren Eckpunkte in der Modelldatei gespeichert sind. Man spricht daher auch von einem *Mesh* (Gitternetz).

Normalerweise werden in einem Spiel diese Dreiecke natürlich ausgefüllt und mit einem Material versehen gezeichnet; Sie können aber in einem Modellierungsprogramm oder auch beim Rendern in XNA die Modelle auch so zeichnen lassen, dass nur die Begrenzungslinien der Dreiecke dargestellt werden (*Drahtgittermodell*).

Abbildung 2.3
Drahtgittermodell



Neben dem Mesh selbst kann die Modelldatei weitere Informationen enthalten, z.B. Materialien, Texturen, hierarchische Beziehungen zwischen den Einzelteilen des Modells, Transformationen (Verschiebungen, Drehungen, Skalierungen), ein »Skelett« für Animationen usw. All dies wird im Verlauf des Buchs noch näher erläutert werden.

Um eine Vorstellung vom Inhalt einer Modelldatei zu bekommen, öffnen Sie einmal eine im Textformat abgespeicherte *.x*-Datei in einem

Texteditor (z.B. Notepad oder Wordpad). Die Datei BOX.X von der Begleit-CD beispielsweise enthält die Definition eines Würfels mit der Kantenlänge 1:

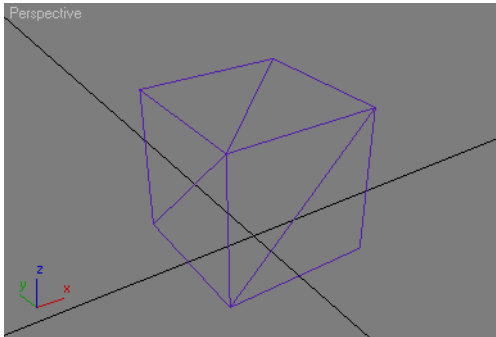


Abbildung 2.4
Der Würfel in 3DStudio
Max

Nach einigen Template-Definitionen, die die Datenstrukturen innerhalb der Datei beschreiben, finden Sie die Definition des Mesh-Objekts mit der Liste der Punkte (*Vertices*) und Indices:

```
Mesh Box01 {
  20;
  -0.500000;-0.500000;0.000000;,
  0.500000;-0.500000;0.000000;,
  -0.500000;0.500000;0.000000;,
  0.500000;0.500000;0.000000;,
  -0.500000;-0.500000;1.000000;,
  0.500000;-0.500000;1.000000;,
  -0.500000;0.500000;1.000000;,
  0.500000;0.500000;1.000000;,
  -0.500000;-0.500000;0.000000;,
  0.500000;-0.500000;0.000000;,
  0.500000;-0.500000;1.000000;,
  -0.500000;-0.500000;1.000000;,
  0.500000;0.500000;0.000000;,
  0.500000;-0.500000;1.000000;,
  0.500000;0.500000;0.000000;,
  -0.500000;0.500000;0.000000;,
  -0.500000;0.500000;1.000000;,
  0.500000;0.500000;1.000000;,
  -0.500000;0.500000;0.000000;,
  -0.500000;-0.500000;1.000000;;
  12;
  3;0,2,3;,
  3;3,1,0;,
  3;4,5,7;,
  3;7,6,4;,
  3;8,9,10;,
  3;10,11,8,;
```

Listing 2.5
Vertices und Indices

```
3;1,12,7;,  
3;7,13,1;,  
3;14,15,16;,  
3;16,17,14;,  
3;18,0,19;,  
3;19,6,18;;  
...  
}
```

Für jeden Punkt sind seine x-, y- und z-Koordinaten angegeben, die Indexliste beginnt in jeder Zeile mit der Zahl 3, was besagt, dass jede Fläche drei Vertices enthält (das Mesh besteht aus lauter Dreiecken). Zwar sind im Prinzip in der Modelldatei auch andere Polygone (z.B. Vierecke) möglich, diese werden aber zur Laufzeit dann doch wieder in Dreiecke aufgelöst; daher sind Dreiecke auch in Modelldateien der häufigste Polygontyp. Nach der 3 folgen für jedes Dreieck die Nummern (*Indices*) der dazugehörigen Punkte, wobei die Zählung bei Null beginnt. Das erste Dreieck besteht also aus den Punkten mit der Nummer 0 (-0.5;-0.5;0.0), 2 (-0.5;0.5;0.0) und 3 (0.5;0.5;0.0).

Hinter den Vertices und Indices kommen die Normalen, Materialien und Texturkoordinaten (hier aus Platzgründen nur schematisch abgedruckt):

Listing 2.6

Andere Informationen in der .x-Datei

```
MeshNormals {  
    ...  
}  
  
MeshMaterialList {  
    ...  
  
    Material {  
    }  
}  
  
MeshTextureCoords {  
    ...  
}
```

Die Bedeutung dieser Daten wird in den folgenden Kapiteln ausführlich diskutiert werden.

Es ist vielleicht schwer, sich vorzustellen, wie aus solch einer Liste von Zahlen ein dreidimensionales Objekt entsteht. Natürlich wird niemand ein komplexes Modell wie z.B. eine Spielerfigur mit einem Texteditor erstellen. Hierfür gibt es spezielle Programme (Modeler) wie

z.B. 3DStudio Max. Eine kurze Einführung in solche Modellierungsprogramme finden Sie im Anhang dieses Buchs.

In der professionellen Spieleprogrammierung existiert normalerweise eine Arbeitsteilung zwischen Künstlern, welche die Modelle erstellen, und Programmierern, die den Programmcode schreiben.

Wenn Sie sich selbst zunächst noch nicht zutrauen, eigene Modelle zu entwerfen, können Sie im Internet eine Vielzahl von fertigen, teils kostenlosen Modellen finden (vgl. Linkliste, ebenfalls im Anhang).

Content einfügen

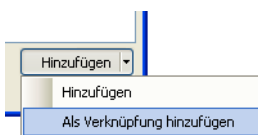
Bevor ein eigenes 3D-Modell in der Szene angezeigt werden kann, müssen Sie zunächst die .x-Datei zum Projekt hinzufügen. Klicken Sie also im Projektmappen-Explorer mit der rechten Maustaste auf einen der Unterordner des CONTENT-Ordners und wählen Sie aus dem Kontextmenü den Befehl HINZUFÜGEN|VORHANDENES ELEMENT. Um vorhandene Medien anzuzeigen, müssen Sie im Dateiauswahl-Dialogfeld unter DATEITYP den Eintrag CONTENT PIPELINE FILES auswählen.

Wählen Sie einmal das Modell EARTH.X aus dem Ordner CONTENT\MODELS von der Begleit-CD. Fügen Sie auch die dazugehörige Texturdatei EARTH.BMP in den CONTENT-Ordner ein.

Content als Verknüpfung einfügen

Beachten Sie den Unterschied zwischen dem Einfügen einer Ressource und dem Einfügen als Verknüpfung: Wenn die Datei sich ursprünglich im Dateisystem in einem anderen Verzeichnis befindet als dem CONTENT-Unterordner, in den sie eingefügt wird, so wird beim Einfügen eine Kopie der Datei in dem zum Projekt zugehörigen Verzeichnis angelegt. Wenn Sie die Datei dagegen als Verknüpfung einfügen, so erhält das Projekt nur einen Verweis auf die Originaldatei. Wird diese Originaldatei später verändert, so verwendet im ersten Fall das Projekt immer noch die unveränderte Kopie, im zweiten dagegen die veränderte Originaldatei.

Durch Anklicken des Pfeils rechts neben der Schaltfläche HINZUFÜGEN können Sie die Option ALS VERKNÜPFUNG HINZUFÜGEN auswählen.



Hinweis

Texturen, auf die in der .x-Datei verwiesen wird, müssen sich nur in dem entsprechenden Verzeichnis befinden (d.h. in demselben Verzeichnis wie die Modelldatei, falls diese nur einen Dateinamen enthält, oder, falls die Modelldatei eine komplette Pfadangabe enthält, an dem angegebenen Speicherort). Sie müssen aber nicht unbedingt ins Projekt aufgenommen werden.

Abbildung 2.5
Hinzufügen als
Verknüpfung

Der Programmcode

Öffnen Sie nun die Datei der Klasse `Game1` (im Folgenden auch einfach *Game*-Klasse genannt) im Editor. Wir werden nun eigenen Code in die verschiedenen vom Assistenten erstellten Funktionsrümpfe einfügen.

Membervariablen

Fügen Sie oben der Klasse eine Membervariable für das anzuzeigende Modell ein:

```
private Model earth;
```

LoadContent

Das Laden des Modells erfolgt in `LoadContent` mittels der `Load`-Methode des `ContentManager`-Objekts. `Load` ist eine *generische* Funktion – ein relativ neues Sprachmerkmal in C# (seit .Net 2.0). Generische Funktionen erlauben die Angabe eines Typparameters in spitzen Klammern – der Compiler kann also verschiedene Versionen dieser Funktion für verschiedene Datentypen generieren, hier z.B. für die Klasse `Model`.

Listing 2.7
LoadContent

```
protected override void LoadContent()  
{  
    earth = Content.Load<Model>("earth");  
}
```

Draw

Der interessanteste Teil dieses ersten Beispiels ist die Methode `Draw`, welche für das Zeichnen des Modells (mit der `Draw`-Methode der Klasse `ModelMesh`) verantwortlich ist. Ergänzen Sie diese wie folgt:

Listing 2.8
Draw

```
protected override void Draw(GameTime gameTime)  
{  
    graphics.GraphicsDevice.Clear(Color.Black);  
  
    // TODO: Add your drawing code here  
    float aspectRatio =  
        (float)GraphicsDevice.Viewport.Width /  
        (float)GraphicsDevice.Viewport.Height;  
    Matrix projection = Matrix.CreatePerspectiveFieldOfView(  
        MathHelper.ToRadians(45.0f), aspectRatio, 1.0f, 1000.0f);  
    Matrix view = Matrix.CreateLookAt(new Vector3(0, 0, 5), Vector3.Zero,  
        Vector3.Up);
```

```

ModelMesh mesh = earth.Meshes[0];
foreach (BasicEffect effect in mesh.Effects)
{
    effect.EnableDefaultLighting();

    effect.View = view;
    effect.Projection = projection;
    effect.World = Matrix.Identity;
}
mesh.Draw();

base.Draw(gameTime);
}

```

World-, View- und Projection-Matrizen

Zunächst werden eine Reihe von *Matrizen* initialisiert:

- projection
- view
- world

Diese werden an den Effekt übergeben und sind letztlich dafür verantwortlich, dass die in der Modelldatei gespeicherten dreidimensionalen Koordinaten der Punkte, aus denen das Modell besteht, in zweidimensionale Bildpunkte (Pixel) auf dem Bildschirm umgerechnet werden.

- Die *World-Matrix* gibt die Position und Orientierung des zu zeichnenden Objekts im Raum an. In obigem Beispiel ist die World-Matrix die *Identitätsmatrix* (`Matrix.Identity`), das heißt, das Objekt liegt im Koordinatenursprung und ist weder verschoben noch rotiert, noch skaliert (vergrößert/verkleinert).
- Die *View-Matrix* kann man sich als eine Zusammenfassung aller Informationen, die mit der Kamera zusammenhängen, vorstellen: Position dieser Kamera, auf welchen Punkt ist die Kamera gerichtet und wie ist die Kamera selbst im Raum orientiert, das heißt, wo ist »oben«.
- Die *Projection-Matrix* schließlich regelt die Projektion auf den Bildschirm (Seitenverhältnis der Projektionsfläche, Abschneiden bei sehr kleinen und sehr großen Entfernungen).

Nähers dazu finden Sie in den Abschnitten 2.8 und 2.9.

Das Modell (Mesh)

Auf das Mesh greifen wir zu über `earth.Meshes[0]`. Das Modell (hier in der Variablen `earth`) kann mehrere Meshes enthalten, welche fort-

laufend durchnummeriert sind und als Elemente der Meshes-Auflistung angesprochen werden können. `earth.Meshes[0]` ist also das erste Mesh im Modell `earth`. Dieses kann wiederum aus mehreren Teilen (*Parts*) mit unterschiedlichen Materialien bestehen. Jedem dieser Teile ist ein *Effekt* zugewiesen, welcher die Darstellung dieses Teils kontrolliert. Nachdem für alle diese Effekte (in der `foreach`-Schleife) die Standard-Beleuchtung eingeschaltet ist und die oben erwähnten Matrizen gesetzt sind, wird mit `mesh.Draw` das Mesh gezeichnet und zuletzt noch die `Draw`-Methode der Basisklasse `Game` aufgerufen.

Versuchen Sie einmal, auch andere `.x`-Dateien zu laden und anzuzeigen.

Hinweis

Sie können sehr schön sehen, wie das Mesh aus einzelnen Dreiecken aufgebaut ist, wenn Sie einmal oben in der Methode `Draw` die folgende Zeile einfügen:

```
GraphicsDevice.  
RenderState.Fill-  
Mode = FillMode.  
Wireframe;  
(Denken Sie aber  
daran, die Zeile für  
die folgenden Bei-  
spiele wieder zu ent-  
fernen.)
```

2.3 BasicEffects

Während in vielen älteren DirectX-Spielen noch die so genannte *Fixed-Function Pipeline* zum Transformieren der einzelnen Punkte (*Vertices*) eines Modells mit ihren Koordinaten, Texturinformationen, Normalen etc. in das fertige Bild verwendet wurde, arbeitet XNA komplett und ausschließlich auf der Basis der *Programmable Pipeline*: Dabei kann der Programmierer selbst mit Hilfe von *Shadern* festlegen, was genau mit jedem Punkt zu geschehen hat, um das resultierende Bildschirmpixel zu berechnen. Shader sind kleine Programme, die in einer eigenen Programmiersprache geschrieben und vom Prozessor der Grafikkarte (der **Graphics Processing Unit**, *GPU*) ausgeführt werden. Vom C#-Programm aus werden diese Shader gesteuert und mit Parametern versorgt. DirectX und XNA verwenden dafür spezielle Klassen, die *Effekte*.

Damit Sie aber nun nicht mit dem Schwierigsten beginnen und für Ihre ersten Programme bereits Shader schreiben müssen, versorgt XNA jedes Modell (bzw. jeden Teil eines Modells) erst einmal mit einem Standard-Effekt. Diese sind Instanzen der Klasse `BasicEffect` und erlauben einen einfachen Zugriff auf Textur, Material und Beleuchtung.

2.4 Textur

Eine *Textur* ist ein Bild, das wie eine Tapete auf ein 3D-Modell »geklebt« wird. Wenn in der Modelldatei eines Objekts eine Textur angegeben ist, wird diese automatisch geladen und angezeigt, wie wir bereits gesehen haben. XNA Game Studio zeigt sogar eine Fehlermeldung an, wenn eine zu einem Modell zugehörige Textur nicht an dem in

der Datei ausgewiesenen Speicherort (meist dasselbe Verzeichnis, in dem sich auch das Modell befindet) vorhanden ist. So ist gewährleistet, dass immer alle Ressourcen vollständig sind, wenn ein Spiel erstellt und ausgeführt wird.

Sie können aber auch (wiederum mit der `Load`-Methode des Content-Manager-Objekts) Texturen explizit laden und einem Mesh bzw. einem Teil davon zur Laufzeit zuweisen:

Fügen Sie oben in die Klasse eine Membervariable für die Textur ein:

```
private Texture2D tex;
```

Laden Sie die Textur in `LoadContent`:

```
protected override void LoadContent()
{
    earth = Content.Load<Model>("earth");
    tex = Content.Load<Texture2D>("earthtex1");
}
```

Listing 2.9
LoadContent

Vor dem Zeichnen muss nun die Textur dem Effekt zugewiesen werden:

```
protected override void Draw(GameTime gameTime)
{
    graphics.GraphicsDevice.Clear(Color.Blue );

    // TODO: Add your drawing code here

    float aspectRatio =
        (float)GraphicsDevice.Viewport.Width /
        (float)GraphicsDevice.Viewport.Height;
    Matrix projection = Matrix.CreatePerspectiveFieldOfView(
        MathHelper.ToRadians(45.Of), aspectRatio, 1.Of, 1000.Of);
    Matrix view = Matrix.CreateLookAt(new Vector3(0, 0, 3),
        Vector3.Zero, Vector3.Up);

    ModelMesh mesh = earth.Meshes[0];
    foreach (BasicEffect effect in mesh.Effects)
    {
        effect.EnableDefaultLighting();

        effect.Texture = tex;

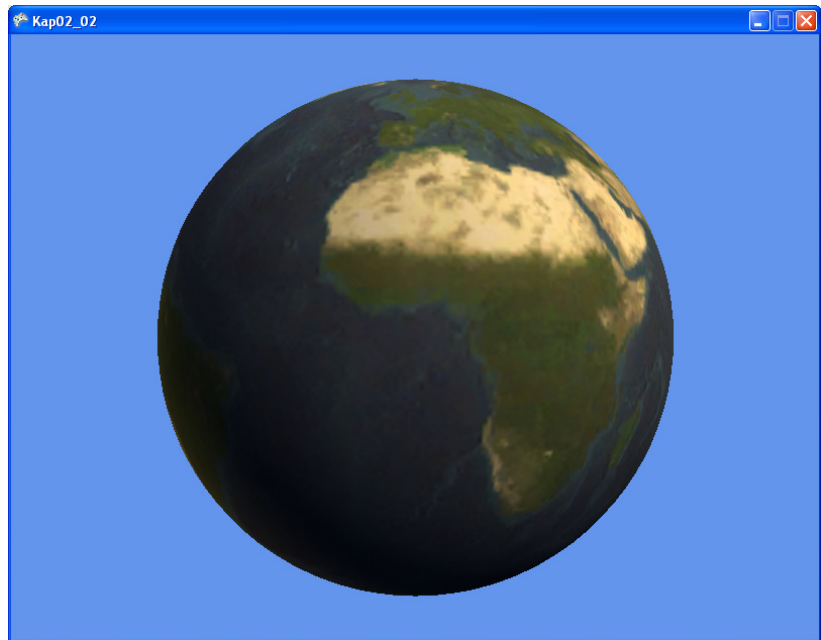
        effect.View = view;
        effect.Projection = projection;
        effect.World = Matrix.Identity;
    }
}
```

Listing 2.10
Zeichnen mit Textur

```
}  
mesh.Draw();  
  
base.Draw(gameTime);  
}
```

Grundsätzlich können Sie alle Arten von Bilddateien in den gängigen Formaten wie BMP, JPG, TGA, DDS etc. als Texturen verwenden. Experimentieren Sie ruhig einmal mit verschiedenen Bildern.

Abbildung 2.6
Modell mit Textur



Um ein Modell ganz ohne Textur anzuzeigen, setzen Sie die Eigenschaft `TextureEnabled` auf `false`:

```
effect.TextureEnabled = false;
```

Texturkoordinaten

Wenn Sie versuchen, verschiedene Texturen auf unsere Erde oder auf andere Modelle zu kleben, werden Sie schnell feststellen, dass nicht jedes Bild ohne Weiteres auf jedes Modell »passt«. Woher weiß XNA, welcher Teil der Textur auf dem Modell wohin gehört? Diese Information muss in der Modelldatei enthalten sein. Modellierungsprogramme wie 3DStudio Max verfügen über ein Feature, das *Texture*

Mapping oder auch *UV-* oder *UVW-Mapping* genannt wird. Dabei wird jedem Punkt (Vertex) im Modell ein Satz Koordinaten zugeordnet (die *Texturkoordinaten*), die angeben, welcher Punkt der Textur diesem Punkt des Modells zugeordnet ist. Normalerweise sind die Koordinaten zweidimensional (da ein Bild ja flach ist) und werden oft nicht wie bei Koordinaten für Punkte im Raum mit x und y , sondern mit u und v bezeichnet (daher der Name UV-Mapping). Es gibt aber auch dreidimensionale Texturkoordinaten, z.B. für Cubemapping, die dann u , v und w heißen.

Die Werte gehen normalerweise von 0 bis 1 für u , v und gegebenenfalls w , wobei ($u = 0$, $v = 0$) der linken oberen und ($u = 1$, $v = 1$) der rechten unteren Ecke der Textur entspricht.

Im Prinzip kann man Texturkoordinaten zwar auch per Programmcode zuweisen, aber nur für einfache Modelle lässt sich dafür ein programmierbarer Algorithmus finden. Bei komplexeren Modellen ist in der Regel »Handarbeit« im Modeler angesagt, was, nebenher bemerkt, auch nicht ganz einfach ist. Auch die Textur muss dann natürlich speziell auf das Modell mit seinen Texturkoordinaten abgestimmt sein. (Öffnen Sie z.B. die Texturen für die Erde einmal mit einem Bildbearbeitungsprogramm.)

Im Moment müssen wir uns einfach damit zufriedengeben, dass ein Modell nur dann vernünftig mit einer Textur angezeigt werden kann, wenn es bereits im Modeler mit Texturkoordinaten versehen wurde.

Vorhandene Modelle können aber leicht verändert werden, indem man die Textur in einem Bildbearbeitungsprogramm »übermalt« – man nennt das auch *Retexturieren* (retexture).

Mipmapping

Wenn eine Textur niedriger Auflösung aus sehr kleiner Entfernung betrachtet wird, wirkt das Objekt »pixelig«: Die Textur wird so stark vergrößert, dass die einzelnen Bildpunkte erkennbar sind. Einerseits hilft dagegen eine höhere Texturauflösung. Andererseits verbrauchen natürlich höher aufgelöste Texturen mehr Speicherplatz und können die Ausführungsgeschwindigkeit stark beeinträchtigen. Eine Lösung dieses Problems besteht darin, dass man die Textur in unterschiedlichen Auflösungen speichert, die dann vom Programm nach Bedarf geladen werden. Diese Technik bezeichnet man als *Mipmapping*.

2.5 Material und Licht

Beispiel

Das Beispiel zu diesem Abschnitt finden Sie auf der Begleit-CD im Ordner `KAP2\KAPO2_02`.

Die Farbe eines Objekts in der Szene (ohne Textur) setzt sich zusammen aus der Farbe des Materials und derjenigen des Lichts. Beide sind selbst wiederum zusammengesetzt aus mehreren Komponenten.

Materialfarben

- Die *Ambient*-Farbe eines Materials bestimmt die Reflexion von *Umgebungslicht*. Umgebungslicht ist unabhängig von den in der Szene vorhandenen Lichtquellen und scheint aus allen Richtungen zu kommen. Die Ausleuchtung eines Objekts ist daher unabhängig von seiner Position und Orientierung.
- Bei matten Oberflächen, die Licht *diffus* reflektieren, ist die Helligkeit einer Fläche nur abhängig von dem Winkel zwischen der Flächennormale (Vektor, der senkrecht auf der Fläche steht) und der Richtung zur Lichtquelle. Die *Diffuse*-Farbe (*Streulichtfarbe*) des Materials kontrolliert diese diffuse Reflexion.

Die *Diffuse*- und *Ambient*-Farben eines Materials sind häufig identisch. Da die meisten Szenen jedoch weitaus mehr Streulicht als Umgebungslicht enthalten, bestimmt die *Diffuse*-Farbe in der Regel das Aussehen des Objekts.

- *Glanzlichter* entstehen auf glänzenden Oberflächen: Beleuchtet man z.B. eine Kugel aus glänzendem Material, so entsteht an einer Stelle ein Fleck in der Farbe des einfallenden Lichts, unabhängig von der Farbe des restlichen Objekts. Die Intensität dieses hellen Flecks hängt nicht nur von der Einfallsrichtung des Lichts, sondern auch von der Richtung zum Auge des Betrachters ab. Die Farbe der Glanzlichter wird durch die *Specular*-Farbe des Materials bestimmt.
- Die *Emissive*-Farbe eines Materials bestimmt die Farbe eines scheinbar selbstleuchtenden Objekts.

Die in den bisherigen Beispielen vorhandene Zeile

```
effect.EnableDefaultLighting ();
```

bewirkte, dass unser Modell mit einem Satz von Standard-Lichtquellen beleuchtet wurde. Um die Wechselwirkung von Licht und Material näher zu untersuchen, entfernen Sie einmal diese Zeile.

Die Materialfarben des `BasicEffect`-Objekts werden als `Vector3` (also als Vektor mit 3 Komponenten) angegeben, wobei die erste Komponente den Rot-Anteil, die zweite den Grün- und die dritte den Blau-Anteil der Farbe bestimmt (jeweils zwischen 0 und 1). Ein Wert von (1, 1, 1) ergibt Weiß; (0, 0, 0) Schwarz.

