9. Graphische Programmierung

Um eine einfache 3D - Szene auf einem Ausgabegerät darzustellen, sind eine Reihe von Schritten erforderlich, die in diesem Kapitel besprochen werden sollen. Die Erstellung des Modells und die Definition von views werden am Beispiel der 3D - Graphik - API OpenGL und dem Dateiformat VRML erläutert.

9.1. OpenGL

OpenGL (Open Graphics Library) ist eine Programmbibliothek (API) für 3D - Graphik. Es ist aus der GL von Silicon Graphics (SGI) weiterentwickelt worden. OpenGL ist heute ein de facto Standard. Die OpenGL Bibliothek enthält etwa 120 Funktionen (die in Variationen verwendbar sind). OpenGL ist dazu geeignet, realistische Graphiken in Echtzeit zu erzeugen.

OpenGL ist Plattform-unabhängig (unabhängig von einem bestimmten Window System). Es ist so konzipiert, dass auch in einer Netzumgebung effizientes Arbeiten möglich ist. Ein Anwendungsprogramm (Client, z.B. PC) kann auf einem Rechner ausgeführt, und die 3D - Graphik auf einer anderen Maschine (Server, z.B. Hochleistungs-Graphik Workstation) ausgegeben werden.

Da OpenGL kein Modellierungssystem ist, müssen komplexe Modelle aus einfachen graphischen Primitiven aufgebaut werden. Stattdessen ist OpenGL bezüglich seiner Performance optimiert. Aus diesem Grund und wegen der fehlenden Eingabemöglichkeiten wird OpenGL auch als ***Rendering*** System bezeichnet. OpenGL - Funktionen werden "sofort" ausgeführt. Diese Arbeitsweise bezeichnet man als "***immediate mode***" (immediate = sofort). Vergleiche hierzu die Arbeitweise von VRML (Kap. 9.2), wo die graphische Information in einer Datenstruktur abgelegt wird und diese "irgend wann" ausgeführt (interpretiert) wird (***retained mode***; retain = bewahren, festhalten). OpenGL ist nicht objektorientiert konzipiert.

9.1.1 Definition von Primitiven

Die in OpenGL verwendeten ***graphischen Primitive***, sind in Abb. 9.1 dargestellt.

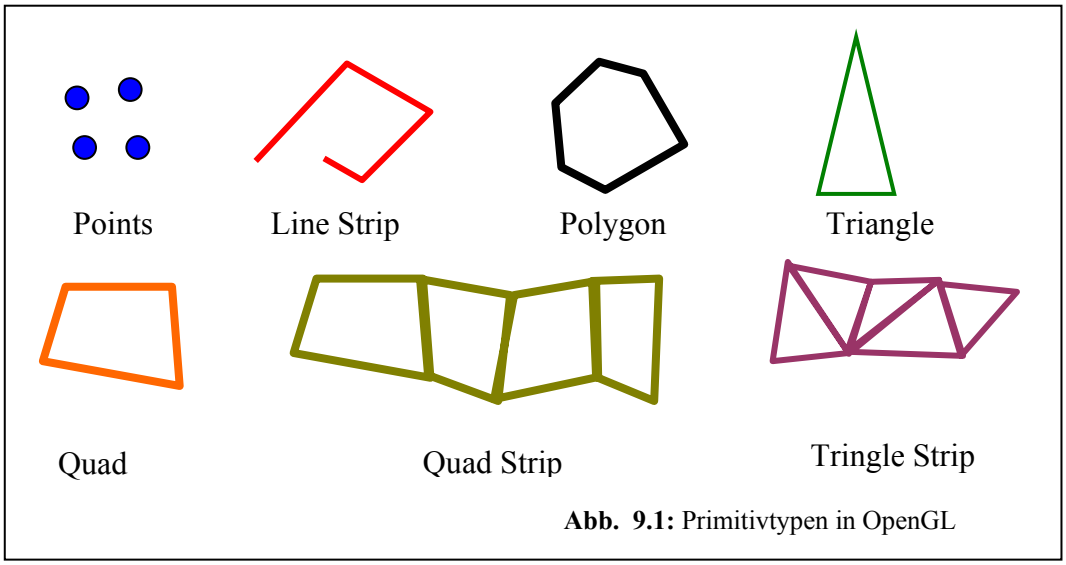


Abb. 9.1: Primitivtypen in OpenGL

Die Definition eines Quadrats, das als Polygon definiert wird, soll die Anwendung veranschaulichen.

Abb. 9.2 zeigt links den hierzu erforderlichen OpenGL -Code. Das Quadrat erhält die Kantenlänge 100. Der Ursprung des Modellierungskoordinatensystems liegt im Mittelpunkt der Fläche. Die Eckpunkte (x,y,z) der 4 Eckpunkte werden in den Funktionen **glVertex3f** definiert. Jede Ecke erhält im Beispiel eine eigene Farbe. Ziel dieses Beispiels ist es, im Folgenden die Hülle eines RGB - Würfels zu erstellen.

```
...
GLfloat KL=100.0f;
...
glBegin ( GL_POLYGON );  //Vorderseite
  glColor4f ( 1.0f, 0.0f, 0.0f, 1.0f ); //ROT
  glVertex3f ( -KL/2.0f, -KL/2.0f, +KL/2.0f );
  glColor4f ( 1.0f, 1.0f, 0.0f, 1.0f );   //GELB
  glVertex3f ( +KL/2.0f, -KL/2.0f, +KL/2.0f );
  glColor4f ( 1.0f, 1.0f, 1.0f, 1.0f );   //WEISS
  glVertex3f ( +KL/2.0f, +KL/2.0f, +KL/2.0f );
  glColor4f ( 1.0f, 0.0f, 1.0f, 1.0f );   //MAGENTA
  glVertex3f ( -KL/2.0f, +KL/2.0f, +KL/2.0f );
glEnd ( );
```