Die Ambient-Farbe des Materials kann beim `BasicEffect` nicht angegeben werden (wohl aber die Farbe des Umgebungslichts, s.u.). Setzen Sie einmal die `DiffuseColor` auf Rot:

```
effect.DiffuseColor = new Vector3(1, 0, 0);
```

Ohne `EnableDefaultLighting()` ist das Objekt gleichmäßig rot, mit `EnableDefaultLighting()` ist die dem Licht abgewandte Seite dunkel. Sie sehen diesen Effekt noch deutlicher, wenn Sie einmal eine oder mehrere der Standard-Lichtquellen deaktivieren:

```
effect.DirectionalLight0.Enabled = false;
```

Der `BasicEffect` verfügt über drei separate Lichtquellen, die das Modell aus verschiedenen Richtungen beleuchten (diese Richtungen können Sie übrigens auch selbst festlegen). Wenn Sie alle drei Lichtquellen abschalten, bleibt das Modell schwarz.

Schalten Sie jetzt allerdings etwas Umgebungslicht ein, indem Sie die `AmbientLightColor` auf Weiß setzen, so wird es wieder hell:

```
effect.DirectionalLight0.Enabled = false;
effect.DirectionalLight1.Enabled = false;
effect.DirectionalLight2.Enabled = false;

effect.AmbientLightColor = new Vector3(1, 1, 1);
```

Wählen Sie einen kleineren Wert, um die Szene dunkler aussehen zu lassen:

```
effect.AmbientLightColor = new Vector3(0.3f, 0.3f, 0.3f);
```

Wenn Sie die `DiffuseColor` des Materials auf Weiß setzen, ergibt sich die Farbe des Objekts ausschließlich aus der Farbe des Lichts (hier z.B. Blau):

```
effect.AmbientLightColor = new Vector3(0, 0, 1);
effect.DiffuseColor = new Vector3(1, 1, 1);
```

Im allgemeinen Fall ergibt sich die Ambient-Komponente der Objektfarbe aus der Materialfarbe in Zusammenarbeit mit dem globalen Umgebungslicht (global ambient) sowie gegebenenfalls den Ambient-Anteilen aller in der Szene vorhandenen Lichtquellen.

Für jede Lichtquelle können theoretisch wie bei Materialien die Farben für Streulicht (Diffuse), für ungerichtetes Umgebungslicht (Ambient) und für Glanzlichter (Specular) separat angegeben werden. Hierfür müssen Sie aber eigene Shader schreiben. Der BasicEffect unterstützt nur DiffuseColor, SpecularColor und EmissiveColor für das Material und sowie DiffuseColor und SpecularColor für die drei Lichtquellen sowie AmbientLightColor für das globale Umgebungslicht.

Deaktivieren Sie nun einmal alle Lichtquellen und geben Sie dem Material eine EmissiveColor (hier Grün):

```
effect.EmissiveColor = new Vector3(0, 1, 0);
```

Wie erwartet ist das Objekt jetzt wieder farbig, da es selbst leuchtet.

Die Farbe von Glanzlichtern ist normalerweise heller als diejenige des Materials. Setzen Sie beispielsweise die DiffuseColor wieder auf Rot und die SpecularColor auf Weiß und beleuchten Sie das Modell mit DirectionalLight0 von vorn:

```
effect.DirectionalLight1.Enabled = false;
effect.DirectionalLight2.Enabled = false;

effect.DiffuseColor = new Vector3(1, 0, 0);
effect.SpecularColor = new Vector3(1, 1, 1);
```

Lassen Sie jetzt einmal das Licht aus einer anderen Richtung kommen (beispielsweise von rechts):

```
effect.DirectionalLight0.Direction = new Vector3(1, 0, 0);
```

Lichttypen

In Spielen mit 3D-Grafik werden häufig unterschiedliche Arten von Lichtquellen eingesetzt:

- Point (Punktlicht)
- Spot (Scheinwerfer)
- Directional (Richtungslicht)

Die Lichtquellen des `BasicEffect` sind Richtungslichter, das heißt, das Licht scheint nur aus einer Richtung zu kommen (alle Strahlen sind parallel), wie bei einer sehr weit entfernten Lichtquelle, z.B. der Sonne.

Andere Arten von Lichtquellen können wir mit eigenen Shadern realisieren.

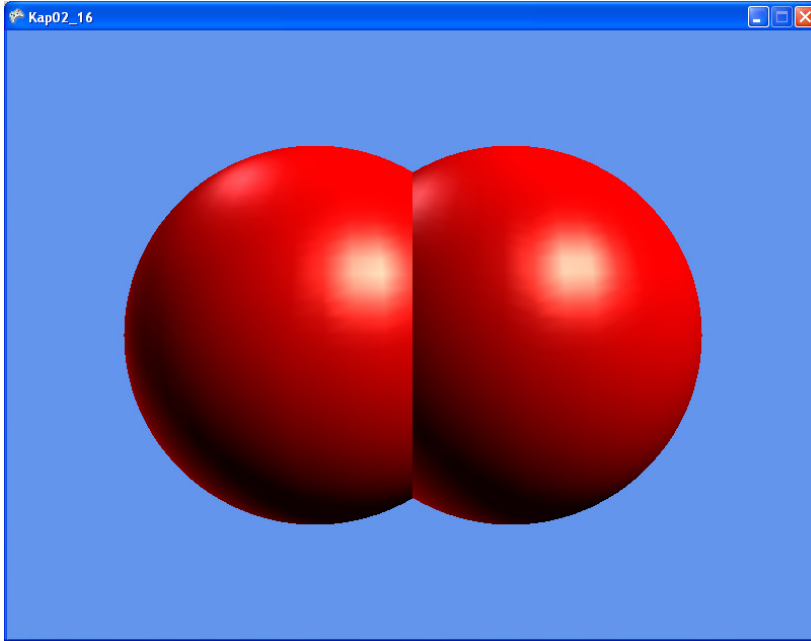


Abbildung 2.7
Glanzlicht

Beispiel: Rotierende Lichtquelle

Man kann die Wirkung von Beleuchtungseffekten sehr schön beobachten, wenn man die Lichtquelle um das beleuchtete Objekt herum rotieren lässt.

Membervariablen

Fügen Sie oben in die Game-Klasse Membervariablen für ein Modell sowie für die jeweils aktuelle Lichtposition ein:

```
private Model earth;  
private Vector3 lightPos;
```

LoadContent

In `LoadContent` wird das Modell (das Sie natürlich wieder in den Content-Ordner des Projekts einfügen müssen) geladen:

Listing 2.11
LoadContent

```
protected override void LoadContent()
{
    earth = Content.Load<Model>("earth");
}
```

Update

In der Update-Methode berechnen wir die Position der Lichtquelle anhand der bisher verstrichenen Zeit (`gameTime`). Die Winkelgeschwindigkeit ω (ω) ist in diesem Beispiel $2\pi/5$, also eine volle Umdrehung in 5 Sekunden. Die hier implementierte Berechnungsvorschrift entspricht gerade einer Kreisbahn (mit Radius 1) in der x/z-Ebene:

$$x = \sin(\omega t)$$

$$z = \cos(\omega t)$$

Man könnte dies auch sehr schön mit Hilfe von Matrizen realisieren (vgl. Abschnitt 2.9).

Listing 2.12
Update

```
protected override void Update(GameTime gameTime)
{
    // Allows the default game to exit on Xbox 360 and Windows
    if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
        this.Exit();

    // TODO: Add your update logic here
    double t = gameTime.TotalGameTime.TotalSeconds;
    double omega = 2 * Math.PI / 5.0f;
    lightPos = new Vector3((float)(Math.Sin(omega * t)), 0,
        (float)(Math.Cos(omega * t)));

    base.Update(gameTime);
}
```

Draw

Die Draw-Methode bringt nichts wesentlich Neues: Die View- und Projection-Matrizen werden gesetzt (die World-Matrix ist wieder einfach die Identitätsmatrix), die Effekt-Eigenschaften werden mit Werten versorgt und das Modell wird gezeichnet. Die Richtung des ersten Richtungslichts wird entsprechend der in Update berechneten Position gesetzt (die Position selbst spielt ja für ein Richtungslicht keine Rolle), und zwar so, dass der Richtungsvektor von der Lichtquelle zum Koordinatenmittelpunkt, also zum Objekt, zeigt. (Wenn `lightPos` der Vektor vom Ursprung zur Lichtquelle ist, ist `-lightPos` gerade der entge-

gegenseitige Vektoren, zeigt also von der Lichtquelle zum Ursprung, vgl. auch Abschnitt 2.9.)

Die anderen Lichtquellen werden abgeschaltet.

```
protected override void Draw(GameTime gameTime)
{
    graphics.GraphicsDevice.Clear(Color.CornflowerBlue);

    float aspectRatio =
        (float)GraphicsDevice.Viewport.Width /
        (float)GraphicsDevice.Viewport.Height;
    Matrix projection = Matrix.CreatePerspectiveFieldOfView(
        MathHelper.ToRadians(45.0f), aspectRatio, 1.0f, 1000.0f);
    Matrix view = Matrix.CreateLookAt(new Vector3(0, 0, 3), Vector3.Zero,
        Vector3.Up);

    ModelMesh mesh = earth.Meshes[0];
    foreach (BasicEffect effect in mesh.Effects)
    {
        effect.EnableDefaultLighting();
        effect.DirectionalLight0.Direction = -lightPos;
        effect.DirectionalLight1.Enabled = false;
        effect.DirectionalLight2.Enabled = false;

        effect.DiffuseColor = new Vector3(1, 0, 0);
        effect.SpecularColor = new Vector3(1, 1, 1);

        effect.View = view;
        effect.Projection = projection;
        effect.World = Matrix.Identity;

        effect.TextureEnabled = false;
    }
    mesh.Draw();

    base.Draw(gameTime);
}
```

Listing 2.13

Draw

2.6 Das GraphicsDevice

In den Draw-Methoden der oben vorgestellten Beispiele wurde ausgiebig Gebrauch gemacht von einem Objekt namens GraphicsDevice. Mit »Device« ist hier einfach ein Gerät gemeint – das GraphicsDevice ist also quasi der Repräsentant Ihrer Grafikkarte im Programmcode.

Beispiel

Das Beispiel zu diesem Abschnitt finden Sie auf der Begleit-CD im Ordner KAP2\KAP02_03.

Viewport

Ein *Viewport* ist der Teil des Bildschirms, auf den gezeichnet wird. Die gleichnamige Eigenschaft des `GraphicsDevice`-Objekts enthält den aktuellen Viewport, und dessen `Width`- und `Height`-Eigenschaften geben folglich die Bildschirmabmessungen an – bzw. desjenigen Teils des Bildschirms, in den zur Zeit gezeichnet wird. Das Arbeiten mit mehreren Viewports für einen geteilten Bildschirm in Multi-Player-Spielen wird in einem späteren Kapitel besprochen.

RenderStates

Die Eigenschaft `RenderState` des `GraphicsDevice` beschreibt den Zustand des Geräts. Den `RenderState FillMode` haben wir bereits im vorangegangenen Abschnitt kennengelernt. Eine Reihe weiterer `RenderStates` kontrollieren Einzelheiten des Zeichenvorgangs. Die meisten davon werden Ihnen im Verlauf des Buchs noch begegnen – vorab sollen hier nur die `RenderStates DepthBufferEnable` und `DepthBufferWriteEnable` vorgestellt werden, die beide mit dem so genannten *Z-Buffering* zu tun haben, sowie die `RenderStates AlphaBlendEnable`, `SourceBlend` und `DestinationBlend`, die wir zum Zeichnen transparenter Objekte benötigen.

Z-Buffering

Im Gegensatz zu 2D-Grafik erstrecken sich dreidimensionale Objekte nicht nur in die Höhe und Breite (y und x), sondern auch in die dritte Raumrichtung, die Tiefe (z). Weiter hinten liegende Objekte werden dabei von den davorliegenden verdeckt. Um dies beim Zeichnen zu berücksichtigen, arbeitet man üblicherweise mit einem *Z-Buffer* (oder *Depth Buffer*). Beim Zeichnen eines 3D-Objekts wird für jeden gezeichneten Punkt die dazugehörige z -Koordinate im `Depth Buffer` abgespeichert. Wenn nun beim Rendern eines zweiten Objekts an denselben Koordinaten wieder ein Punkt zu zeichnen ist, so wird dessen z -Koordinate mit derjenigen im `Z-Buffer` verglichen, und nur wenn der neue Punkt weiter vorn liegt als der bereits vorhandene, wird dieser überschrieben.

Die Eigenschaft `DepthBufferEnable` des `GraphicsDevice`-Objekts legt fest, ob das Schreiben in den `Z-Buffer` erfolgt (Voreinstellung `true`) und die Eigenschaft `DepthBufferWriteEnable` gibt an, ob beim Zeichnen vorhandene Daten im `Z-Buffer` berücksichtigt werden (ebenfalls stan-

dardmäßig true). In bestimmten Situationen (z.B. beim Zeichnen durchsichtiger Objekte) schaltet man das Depth-Buffering ab – was zur Folge hat, dass später gezeichnete Objekte die vorher gezeichneten verdecken, unabhängig von der z-Koordinate.

Sie können die Auswirkung des Z-Buffering leicht testen, wenn Sie in unserem Beispiel die Kugel einfach zweimal zeichnen, und zwar so, dass sich die Modelle überlappen. Ändern Sie dazu die Draw-Methode wie folgt:

```
protected override void Draw(GameTime gameTime)
{
    graphics.GraphicsDevice.Clear(Color.CornflowerBlue);

    float aspectRatio =
        (float)GraphicsDevice.Viewport.Width /
        (float)GraphicsDevice.Viewport.Height;
    Matrix projection = Matrix.CreatePerspectiveFieldOfView(
        MathHelper.ToRadians(45.Of), aspectRatio, 1.Of, 1000.Of);
    Matrix view = Matrix.CreateLookAt(new Vector3(0, 0, 3), Vector3.Zero,
        Vector3.Up);

    Matrix world1 = Matrix.CreateTranslation(new Vector3(-0.5f, 0, -1));
    Matrix world2 = Matrix.CreateTranslation(new Vector3( 0.5f, 0, -1));
    DrawModel(earth, ref world1, ref view, ref projection);
    DrawModel(earth, ref world2, ref view, ref projection);

    base.Draw(gameTime);
}
```

Listing 2.14

Draw

Um den Code etwas übersichtlicher zu halten, haben wir für das Zeichnen des Modells, das ja jetzt zweimal mit verschiedenen World-Matrizen ausgeführt werden muss, eine Unterfunktion angelegt:

```
private void DrawModel(Model model,
    ref Matrix world, ref Matrix view, ref Matrix projection)
{
    ModelMesh mesh = model.Meshes[0];
    foreach (BasicEffect effect in mesh.Effects)
    {
        effect.EnableDefaultLighting();

        effect.DiffuseColor = new Vector3(1, 0, 0);
        effect.SpecularColor = new Vector3(1, 1, 1);

        effect.View = view;
        effect.Projection = projection;
    }
}
```

Listing 2.15

DrawModel

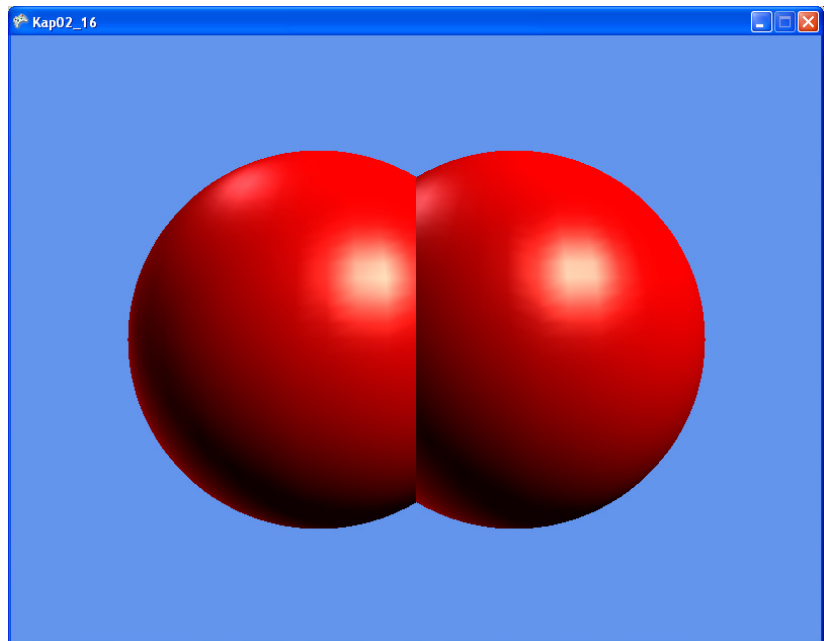
```
    effect.World = world;

    effect.TextureEnabled = false;
}
mesh.Draw();
}
```

Die Matrizen (die keine Objekte, sondern Strukturen sind) werden aus Performancegründen per Referenz an die Funktion übergeben; bei dem Modell ist das nicht nötig, da Objektvariablen ja ohnehin selbst immer Referenzen sind.

Mit Z-Buffering (also im Normalfall) sieht man von beiden Kugeln jeweils diejenigen Teile, die am weitesten vorn sind.

Abbildung 2.8
Überlappende Kugeln
mit Z-Buffering



Deaktivieren Sie nun das Z-Buffering, indem Sie oben in die Draw-Methode die folgende Zeile einfügen:

```
GraphicsDevice.RenderState.DepthBufferEnable = false;
```

Die zweite Kugel überdeckt jetzt die erste – auch diejenigen Teile, die eigentlich weiter vorn liegen.

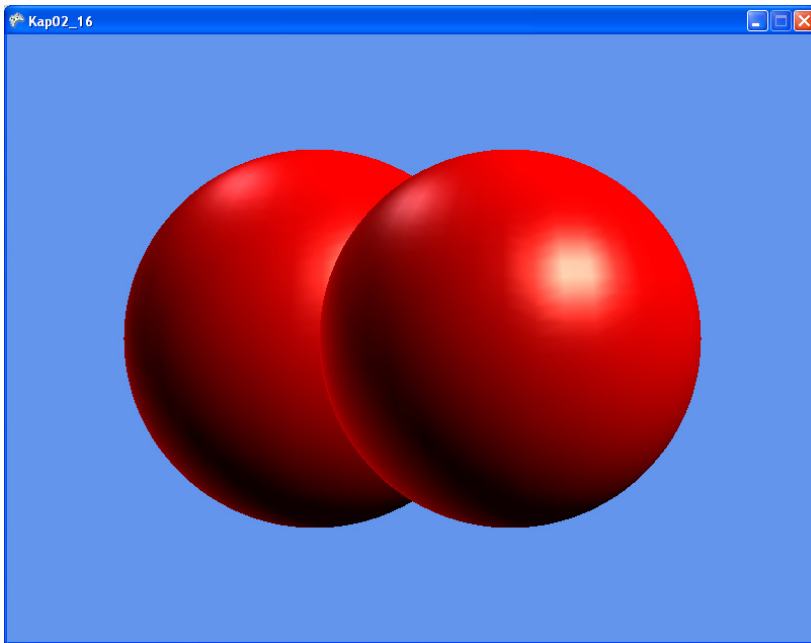


Abbildung 2.9
Überlappende Kugeln
ohne Z-Buffering

Alpha-Blending

Unter *Alpha-Blending* versteht man das Mischen (*Blending*) der Farbwerte eines zu zeichnenden Bildpunktes – der *Quelle (Source)* – mit dem Farbwert eines eventuell an dieser Stelle schon vorhandenen Punktes – des *Ziels (Destination)* – unter Berücksichtigung von Transparenzinformationen (*Alpha*). Wenn also ein Hintergrundobjekt bereits gezeichnet wurde und ein zweites Objekt jetzt im Vordergrund darüber gezeichnet werden soll, dann sind die Bildpunkte dieses zweiten Objekts die Quelle und die des ersten das Ziel.

Das Alpha-Blending müssen Sie zunächst einmal aktivieren und dann können Sie auch noch mit den Eigenschaften `RenderState.SourceBlend` und `RenderState.DestinationBlend` genau festlegen, wie dieses Mischen erfolgen soll.

Dabei sind für beide jeweils die folgenden Einstellungen möglich (und sogar noch ein paar exotische zusätzlich):

Zero	Der Wert wird nicht berücksichtigt.
One	Der Wert wird voll berücksichtigt.
SourceColor	Farbe der Quelle
InverseSourceColor	Inverse Farbe der Quelle

DestinationColor	Farbe des Ziels
InverseDestinationColor	Inverse Farbe des Ziels
SourceAlpha	Alpha der Quelle
InverseSourceAlpha	Inverses Alpha der Quelle
DestinationAlpha	Alpha des Ziels
InverseDestinationAlpha	Inverses Alpha des Ziels
BlendFactor	Konstanter Blend-Faktor, der durch die Eigenschaft <code>BlendFactor</code> angegeben wird
InverseBlendFactor	Das Inverse des konstanten Blend-Faktors, der durch die Eigenschaft <code>BlendFactor</code> angegeben wird

Die mit Fettdruck gekennzeichneten Einstellungen sind diejenigen, die in diesem Buch verwendet werden.

Hinweis

Mit den *Inversen* eines Wertes x ist hier nicht das Inverse hinsichtlich der Multiplikation ($1/x$) gemeint, sondern das Inverse bezüglich der Addition, also $1-x$.

Die Kombination der Werte in der obigen Tabelle für Quelle und Ziel ergibt eine verwirrende Anzahl von Möglichkeiten, die nicht alle sinnvoll sind.

Blend.One/Blend.Zero

Die Voreinstellung ist `Blend.One` für die Source und `Blend.Zero` für die Destination – der neue Punkt überschreibt also vollständig den vorhandenen. Damit hat man also auch keinen Transparenzeffekt.

Blend.SourceAlpha/Blend.InverseSourceAlpha

Die übliche Einstellung für ein teiltransparentes Quellobjekt ist `Blend.SourceAlpha` für `SourceBlend` und `Blend.InverseSourceAlpha` für `DestinationBlend`. Dann wird die Farbe der Quelle mit dem Alpha-Wert der Quelle multipliziert und die Farbe des Ziels mit dem Inversen ebendieses Alpha-Wertes – je durchsichtiger also das vordere Objekt ist, desto mehr scheint das hintere hindurch.

Blend.One/Blend.One

Auch die Kombination `Blend.One` für Quelle *und* Ziel liefert interessante Ergebnisse: Dann werden die Farbwerte von Hintergrund- und Vordergrund-Objekt einfach addiert. Schwarze Punkte (Farbwert (0, 0, 0)) wirken damit transparent, während sich bei farbigen Punkten die Helligkeiten addieren. So sehen auch texturierte Objekte, deren Texturen keinen Alpha-Kanal, aber einen schwarzen Hintergrund haben, einigermmaßen durchsichtig aus.

Beispiel

Um ein Material teilweise durchsichtig zu machen, weisen Sie ihm einen Alpha-Wert zu, der kleiner als 1 ist. (Alpha = 1 bedeutet, das

Material ist komplett undurchsichtig (*opak*), Alpha = 0 bedeutet, das Material ist vollständig durchsichtig (*transparent*.)

Beim `BasicEffect` setzen Sie dazu einfach die Eigenschaft `Alpha`, z.B. im Beispiel des vorhergehenden Abschnitts mit den zwei Kugeln in der Methode `DrawModel`:

```
effect.Alpha = 0.5f;
```

Außerdem muss (in der Regel in der `Draw`-Methode) der `RenderState` `AlphaBlendEnable` auf `true` gesetzt und die Eigenschaften `SourceBlend` und `DestinationBlend` müssen eingestellt werden. Meist setzt man zusätzlich `DepthBufferEnable` auf `false`, damit auch diejenigen Teile des hinteren Objekts gezeichnet werden, die von dem vorderen (ganz oder teilweise durchsichtigen) Objekt verdeckt werden. Dabei ist außerdem zu gewährleisten, dass das durchsichtige Objekt nach allen anderen in der Szene gezeichnet wird.

```
GraphicsDevice.RenderState.DepthBufferEnable = false;  
  
GraphicsDevice.RenderState.AlphaBlendEnable = true;  
GraphicsDevice.RenderState.SourceBlend = Blend.SourceAlpha;  
GraphicsDevice.RenderState.DestinationBlend = Blend.InverseSourceAlpha;
```

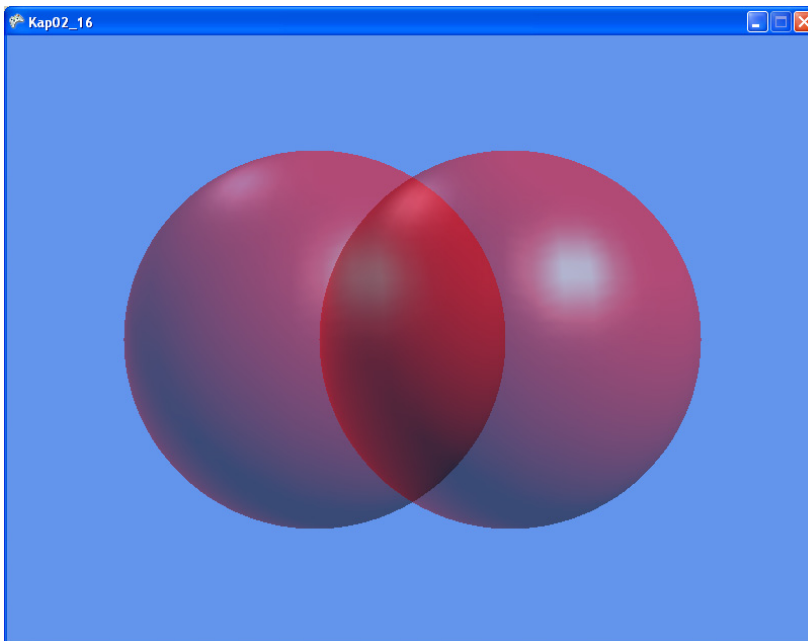


Abbildung 2.10
Halbdurchsichtige
Kugeln

Probieren Sie auch einmal das Zusammenwirken von Alpha-Blending mit Texturen aus, indem Sie die Zeile

```
effect.TextureEnabled = false;
```

aus der Methode `DrawModel` entfernen.

2.7 Eingabegeräte

Bevor unser Spiel irgendetwas Sinnvolles tun kann, müssen wir herausfinden, wie man die Eingabegeräte (Tastatur, Maus und Xbox-Controller) abfragt. Das XNA-Framework enthält dafür eine Reihe von Klassen, mit deren Methoden Sie den Zustand aller Eingabegeräte auslesen und teilweise auch festlegen können.

Wenn Sie die Eingabemethoden in einer Datei verwenden, die nicht mit dem Assistenten erstellt wurde, müssen Sie daran denken, eine `using`-Direktive für den Namespace `Microsoft.Xna.Framework.Input` in die Datei einzufügen:

```
using Microsoft.Xna.Framework.Input;
```

Tastatur

Beispiel

Das Beispiel zu diesem Abschnitt finden Sie auf der Begleit-CD im Ordner `KAP2\KAPO2_04`.

Die Klassen `Keyboard` und `KeyboardState`

Die Klasse `Keyboard` stellt uns Methoden zum Zugriff auf die Tastatur zur Verfügung.

GetState

Die statische Methode `GetState` der `Keyboard`-Klasse liefert ein `KeyboardState`-Objekt, über dessen Eigenschaften Sie den Zustand der Tastatur abfragen können.

Beispiel

Wir wollen die Funktionstasten `F1` bis `F3` verwenden, um die drei Richtungslichter des `BasicEffect` ein- und auszuschalten.

Dazu fügen wir zunächst drei Hilfsvariablen oben in die Klasse ein:

```
private bool isLight00n = false;  
private bool isLight10n = false;  
private bool isLight20n = false;
```

Da uns normalerweise vor allem Veränderungen des Zustands einer Taste interessieren (gedrückt/losgelassen), erhält unsere Klasse außerdem eine Variable für den vorherigen Zustand:

```
private KeyboardState oldKeyboardState;
```

In `Update` werden die Hilfsvariablen umgeschaltet, wenn die zugehörige Taste gedrückt wurde. Dabei genügt es nicht, abzufragen, ob die Taste aktuell gedrückt ist, denn dies wird in der Regel über mehrere Frames hinweg der Fall sein, selbst wenn der Anwender die Taste nur kurz antippt. Damit also der Zustand unserer Lampen nicht ständig umgeschaltet wird, solange die entsprechende Taste gedrückt bleibt, müssen wir den neuen Zustand der Taste mit dem vorhergehenden vergleichen: Nur wenn die Taste vorher nicht gedrückt war, jetzt aber wohl, wird die Lampe umgeschaltet.

```
protected override void Update(GameTime gameTime)
{
    // Allows the default game to exit on Xbox 360 and Windows
    if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
        this.Exit();

    KeyboardState newState = Keyboard.GetState();
    if ((newState.IsKeyDown(Keys.F1)) &&
        !(oldKeyboardState.IsKeyDown(Keys.F1)))
    {
        isLight00n = !isLight00n;
    }
    if ((newState.IsKeyDown(Keys.F2)) &&
        !(oldKeyboardState.IsKeyDown(Keys.F2)))
    {
        isLight10n = !isLight10n;
    }
    if ((newState.IsKeyDown(Keys.F3)) &&
        !(oldKeyboardState.IsKeyDown(Keys.F3)))
    {
        isLight20n = !isLight20n;
    }
    oldKeyboardState = newState;

    base.Update(gameTime);
}
```

Listing 2.16

Update

In `Draw` müssen wir jetzt nur noch entsprechend die Lichter aktivieren:

Listing 2.17
Draw

```
protected override void Draw(GameTime gameTime)
{
    graphics.GraphicsDevice.Clear(Color.Black);

    float aspectRatio =
        (float)GraphicsDevice.Viewport.Width /
        (float)GraphicsDevice.Viewport.Height;
    Matrix projection = Matrix.CreatePerspectiveFieldOfView(
        MathHelper.ToRadians(45.0f), aspectRatio, 1.0f, 1000.0f);
    Matrix view = Matrix.CreateLookAt(new Vector3(0, 0, 5), Vector3.Zero,
        Vector3.Up);

    ModelMesh mesh = earth.Meshes[0];
    foreach (BasicEffect effect in mesh.Effects)
    {
        effect.EnableDefaultLighting();
        effect.AmbientLightColor = new Vector3(0, 0, 0);

        effect.DirectionalLight0.Enabled = isLight0On;
        effect.DirectionalLight1.Enabled = isLight1On;
        effect.DirectionalLight2.Enabled = isLight2On;

        effect.View = view;
        effect.Projection = projection;
        effect.World = Matrix.Identity;
    }
    mesh.Draw();

    base.Draw(gameTime);
}
```

Ereignisgesteuerte Tastaturabfrage

Wenn Sie von der Windows-Forms-Programmierung mit C# her daran gewöhnt sind, dass Tastatureingaben Ereignisse auslösen, so können Sie dies mit ein wenig Programmieraufwand auch in XNA so realisieren. Dazu müssen Sie in Ihrer Game-Klasse selbst Ereignisse definieren, auslösen und dafür dann entsprechende Ereignishandler anfügen.

Das vorhergehende Beispiel (Ein- und Ausschalten eines der drei BasicEffect-Richtungslichter) könnte etwa wie folgt aufgebaut werden:

Definieren Sie zunächst eine von EventArgs abgeleitete Klasse und einen dazu passenden Delegattyp:

```
public class KeyboardEventArgs: EventArgs
{
    private Keys key;

    public Keys Key
    {
        get { return key; }
    }

    public KeyboardEventArgs(Keys key)
    {
        this.key = key;
    }
}
```

Listing 2.18

Die Klasse Keyboard-EventArgs

```
public delegate void KeyboardEventHandler(object sender, KeyboardEventArgs e);
```

In die Game-Klasse wird das zugehörige Ereignis eingefügt:

```
public event KeyboardEventHandler KeyPressed;
```

Bei Änderung des Tastenzustands lösen Sie das Ereignis aus:

```
protected override void Update(GameTime gameTime)
{
    if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
        this.Exit();

    KeyboardState newState = Keyboard.GetState();
    if ((newState.IsKeyDown(Keys.F1)) &&
        !(oldKeyboardState.IsKeyDown(Keys.F1)))
    {
        if (KeyPressed != null)
        {
            KeyboardEventArgs e = new KeyboardEventArgs(Keys.F1);
            KeyPressed(this, e);
        }
    }

    //...

    oldKeyboardState = newState;

    base.Update(gameTime);
}
```

Listing 2.19

Update

Verknüpfen Sie nun im Konstruktor von `Game1` einen Ereignishandler mit dem Ereignis:

Listing 2.20
Der Konstruktor der Klasse `Game1`

```
public Game1()
{
    graphics = new GraphicsDeviceManager(this);
    Content.RootDirectory = "Content";

    oldKeyboardState = Keyboard.GetState();
    this.KeyPressed += new KeyboardEventHandler(Game1_KeyPressed );
}
```

Zuletzt muss nur noch wie gewohnt der Ereignishandler selbst geschrieben werden:

Listing 2.21
`Game1_KeyPressed`

```
void Game1_KeyPressed (object sender, KeyboardEventArgs e)
{
    switch (e.Key)
    {
        case Keys.F1:
            isLight00n = !isLight00n;
            break;
        //...
    }
}
```

Diese Methode bringt natürlich einen gewissen Overhead mit sich, ist aber für den Programmierer manchmal leichter zu handhaben.

Maus

Beispiel

Das Beispiel zu diesem Abschnitt finden Sie auf der Begleit-CD im Ordner `KAP2\KAPO2_05`.

Für Eingaben in Dialogfeldern (Einstellen von Spieloptionen etc.), aber auch für die Steuerung der Spielerposition und der Kamera ist die Maus ebenfalls ein wichtiges Eingabegerät.

Die Klassen `Mouse` und `MouseState`

Die Klasse `Mouse` stellt uns unter Windows Methoden zum Zugriff auf die PC-Maus zur Verfügung. Auf der Xbox funktionieren diese Methoden natürlich nicht, da die Xbox keine Maus hat. Wenn Sie portablen Code entwickeln wollen, sollten Sie daher als Alternative zur Maus immer auch eine Steuerung über den Xbox-Controller (z.B. die Analogsticks) vorsehen.

GetState

Die statische Methode `GetState` der `Mouse`-Klasse liefert ein `MouseState`-Objekt, über dessen Eigenschaften Sie die Position des Mauszeigers und den Zustand der Maustasten abfragen können.

Mausposition

Die Mausposition ist einfach in den Eigenschaften `X` und `Y` des `MouseState` enthalten.

SetPosition

Mit Hilfe der Methode `SetPosition` der Klasse `Mouse` können Sie die Position des Mauszeigers sogar festlegen. Wir werden davon bei der Kamerasteuerung Gebrauch machen, indem wir den (unsichtbaren) Mauszeiger in jedem Frame in die Bildschirmmitte zurücksetzen und dann die Verschiebung gegenüber dieser Position auswerten, um die Kamera zu drehen.

Maustatzen

Die folgenden Eigenschaften der `MouseState`-Klasse liefern jeweils einen Wert aus der `ButtonState`-Aufzählung:

- `LeftButton`
- `RightButton`
- `MiddleButton`
- `XButton1`
- `XButton2`

Die möglichen `ButtonStates` sind:

- `Pressed`
- `Released`

Das Mousrad

Die Eigenschaft `ScrollWheelValue` des `MouseState`-Objekts liefert einen Wert, der der Drehung des Mousrads entspricht. Es handelt sich dabei um einen absoluten Wert – in der Regel wird man diesen mit demjenigen aus dem vorhergehenden Frame vergleichen, um festzustellen, ob das Rad seitdem gedreht wurde.

Den Mauszeiger sichtbar machen

Standardmäßig ist bei XNA-Projekten der Mauszeiger ausgeblendet (auf der Xbox haben Sie ja keine Maus und auf dem PC wird die Maus

meist für die Rundumsicht der Kamera verwendet, wofür Sie ebenfalls keinen Mauszeiger brauchen).

Wenn Sie aber trotzdem einmal den Mauszeiger sichtbar machen möchten (z.B. während ein Dialogfeld angezeigt wird oder weil Sie in einem Windows-Spiel den Mauszeiger zum Zielen benutzen möchten), setzen Sie einfach in der Game-Klasse die Eigenschaft `IsMouseVisible` auf `true`:

```
IsMouseVisible = true;
```

Beispiel

Im folgenden Beispiel sollen Informationen über die Maus in der Titelleiste des Fensters angezeigt werden. Da wir noch nicht über grafische Benutzeroberflächen gesprochen haben, ist das derzeit die einfachste Möglichkeit, überhaupt Text auszugeben. (Vorab sei aber gesagt: Wenn Sie bereits Erfahrung mit der Windows-Forms-Klassenbibliothek haben, können Sie diese, zumindest für Windows-Projekte, auch mit XNA nutzen.)

Membervariablen

Wie bei der Tastaturabfrage interessieren uns häufig nur Änderungen des Zustands, daher fügen wir wieder eine Variable für den vorherigen Mauszustand in die Game-Klasse ein. Außerdem benötigen wir noch eine `String`-Variable, die eine Meldung aufnehmen wird.

```
private MouseState oldMouseState;  
private String message = "Left Button is up.";
```

Im Konstruktor wird die Maus erstmalig abgefragt. Dies ist besonders dann wichtig, wenn Sie Mausbewegungen auswerten: In der Regel werden Sie dann den Abstand von der alten zur neuen Mausposition messen – dazu muss natürlich die alte Mausposition bereits am Anfang einen korrekten Wert enthalten.

Listing 2.22
Der Konstruktor der
Klasse `Game1`

```
public Game1()  
{  
    graphics = new GraphicsDeviceManager(this);  
    Content.RootDirectory = "Content";  
  
    oldMouseState = Mouse.GetState();  
}
```

Update

Auch der Mauszustand wird normalerweise in der Update-Methode abgefragt. Zunächst prüfen wir, ob der Zustand der Buttons sich geändert hat, und setzen unseren Meldungstext entsprechend um. (Hiefür wäre das Abspeichern des alten Zustands streng genommen nicht notwendig, man könnte die Variable `message` einfach jedes Mal neu setzen, egal ob sich der Zustand geändert hat oder nicht.)

Zuletzt werden die Koordinaten des Mauszeigers in die Titelzeile geschrieben:

```
protected override void Update(GameTime gameTime)
{
    // Allows the default game to exit on Xbox 360 and Windows
    if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
        this.Exit();

    MouseState newMouseState = Mouse.GetState();

    if ((newMouseState.LeftButton == ButtonState.Pressed) &&
        (oldMouseState.LeftButton == ButtonState.Released))
    {
        message = "Left Button is down.";
    }

    if ((newMouseState.LeftButton == ButtonState.Released) &&
        (oldMouseState.LeftButton == ButtonState.Pressed))
    {
        message = "Left Button is up.";
    }

    Window.Title = String.Format("X: {0}, Y: {1} ({2})",
        newMouseState.X.ToString(), newMouseState.Y.ToString(), message);

    oldMouseState = newMouseState;
    base.Update(gameTime);
}
```

Listing 2.23

Update

Xbox-Controller

Wenn Sie für die Xbox programmieren, werden Sie an der Abfrage von Tastatur und Maus weniger interessiert sein, sondern Ihr Spiel mit Hilfe des Xbox-Controller (Gamepad) steuern. Aber auch unter Windows bringt es ein ganz anderes Spielgefühl, ein Gamepad in der Hand zu halten, das eine speziell an Spiele angepasste Anordnung der Steuerungselemente aufweist, wie man sie auch von anderen Konsolen her

Beispiel

Das Beispiel zu diesem Abschnitt finden Sie auf der Begleit-CD im Ordner `KAP2\KAPO2_04`.

kennt, und das womöglich noch mit Vibrationseffekten auf Zusammenstöße, Treffer oder Ähnliches reagiert. Xbox-Controller sind im Handel (z.B. bei Online-Versandhändlern) auch einzeln erhältlich, teilweise sogar mit mitgelieferten Treibern für Windows. Die Anschaffung ist (im Vergleich mit PC-Eingabegeräten) nicht ganz billig, lohnt sich aber definitiv, und wenn man auch für die Xbox programmieren möchte, kommt man ohnehin nicht darum herum.

Wie man den Xbox-Controller abfragt, hat uns der Assistent bereits vorgemacht: Beim Erstellen eines neuen Projekts wird der folgende Code in die Update-Methode der Game-Klasse eingefügt:

Listing 2.24
Update

```
protected override void Update(GameTime gameTime)
{
    // Allows the default game to exit on Xbox 360 and Windows
    if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
        this.Exit();

    //...
}
```

Die Klassen `GamePad` und `GamePadState`

Die Klasse `GamePad` stellt uns Methoden zum Zugriff auf den Xbox-Controller zur Verfügung.

GetState

Die statische Methode `GetState` der `GamePad`-Klasse liefert ein `GamePadState`-Objekt, über dessen Eigenschaften Sie den Zustand der verschiedenen Buttons Ihres Controllers abfragen können.

SetVibration

Mittels der statischen Methode `SetVibration` der `GamePad`-Klasse können Sie die Vibrationsmotoren des Xbox-Controllers (*Rumble*) steuern.

Der linke Motor erzeugt niederfrequente Schwingungen, der zweite hochfrequente.

Zugriff auf Controller-Elemente

Die folgenden Eigenschaften des `GamePadState`-Objekts bieten Zugriff auf den Zustand des Controllers:

- Buttons
- DPad

- ThumbSticks
- Triggers

Beispiel

Fügen Sie zum Testen der Steuerungsmöglichkeiten mit dem Xbox-Controller die folgenden Anweisungen in die `Update`-Methode Ihrer `Game`-Klasse ein:

Einschalten beider Motoren mit dem linken Shoulder-Button:

```
if (GamePad.GetState(PlayerIndex.One).Buttons.LeftShoulder ==
    ButtonState.Pressed)
{
    GamePad.SetVibration(PlayerIndex.One, 1.0f, 1.0f);
}
```

Abschalten beider Motoren mit dem rechten Shoulder-Button:

```
if (GamePad.GetState(PlayerIndex.One).Buttons.RightShoulder ==
    ButtonState.Pressed)
{
    GamePad.SetVibration(PlayerIndex.One, 0.0f, 0.0f);
}
```

Einschalten des linken Motors durch Niederdrücken des linken Thumbsticks:

```
if (GamePad.GetState(PlayerIndex.One).Buttons.LeftStick ==
    ButtonState.Pressed)
{
    GamePad.SetVibration(PlayerIndex.One, 1.0f, 0.0f);
}
```

Einschalten des rechten Motors durch Niederdrücken des rechten Thumbsticks:

```
if (GamePad.GetState(PlayerIndex.One).Buttons.RightStick ==
    ButtonState.Pressed)
{
    GamePad.SetVibration(PlayerIndex.One, 0.0f, 1.0f);
}
```

Regeln der Vibrationsstärke der Motoren mit den Thumbsticks:

```
GamePad.SetVibration(PlayerIndex.One,
    (float) Math.Abs(GamePad.GetState(PlayerIndex.One).ThumbSticks.Left.Y),
    (float) Math.Abs(GamePad.GetState(PlayerIndex.One).ThumbSticks.Right.Y));
```

Bei obigen Codeschnipseln handelt es sich natürlich nur um Beispiele; anstelle der Gamepad-Motoren hätten wir genauso gut die Lichtintensität oder die Position und Orientierung irgendeines Objekts mit dem Xbox-Controller regeln können (vgl. auch das Beispiel im Abschnitt 2.9). Was Sie tatsächlich mit welchem Controllerelement steuern, hängt vom Spiel ab und kann im Idealfall (z.B. mittels einer Konfigurationsdatei) vom Spieler selbst eingestellt werden.

2.8 Kamera

Bisher haben wir unsere Szene nur aus jeweils einer festen Richtung betrachtet. Um die 3D-Welt von verschiedenen Seiten betrachten und den Spieler darin umherbewegen zu können, ist eine Kamerasteuerung unverzichtbar.

Hintergrundwissen

Beispiel

Das Beispiel zu diesem Abschnitt finden Sie auf der Begleit-CD im Ordner `KAP2\KAPO2_o6`.

Wir haben bereits gesehen, dass wir für die Umrechnung der Positionen unserer Vertices in Pixel auf dem Bildschirm den Effekt immer mit einer Reihe von Transformationsmatrizen versorgen müssen:

- World (Position, Rotation und Skalierung des Objekts im Raum)
- View (Kameraposition und Blickrichtung)
- Projection (Abbildung des Kamerabildes auf den Bildschirm)

Während die World-Matrix eines Modells nur dieses Modell selbst betrifft und verschiedene Modelle sich in der Regel auch an verschiedenen Positionen befinden und daher verschiedene World-Matrizen haben, sind die View- und Projection-Matrizen für die ganze Szene gleich. Die Projection-Matrix wird sich im Verlauf des Spiels überhaupt nur selten ändern; meist wird sie einmal initialisiert und dann nicht weiter beachtet. Die View-Matrix dagegen verändert sich sehr häufig – wenn der Spieler sich durch die Szene bewegt, betrachtet er diese ja ständig unter neuen Blickwinkeln und von verschiedenen Positionen aus.

Eine eigene Kameraklasse

Es bietet sich an, alle diejenigen Informationen, die in die Berechnung der View- und Projection-Matrizen eingehen – also Position und

Blickrichtung des Beobachters, Seitenverhältnis der Projektionsfläche (des Bildschirms), Far und Near Plane –, in einer eigenen Klasse zu speichern und nur dann neu zu berechnen, wenn sich wirklich etwas geändert hat. Bei einem Film oder Foto würden diese Parameter durch die verwendete Kamera bestimmt: ihre Position, Linsen usw. Man spricht daher auch bei 3D-Grafik in diesem Zusammenhang von einer Kamera und wir wollen unsere Klasse ebenfalls `Camera` nennen.

```
public class Camera
{
    private Matrix view;
    private Matrix projection;

    protected Vector3 eye;
    protected Vector3 lookAt;
    protected Vector3 up;

    protected float aspectRatio = 1.0f;
    protected float nearPlane = 1.0f;
    protected float farPlane = 1000.0f;

    public Camera()
    {
    }

    //...
}
```

Listing 2.25
Grundgerüst der
Camera-Klasse

Projektionsparameter

Die Projection-Matrix wird durch drei Größen bestimmt: das Seitenverhältnis der Projektionsfläche (*Aspect Ratio*), das normalerweise mit dem Seitenverhältnis der Bildschirmabmessungen übereinstimmt, die so genannte *Near Plane* und die *Far Plane*.

Near Plane und Far Plane

Die Near Plane und Far Plane begrenzen den sichtbaren Bereich (auch *View Volume* oder *View Frustum* genannt) nach vorn und hinten. Seitlich ergibt sich die Begrenzung des View Frustum durch den Öffnungswinkel `fieldOfView`, der ebenfalls in die Berechnung der Projektionsmatrix mit eingeht.

Objekte, die näher an der Kamera liegen als die Near Plane, werden nicht gerendert, ebenso Objekte, die weiter entfernt sind als die Far Plane.

Methoden und Eigenschaften

Für den Zugriff auf diese Daten fügen wir Set-Methoden ein sowie zum direkten Lesen und Setzen der Matrix eine Eigenschaft:

Listing 2.26
Projektionsparameter

```
public void SetAspectRatio(float aspectRatio)
{
    this.aspectRatio = aspectRatio;

    projection = Matrix.CreatePerspectiveFieldOfView(
        MathHelper.ToRadians(45.0f), aspectRatio, nearPlane, farPlane);
}

public void SetNearPlane(float nearPlane)
{
    this.nearPlane = nearPlane;

    projection = Matrix.CreatePerspectiveFieldOfView(
        MathHelper.ToRadians(45.0f), aspectRatio, nearPlane, farPlane);
}

public void SetFarPlane(float farPlane)
{
    this.farPlane = farPlane;

    projection = Matrix.CreatePerspectiveFieldOfView(
        MathHelper.ToRadians(45.0f), aspectRatio, nearPlane, farPlane);
}

public Matrix Projection
{
    get { return projection; }
    set
    {
        projection = value;
    }
}
```

View-Parameter

Die View-Matrix legt man fest durch die Vektoren *Eye Location*, *Lookat Vector* und *Up Vector*.

Eye Location und Lookat Vector

Die Position der Kamera (Eye Location) wird angegeben durch den Vektor `eye`; die Blickrichtung ist festgelegt durch den Vektor `lookAt` (die Position des betrachteten Objekts, wobei es hier auf die Entfernung nicht ankommt).

Up Vector

Um die Orientierung der Kamera eindeutig festzulegen (sie könnte ja noch um die z-Achse gedreht sein), benötigen wir einen dritten Vektor, der immer in die Richtung zeigt, die für die Kamera »oben« ist (also da, wo sich bei einer echten Kamera die Oberseite des Kameragehäuses befindet, wie auch immer diese gerade gedreht ist).

Methoden und Eigenschaften

Für die drei View-Vektoren sowie für die View-Matrix selbst (die daraus berechnet wird) fügen wir Eigenschaften in unsere Kameraklasse ein.

```
public Vector3 EyeLocation
{
    get { return eye; }
}

public Vector3 UpVector
{
    get { return up; }
}

public Vector3 LookAt
{
    get { return lookAt; }
}

public Matrix View
{
    get{return view;}
}
```

Mit der Methode `SetViewParameters` kann außerdem die View-Matrix als Ganzes gesetzt werden:

```
public void SetViewParameters(Vector3 eye, Vector3 lookAt, Vector3 up)
{
    this.eye = eye;
    this.lookAt = lookAt;
    this.up = up;
    view = Matrix.CreateLookAt(eye, lookAt, up);
}
```

Listing 2.27
View-Parameter

Die Voreinstellungen für alle Kamera-Parameter können mit der Methode `Initialize` vorgenommen werden:

Listing 2.28
Initialize

```
public override void Initialize()
{
    IGraphicsDeviceService graphicsService =
        (IGraphicsDeviceService)Game.Services.GetService(typeof(
            IGraphicsDeviceService));
    GraphicsDevice device = graphicsService.GraphicsDevice;

    float aspectRatio =
        (float)device.Viewport.Width /
        (float)device.Viewport.Height;
    SetAspectRatio(aspectRatio);
    SetViewParameters(new Vector3(0, 0, 10), Vector3.Zero, Vector3.Up);
}
```

Verwendung

Die Klasse `Game1` erhält eine `Membervariable` für die Kamera:

```
private Camera camera;
```

Sie kann die Kamera in ihrer `Initialize`-Methode erstellen und gegebenenfalls die voreingestellten Kamera-Parameter verändern:

Listing 2.29
Initialize-Methode
der Game-Klasse

```
protected override void Initialize()
{
    camera = new Camera();
    camera.Initialize();
    camera.SetViewParameters(new Vector3(0, 0, 5), Vector3.Zero, Vector3.Up);

    base.Initialize();
}
```

In `Draw` werden jetzt `View`- und `Projection`-Eigenschaften des `Camera`-Objekts verwendet:

Listing 2.30
`Draw`

```
protected override void Draw(GameTime gameTime)
{
    graphics.GraphicsDevice.Clear(Color.CornflowerBlue);

    ModelMesh mesh = earth.Meshes[0];

    foreach (BasicEffect effect in mesh.Effects) //earth: nur ein Effekt
    {
        effect.EnableDefaultLighting();
        effect.View = camera.View;
        effect.Projection = camera.Projection;
        effect.World = Matrix.Identity;
    }
    mesh.Draw();
}
```

```
base.Draw(gameTime);
}
```

Bewegen der Kamera

Wirklich nützlich wird unsere Kamera erst, wenn wir deren Position und Blickrichtung mit der Tastatur und gegebenenfalls der Maus bzw. mit dem Xbox-Controller steuern können.

Vorwärtsbewegung und Drehung um die y-Achse

Im einfachsten Fall bewegt sich die Kamera nur vorwärts und rückwärts und kann um die y-Achse gedreht werden.

Wir fügen oben in die Klasse eine Membervariable für die Blickrichtung ein (cameraReference ist die Richtung, zu der relativ die Rotation der Kamera gemessen wird – zu Beginn zeigt dieser Vektor einfach von der Kamera zum betrachteten Objekt).

Außerdem benötigen wir Variablen für die Vorwärts- und Rotationsgeschwindigkeit sowie für den aktuellen Drehwinkel um die y-Achse (Yaw):

```
private Vector3 cameraReference = new Vector3(0, 0, -1);
private float rotationSpeed = 0.1f / 60f;
private float forwardSpeed = 1.0f / 60f;

private float yaw = 0;
private float pitch = 0;
```

Damit die Rotations- und die Vorwärtsgeschwindigkeit auch von der Game-Klasse aus gesetzt werden können (in größeren Szenen möchte man ja sicherlich die Kamera schneller bewegen als in kleinen), stellen wir entsprechende Properties zur Verfügung (man könnte auch einfach die Felder öffentlich machen):

```
public float RotationSpeed
{
    get { return rotationSpeed; }
    set { rotationSpeed = value; }
}

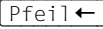
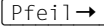
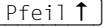
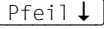
public float ForwardSpeed
{
    get { return forwardSpeed; }
    set { forwardSpeed = value; }
}
```

Listing 2.31

RotationSpeed und ForwardSpeed

Für die aus dem Drehwinkel berechnete Rotationsmatrix sehen wir ebenfalls eine Variable vor:

```
private Matrix rotationMatrix;
```

In der Update-Methode wird die Tastatur abgefragt. Die - und -Tasten verändern den Drehwinkel, die - und -Tasten bewegen die Kamera vorwärts in die aktuelle Blickrichtung oder rückwärts in die entgegengesetzte Richtung. Der Vektor, der von der Kameraposition aus nach vorn (d.h. zunächst in die negative z-Richtung) zeigt, muss dazu mit der aus dem Drehwinkel berechneten Rotationsmatrix transformiert werden, damit »vorwärts« eben immer in Blickrichtung bedeutet.

Zuletzt wird mit Hilfe der Methode `UpdateMatrices` die View-Matrix aus der Position und Blickrichtung neu berechnet.

Listing 2.32
Update

```
public void Update(GameTime gameTime)
{
    KeyboardState keyboardState = Keyboard.GetState();
    GamePadState currentState = GamePad.GetState(PlayerIndex.One);

    if (keyboardState.IsKeyDown(Keys.Left) ||
        (currentState.DPad.Left == ButtonState.Pressed))
    {
        yaw += rotationSpeed;
    }

    if (keyboardState.IsKeyDown(Keys.Right) ||
        (currentState.DPad.Right == ButtonState.Pressed))
    {
        yaw -= rotationSpeed;
    }

    if (keyboardState.IsKeyDown(Keys.Up) ||
        (currentState.DPad.Up == ButtonState.Pressed))
    {
        Vector3 v = cameraReference * forwardSpeed;

        Matrix forwardMovement = Matrix.CreateRotationY(yaw);
        v = Vector3.Transform(v, forwardMovement);
        MoveForward(v);
    }

    if (keyboardState.IsKeyDown(Keys.Down) ||
        (currentState.DPad.Down == ButtonState.Pressed))
    {

```

```

        Vector3 v = -cameraReference * forwardSpeed);

        Matrix forwardMovement = Matrix.CreateRotationY(yaw);
        v = Vector3.Transform(v, forwardMovement);
        MoveForward(v);
    }

    UpdateMatrices();
}

```

```

protected void UpdateMatrices()
{
    rotationMatrix = Matrix.CreateRotationY(yaw);
    Vector3 transformedReference = Vector3.Transform(
        cameraReference, rotationMatrix);
    Vector3 lookAt = eye + transformedReference;

    view = Matrix.CreateLookAt(eye, lookAt, up);
}

```

Listing 2.33
UpdateMatrices

Die Vorwärtsbewegung selbst wird in einer virtuellen Funktion durchgeführt, die in abgeleiteten Klassen überschrieben werden kann. (Wir werden später beim Entwickeln einer Kamera mit Kollisionserkennung hiervon noch Gebrauch machen.)

```

protected virtual void MoveForward(Vector3 v)
{
    eye.Z += v.Z;
    eye.X += v.X;
}

```

Listing 2.34
MoveForward

Die Methode `SetViewParameters` muss jetzt noch ergänzt werden, um bei einer Änderung der View-Einstellungen den Vektor `cameraReference` anzupassen und den `yaw`-Wert auf 0 zurückzusetzen:

```

public void SetViewParameters(Vector3 eye, Vector3 lookAt, Vector3 up)
{
    this.eye = eye;
    this.lookAt = lookAt;
    this.up = up;
    view = Matrix.CreateLookAt(eye, lookAt, up);

    Vector3 d = lookAt - eye;
    d.Normalize();
    cameraReference = d;
    yaw = 0;
}

```

Listing 2.35
SetViewParameters

Aktualisierung

In der Update-Methode der Game-Klasse müssen wir nun Update für die Kamera aufrufen:

Listing 2.36

Update

```
protected override void Update(GameTime gameTime)
{
    if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
        this.Exit();

    camera.Update(gameTime);

    base.Update(gameTime);
}
```

Drehen der Kamera um die x-Achse (Pitch)

Während für eine Innen- oder Außenszene, in der die Spielerfigur herumläuft, die Drehung um die y-Achse durchaus ausreicht, kann ein Raumschiff oder ein Flugzeug natürlich seine »Nase« auch nach oben oder unten neigen (also um die x-Achse drehen (Pitch)). Wir wollen hierfür die Tasten **Bild ↑** und **Bild ↓** verwenden und fügen eine entsprechende Membervariable in die Kameraklasse ein:

```
float pitch = 0;
```

Nun müssen wir noch die Update-Methode erweitern:

Listing 2.37

Drehen der Kamera um die x-Achse

```
if (keyboardState.IsKeyDown(Keys.PageDown))
{
    pitch += rotationSpeed;
}

if (keyboardState.IsKeyDown(Keys.PageUp))
{
    pitch -= rotationSpeed;
}
```

Sie könnten für die unterschiedlichen Drehungen verschiedene Rotationsgeschwindigkeiten festlegen; wir begnügen uns hier mit einem einheitlichen Wert.

Die Methode UpdateMatrices wird ebenfalls ergänzt, um auch den Pitch-Wert zu berücksichtigen:

Listing 2.38

UpdateMatrices

```
protected void UpdateMatrices()
{
```

```

rotationMatrix = Matrix.CreateRotationX(pitch) *
    Matrix.CreateRotationY(yaw);
Vector3 transformedReference = Vector3.Transform(cameraReference,
    rotationMatrix);
lookAt = eye + transformedReference;
view = Matrix.CreateLookAt(eye, lookAt, up);
}

```

Fügen Sie auch eine Zeile in `SetViewParameters` ein, um die `pitch`-Variable zurückzusetzen:

```
pitch = 0;
```

Umsehen mit der Maus

In den meisten Spielen wird die Position der Kamera über die Tastatur gesteuert; oft ist es aber praktisch, wenn der Spieler zusätzlich die Möglichkeit hat, sich umzusehen, ohne seine Position zu verändern. Unter Windows kann man dafür sehr gut die Maus verwenden.

Damit dieses Verhalten nach Wunsch an- und abgeschaltet werden kann, fügen wir eine `bool`-Variable in die Klasse ein. Außerdem benötigen wir Variablen für die jeweils vorherige Position des Mauszeigers, da die Drehung von der Verschiebung, also der Differenz zwischen neuer und alter Position, abhängen wird:

```

public bool mouseRotate = false;
private int oldX, oldY;

```

In die `Update`-Methode wird (vor der Tastaturabfrage) entsprechender Code für die Auswertung der Mausbewegung eingefügt:

```

if (!Game.IsActive) return;
if (mouseRotate)
{
    MouseState mouseState = Mouse.GetState();

    int dx = mouseState.X - oldX;
    yaw += - 0.01f * dx;

    int dy = mouseState.Y - oldY;
    pitch += 0.01f * dy;

    ResetMouseCursor();
}

```

Listing 2.39
Drehen der Kamera mit
der Maus

Zum Umschalten der `mouseRotate`-Variablen mittels einer Funktionstaste können Sie ebenfalls Code in `Update` einfügen:

Listing 2.40
Umschalten der
Variablen `mouseRotate`

```
if (keyboardState.IsKeyDown(Keys.F8) && !oldKeyboardState.
    IsKeyDown(Keys.F8))
{
    mouseRotate = !mouseRotate;
}
```

Die Variable `oldKeyboardState` wird wieder wie in früheren Beispielen in der `Game`-Klasse oben in der Klasse deklariert und in `Initialize` initialisiert.

Wenn die Anwendung nicht aktiv ist, soll übrigens keine Aktualisierung der Kamera erfolgen. Bei der Tastatursteuerung fällt es selten auf, wenn man daran nicht denkt, aber die mausgesteuerte Kamera schaut in diesem Fall garantiert in irgendeine völlig unsinnige Richtung, wenn Sie beispielsweise mit der Tastenkombination `Alt`+`↔` zwischen den Anwendungen hin- und herschalten.

Zurücksetzen des Mauszeigers

Die Methode `ResetMouseCursor` setzt nach der Auswertung der Bewegung den Mauszeiger wieder in die Bildschirmmitte zurück, da man sonst beim Drehen schnell den Bildschirmrand erreicht und nicht weiter rotieren kann.

Listing 2.41
`ResetMouseCursor`

```
private void ResetMouseCursor()
{
    IGraphicsDeviceService graphicsService =
        (IGraphicsDeviceService)game.Services.GetService(typeof(
            IGraphicsDeviceService));
    GraphicsDevice device = graphicsService.GraphicsDevice;
    int centerX = device.Viewport.Width / 2;
    int centerY = device.Viewport.Height / 2;
    Mouse.SetPosition(centerX, centerY);
    oldX = centerX;
    oldY = centerY;
}
```

Das Zurücksetzen erfolgt auch jedes Mal, wenn die Anwendung nach dem Wechsel zu anderen Anwendungen wieder aktiviert wird:

Listing 2.42
`game_Activated`

```
void game_Activated(object sender, EventArgs e)
{
    ResetMouseCursor();
}
```

Dieser Ereignishandler wird im Konstruktor der Klasse an das Ereignis `Activated` der `Game`-Klasse angehängt. Außerdem wird dort ein Verweis auf das `Game`-Objekt festgehalten, damit wir in `ResetMouseCursor` über das `GraphicsDevice`-Objekt auf die Bildschirmabmessungen zugreifen können. Eine Membervariable für dieses `Game`-Objekt fügen wir oben in die Klasse ein:

```
private Game game;
```

Der Konstruktor sieht dann wie folgt aus:

```
public Camera(Game game):base(game)
{
    this.game = game;
    game.Activated += new EventHandler(game_Activated);}

```

Listing 2.43
Konstruktor der
`Camera`-Klasse

Aktivieren der Maussteuerung

Standardmäßig ist die Kamerasteuerung mit der Maus deaktiviert – um sie zu aktivieren, brauchen Sie nur in der `Game`-Klasse die Variable `mouseRotate` auf `true` einzustellen:

```
camera.mouseRotate = true;
```

Hinweis

Die Kameraklasse wird später noch eine weitere Vereinfachung erfahren, wenn wir `Game Components` kennengelernt haben.

Third-Person-Kamera

Die bisher diskutierte Kameraansicht wird gemeinhin auch als *First Person Camera* bezeichnet: Diese Kamera zeigt die Szene aus der Sicht des Spielers und wird mit diesem mitbewegt. Dabei sieht der Spieler sich natürlich nicht selbst, sondern allenfalls seine nach vorn ausgestreckte Hand – bei *First Person Shootern (FPS)* in der Regel mit einer Waffe. (Der Begriff *Ego-Shooter* ist übrigens nur im deutschen Sprachraum gebräuchlich.)

Bei der *Third Person Camera* dagegen sieht der Spieler seine eigene Spielfigur (*Avatar*), die Kamera folgt dieser meist etwas nach hinten und nach oben versetzt.

Eine sehr einfache Methode, eine *Third-Person-Ansicht* zu realisieren, besteht darin, dass man den Spieleravatar von vorneherein im Koordinatensystem der Kamera rendert (zur Transformation zwischen verschiedenen Koordinatensystemen vgl. auch Abschnitt 2.9). Dazu verwendet man eine *View-Matrix*, die die Kamera im Koordinatenursprung positioniert und in die negative *z*-Richtung blicken lässt. Die World-

Matrix gibt dann die Position des Avatars nur noch relativ zur Kamera an, also etwas vor und ggf. unter dem Koordinatenursprung.

Diese Technik klappt allerdings bei Bewegungen in einer Ebene (z.B. für einen Character in einem Level) nur so lange, wie die Kamera nicht nach oben oder unten schaut (weil dann der Avatar sich in die Luft erhebt bzw. im Boden versinkt) – Sie müssen also für solche Szenen die Drehung um die x-Achse (Pitch) der Kamera deaktivieren.

Beispiel

Beispiel

Das Beispiel zu diesem Abschnitt finden Sie auf der Begleit-CD im Ordner KAP2\KAPO2_07.

Um das Rendern eines Spieleravatars in der Third-Person-View einmal auszuprobieren, fügen Sie in die Game-Klasse zunächst eine Variable für eine gewöhnliche Kamera ein (wir können dafür die oben entwickelte Kameraklasse verwenden). Außerdem benötigen wir das Modell des Avatars selbst – hier ein Flugzeug. Zudem sollte die Szene noch ein paar andere Objekte enthalten, damit man überhaupt bemerkt, dass die Kamera sich bewegt – wir können dafür beispielsweise wieder einmal unsere Erdkugel benutzen.

```
private Camera camera;
private Model fighter;
private Model earth;
```

Die Kamera wird wie in den vorhergehenden Beispielen initialisiert:

Listing 2.44
Initialize

```
protected override void Initialize()
{
    camera = new Camera(this);
    camera.Initialize();
    camera.SetNearPlane(0.1f);

    base.Initialize();
}
```

In LoadContent laden wir die Modelle:

Listing 2.45
LoadContent

```
protected override void LoadContent()
{
    earth = Content.Load<Model>("earth");
    fighter = Content.Load<Model>("fighter");
}
```

Das Aktualisieren der Kamera erfolgt ebenfalls wie in den bisherigen Beispielen in Update:

```
protected override void Update(GameTime gameTime)
{
    if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
        this.Exit();

    camera.Update(gameTime);

    base.Update(gameTime);
}
```

Listing 2.46
Update

Die Draw-Methode enthält nun den interessanten Teil: Zunächst rendern wir die Szene (hier nur den Planeten) mit der View-Matrix der Kamera. Beim Rendern des Avatars werden aber andere View- und World-Matrizen verwendet – nämlich die für das Koordinatensystem der Kamera:

```
protected override void Draw(GameTime gameTime)
{
    graphics.GraphicsDevice.Clear(Color.Black);

    foreach (ModelMesh mesh in earth.Meshes)
    {
        foreach (BasicEffect effect in mesh.Effects)
        {
            effect.EnableDefaultLighting();
            effect.View = camera.View;
            effect.Projection = camera.Projection;
            effect.World = Matrix.Identity;
        }
        mesh.Draw();
    }

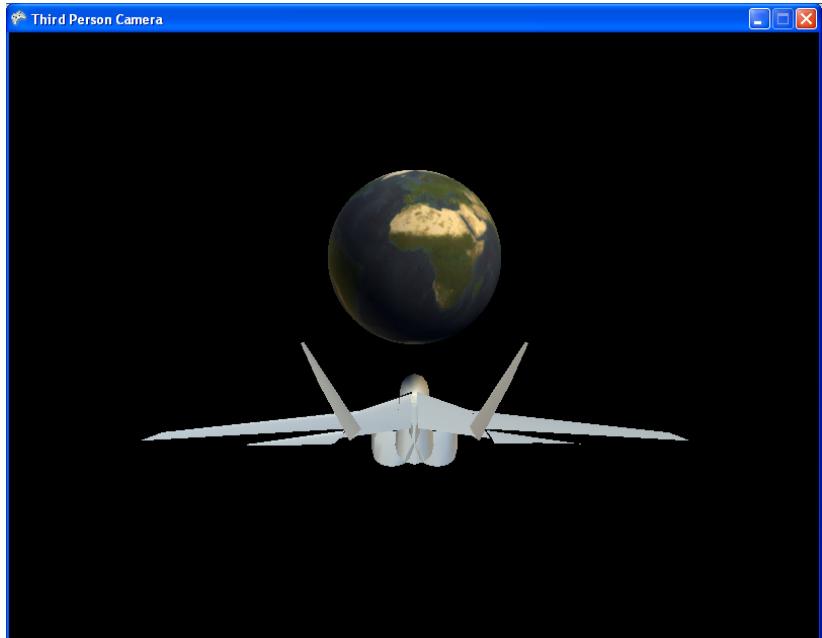
    foreach (ModelMesh mesh in fighter.Meshes)
    {
        foreach (BasicEffect effect in mesh.Effects)
        {
            effect.EnableDefaultLighting();
            effect.View = Matrix.CreateLookAt(Vector3.Zero,
                new Vector3(0, 0, -1), Vector3.Up); ;
            effect.Projection = camera.Projection;
            effect.World =
                Matrix.CreateScale(0.05f) *
                Matrix.CreateTranslation(0, -0.01f, -0.15f);
        }
        mesh.Draw();
    }

    base.Draw(gameTime);
}
```

Listing 2.47
Draw

Das Zeichnen des Mesh-Objekts erfolgt hier übrigens anders als in den vorhergehenden Beispielen in einer `foreach`-Schleife. Da die hier verwendeten Modelle zwar ohnehin nur jeweils ein Mesh enthalten, macht das zwar in diesem Fall keinen Unterschied – von jetzt an soll aber die allgemeinere Formulierung verwendet werden, damit man ohne Änderungen am Code auch Modelle mit mehreren Meshes verwenden kann.

Abbildung 2.11
Third Person Camera



2.9 Grundlagen der 3D-Mathematik

Wenn auch durch moderne Frameworks wie XNA die Spieleprogrammierung recht unkompliziert geworden ist, so stößt man doch schon bei den ersten Gehversuchen auf Konzepte wie Vektoren und Matrizen. Für einfache Spiele genügt es sicherlich auch, eine ungefähre Vorstellung davon zu haben, wie die Position, Orientierung und Größe eines Objekts mit Hilfe der World-Matrix festgelegt und wie durch Matrizenmultiplikation verschiedene Transformationen (Verschiebung, Rotation, Skalierung) miteinander kombiniert werden können.

Fortgeschrittenere Programmierer werden aber (beispielsweise bei der Entwicklung von Shadern) immer wieder auf Problemstellungen stoßen, die ein tiefer gehendes Verständnis dieser Operationen erfordern. Der vorliegende Abschnitt soll die Grundlagen hierfür vermitteln. Wenn Sie es vielleicht im ersten Durchgang nur überfliegen möchten, können Sie später darauf zurückkommen, um gezielt bestimmte Informationen nachzuschlagen.

Koordinatensystem

Wie bereits in den vorangehenden Abschnitten diskutiert, ist ein 3D-Objekt in erster Linie eine Ansammlung von Dreiecken, die durch die Koordinaten ihrer Eckpunkte beschrieben werden. Wir haben mit solchen Punkten bereits gearbeitet: Die drei Komponenten x , y und z beschreiben die Entfernung vom Koordinatenursprung in Richtung der jeweiligen Achse. x gibt also die Verschiebung nach rechts, y nach oben und z nach vorn an (aus dem Bildschirm heraus zum Betrachter hin).

Rechtshändige und linkshändige Koordinatensysteme

Die oben angegebene Orientierung der Koordinatenachsen ist keineswegs gottgegeben: Viele Programme (sowohl Modeler-Programme als auch Grafik- und Game-Engines) lassen z nach oben und y nach vorn zeigen. Dass man z nach oben zeigen lässt, ist dabei für sich allein genommen noch nicht so dramatisch: Wenn man das oben beschriebene Koordinatensystem um die x -Achse nach hinten dreht, weist die z -Achse nach oben – allerdings müsste y jetzt nach *hinten* zeigen. Die beiden Koordinatensysteme, bei denen y und z genau vertauscht sind, gehen also nicht durch eine Drehung auseinander hervor und sind somit nicht äquivalent. Man spricht in diesem Zusammenhang von der *Händigkeit* des Koordinatensystems. Wenn man bei der zuerst beschriebenen Version den Daumen der rechten Hand in Richtung der x -Achse (nach rechts) und den Zeigefinger in Richtung der y -Achse (nach oben) hält, dann zeigt der abgewinkelte Mittelfinger in Richtung der z -Achse nach vorn. Das gilt auch dann noch, wenn man die ganze Hand (das ganze Koordinatensystem) dreht. Bei dem Koordinatensystem mit x nach rechts, z nach oben und y nach vorn klappt dasselbe mit der rechten Hand nicht – wohl aber mit der linken. Das erstere Koordinatensystem wird daher als rechtshändiges, das zweite als linkshändiges bezeichnet. Die Tatsache, dass beide in der 3D-Grafik oft und gern verwendet werden, ist leider eine nicht enden wollende Quelle der Verwirrung.

DirectX selbst bietet die Möglichkeit, wahlweise mit einem rechtshändigen oder einem linkshändigen Koordinatensystem zu arbeiten, wobei aber die Konvention (etwa die Beispiele in der Dokumentation) eher zu linkshändigen Koordinaten tendiert.

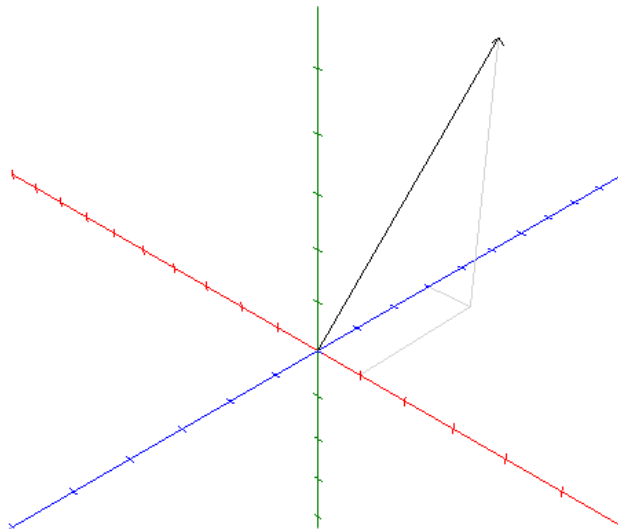
XNA dagegen ist »zwangsweise« rechtshändig, was den Umstieg und die Verwendung von vorhandenem Code nicht gerade erleichtert.

Vektoren

Eine *Vektor* ist eine Zusammenfassung von mehreren (in der 3D-Grafik meist drei oder vier) Komponenten. Ein Dreivektor kann z.B. einen Punkt im dreidimensionalen Raum beschreiben, indem man nämlich wie oben beschrieben die Entfernung vom Koordinatenursprung in x-, y- und z-Richtung angibt. Die Komponenten des Vektors werden daher meist auch mit x, y und z bezeichnet. So stellt etwa der Vektor (1, 5, -3) einen Punkt dar, der vom Koordinatenursprung aus eine Einheit nach rechts, fünf Einheiten nach oben und drei Einheiten nach hinten (vom Betrachter weg) verschoben ist.

Oft werden Vektoren auch zur Angabe einer Richtung (z.B. einer Bewegungsrichtung) verwendet. Zur Veranschaulichung stellt man Vektoren häufig durch Pfeile dar, die vom Koordinatenursprung zu dem Punkt mit den Koordinaten x, y, z zeigt. Ein solcher Pfeil beschreibt offensichtlich sowohl eine Richtung als auch einen Punkt (nämlich seinen Endpunkt).

Abbildung 2.12
Ein Vektor als Pfeil
gezeichnet



Im Programmcode wird ein Vektor durch die Struktur `Vector3` beschrieben.

XNA verwendet diese Struktur nicht nur für Punkte und Richtungen, sondern auch für Farben (die ja ebenfalls durch drei Zahlenwerte, nämlich den Rot-, Grün- und Blau-Anteil, plus evtl. einen vierten Wert für den Alpha-Anteil, festgelegt werden).

In Gleichungen werden Vektoren (im Gegensatz zu Skalaren, also einfachen Zahlen) oft auch dadurch gekennzeichnet, dass man einen Pfeil über das Symbol schreibt: \vec{v}

Länge

Nach Pythagoras gilt $x^2 + y^2 + z^2 = l^2$, wenn l die Länge des Vektors ist.

Demnach ist die Länge gleich der Quadratwurzel aus $x^2 + y^2 + z^2$. In der Mathematik bezeichnet man die Länge eines Vektors v mit $|v|$. Es gilt also:

$$\left| \begin{pmatrix} x \\ y \\ z \end{pmatrix} \right| = \sqrt{x^2 + y^2 + z^2}$$

Bei XNA berechnen Sie die Länge eines Vektors mit der Methode `Length` der Struktur `Vector3`.

Einheitsvektoren

Ein Einheitsvektor ist ein Vektor der Länge 1. Da bei Richtungsvektoren die Länge ohnehin nicht interessiert, ist es üblich, diese zu *normalisieren*, indem man jede Komponente durch die Länge teilt. Sie können anhand der obigen Formel leicht nachrechnen, dass dabei in der Tat ein Vektor mit Länge 1 herauskommt.

Mit Hilfe der Methode `Normalize` der Struktur `Vector3` wird ein Vektor normalisiert.

Addition

Die Addition zweier Vektoren kann man grafisch einfach durch Aneinanderhängen der Pfeile darstellen.

Mathematisch ist dies äquivalent mit der komponentenweisen Addition der Vektoren:

$$v_1 + v_2 = \begin{pmatrix} v_1 \cdot x + v_2 \cdot x \\ v_1 \cdot y + v_2 \cdot y \\ v_1 \cdot z + v_2 \cdot z \end{pmatrix}$$

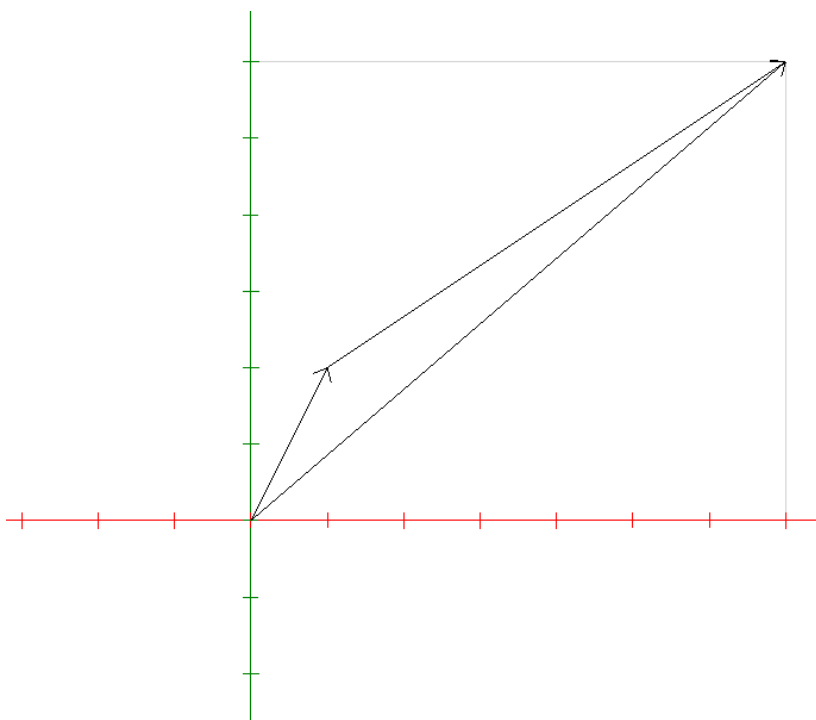
Beispiel:

$$\begin{pmatrix} 1 \\ 2 \\ 0 \end{pmatrix} + \begin{pmatrix} 6 \\ 4 \\ 0 \end{pmatrix} = \begin{pmatrix} 1+6 \\ 2+4 \\ 0+0 \end{pmatrix} = \begin{pmatrix} 7 \\ 6 \\ 0 \end{pmatrix}$$

Eine typische Anwendung für die Vektoraddition ist die Addition von Kräften in der Physik.

Bei XNA (also in Ihrem C#-Programmcode) wird eine Addition auch für Vektoren einfach mit dem Plus-Operator (+) durchgeführt.

Abbildung 2.13
Vektoraddition



Subtraktion

Da die Subtraktion einfach die Umkehroperation zur Addition ist, kann man aus obiger Grafik auch das Ergebnis einer Subtraktion zweier Vektoren entnehmen:

Wenn $v_1 + v_2 = v_3$ gilt, dann muss folglich $v_2 = v_3 - v_1$ sein.

Anders ausgedrückt hängt man die Spitze des abzuziehenden Vektors (v_2) an die Spitze des Vektors, von dem abgezogen werden soll (v_3). Der Vektor vom Fuß von v_3 zum Fuß von v_2 ist dann das Ergebnis der Addition (v_1).

Mathematisch ist diese Operation wiederum äquivalent mit der komponentenweisen Subtraktion der Vektoren:

$$v_1 - v_2 = \begin{pmatrix} v_1 \cdot x - v_2 \cdot x \\ v_1 \cdot y - v_2 \cdot y \\ v_1 \cdot z - v_2 \cdot z \end{pmatrix}$$

Beispiel:

$$\begin{pmatrix} 7 \\ 6 \\ 0 \end{pmatrix} - \begin{pmatrix} 6 \\ 4 \\ 0 \end{pmatrix} = \begin{pmatrix} 7-6 \\ 6-4 \\ 0-0 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \\ 0 \end{pmatrix}$$

Im Programmcode verwenden Sie einfach das Minuszeichen (-).

Abstand

Wenn die beiden zu subtrahierenden Vektoren die Position zweier Objekte angeben, dann ist der Differenzvektor gerade der Vektor vom ersten Objekt zum zweiten, und die Länge dieses Differenzvektors ist der Abstand der Objekte. Da die Operation in der Spieleprogrammierung recht häufig benötigt wird, stellt die Struktur `Vector3` dafür die Methode `Distance` zur Verfügung. Sie können aber genauso gut auch selbst die Länge des Abstandsvektors bestimmen.

Multiplikation mit einer Zahl

Die Multiplikation eines Vektors mit einer Zahl (einem *Skalar*) wird ebenfalls komponentenweise ausgeführt:

$$s \cdot v = \begin{pmatrix} s \cdot v \cdot x \\ s \cdot v \cdot y \\ s \cdot v \cdot z \end{pmatrix},$$

wenn s ein Skalar ist.

Beispiel:

$$3 \cdot \begin{pmatrix} 1 \\ 12 \\ 17 \end{pmatrix} = \begin{pmatrix} 3 \cdot 1 \\ 3 \cdot 12 \\ 3 \cdot 17 \end{pmatrix} = \begin{pmatrix} 3 \\ 36 \\ 51 \end{pmatrix}$$

Die Richtung des Vektors ändert sich dadurch nicht, die Länge wächst aber um den Faktor s . Die Multiplikation eines Vektors mit einem Skalar entspricht daher einer Skalierung (Streckung für $s > 1$ oder Stauchung für $s < 1$).

Bei XNA können Sie den gewöhnlichen Multiplikationsoperator verwenden, um einen Vektor mit einer Zahl zu multiplizieren.

Skalarprodukt

Nachdem die Addition und Subtraktion zweier Vektoren so einfach komponentenweise durchzuführen war, kann man sich fragen, was bei der komponentenweisen Multiplikation zweier Vektoren herauskommt. Man kann diese sicherlich auch formal durchführen, leider hat das Ergebnis aber keinerlei irgendwie anschauliche, mathematisch sinnvolle oder auch nur für die Spieleprogrammierung nützliche Bedeutung.

Stattdessen definiert man für Vektoren zwei andere Arten der Multiplikation: das Skalarprodukt und das Kreuzprodukt.

Beim *Skalarprodukt* (oder auch *Punktprodukt*, im Englischen *Dot Product*) wird tatsächlich jede Komponente des ersten Vektors mit der entsprechenden Komponente des zweiten Vektors multipliziert. Danach werden die Ergebnisse dieser drei Multiplikationen aber addiert – man erhält als Gesamtergebnis also nicht einen Vektor, sondern einen Skalar (d.h. eine einfache Zahl).

$$v_1 \cdot v_2 = v_1.x \cdot v_2.x + v_1.y \cdot v_2.y + v_1.z \cdot v_2.z$$

Beispiel:

$$\begin{pmatrix} 1 \\ 12 \\ 17 \end{pmatrix} \cdot \begin{pmatrix} 3 \\ 4 \\ 8 \end{pmatrix} = 1 \cdot 3 + 12 \cdot 4 + 17 \cdot 8 = 3 + 48 + 136 = 187$$

Man kann zeigen, dass gilt

$$v_1 \cdot v_2 = |v_1| \cdot |v_2| \cdot \cos \alpha,$$

wenn der Winkel zwischen den beiden Vektoren ist und $|v_1|$ die Länge des Vektors v_1 bzw. $|v_2|$ die Länge des Vektors v_2 bezeichnet.

Somit ist das Skalarprodukt ein Maß für die *Projektion* des zweiten Vektors auf den ersten (oder umgekehrt), also für die Komponente des zweiten in Richtung des ersten Vektors. So ist z.B. das Skalarprodukt eines beliebigen Vektors mit dem Einheitsvektor in x-Richtung $(1, 0, 0)$ genau die x-Komponente dieses Vektors.

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} = x \cdot 1 + y \cdot 0 + z \cdot 0 = x$$

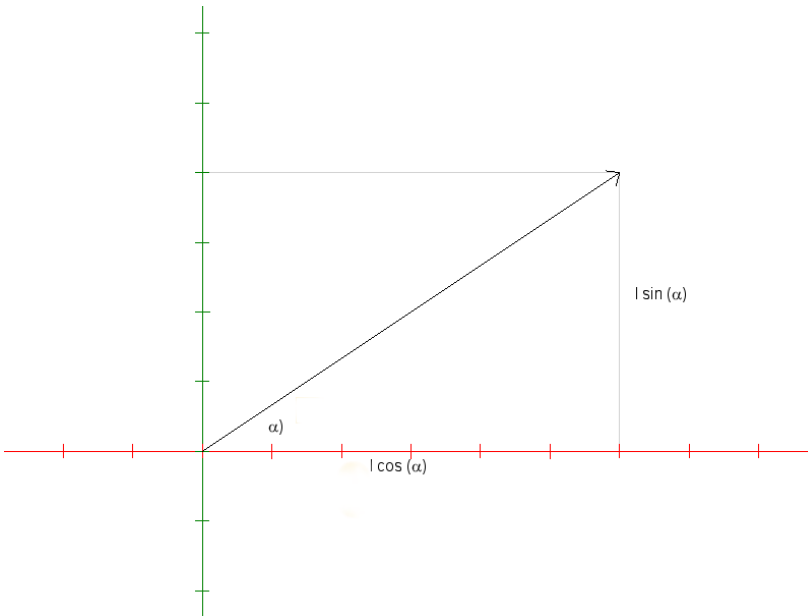


Abbildung 2.14

Die Projektion des Vektors auf die x-Achse ist gleich der Länge multipliziert mit dem Kosinus des mit der Achse eingeschlossenen Winkels.

Bei parallelen Vektoren ist der Kosinus des Winkels 1, das Skalarprodukt nimmt also den größtmöglichen Wert an (für Einheitsvektoren ebenfalls 1), bei zueinander senkrechten Vektoren ist der Kosinus des Winkels 0 und damit auch das Skalarprodukt immer 0.

Bei XNA berechnen Sie das Skalarprodukt mit der statischen Methode `Dot` der `Vector3`-Struktur.

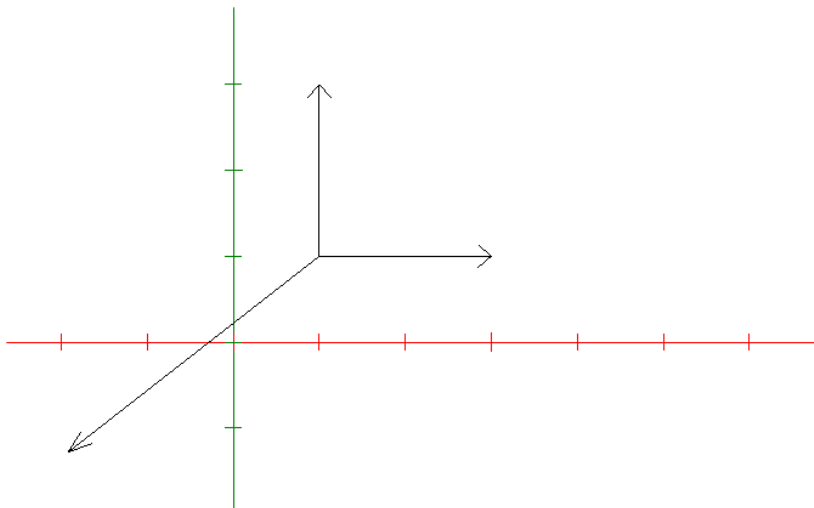
Kreuzprodukt

Eine andere Möglichkeit, Vektoren miteinander zu multiplizieren, ist das *Kreuzprodukt* oder auch einfach *Vektorprodukt* (weil das Ergebnis im Gegensatz zum Skalarprodukt ein Vektor ist), im Englischen *Cross Product*.

Geometrisch gesehen ist das Kreuzprodukt ein Vektor, der senkrecht auf den beiden miteinander multiplizierten Vektoren steht.

Abbildung 2.15

Das Kreuzprodukt (hier nach vorn zeigend) steht auf den beiden Vektoren senkrecht; seine Länge ist gleich dem Produkt der Längen, multipliziert mit dem Sinus des eingeschlossenen Winkels.



Die Länge dieses Vektors berechnet sich ähnlich wie das Skalarprodukt, nur nicht mit dem Kosinus, sondern mit dem Sinus des eingeschlossenen Winkels:

$$|\mathbf{v}_1 \times \mathbf{v}_2| = |\mathbf{v}_1| \cdot |\mathbf{v}_2| \cdot \sin \alpha$$

Folglich ist bei parallelen Vektoren das Kreuzprodukt immer 0, bei zueinander senkrechten Vektoren nimmt es den größtmöglichen Wert an (1 für Einheitsvektoren).

Die Richtung des Ergebnisvektors kann man sich wiederum mit der »Rechte-Hand-Regel« (s.o.) verdeutlichen: Wenn der Daumen der rechten Hand in Richtung \mathbf{v}_1 und der Zeigefinger in Richtung \mathbf{v}_2 weist, dann gibt der abgewinkelte Mittelfinger die Richtung des Kreuzprodukts an.

Vielleicht kennen Sie diese Regel noch aus der Schulphysik: In der Elektrodynamik spielt das Kreuzprodukt eine wichtige Rolle (z.B. Lorenzkraft).



Abbildung 2.16
Rechte-Hand-Regel

In der Spieleprogrammierung verwenden wir das Kreuzprodukt häufig dazu, einen Vektor zu finden, der auf zwei gegebenen Vektoren senkrecht steht (z.B. einen Normalenvektor, der auf einer Fläche senkrecht steht, s.u.).

Die Berechnungsvorschrift ist komplizierter als die für das Skalarprodukt und soll hier ohne Begründung angegeben werden:

$$v_1 \times v_2 = \begin{pmatrix} v_1 \cdot y \cdot v_2 \cdot z - v_1 \cdot z \cdot v_2 \cdot y \\ v_1 \cdot z \cdot v_2 \cdot x - v_1 \cdot x \cdot v_2 \cdot z \\ v_1 \cdot x \cdot v_2 \cdot y - v_1 \cdot y \cdot v_2 \cdot x \end{pmatrix}$$

Beispiel:

$$\begin{pmatrix} 1 \\ 12 \\ 17 \end{pmatrix} \times \begin{pmatrix} 3 \\ 4 \\ 5 \end{pmatrix} = \begin{pmatrix} 12 \cdot 5 - 17 \cdot 4 \\ 17 \cdot 3 - 1 \cdot 5 \\ 1 \cdot 4 - 12 \cdot 3 \end{pmatrix} = \begin{pmatrix} -8 \\ 46 \\ -32 \end{pmatrix}$$

Bei XNA berechnen Sie das Kreuzprodukt mit der statischen Methode `Cross` der `Vector3`-Struktur.

Normalen

Die *Normale* einer Fläche ist ein Vektor, der auf der Fläche senkrecht steht und die Länge 1 hat. Normalenvektoren spielen eine wichtige Rolle bei der Beleuchtung von 3D-Modellen, also der Berechnung von Licht und Schatten.

Dazu genügt es aber normalerweise nicht, für jedes Dreieck die Normale zu kennen, da man in diesem Fall (vor allem bei gerundeten

Objekten) die Kanten an den Übergängen zwischen den Dreiecken allzu deutlich sehen würde. Stattdessen speichert man die Normalen für jeden Vertex einzeln ab. Bei Vertices, die mehreren Flächen angehören, wird aus den Normalen dieser Flächen ein Mittelwert gebildet. So entstehen weiche Übergänge an den Kanten. Wünscht man bei eckigen Objekten dagegen harte Kanten, so muss derselbe Punkt mehrmals abgespeichert werden, mit zwar identischer Position, aber unterschiedlichen Normalenvektoren.

Matrizen

So wie die Komponenten eines Vektors nebeneinander in einer Zeile (oder untereinander in einer Spalte) geschrieben werden können, so bildet eine Anordnung von Zahlen in Zeilen und Spalten eine *Matrix*.

Beispiel:

$$\begin{pmatrix} 1 & 12 & 17 \\ 3 & 4 & 8 \\ 1 & 2 & 3 \end{pmatrix}$$

Die Komponenten einer Matrix werden meist durch Angabe der Zeilen- und Spaltenindices angesprochen, also etwa M_{12} für das Element in der ersten Zeile und zweiten Spalte der Matrix M (in der Beispielmatrix die Zahl 12).

$$M = \begin{pmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{pmatrix}$$

Bei drei Spalten spricht man von einer 3×3 -Matrix. Daneben spielen in der 3D-Grafikprogrammierung auch noch 4×4 -Matrizen mit 4 Zeilen und 4 Spalten eine wichtige Rolle.

Ein Zeilen-Dreiervektor kann als eine 1×3 -Matrix, ein Spaltenvektor als 3×1 -Matrix aufgefasst werden.

Addition, Subtraktion und Multiplikation von Matrizen mit Skalaren erfolgen wie bei Vektoren komponentenweise und haben auch bis auf Letztere keine allzu große praktische Bedeutung, daher werden wir darauf hier nicht im Einzelnen eingehen.

XNA stellt neben den gewöhnlichen Operatoren `+`, `-` und `*` die Methoden `Add`, `Subtract` und `Multiply` der Struktur `Matrix` zur Verfügung, um diese Operationen auszuführen.

Interessanter ist die Multiplikation von Matrizen, da sie in der Spieleprogrammierung für Transformationen (Verschiebung, Rotation, Skalierung und auch die Projektion auf die zweidimensionale Bildschirmfläche) außerordentlich wichtig ist.

Matrizenmultiplikation

Bei der Multiplikation zweier Matrizen ergibt sich das Element der Ergebnismatrix in der Zeile i und Spalte j , indem man die Zeile i der ersten Matrix mit der Zeile j der zweiten Matrix Komponente für Komponente multipliziert und die Ergebnisse addiert (wie beim Skalarprodukt zwischen Vektoren), hier am Beispiel für 2×2 -Matrizen:

$$\begin{pmatrix} 3 & 4 \\ 2 & 7 \end{pmatrix} \cdot \begin{pmatrix} 1 & 5 \\ 6 & 8 \end{pmatrix} = \begin{pmatrix} 3 \cdot 1 + 4 \cdot 6 & 3 \cdot 5 + 4 \cdot 8 \\ 2 \cdot 1 + 7 \cdot 6 & 2 \cdot 5 + 7 \cdot 8 \end{pmatrix} = \begin{pmatrix} 27 & 47 \\ 44 & 66 \end{pmatrix}$$

Für 3×3 - oder 4×4 -Matrizen funktioniert das Verfahren genauso, die Ausdrücke werden nur entsprechend länger.

Bei XNA können Sie Matrizen einfach mit dem Multiplikationsoperator `*` oder auch mit der statischen `Multiply`-Methode der `Matrix`-Struktur multiplizieren.

Einheitsmatrix

Die *Einheitsmatrix* oder *Identitätsmatrix* ist diejenige Matrix, die andere Matrizen oder Vektoren bei der Multiplikation unverändert lässt. Sie hat lauter Einsen in der Diagonale und ansonsten nur Nullen. Hier z.B. die 4×4 -Einheitsmatrix:

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

In XNA erhalten Sie über die (statische) `Identity`-Eigenschaft der Struktur `Matrix` eine Einheitsmatrix.

Multiplikation eines Vektors mit einer Matrix

Da Vektoren auch als Matrizen aufgefasst werden können, ist die Multiplikation eines Vektors mit einer Matrix gegenüber dem bisher Gesagten nichts eigentlich Neues.

Bei XNA erfolgt die Multiplikation eines Vektors mit einer Matrix mittels der Methode `Transform` der `Vector3`-Struktur. Mit Hilfe von Matrizenmultiplikationen können Vektoren verschoben, gedreht und skaliert werden, wie im folgenden Abschnitt besprochen.

Inverse Matrix

Die inverse Matrix M^{-1} einer Matrix M ist diejenige, die mit M multipliziert die Einheitsmatrix ergibt.

Bei XNA ermitteln Sie die inverse Matrix mit der statischen Methode `Invert` der Struktur `Matrix`.

Transponierte Matrix

Die transponierte Matrix M^T einer Matrix M ergibt sich dadurch, dass man die Zeilen und Spalten vertauscht.

Beispiel:

Die transponierte Matrix zu

$$\begin{pmatrix} 1 & 12 & 17 \\ 3 & 4 & 8 \\ 1 & 2 & 3 \end{pmatrix}$$

ist

$$\begin{pmatrix} 1 & 3 & 1 \\ 12 & 4 & 2 \\ 17 & 8 & 3 \end{pmatrix}$$

Bei XNA ermitteln Sie die transponierte Matrix mit der statischen Methode `Transpose` der Struktur `Matrix`.

Transformationen

Wenn im Verlauf eines Spiels ein Objekt verschoben, gedreht, vergrößert oder verkleinert wird, müssen die Koordinaten aller Punkte entsprechend dieser Transformation verändert werden, wobei es durchaus

möglich ist, dass auch mehrere Transformationen zusammen auftreten (z.B. eine Drehung und eine Verschiebung). Auch die Betrachtung des Objekts aus unterschiedlichen Entfernungen und Blickwinkeln sowie letztendlich die Projektion auf den Bildschirm erfordern umfangreiche Berechnungen, die für alle Punkte des Modells durchgeführt werden möchten. Glücklicherweise müssen wir nicht in einer Abfolge von Schleifen nacheinander zuerst für alle Punkte die Skalierung berechnen, dann nochmals für alle Punkte die Rotation, danach vielleicht noch für alle Punkte die Drehung und schließlich unter Berücksichtigung der Kamera die Projektion auf den Bildschirm, und das möglichst mehrere hundert Mal in der Sekunde.

Mit Hilfe von Matrizen können wir alle Transformationen, denen ein Objekt unterzogen werden muss, zu einer einzigen Matrix kombinieren, die wir dann, zusammen mit zwei weiteren Matrizen für die Kamera und die Projektion, an die Grafikkarte übergeben – die erledigt dann den Rest.

Homogene Koordinaten

Allein durch die Kombination von Dreiervektoren mit 3×3 -Matrizen lassen sich zwar Drehungen und Skalierungen, aber leider keine Verschiebung eines Punktes realisieren. Wir müssen deshalb unsere Dreiervektoren zu Vierervektoren erweitern, indem die vierte Komponente auf 1 gesetzt wird (bzw. für Richtungsvektoren auf 0, weil für diese eine Verschiebung keinen Sinn macht und durch das Setzen auf 0 gerade der Verschiebungsanteil dann nicht beiträgt, s.u.)

World-Matrix

Die *World-Matrix* eines Objekts ergibt sich durch Kombination der Matrizen für die Position, Orientierung und ggf. eine Skalierung.

Um einen Zeilenvektor mit einer solchen kombinierten Matrix zu transformieren, wird diese *von rechts* mit dem Vektor multipliziert:

$$v' = v \cdot M, \text{ wenn } v \text{ der ursprüngliche und } v' \text{ der neue Vektor ist.}$$

Die Methode `Transform` der `Vector3`-Struktur ergänzt den Vektor mit einer 1 für die vierte Komponente und führt diese Transformation durch.

Position

Die Position eines Objekts wird mit Hilfe einer Translation (Verschiebung) festgelegt. (Ohne Verschiebung liegt es im Koordinatenursprung.)

Eine Translationsmatrix hat in der vierten Zeile die Komponenten des Verschiebungsvektors und sieht ansonsten aus wie eine Einheitsmatrix, also für eine Verschiebung um den Vektor mit den Komponenten (t_x, t_y, t_z) :

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ t_x & t_y & t_z & 1 \end{pmatrix}$$

Da diese bei der Multiplikation gerade mit der 1 in der vierten Zeile des Vektors malgenommen werden, sieht man leicht, dass dabei tatsächlich ein entsprechender Wert zu den Komponenten des Vektors hinzugeaddiert wird.

Die statische Methode `CreateTranslation` der `Matrix`-Struktur erzeugt eine Translationsmatrix.

Rotation

For every thing, turn, turn turn

There is as season, turn, turn turn ...

Die Matrix für eine Rotation (Drehung) ist weitaus komplizierter als die Translationsmatrix und wird für den allgemeinen Fall einer Rotation um beliebige Achsen ziemlich unübersichtlich. Hier sollen aber zumindest die drei Spezialfälle einer Rotation um eine der Koordinatenachsen angegeben werden. Für diese Rotationen werden in der Spieleprogrammierung gern die Begriffe *Pitch*, *Yaw* und *Roll* verwendet: Man kann sich das sehr schön an einem Flugzeug (oder Raumschiff) veranschaulichen:

- *Pitch* bedeutet eine Rotation um die x-Achse (das Flugzeug neigt seine Nase nach oben oder unten).
- *Yaw* ist eine Rotation um die y-Achse (nach rechts oder links).
- *Roll* ist eine Drehung um die z-Achse (beim Flugzeug also die Längsachse – das Flugzeug »legt sich in die Kurve«).

In der Spieleprogrammierung sind für diese Winkel (auch *Euler-Winkel* genannt) die englischen Bezeichnungen üblich. In der deutschen Luftfahrtnorm (DIN 9300) heißt der *Pitch* *Nickwinkel*, der *Yaw* *Gierwinkel* und der *Roll* *Rollwinkel*.

Pitch

Rotation um den Winkel θ um die x-Achse:

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & \sin\theta & 0 \\ 0 & -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Yaw

Rotation um den Winkel θ um die y-Achse:

$$\begin{pmatrix} \cos\theta & 0 & -\sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ \sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Roll

Rotation um den Winkel θ um die z-Achse:

$$\begin{pmatrix} \cos\theta & \sin\theta & 0 & 0 \\ -\sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Die statischen Methoden `CreateRotationX`, `CreateRotationY` und `CreateRotationZ` der `Matrix`-Struktur erzeugen eine Rotationsmatrix. Der Drehwinkel um die jeweilige Achse wird dabei im Bogenmaß angegeben. Eine Winkelangabe in Grad ist dazu mit dem Faktor `/ 180` zu multiplizieren (für den Vollkreis mit Radius 1 ist die Länge des Bogens gerade der Umfang, also 2).

Beispiel

Das folgende Beispielprogramm erlaubt es, Yaw, Pitch und Roll eines Flugzeugs (das allerdings nicht fliegt, sondern nur fest im Raum stehen bleibt) mit den Thumbsticks des Gamepads einzustellen.

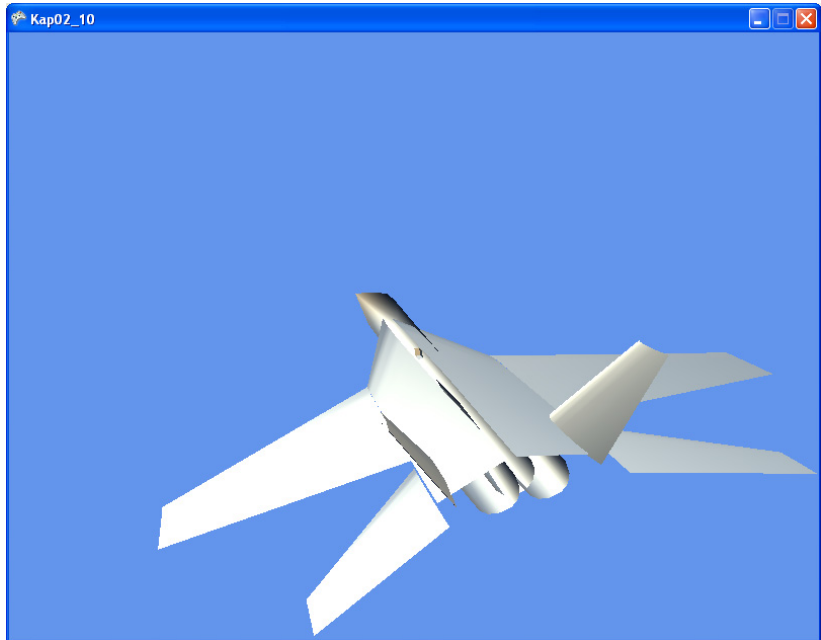
Oben in die `Game`-Klasse fügen wir Variablen für die Kamera, den Flieger selbst, seine `World`-Matrix sowie für die drei Rotationswinkel ein:

Beispiel

Das Beispiel zu diesem Abschnitt finden Sie auf der Begleit-CD im Ordner `KAP2\KAPO2_08`.

```
private Camera camera;  
private Model fighter;  
Matrix world;  
private float yaw, pitch, roll;
```

Abbildung 2.17
Probieren Sie Yaw, Pitch
und Roll einfach aus.



In `Initialize` wird die Kamera initialisiert:

Listing 2.48
`Initialize`

```
protected override void Initialize()  
{  
    camera = new Camera(this);  
    float aspectRatio =  
        (float)GraphicsDevice.Viewport.Width /  
        (float)GraphicsDevice.Viewport.Height;  
    camera.SetAspectRatio(aspectRatio);  
    camera.SetViewParameters(new Vector3(0, 0, 2.5f), Vector3.Zero,  
        Vector3.Up);  
  
    base.Initialize();  
}
```

In `LoadContent` laden wir das Flugzeug:

Listing 2.49
`LoadContent`

```
protected override void LoadContent()  
{  
    fighter = Content.Load<Model>("fighter");  
}
```

In Update werden die Eingabegeräte abgefragt und die World-Matrix wird aus den Rotationswinkeln berechnet:

```
protected override void Update(GameTime gameTime)
{
    // Allows the default game to exit on Xbox 360 and Windows
    if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
        this.Exit();

    camera.Update(gameTime);

    float scale = 0.025f;
    yaw += -scale * GamePad.GetState(PlayerIndex.One).
        ThumbSticks.Left.X * (float)Math.PI / 2.0f;
    pitch += scale * GamePad.GetState(PlayerIndex.One).
        ThumbSticks.Left.Y * (float)Math.PI / 2.0f;
    roll += scale * GamePad.GetState(PlayerIndex.One).
        ThumbSticks.Right.X * (float)Math.PI / 2.0f;

    float speed = (GamePad.GetState(PlayerIndex.One).ThumbSticks.
        Right.Y + 1.0f) / 2.0f;
    GamePad.SetVibration(PlayerIndex.One, speed, speed);

    world =
        Matrix.CreateRotationZ(roll) *
        Matrix.CreateRotationX(pitch) *
        Matrix.CreateRotationY(yaw);

    base.Update(gameTime);
}
```

Listing 2.50

Update

Wie Sie an dem Programmcode sehen können, steuert der linke Thumbstick Yaw und Pitch und der rechte Roll. Spaßeshalber (und weil eine Bewegungsrichtung des rechten Thumbsticks sonst keine Verwendung gehabt hätte) simulieren wir die Motorleistung des Fliegers mit Hilfe der Vibration des Gamepads. Bewegung des rechten Sticks nach oben (vom Spieler weg) erhöht diese, Bewegung nach unten reduziert sie.

Die Draw-Methode bringt nichts wesentlich Neues – dort wird der Flieger mit der eben berechneten World-Matrix gezeichnet:

```
protected override void Draw(GameTime gameTime)
{
    graphics.GraphicsDevice.Clear(Color.CornflowerBlue);

    foreach (ModelMesh mesh in fighter.Meshes)
```

Listing 2.51

Draw

```
{
    foreach (BasicEffect effect in mesh.Effects)
    {
        effect.EnableDefaultLighting();
        effect.View = camera.View;
        effect.Projection = camera.Projection;
        effect.World = world;
    }
    mesh.Draw();
}
```

Wenn Sie sich die verschiedenen Drehachsen noch nicht so recht vorstellen können, laden Sie einfach das Programm von der CD und spielen etwas damit herum. Natürlich können Sie die Belegung der Thumbsticks oder anderer Elemente des Gamepads auch Ihren eigenen Gewohnheiten (z.B. von anderen Spielen) anpassen oder die Steuerung auf Tastatureingabe umprogrammieren. (Kommerzielle Spiele erlauben in der Regel eine individuelle Tastaturbelegung mittels Konfigurationsdateien oder Skripting – hier müssen Sie fürs Erste noch selbst den Quelltext ändern.)

Inverse Matrix

Für Rotationsmatrizen gilt $M^{-1} = M^T$.

Skalierung

Eine Skalierungsmatrix ist wiederum sehr einfach aufgebaut: Sie hat in der Diagonalen die Faktoren, mit denen die einzelnen Komponenten des Vektors multipliziert werden sollen (die also angeben, wie das Objekt in jeder Richtung gestreckt oder gestaucht wird), unten rechts eine Eins und sonst überall Nullen. Für eine Skalierung mit den Faktoren s_x , s_y und s_z sieht das demnach wie folgt aus:

$$\begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Die statische Methode `CreateRotation` der `Matrix`-Struktur erzeugt eine Skalierungsmatrix.

Transponierte Matrix

Für Skalierungsmatrizen gilt $M^T = M$.

Transformationen kombinieren

Beim Zusammensetzen von Transformationen durch Matrizenmultiplikation ist es wichtig, die Reihenfolge zu beachten: Wenn Sie beispielsweise zuerst eine Rotation und dann eine Translation durchführen, rotiert das Objekt (das sich ja zunächst im Koordinatenursprung befindet) um seine eigene Achse und wird dann verschoben. Wenn Sie das Objekt aber zuerst verschieben und dann rotieren, so erfolgt die Rotation immer noch um den Koordinatenursprung (der aber jetzt nicht mehr mit dem Mittelpunkt des Objekts übereinstimmt) – es wird also einen Kreis beschreiben, dessen Radius die Länge des Verschiebungsvektors ist (so, wie sich beispielsweise die Erde um die Sonne dreht).

Noch dramatischer sind die Effekte bei einer Skalierung: Diese sollte normalerweise immer zuerst durchgeführt werden, da sich ansonsten Verzerrungen ergeben.

Bedenken Sie dabei aber auch, dass die Matrizen ja von rechts an die Positionsvektoren der Punkte heranmultipliziert werden. Die am weitesten links stehende Matrix in einer Reihe von Multiplikationen wird also im Endeffekt zuerst auf das Objekt angewandt. Normalerweise (d.h. bei einer Drehung um die eigene Achse) ist die Reihenfolge deshalb wie folgt:

$M_S \cdot M_R \cdot M_T$, wenn M_S die Skalierungsmatrix, M_R die Rotationsmatrix und M_T die Translationsmatrix sind.

Transformation von Richtungsvektoren

Wie bereits erwähnt, können Richtungsvektoren nicht verschoben werden und erhalten deshalb bei der Ergänzung zum Vierervektor nicht eine 1, sondern eine 0 in der vierten Komponente, was dazu führt, dass nur der 3x3-Anteil der Matrix, nicht aber der Verschiebungsanteil in die Multiplikation eingeht. Richtungsvektoren werden außerdem mit der transponierten invertierten Matrix transformiert.

Sie brauchen das normalerweise (außer evtl. beim Schreiben von Shadern) nicht selbst zu berücksichtigen, sondern Sie müssen nur daran denken, Richtungsvektoren (hauptsächlich sind das Normalen) nicht mit `Transform`, sondern mit `TransformNormal` zu transformieren.

Transformationen als Wechsel des Koordinatensystems

Die Verschiebungen, Rotationen und Skalierungen, die bei der Anwendung von Transformationsmatrizen auf die Koordinaten der Vertices unserer Szene angewandt werden, kann man auch als einen Wechsel des

Koordinatensystems verstehen: Wenn beispielsweise ein Punkt, der sich im Ursprung des Koordinatensystems befindet, um fünf Einheiten in die positive z-Richtung verschoben wurde, dann befindet sich dieser Punkt im Ursprung eines anderen, um eben diese fünf Einheiten verschobenen Koordinatensystems. Oder andersherum: Die neuen Koordinaten (0, 0, 5) kann man verstehen als die Koordinaten des verschobenen Punktes im ursprünglichen Koordinatensystem oder aber auch als die Koordinaten des unverschobenen Punktes in einem wiederum anderen, nämlich um fünf Einheiten nach *hinten* verschobenen Koordinatensystem.

Entsprechend ist ein Vektor, der im ursprünglichen Koordinatensystem parallel zu einer Achse verläuft und dann gedreht wird, in einem anderen, nämlich ebenfalls gedrehten Koordinatensystem wieder parallel zur Achse. Die neuen Koordinaten des gedrehten Vektors im ursprünglichen Koordinatensystem sind einfach die Koordinaten desselben Vektors in einem *in die andere Richtung* gedrehten Koordinatensystem.

Das gilt auch für die Skalierung eines Objekts: Man kann diese genauso gut als eine Änderung der Maßeinheit auf den Achsen des Koordinatensystems verstehen.

Jede Transformation ist also im Prinzip nichts anderes als eine Umrechnung der Koordinaten in ein anderes Koordinatensystem. Diese Feststellung erleichtert insbesondere das Verständnis der World-View-Projection-Matrizen, mit denen die Vertices eines 3D-Objekts für die Darstellung auf dem Bildschirm umgerechnet werden.

Die World-Matrix rechnet demnach einfach die Koordinaten der einzelnen Punkte, die zunächst im lokalen Koordinatensystem des Objekts angegeben sind (z.B. relativ zu dessen Mittelpunkt), anhand der Position, Rotation und Skalierung des Objekts um in das globale Welt-Koordinatensystem.

View-Matrix

Die View-Matrix transformiert die Vertices einer Szene in das Koordinatensystem der Kamera (auch als *View Space* bezeichnet). In diesem Koordinatensystem befindet sich die Kamera im Koordinatenursprung und blickt in die negative z-Richtung. Wenn nun die Kamera beispielsweise im Welt-Koordinatensystem um fünf Einheiten in die positive z-Richtung (d.h. aus dem Bildschirm heraus) verschoben ist, dann liegt ein Vertex, der sich im Ursprung des Welt-Koordinatensystems befindet, im Kamera-Koordinatensystem um fünf Einheiten in die *negative* z-Richtung (in den Bildschirm hinein) verschoben.

Dasselbe gilt für die Orientierung von Vektoren: Wenn die Kamera gegenüber dem Welt-Koordinatensystem gedreht ist, dann sind alle Objekte im Kamera-Koordinatensystem um denselben Winkel in die entgegengesetzte Richtung gedreht.

Die View-Matrix ist demnach das Inverse der World-Matrix eines Objekts, das sich genau an der Kameraposition in der Szene befindet und genauso gedreht ist wie diese.

Projection-Matrix

Die Projektionsmatrix sorgt für die *Perspektive* unserer Szene: Objekte, die sich näher an der Kameraposition befinden, erscheinen größer als Objekte, die weiter weg sind. Um diese perspektivische Skalierung zu realisieren, werden die x- und y-Koordinaten aller Objekte einfach durch die zugehörigen z-Koordinaten dividiert. Außerdem werden die z-Koordinaten durch die Projektionsmatrix so skaliert, dass sich die Near Plane (vgl. Abschnitt 2.8) bei $z = 0$ und die Far Plane bei $z = 1$ befinden, wodurch der nächste Schritt in der so genannten *Rendering Pipeline* vereinfacht wird, das *Clipping*. Dabei werden nur diejenigen Vertices gerendert, die sich innerhalb des View Frustrums befinden.

Quaternionen

Quaternionen stellen eine alternative Beschreibung von Rotationen dar (anstelle der Verwendung von Matrizen oder Euler-Winkeln). Ein Quaternion besteht aus vier Einzelkomponenten, denen zwar einzeln keine direkte geometrische Bedeutung zukommt, die zusammengenommen aber eine Achse und eine Rotation um diese Achse darstellen. Wie bei Rotationsmatrizen können auch bei Verwendung von Quaternionen mehrere Rotationen durch einfache Multiplikation kombiniert werden. Der Hauptvorteil von Quaternionen besteht darin, dass sie im Vergleich zu Rotationsmatrizen weniger Speicherplatz verbrauchen (nur 4 statt 14 Fließkommazahlen).

Wichtig ist außerdem, dass man zwischen Quaternionen interpolieren (also kontinuierliche Zwischenwerte zwischen zwei Orientierungen berechnen) kann – sie werden deshalb hauptsächlich für skelettbasierte Animationen eingesetzt, wo sie beispielsweise die Orientierung der einzelnen Knochen eines Skeletts zu jedem Zeitpunkt angeben können.

Die Komponenten eines Quaternions werden als x , y , z und w bezeichnet.

Identität

Auch bei Quaternionen gibt es, wie bei Matrizen die Einheitsmatrix, ein Identitätsobjekt (Einheitsquaternion, wenn Sie so wollen), das bei der Multiplikation mit einem anderen Quaternion dieses unverändert lässt – das also einer Rotation mit dem Winkel 0 entspricht (genau genommen gibt es davon sogar mehrere, aber wir brauchen nur eines zu kennen). Es hat die Komponenten x , y und z gleich 0 und nur w gleich 1.

$$\begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}$$

Die (statische) `Identity`-Eigenschaft der Struktur `Quaternion` liefert ein Identitätsquaternion.

Invertieren

Das Inverse eines Quaternion ergibt mit diesem multipliziert das Identitätsquaternion. Die Komponentendarstellung soll uns hier nicht weiter interessieren; XNA berechnet das Inverse eines Quaternion mit Hilfe der statischen Methode `Inverse`.

Axis-Angle-Darstellung

Wie bereits oben erwähnt, kann man Quaternionen als Darstellung einer Drehung durch eine Achse und einen Winkel auffassen. Leider ist es aber *nicht so*, dass drei der Komponenten die Richtung der Achse und die vierte den Winkel angeben, sondern die Berechnung von Achse und Winkel aus den Komponenten ist komplizierter. Die statische Methode `CreateFromAxisAngle` der Struktur `Quaternion` erlaubt aber, zu einer gegebenen Achse und einem Winkel ein entsprechendes Quaternion zu erstellen.

Umrechnung in Matrix

Mit der statischen Methode `CreateFromQuaternion` der Struktur `Matrix` können Sie aus einem Quaternion eine Rotationsmatrix erzeugen. Umgekehrt erzeugt die Methode `CreateFromRotationMatrix` der Quaternion-Struktur aus einer Rotationsmatrix ein Quaternion.

Multiplikation von Quaternionen

Die Kombination zweier Rotationen durch Multiplikation der entsprechenden Quaternionen können Sie einfach mit dem Multiplikations-

operator `*` oder mit Hilfe der statischen Methode `Multiply` der Quaternion-Struktur durchführen.

Interpolation

Die statische Methode `Lerp` der Struktur `Quaternion` führt die lineare Interpolation zwischen zwei Quaternionen aus.

2.10 Bewegte Objekte

Nach den vorhergehenden Ausführungen über Transformationen und World-Matrix sind wir in der Lage, Objekte beliebig in der Szene zu positionieren und zu bewegen. Der vorliegende Abschnitt stellt dazu einige Anwendungsbeispiele vor.

Das Beispielprogramm

Wir verwenden für alle Beispiele dasselbe Programmgrundgerüst: Ein Raumschiffmodell wird durch die Szene bewegt und unsere wohlbekannte Erdkugel steht als Referenzpunkt fest im Raum.

Die Variablen für beide Modelle werden als Membervariablen in die Game-Klasse eingefügt:

```
public class Game1 : Microsoft.Xna.Framework.Game
{
    GraphicsDeviceManager graphics;

    private Model earth;
    private Model ship;

    private Matrix worldShip = Matrix.CreateTranslation(1, 0, 0);
    private Matrix worldEarth = Matrix.Identity;

    public Game1()
    {
        graphics = new GraphicsDeviceManager(this);
        Content.RootDirectory = "Content";
    }

    //...
}
```

Beispiel

Das Beispiel zu diesem Abschnitt finden Sie auf der Begleit-CD im Ordner `KAP2\KAPO2_09`.

Listing 2.52
Grundgerüst der Klasse `Game1`

In `LoadContent` werden die beiden Modelle geladen:

Listing 2.53
LoadContent

```
protected override void LoadContent()
{
    earth = Content.Load<Model>("earth");
    ship = Content.Load<Model>("raumschiff");
}
```

In Draw zeichnen wir die Modelle:

Listing 2.54
Draw

```
protected override void Draw(GameTime gameTime)
{
    graphics.GraphicsDevice.Clear(Color.Black);

    float aspectRatio =
        (float)GraphicsDevice.Viewport.Width /
        (float)GraphicsDevice.Viewport.Height;
    Matrix projection = Matrix.CreatePerspectiveFieldOfView(
        MathHelper.ToRadians(45.0f), aspectRatio, 1.0f, 1000.0f);
    Matrix view = Matrix.CreateLookAt(new Vector3(0, 0, 10),
        Vector3.Zero, Vector3.Up);

    DrawModel(earth, ref worldEarth, ref view, ref projection);
    DrawModel(ship, ref worldShip, ref view, ref projection);

    base.Draw(gameTime);
}
```

Um den Code etwas übersichtlicher zu halten, haben wir wieder für die Programmteile, die für beide Modelle auszuführen sind, eine Unterfunktion angelegt:

Listing 2.55
DrawModel

```
private void DrawModel(Model model, ref Matrix world, ref Matrix view, ref
Matrix projection)
{
    foreach (ModelMesh mesh in model.Meshes)
    {
        foreach (BasicEffect effect in mesh.Effects)
        {
            effect.EnableDefaultLighting();
            effect.View = view;
            effect.Projection = projection;
            effect.World = world;
        }
        mesh.Draw();
    }
}
```

Die anderen Methoden der Klasse bleiben zunächst unverändert wie vom Assistenten erstellt und werden deshalb hier nicht abgedruckt.

In den folgenden Abschnitten werden wir jeweils die Update-Methode verändern und dabei dieses Grundgerüst beibehalten.

Bewegungen mit konstanter Geschwindigkeit

Um einen Gegenstand zu bewegen, müssen wir offensichtlich seine Position verändern. Die naheliegendste Idee dazu ist vielleicht, diese in jedem Frame um einen gewissen, konstanten Betrag zu verschieben. Das wurde früher auch oft so gemacht, hat aber einen entscheidenden Nachteil: Auf schnellen Systemen mit hoher Framerate läuft die Bewegung dann wesentlich schneller ab als auf langsamen Systemen mit niedriger Framerate. Wenn Sie schon einmal auf einem modernen Rechner ein altes 2D-Zeichentrickspiel gespielt haben, kennen Sie diesen Effekt vielleicht: Alles läuft dann so schnell ab, dass der Spieler praktisch keine Chance mehr hat. Während man das bei Single-Player-Spielen in gewissen Grenzen vielleicht noch tolerieren könnte, hört bei Multi-Player-Spielen der Spaß auf, wenn auf dem Rechner des einen Spielers die Kugel den Gegner trifft, während dieser sich auf dem Rechner des anderen Spielers schon längst woanders befindet.

TotalGameTime und ElapsedGameTime

In modernen Spielen berücksichtigt man aus den oben genannten Gründen bei der Berechnung von Bewegungen die Zeit, die zwischen den einzelnen Frames vergangen ist, und verschiebt beispielsweise Gegenstände um einen Betrag, der sich aus der (konstanten) Geschwindigkeit, multipliziert mit der verstrichenen Zeit, ergibt.

Wenn Sie die Bahnkurve des Objekts als geschlossene Formel angeben können (was ja bei einer konstanten Geschwindigkeit der Fall ist), brauchen Sie genau genommen gar nicht die Verschiebung zu berechnen, sondern rechnen einfach aus der *insgesamt* (seit Beginn des Spiels) vergangenen Zeit die aktuelle Position aus und setzen das Objekt dorthin. Da für verschiedene Anwendungen mal die eine, mal die andere Variante sinnvoller ist, bietet Ihnen XNA beide Möglichkeiten: Das vom Framework an die Update-Methode übergebene `GameTime`-Objekt enthält in den Eigenschaften `ElapsedGameTime` die seit dem letzten Frame verstrichene und in `TotalGameTime` die insgesamt verstrichene Zeit.

Geradlinige Bewegungen

Bei Bewegung mit konstanter Geschwindigkeit in eine feste Richtung brauchen wir nur in jedem Frame zur Position des Objekts einen konstanten Vektor, malgenommen mit der verstrichenen Zeit, hinzuaddieren. Die Länge dieses Vektors bestimmt die Geschwindigkeit, mit der das Objekt sich bewegt, und seine Richtung die Bewegungsrichtung.

Fügen Sie für die Position des Raumschiffs und diesen Geschwindigkeitsvektor je eine Membervariable oben in die Klasse ein:

```
private Vector3 shipPosition = new Vector3(-3, 0, 0);
private Vector3 v = new Vector3(1, 0, 0);
```

Da wir die Zeit in Sekunden (mit Nachkommastellen) angeben, entspricht eine Geschwindigkeit von (1, 0, 0) einer Bewegung von einer Einheit pro Sekunde in x-Richtung.

Fügen Sie jetzt in die Update-Methode (vor dem Aufruf der Basisklassen-Methode `base.Update()`) die folgenden Zeilen ein:

```
float elapsedTime = (float) gameTime.ElapsedGameTime.TotalSeconds;
shipPosition += elapsedTime * v;
worldShip = Matrix.CreateTranslation(shipPosition);
```

Das Raumschiff fliegt von links nach rechts über den Bildschirm (durch die Erde hindurch) und verschwindet nach rechts aus dem Bild.

Umkehrpunkte

Wenn unser Objekt jetzt zwischen zwei bestimmten Punkten hin- und herpendeln soll, müssen wir die Geschwindigkeit irgendwann umkehren. Technisch geht das auch sehr einfach mit der Anweisung $v = -v$.

Nur wann genau soll dieses Umkehren der Geschwindigkeit erfolgen?

Umkehr nach einer bestimmten Zeitspanne

Die einfachste Lösung besteht darin, das Raumschiff immer nach Ablauf einer bestimmten Zeitspanne wenden zu lassen.

Um herauszufinden, wie lange das Schiff bereits in eine bestimmte Richtung unterwegs ist, müssen wir seinen Startzeitpunkt festhalten. Fügen Sie dafür eine weitere Membervariable in die Game-Klasse ein:

```
private float startTime = 0;
```

In `Update` prüfen wir, ob zwischen dem Startzeitpunkt und der aktuellen Game-Time mehr als acht Sekunden verstrichen sind. Wenn ja, keh-

ren wir die Bewegungsrichtung um und setzen die Startzeit auf die aktuelle Zeit.

```
float totalTime = (float)gameTime.TotalGameTime.TotalSeconds;
if (totalTime - startTime > 8.0f)
{
    v = -v;
    startTime = totalTime;
}
```

Umkehr bei einer bestimmten Entfernung

Bei einer konstanten Geschwindigkeit legt unser Raumschiff in einer bestimmten Zeit immer die gleiche Strecke zurück, deshalb entspricht eine Umkehr nach einer bestimmten Zeit einer Umkehr bei einem bestimmten Abstand. Bei beschleunigten oder abgebremsten Bewegungen ist das aber normalerweise nicht der Fall. Wenn Sie trotzdem möchten, dass das Raumschiff genau bei einer bestimmten Distanz wendet, können Sie auch direkt die Entfernung vom Startpunkt messen. Dazu müssen wir natürlich die Startposition in einer Membervariablen festhalten:

```
private Vector3 shipStart = new Vector3(-3, 0, 0);
```

Bei einer Bewegung nur in x-Richtung könnten wir uns damit begnügen, die x-Komponente des Positionsvektors mit derjenigen der Startposition zu vergleichen. Bei einer Bewegung in beliebige Richtung müssen wir, um den Abstand vom Startpunkt zu ermitteln, die Differenz der Vektoren bilden. Die Länge dieses Differenzvektors ist genau der gesuchte Abstand.

Das Raumschiff kehrt jetzt um, wenn es sechs Längeneinheiten zurückgelegt hat:

```
float distance = (shipPosition - shipStart).Length();
if (distance > 8.0f)
{
    v = -v;
    shipStart = shipPosition;
}
```

Wenn Sie beim Umkehren den Startpunkt nicht neu setzen, kehrt das Schiff jedes Mal um, wenn es sechs Längeneinheiten von dieser Anfangsposition entfernt ist. Sie können dann die Anfangsgeschwindigkeit in eine beliebige Richtung zeigen lassen: Das Schiff bleibt immer in einem bestimmten Raumbereich und prallt quasi ab von den

unsichtbaren Wänden einer Kugel mit Radius sechs. So kann man die Szene leicht mit einer Anzahl an Raumschiffen füllen, die scheinbar ziellos umherfliegen, aber trotzdem nicht »verloren gehen«.

Rotationsbewegungen

So wie wir mit Hilfe von Translationsmatrizen Objekte zeitabhängig im Raum verschieben können, so können wir mit Rotationsmatrizen Drehungen ausführen.

Rotation eines Objekts um die eigene Achse

Wie bereits in Abschnitt 2.9 diskutiert, dreht ein Objekt, das zuerst gedreht und dann ggf. noch verschoben wird, sich um seine eigene Achse.

Für eine Drehung mit der Winkelgeschwindigkeit fügen Sie die Membervariable `omega` oben in die Klasse ein:

```
private float omega = MathHelper.TwoPi / 5.0f;
```

Ersetzen Sie dann den Code für die Translationsbewegung in der `Update`-Methode durch den folgenden:

```
float totaltime = (float)gameTime.TotalGameTime.TotalSeconds;
worldShip = Matrix.CreateRotationY(omega * totaltime % MathHelper.TwoPi) *
    Matrix.CreateTranslation(shipPosition);
```

Hinweis

Die Klasse `MathHelper` stellt häufig benutzte Konstanten wie beispielsweise `TwoPi` zur Verfügung. Gegenüber den entsprechenden Konstanten der `.Net-Framework`-Klasse `Math` haben diese den Vorteil, dass sie nicht den Datentyp `double` sondern `float` haben, den wir bei XNA-Anwendungen normalerweise benötigen.

Mit Hilfe des Modulo-Operators `%` begrenzen wir den Winkel auf maximal 2, also eine volle Umdrehung – danach geht es wieder bei 0 los, weil die trigonometrischen Funktionen, die bei der Berechnung der Rotationsmatrix Anwendung finden, für große Winkel ungenau werden.

Natürlich können Sie die Drehbewegung auch mit der zeitabhängigen Veränderung der Schiffsposition kombinieren.

Kreisbahnen

Wenn unser Raumschiff eine Kreisbahn beschreiben soll, müssen wir es zuerst verschieben und dann drehen:

```
float totaltime = (float)gameTime.TotalGameTime.TotalSeconds;
float omega = MathHelper.TwoPi / 5.0f;
worldShip = Matrix.CreateTranslation(shipPosition) *
    Matrix.CreateRotationY(omega * totaltime % MathHelper.TwoPi);
```

Das Schiff kreist jetzt in einem Abstand von drei Einheiten um die Erde.

Beschleunigte Bewegung

Bei einer beschleunigten Bewegung ist die Geschwindigkeit selbst nicht konstant, sondern ändert sich mit der Zeit. Wir wollen zum Beschleunigen und Abbremsen unseres Raumschiffs die Pfeiltasten `Pfeil→` und `Pfeil←` verwenden: Solange die Taste `Pfeil→` gedrückt ist, wird zur Geschwindigkeit in x-Richtung in jedem Frame ein konstanter Betrag hinzuaddiert, wenn dagegen die Taste `Pfeil←` gedrückt ist, ein konstanter Betrag abgezogen. Die aktuelle Geschwindigkeit zeigen wir zur Kontrolle in der Titelleiste des Fensters an.

```
float elapsedTime = (float)gameTime.ElapsedGameTime.TotalSeconds;
shipPosition += elapsedTime * v;
worldShip = Matrix.CreateTranslation(shipPosition);

float distance = (shipPosition - shipStart).Length();
if (distance > 6.0f)
{
    v = -v;
}

KeyboardState keyboardState = Keyboard.GetState();
if (keyboardState.IsKeyDown(Keys.Right))
{
    v += new Vector3(0.1f, 0, 0);
}

if (keyboardState.IsKeyDown(Keys.Left))
{
    v -= new Vector3(0.1f, 0, 0);
}

Window.Title = v.ToString();
```

Wenn das Raumschiff gerade nach rechts fliegt, bewirkt natürlich das Drücken der `Pfeil→`-Taste eine Beschleunigung, fliegt es dagegen gerade nach links, so wird es abgebremst.

Wie Sie sehen, ist die Beschleunigung (der Betrag, um den sich die Geschwindigkeit ändert) selbst ein Vektor. Sie können das Schiff auch in eine andere als die ursprüngliche Bewegungsrichtung beschleunigen und damit seinen Kurs ändern:

```
if (keyboardState.IsKeyDown(Keys.Up))
{
    v += new Vector3(0, 0, -0.1f);
}
```

```
if (keyboardState.IsKeyDown(Keys.Down))
{
    v -= new Vector3(0, 0, -0.1f);
}
```

Die **Pfeil ↑**-Taste beschleunigt das Raumschiff in Richtung der negativen z-Achse, also vom Betrachter weg, die **Pfeil ↓**-Taste in die entgegengesetzte Richtung. (Anstatt einen negativen Wert abzuziehen, kann man natürlich auch einen positiven Wert addieren.)

Damit das Raumschiff nicht zu leicht außer Kontrolle gerät, bauen wir noch die Möglichkeit ein, mit der **Leertaste** die Geschwindigkeit auf 0 zu reduzieren:

```
if (keyboardState.IsKeyDown(Keys.Space))
{
    v = Vector3.Zero;
}
```

Beschleunigte Rotationsbewegungen

Auch Rotationsbewegungen können natürlich beschleunigt oder abgebremst werden, indem man die Winkelgeschwindigkeit verändert. In diesem Fall darf allerdings auch der Drehwinkel nicht aus der insgesamt verstrichenen Zeit berechnet werden, sondern muss, so wie wir es bei der Position getan haben, in jedem Frame um einen von der seit dem vorhergehenden Frame verstrichenen Zeit abhängigen Betrag erhöht werden.

Fügen Sie oben in die Game-Klasse eine Membervariable für den aktuellen Drehwinkel ein:

```
private float angle = 0;
```

In Update wird dieser Winkel entsprechend der aktuellen Winkelgeschwindigkeit verändert:

```
float elapsetime = (float)gameTime.ElapsedGameTime.TotalSeconds;
angle += omega * elapsetime;

worldShip = Matrix.CreateRotationY(angle % MathHelper.TwoPi) *
    Matrix.CreateTranslation(shipPosition);

KeyboardState keyboardState = Keyboard.GetState();
if (keyboardState.IsKeyDown(Keys.Up))
```

```

{
    omega += 0.01f;
}

if (keyboardState.IsKeyDown(Keys.Down))
{
    omega -= 0.01f;
}

```

Um statt einer Rotation um die eigene Achse eine Kreisbewegung durchzuführen, brauchen Sie wieder nur die Reihenfolge der Matrizenmultiplikation zu vertauschen:

```

worldShip = Matrix.CreateTranslation(shipPosition) *
Matrix.CreateRotationY(angle % MathHelper.TwoPi);

```

2.11 Physik

Um physikalische Verhalten von Objekten zu simulieren, genügt es nicht, die Position zu jedem Zeitpunkt aus der Geschwindigkeit bzw. Beschleunigung zu berechnen. Wenn Sie beispielsweise die Beschleunigung eines Raumschiffs aus der Schubkraft des Antriebs und der Masse berechnen oder die Kollision zweier Objekte physikalisch überzeugend darstellen möchten, müssen Sie sich ein paar Grundlagen der Mechanik ins Gedächtnis rufen.

Kraft

Nach Newton ist die Kraft das Produkt aus Masse und Beschleunigung:

$$F = ma$$

Die Kraft, die benötigt wird, um einen Körper zu beschleunigen, hängt demnach von seiner Masse ab bzw. gilt bei einer vorgegebenen Kraft für die erzielte Beschleunigung.

$$a = \frac{F}{m}$$

Diese Gleichungen sind Vektorgleichungen, das heißt, F und a können auch dreidimensionale Vektoren sein.

Energie

Die kinetische Energie eines Objekts, das sich mit Geschwindigkeit v bewegt, ist

$$W = \frac{1}{2}mv^2$$

Kollision und Impulserhaltung

Unter dem *Impuls* versteht man in der Physik das Produkt aus Masse und Geschwindigkeit:

$$p = mv$$

Bei der elastischen Kollision zweier Objekte gilt der Impulserhaltungssatz:

$$m_1v_1 + m_2v_2 = m_1v'_1 + m_2v'_2$$

Wenn die Bewegung nur entlang einer Achse erfolgt, kann man hieraus und aus dem Energieerhaltungssatz

$$\frac{1}{2}m_1v_1^2 + \frac{1}{2}m_2v_2^2 = \frac{1}{2}m_1v'^2_1 + \frac{1}{2}m_2v'^2_2$$

die Geschwindigkeiten beider Objekte nach dem Zusammenstoß berechnen, wenn die Anfangsgeschwindigkeiten und die Massen bekannt sind:

$$v'_1 = \frac{(m_1 - m_2)v_1 + 2m_2v_2}{m_1 + m_2}$$

$$v'_2 = \frac{(m_2 - m_1)v_2 + 2m_1v_1}{m_1 + m_2}$$

Das Beispielprogramm

Das Grundgerüst des Beispielprogramms gleicht demjenigen aus dem vorhergehenden Abschnitt, nur dass wir hier keine Raumschiffe, sondern einfach zwei Kugeln bewegen.

Wir fügen in die Game-Klasse eine Membervariable für das Modell ein (obwohl zwei Kugeln gezeichnet werden, brauchen wir das

Beispiel

Das Beispiel zu diesem Abschnitt finden Sie auf der Begleit-CD im Ordner `KAP2\KAPO2_10`.

Modell nur einmal), außerdem Variablen für die Massen, Positionen und Geschwindigkeiten der Kugeln sowie für die World-Matrizen.

```
public class Game1 : Microsoft.Xna.Framework.Game
{
    GraphicsDeviceManager graphics;

    private Model sphere;

    private float m1 = 1.0f;
    private float m2 = 1.0f;

    private Vector3 pos1;
    private Vector3 pos2;

    private Vector3 v1;
    private Vector3 v2;

    private Matrix world1;
    private Matrix world2;

    //...
}
```

Listing 2.56
Grundgerüst der
Game-Klasse

Das Modell wird wieder in LoadContent geladen:

```
protected override void LoadContent()
{
    sphere = Content.Load<Model>("sphere");
}
```

Listing 2.57
LoadContent

Für die Initialisierung der physikalischen Größen verwenden wir eine eigene Methode Reset, die zunächst im Konstruktor aufgerufen wird:

```
public Game1()
{
    graphics = new GraphicsDeviceManager(this);
    Content.RootDirectory = "Content";
    Reset();
}
```

Listing 2.58
Konstruktor der
Game-Klasse

```
private void Reset()
{
    pos1 = new Vector3(-25, 0, -50);
    pos2 = new Vector3(0, 0, -50);

    v1 = new Vector3(5, 0, 0);
    v2 = new Vector3(0, 0, 0);
}
```

Listing 2.59
Reset

Später können wir diese Methode auch beim Drücken der Leertaste aufrufen, um die Anfangssituation wieder herzustellen:

Listing 2.60
Update

```
protected override void Update(GameTime gameTime)
{
    if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
        this.Exit();

    KeyboardState keyboardState = Keyboard.GetState();
    if (keyboardState.IsKeyDown(Keys.Space))
    {
        Reset();
    }

    //...
    base.Update(gameTime);
}
```

In der Draw-Methode werden wie im Beispiel des vorhergehenden Abschnitts die Modelle unter Verwendung der Methode `DrawModel` gezeichnet:

Listing 2.61
Draw

```
protected override void Draw(GameTime gameTime)
{
    graphics.GraphicsDevice.Clear(Color.CornflowerBlue);

    float aspectRatio =
        (float)GraphicsDevice.Viewport.Width /
        (float)GraphicsDevice.Viewport.Height;
    Matrix projection = Matrix.CreatePerspectiveFieldOfView(
        MathHelper.ToRadians(45.0f), aspectRatio, 1.0f, 1000.0f);
    Matrix view = Matrix.CreateLookAt(new Vector3(0, 0, 10),
        Vector3.Zero, Vector3.Up);

    DrawModel(sphere, ref world1, ref view, ref projection);
    DrawModel(sphere, ref world2, ref view, ref projection);

    base.Draw(gameTime);
}
```

Listing 2.62
DrawModel

```
private void DrawModel(Model model, ref Matrix world, ref Matrix view, ref
Matrix projection)
{
    foreach (ModelMesh mesh in model.Meshes)
    {
        foreach (BasicEffect effect in mesh.Effects)
        {
```

```

        effect.EnableDefaultLighting();
        effect.View = view;
        effect.Projection = projection;
        effect.World = world;
    }
    mesh.Draw();
}
}

```

In Update erfolgt wieder die Aktualisierung der Positionen für die beiden Kugeln. Außerdem prüfen wir, ob eine Kollision eingetreten ist (nämlich dann, wenn der Abstand der Kugeln kleiner ist als der zweifache Radius). Fügen Sie also die folgenden Zeilen in die Methode ein:

```

float elapsedtime = (float)gameTime.ElapsedGameTime.TotalSeconds;

pos1 += elapsedtime * v1;
world1 = Matrix.CreateTranslation(pos1);

pos2 += elapsedtime * v2;
world2 = Matrix.CreateTranslation(pos2);

Collide();

```

Die eigentliche Physik steckt in der Methode Collide, die die neuen Geschwindigkeiten der Kugeln berechnet:

```

private void Collide()
{
    if (Vector3.Distance(pos1, pos2) <=
        2 * sphere.Meshes[0].BoundingSphere.Radius)
    {
        Vector3 v1New = ((m1 - m2) * v1 + 2 * m2 * v2) / (m1 + m2);
        Vector3 v2New = ((m2 - m1) * v2 + 2 * m1 * v1) / (m1 + m2);
        v1 = v1New;
        v2 = v2New;
    }
}

```

Listing 2.63
Collide

Die übrigen Methoden der Klasse, die so bleiben, wie vom Assistenten erstellt, werden auch diesmal wieder nicht mit abgedruckt.

Verändern Sie einmal probeweise die Massen und Anfangsgeschwindigkeiten der Kugeln (in der Reset-Methode). Nur wenn die Massen gleich sind, bleibt beispielsweise die erste Kugel nach dem Stoß liegen.

Kollision bei Bewegung in beliebige Richtungen

Für den dreidimensionalen Fall muss man die Geschwindigkeiten der Objekte zerlegen in einen Anteil in Stoßrichtung und einen Anteil senkrecht dazu.

Da die Impulsübertragung nur entlang des gemeinsamen Normalenvektors der sich berührenden Flächen (also für zwei Kugeln einfach entlang der Verbindungslinie zwischen den beiden Mittelpunkten) erfolgt, gilt für den Anteil der Geschwindigkeiten entlang dieser Verbindungslinie wieder die obige Gleichung, während der Anteil senkrecht dazu unverändert bleibt.

Eine Implementierung finden Sie in Abschnitt 3.7.

2.12 Sound

XACT

Beispiel

Das Beispiel zu diesem Abschnitt finden Sie auf der Begleit-CD im Ordner KAP2\KAPO2_11.

Alle zu einem Spiel zugehörigen Sounds werden in einer .XAP-Datei verwaltet. Sie erstellen eine solche Datei mit Hilfe des MICROSOFT CROSS-PLATFORM AUDIO CREATION TOOL (XACT). Da es sich um einfache Textdateien handelt, können Sie diese auch in einem beliebigen Texteditor anzeigen und bearbeiten.

Erstellen eines XACT-Projekts

Sie finden das XACT-Programm im Startmenü in der MICROSOFT XNA GAME STUDIO EXPRESS-Programmgruppe im Ordner TOOLS. Beim Starten des Programms wird automatisch ein neues Projekt angelegt.

Sie benötigen mindestens eine Sound-Bank und eine Wave-Bank. Erstellen Sie zunächst eine Wave-Bank, indem Sie in der Strukturansicht am linken Bildschirmrand mit der rechten Maustaste auf den Eintrag WAVE BANKS klicken und aus dem Kontextmenü den Befehl NEW WAVE BANK wählen. Im rechten Bildschirmbereich wird ein neues Fenster angezeigt. Erstellen Sie analog eine neue Sound-Bank. Auch dafür wird ein eigenes Fenster angezeigt.

Klicken Sie nun in der Strukturansicht mit der rechten Maustaste auf die neue Wave-Bank und wählen Sie aus dem Kontextmenü den Befehl INSERT WAVE FILE(S).

Fügen Sie die gewünschten Sounddateien (die in einem der Formate .WAV, .AIF oder .AIFF vorliegen müssen) in das Projekt ein.

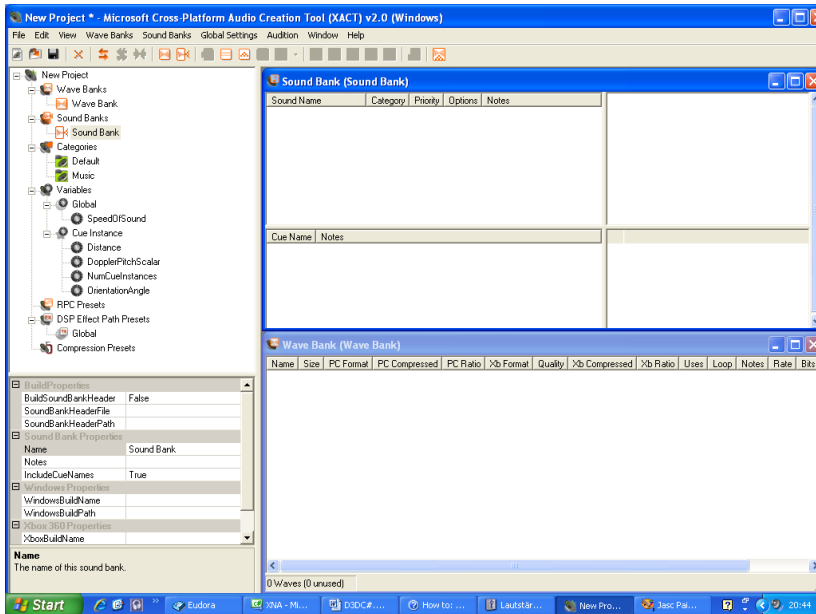


Abbildung 2.18
Das XACT-Werkzeug

Hinweis

Eine Fülle an (teilweise lizenzfreien) Musikstücken finden Sie unter <http://www.artist-server.com>.

Ziehen Sie dann jede der eingefügten Sounddateien mit festgehaltener Maustaste in das Sound-Bank-Fenster und legen Sie sie auf dem CUE NAME-Panel ab.

Sie können das Projekt jetzt speichern und das XACT-Programm beenden. Fügen Sie die .XAP-Datei und die zugehörigen Sounddateien in Ihr Projektverzeichnis ein und fügen Sie im Projektmappen-Explorer die .XAP-Datei zum Projekt hinzu.

In der erzeugten .XAP-Datei finden Sie die Wave-Bank und die Sound-Bank wieder:

```
Wave Bank
{
    Name = Wave Bank;
    Xbox File = Xbox\Wave Bank.xwb;
    Windows File = Win\Wave Bank.xwb;
    Seek Tables = 1;
    Compression Preset Name = <none>;

    Wave
    {
        Name = intro;
```

Listing 2.64
Auszug aus der
.XAP-Datei

```
File = intro.wav;
Build Settings Last Modified Low = 2843233676;
Build Settings Last Modified High = 29843094;

Cache
{
    //...
}

}

Sound Bank
{
    Name = Sound Bank;
    Xbox File = Xbox\Sound Bank.xsb;
    Windows File = Win\Sound Bank.xsb;

    Sound
    {
        Name = intro;
        //...
        Category Entry
        {
            Name = Default;
        }

        Track
        {
            Volume = 0;
            Play Wave Event
            {
                //...
                Wave Entry
                {
                    Bank Name = Wave Bank;
                    Bank Index = 0;
                    Entry Name = intro;
                    //...
                }
            }
        }
    }

    Cue
    {
        Name = intro;
        //...

        Sound Entry
        {
```

```

        Name = intro;
        Index = 0;
        //...
    }
}
}

```

Abspielen von Sounds

Um die Sounds aus der .XAP-Datei abspielen zu können, fügen Sie Membervariablen für die Audio-Engine, die Wave-Bank und die Sound-Bank in die Game-Klasse ein:

```

AudioEngine audioEngine;
WaveBank waveBank;
SoundBank soundBank;

```

In `Initialize` werden diese Variablen initialisiert:

```

protected override void Initialize()
{
    audioEngine = new AudioEngine("Content\\Raumflug.xgs");
    waveBank = new WaveBank(audioEngine, "Content\\Wave Bank.xwb");
    soundBank = new SoundBank(audioEngine, "Content\\Sound Bank.xsb");

    soundBank.PlayCue("intro");

    base.Initialize();
}

```

Listing 2.65
Initialize

Die .XGS-, .XWB- und .XSB-Dateien entstehen beim Erstellen des Projekts. Der Name der .XGS-Datei stimmt normalerweise mit demjenigen des XACT-Projekts überein (hier RAUMFLUG). Der Name der .XWB-Datei ist der Name der Wave-Bank und der Name der .XSB-Datei ist der Name der Sound-Bank.

Keinesfalls dürfen Sie vergessen, in `Update` die `Update`-Methode der Audio-Engine aufzurufen:

```

protected override void Update(GameTime gameTime)
{
    if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
        this.Exit();

    audioEngine.Update();

    base.Update(gameTime);
}

```

Listing 2.66
Update

Falls beim Starten die .WAV-Datei nicht gefunden wird, überprüfen Sie die Zeile

```
File = intro.wav;
```

in der .XAP-Datei – wenn sich die Datei nicht in demselben Verzeichnis befindet wie das Projekt, schreibt das XACT-Tool dort einen absoluten Pfad hinein, was für die Weitergabe des Spiels nicht unbedingt wünschenswert ist. Ändern Sie den Eintrag gegebenenfalls einfach im Texteditor.

2.13 Sprites

Sprites sind 2D-Grafiken, die ohne Berücksichtigung von World-, View- und Projection-Matrizen einfach flach auf den Bildschirm gezeichnet werden (in einem 3D-Spiel normalerweise entweder als Hintergrundbild hinter der Szene oder als Overlay (z.B. Dialogfeld) im Vordergrund). XNA verwendet *SpriteBatch*-Objekte zum Zeichnen solcher Sprites, wobei das Bild selbst einfach eine 2D-Textur ist. Mit einem *SpriteBatch*-Objekt können auch mehrere Bilder gezeichnet werden, wobei Sie jeweils die Position sowie zusätzlich eine Farbe angeben können, die zu den Farbwerten der Textur hinzuaddiert wird. Der Name *Batch* kommt daher, dass zwischen den Aufrufen der *Begin*- und der *End*-Methode des *SpriteBatch*-Objekts die Einstellungen des *GraphicsDevice* erhalten bleiben, so dass mehrere Zeichenvorgänge in einem Stapel (*Batch*) ausgeführt werden, ohne dass dazwischen ein zeitaufwendiges Umschalten dieser Einstellungen erfolgen muss.

Ein Sprite zeichnen

Beispiel

Das Beispiel zu diesem Abschnitt finden Sie auf der Begleit-CD im Ordner KAP2\KAPO2_12.

Als Beispiel soll im Vordergrund der Raumflugszene ein Cockpit angezeigt werden. Da das Cockpit sich natürlich mit dem Raumschiff mitbewegt, wird das Bild immer an derselben Stelle gezeichnet, unabhängig von der Kameraposition.

Fügen Sie oben in die Klasse *Game1* Variablen für die Kamera, für ein Modell sowie für eine Textur ein. Die Variable für das *SpriteBatch*-Objekt hat ja der Assistent bereits angelegt:

```
private Camera camera;
private Model earth;

private Texture2D spriteTexture;
private SpriteBatch spriteBatch;
```

Auch diese Objekte werden in LoadContent geladen:

```
protected override void LoadContent()
{
    earth = Content.Load<Model>("earth");

    spriteBatch = new SpriteBatch(GraphicsDevice);
    spriteTexture = Content.Load<Texture2D>("textures\\cockpit02");
}
```

Listing 2.67
LoadContent

In Draw zeichnen wir zunächst die Szene und dann darüber das Sprite:

```
protected override void Draw(GameTime gameTime)
{
    graphics.GraphicsDevice.Clear(Color.Black);

    foreach (ModelMesh mesh in earth.Meshes)
    {
        foreach (BasicEffect effect in mesh.Effects)
        {
            effect.EnableDefaultLighting();
            effect.View = camera.View;
            effect.Projection = camera.Projection;
            effect.World = Matrix.Identity;
        }
        mesh.Draw();
    }

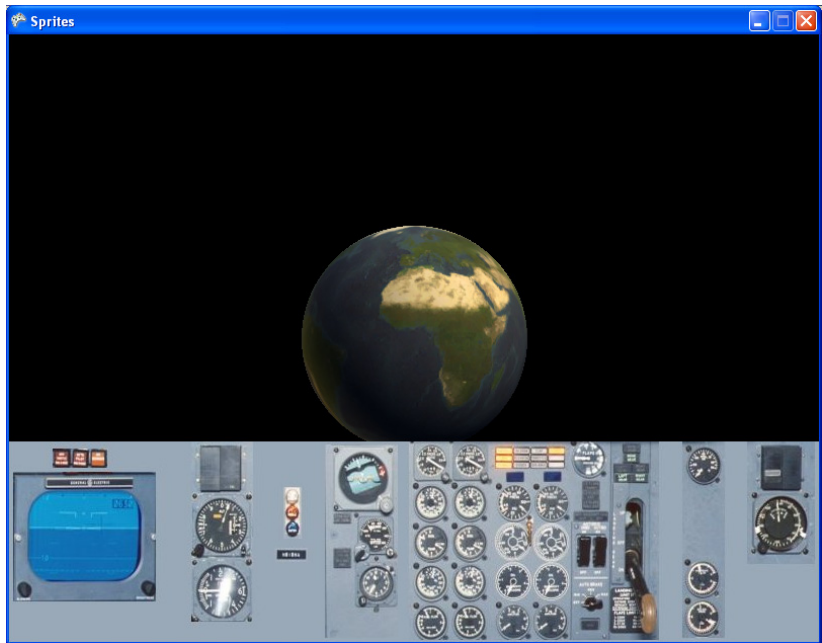
    spriteBatch.Begin();
    Vector2 pos = new Vector2(GraphicsDevice.Viewport.X,
        GraphicsDevice.Viewport.Height - spriteTexture.Height);

    spriteBatch.Draw(spriteTexture, pos, Color.White);
    spriteBatch.End();

    base.Draw(gameTime);
}
```

Listing 2.68
Draw

Abbildung 2.19
Blick aus dem Cockpit
des Raumschiffs



Title-Safe-Bereich

Wenn Sie für die Xbox programmieren, sollten Sie berücksichtigen, dass deren Ausgabe auf einem Fernseher dargestellt wird. Das kann möglicherweise dazu führen, dass ein kleiner Bereich am Rand des Displays nicht sichtbar ist. Wenn Ihr Sprite wichtige Informationen enthält (z.B. Texte), sollten Sie es innerhalb des garantiert sichtbaren Bereichs zeichnen – der so genannten *Title Safe Area*. Ändern Sie die `Draw`-Methode so ab, dass nur innerhalb des »sicheren« Bereichs gezeichnet wird:

```
spriteBatch.Begin();
Rectangle titlesafe = GetTitleSafeArea(.8f);

Vector2 pos = new Vector2(titlesafe.Left,
    titlesafe.Bottom - spriteTexture.Height);
spriteBatch.Draw(spriteTexture, pos, Color.White);
spriteBatch.End();
```

Für die Berechnung des sicheren Rechtecks fügen wir die Methode `GetTitleSafeArea` in die Klasse ein:

Listing 2.69
`GetTitleSafeArea`

```
protected Rectangle GetTitleSafeArea(float percent)
{
```

```

Rectangle retval = new Rectangle(GraphicsDevice.Viewport.X,
    GraphicsDevice.Viewport.Y,
    GraphicsDevice.Viewport.Width,
    GraphicsDevice.Viewport.Height);
#if XBOX
float border = (1 - percent) / 2;
retval.X = (int)(border * retval.Width);
retval.Y = (int)(border * retval.Height);
retval.Width = (int)(percent * retval.Width);
retval.Height = (int)(percent * retval.Height);
return retval;
#else
return retval;
#endif
}

```

Durch die Direktive `#if XBOX` werden die durch dieses `#if` eingeschlossenen Programmteile nur dann verwendet, wenn man das Programm für die Xbox kompiliert.

Parameter von `SpriteBatch.Begin`

Der Methode `Begin` der `SpriteBatch`-Klasse können optional eine Reihe von Parametern übergeben werden, die das Zeichnen der Sprites kontrollieren.

Beispiel:

```
batch.Begin(SpriteBlendMode.AlphaBlend, SpriteSortMode.Texture,
SaveStateMode.SaveState);
```

Teilweise durchsichtige Texturen

Wenn Sie teilweise durchsichtige Sprites zeichnen möchten, müssen Sie den ersten Parameter `BlendMode` auf den Wert `SpriteBlendMode.AlphaBlend` einstellen:

```
spriteBatch.Begin(SpriteBlendMode.AlphaBlend);
```

Natürlich muss die Textur dazu auch einen Alpha-Kanal haben. Wenn beim Zeichnen der Sprites über der Szene einfach nur die Farbwerte von Vordergrund und Hintergrund addiert werden sollen, können Sie `SpriteBlendMode.Additive` verwenden: Dabei erscheinen schwarze Punkte (Farbwert 0) in der Vordergrundtextur durchsichtig, während Punkte, die sowohl im Hintergrund als auch im Vordergrund farbig sind, aufgehellt erscheinen (vgl. auch Abschnitt 2.6). Natürlich sind

Vordergrund und Hintergrund bei diesem Verfahren völlig gleichberechtigt – also erscheinen auch schwarze Bildteile des Hintergrunds durchsichtig, weil dort nur die Vordergrundfarbe zu sehen ist.

Sortieren von Sprites

Der zweite optionale Parameter `sortMode` der `Begin`-Methode gibt an, in welcher Reihenfolge die Sprites vor dem Zeichnen sortiert werden. Erinnern Sie sich in diesem Zusammenhang, dass ein `SpriteBatch`-Objekt zwischen `Begin` und `End` auch mehrere Sprites zeichnen kann. Von der `Draw`-Methode existieren neben der in obigem Beispiel verwendeten mehrere überladene Varianten, bei denen u.a. ein zusätzlicher Parameter `layerDepth` angegeben werden kann. Damit wird dem gezeichneten Bild eine Tiefeninformation zugeordnet (mit Werten zwischen 0 und 1), so dass Sie für überlappende Bilder festlegen können, in welcher Reihenfolge diese gezeichnet werden – wobei natürlich die zuerst gezeichneten von den später gezeichneten teilweise überdeckt werden.

Die möglichen Einstellungen für den Parameter `sortMode` sind:

- `BackToFront`: Sprites mit unterschiedlichen z-Werten (Depth) werden von hinten nach vorn gezeichnet. Empfohlen für teilweise durchsichtige Sprites – wenn das Hintergrundbild durch die durchsichtigen Teile des Vordergrundbilds sichtbar sein soll, muss der Hintergrund zuerst gezeichnet werden.
- `Deferred`: Voreinstellung. Das eigentliche Zeichnen erfolgt erst nach dem Aufruf von `End`, in der Reihenfolge, in der auch die `Draw`-Aufrufe erfolgen.
- `FrontToBack`: Sprites mit unterschiedlichen z-Werten (Depth) werden von vorn nach hinten gezeichnet. Empfohlen für undurchsichtige Sprites – wenn der Vordergrund den Hintergrund verdeckt, brauchen Hintergrundpixel dort, wo sich bereits Vordergrundpixel befinden, gar nicht erst gezeichnet zu werden.
- `Immediate`: Das Zeichnen erfolgt unmittelbar bei jedem `Draw`-Aufruf.
- `Texture`: Sprites werden nach der Textur sortiert. Empfohlen für nicht überlappende Sprites mit gleichen z-Werten – in diesem Fall ist das Ergebnis unabhängig von der Reihenfolge – die Sortierung beeinflusst lediglich die Performance.

Auch bei den Einstellungen `BackToFront`, `FrontToBack` und `Texture` erfolgt das Zeichnen im Gegensatz zur Einstellung `Immediate` erst nach dem Aufruf von `End`.

Speichern des `RenderState`

Der Parameter `SaveStateMode` schließlich kontrolliert, ob der Zustand (`RenderState`) des `GraphicsDevice`-Objekts vor dem Zeichnen gespeichert und nach dem Zeichnen wieder zurückgesetzt wird. Zwar kostet das etwas an Performance, aber gerade beim Zeichnen mit Alpha-Blending kann es sonst leicht passieren, dass man beim Zeichnen des nächsten Objekts vergisst, den `RenderState` wieder zu den Standardeinstellungen zurückzusetzen. Wenn also plötzlich Objekte in der Szene durchsichtig werden oder ohne Z-Buffering gerendert werden, die eigentlich nicht durchsichtig sein dürften, dann sollten Sie immer prüfen, ob Sie vorher ein Sprite gezeichnet haben, ohne dabei den Parameter `SaveStateMode` auf `SaveState` zu setzen.

Rendern in ein Rechteck bestimmter Größe

Bei der oben vorgestellten Version der `Draw`-Methode in der `SpriteBatch`-Klasse haben wir nur die Zielposition angegeben – die Textur wird dann in ihrer Originalgröße gezeichnet. Sie können aber auch als Ziel ein Rechteck angeben, um die Textur auf eine bestimmte Größe zu strecken oder zu stauchen.

Das folgende Codeschnipsel zeichnet eine Textur mit schwarzem Hintergrund mittels `SpriteBlendMode.Additive` in ein vorgegebenes Rechteck hinein (der Rest des Programms bleibt unverändert wie im vorhergehenden Beispiel, bis auf den Namen der Texturdatei in `LoadContent`):

```
spriteBatch.Begin(SpriteBlendMode.Additive);

Vector2 pos = new Vector2(
    GraphicsDevice.Viewport.Width / 2
    - spriteTexture.Width / 2,
    GraphicsDevice.Viewport.Height / 2
    - spriteTexture.Height / 2);

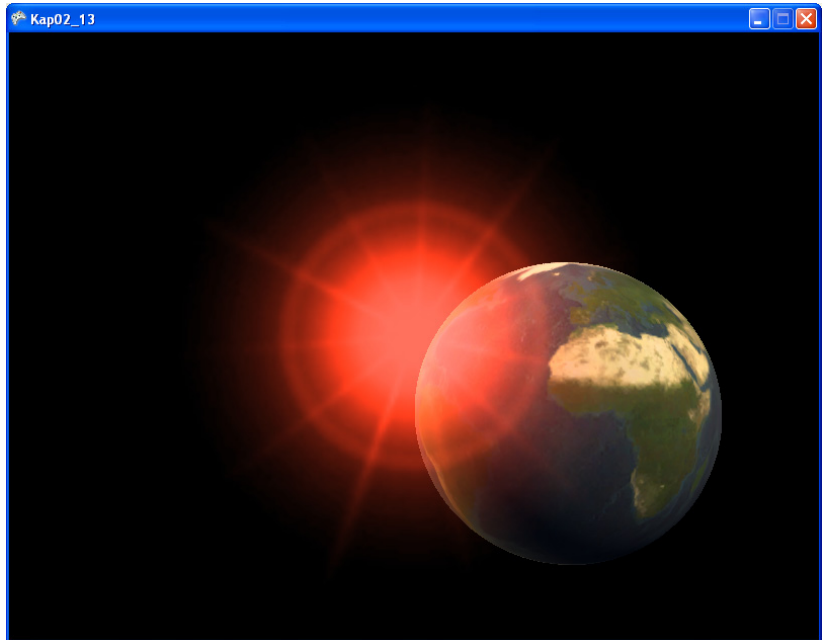
int width = 500;
int height = 500;
Rectangle dest = new Rectangle(
```

Beispiel

Das Beispiel zu diesem Abschnitt finden Sie auf der Begleit-CD im Ordner `KAP2\KAPO2_13`.

```
GraphicsDevice.Viewport.Width / 2 - width / 2,  
GraphicsDevice.Viewport.Height / 2 - height / 2,  
width, height);  
  
spriteBatch.Draw(spriteTexture, dest, Color.White);  
spriteBatch.End();
```

Abbildung 2.20
Sprite mit additivem
Color Blending



2.14 Textausgabe

Beispiel

Das Beispiel zu diesem Abschnitt finden Sie auf der Begleit-CD im Ordner KAP2\KAPO2_14.

Mit Hilfe der Klasse `SpriteFont` können Sie in XNA Text am Bildschirm anzeigen. Sie können dafür alle auf dem System registrierten Fonts (Zeichensätze) verwenden. Informationen zur Verwendung (z.B. die Schriftgröße) werden in XML-Dateien gespeichert, die Sie mit XNA Game Studio Express leicht erzeugen können:

Klicken Sie mit der rechten Maustaste auf den Content-Ordner Ihres Projekts und wählen Sie aus dem Kontextmenü den Befehl HINZUFÜGEN\NEUES ELEMENT. Im Dialogfeld NEUES ELEMENT HINZUFÜGEN klicken Sie dann auf den Eintrag SPRITE FONT.

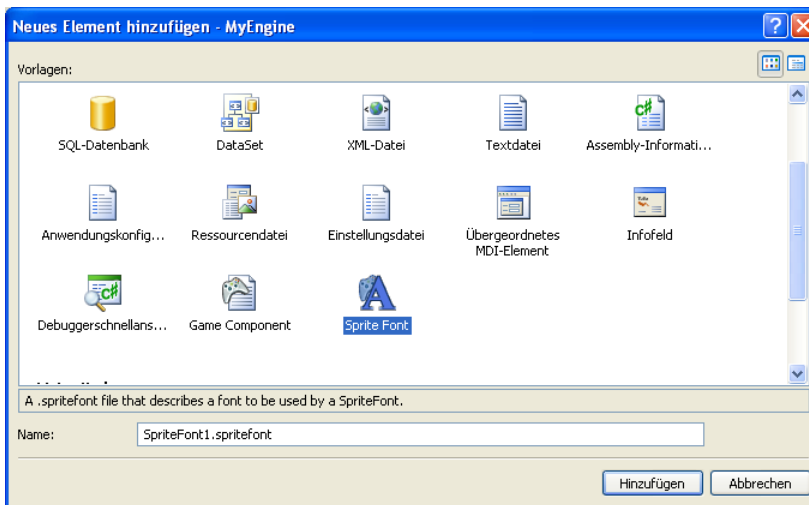


Abbildung 2.21
Einfügen eines
Zeichensatzes

Ändern Sie den vom Assistenten vorgeschlagenen Namen z.B. in ARIAL. SPRITEFONT. Der Asset-Name, unter dem Sie den Font mit dem Contentmanager laden können, wird dann vom Assistenten auf ARIAL eingestellt.

Der Assistent passt auch innerhalb der XML-Datei das Element `<FontName>` entsprechend an. Hier muss der Name eingetragen werden, unter dem der Font auch in der Systemsteuerung (unter SCHRIFTARTEN) geführt wird. Außerdem können Sie in der XML-Datei auch andere Eigenschaften des Zeichensatzes einstellen, z.B. die Schriftgröße.

Das Element `<CharacterRegion>` legt fest, welche Zeichen aus dem Gesamtvorrat des Zeichensatzes überhaupt verfügbar gemacht werden. Voreinstellung ist 32 bis 236 – da sind aber leider die deutschen Umlaute nicht enthalten. Ändern Sie also vorsichtshalber den Wert des `<End>`-Elements auf 255:

```
<XnaContent xmlns:Graphics="Microsoft.Xna.Framework.Content.Pipeline.Graphics">
  <Asset Type="Graphics:FontDescription">

    <FontName>Arial</FontName>
    <Size>14</Size>
    <Spacing>2</Spacing>
    <Style>Regular</Style>

    <CharacterRegions>
      <CharacterRegion>
        <Start>&#32;</Start>
```

Listing 2.70
Font-Definition in der
XML-Datei

```

        <End>&#255;</End>
    </CharacterRegion>
</CharacterRegions>
</Asset>
</XnaContent>

```

Hinweis

In XML-Dateien dienen (wie in HTML-Dateien) die Zeichen <!-- und --> zur Begrenzung von Kommentaren.

Da das Rendern von Text (wie der Name der Klasse `SpriteFont` schon andeutet) mit Hilfe eines Sprites erfolgt, benötigen wir neben dem `SpriteFont` selbst auch noch ein `SpriteBatch`-Objekt. Fügen Sie eine Variable für den `SpriteFont` in die Game-Klasse ein. Die Variable für das `SpriteBatch`-Objekt hat der Assistent bereits eingefügt:

```

private SpriteFont font;
private SpriteBatch spriteBatch;

```

In `LoadContent` wird das `SpriteBatch`-Objekt erstellt und der Font geladen:

Listing 2.71
LoadContent

```

protected override void LoadContent()
{
    spriteBatch = new SpriteBatch(GraphicsDevice);
    font = Content.Load<SpriteFont>("Arial");
}

```

In `Draw` können Sie nun den Text zeichnen:

Listing 2.72
Draw

```

protected override void Draw(GameTime gameTime)
{
    graphics.GraphicsDevice.Clear(Color.CornflowerBlue);

    spriteBatch.Begin(SpriteBlendMode.AlphaBlend,
        SpriteSortMode.Texture, SaveStateMode.SaveState);
    spriteBatch.DrawString(font, "Hallo, Welt", new Vector2(10, 10),
        Color.Black);
    spriteBatch.End();

    base.Draw(gameTime);
}

```