Abb. 9.2: Definition eines Primitivs

Wir wollen nun annehmen, dass in der Funktion *Wuerfel()* auch die übrigen 5 Seiten des RGB-Würfels auf die selbe Weise wie die Vorderseite definiert wurden. Beim Aufruf dieser Funktion beim Rendern würde sich keine Veränderung der Ausgabe gegenüber Abb 9.2 ergeben, da wir den Würfel noch von vorne sehen.

9.1.2 Transformationen

Abb. 9.3: Drehung um die y-Achse

Damit wir sehen, dass der Würfel tatsächlich mehrere Seiten besitzt, drehen wir ihn um 30 Grad um seine y-Achse. Im OpenGL - Programm stellen wir dem Wuerfel - Aufruf eine Rotation voraus:

```
glRotatef ( 30.0f, 0.0f, 1.0f, 0.0f );
Wuerfel ( );
```

Die Parameter 0.0, 1.0, 0.1 im Rotationsaufruf sagen aus, dass die Drehung um einen Vektor mit diesen Koordinaten, also hier um die y-Achse, erfolgen soll. Die ***Transformationshierarchie*** ist in Abb. 9.3 rechts neben dem gerenderten Ergebnis dargestellt.

Abb. 9.4: zusätzliche Drehung um die x-Achse um 20°

Da aus der Darstellung immer noch nicht hervorgeht, dass es sich bei dem Würfel um ein 3D-Objekt handelt, fügen wir eine weitere Drehung um die x-Achse um 20° (Abb. 9.4) hinzu:

```
glRotatef ( 20.0f, 1.0f, 0.0f, 0.0f );
glRotatef ( 30.0f, 0.0f, 1.0f, 0.0f );
Wuerfel ( );
```

Beachten Sie, dass aus der Transformationshierarchie in Abb. 9.4 klar hervorgeht, dass die Drehung um die x-Achse tatsächlich auf das bereits um die y-Achse rotierte Objekt ausgeführt wird. In OpenGL müssen Sie das wie gezeigt lösen. Sie "lesen" dann den Code am besten entweder wie folgt:

- "rotiere um die x-Achse alles was folgt....", oder
- "rotiere um die x-Achse, davor aber um die y-Achse...."

Im nächsten Schritt soll aus dem Würfel ein Quader entstehen, denn am Ende soll ein einfacher Tisch definiert werden. Die Kantenlänge des Würfels war mit 100 ziemlich willkürlich gewählt. Wir wollen dem Tischbein durch Skalierung die Maße 4x4x70 geben. Natürlich hätten wir auch unser Primitiv gleich mit diesen Maßen definieren können.

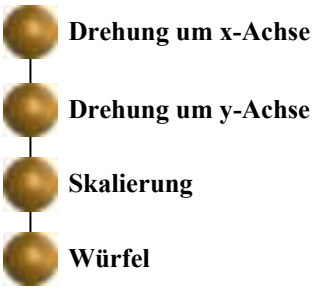


Abb. 9.5: Gedrehter Quader

```
glRotatef ( 20.0f, 1.0f, 0.0f, 0.0f );
glRotatef ( 30.0f, 0.0f, 1.0f, 0.0f );
glScalef ( 0.04f, 0.7f, 0.04f );
Wuerfel ( );
```

Beachten Sie, dass unser Teilmodell nun der (skalierte) Quader (= Tischbein) ist, und dieses Teilmodell um die beiden Achsen (wie gehabt) gedreht wird (Abb. 9.5). Im nächsten Schritt müssen 3 weitere Tischbeine erzeugt und zusammen mit dem ersten im Teilmodell "Tisch" richtig positioniert werden.

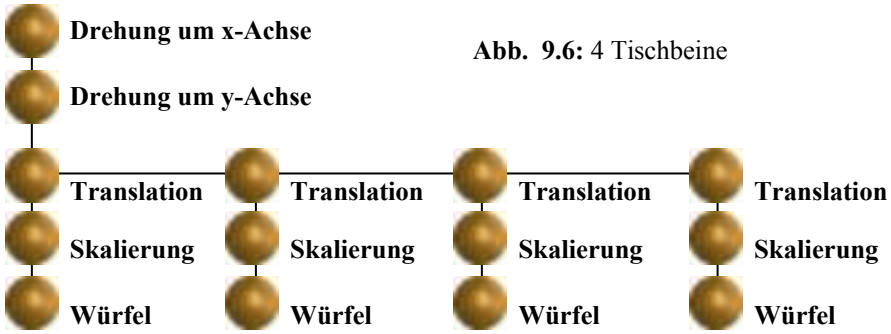
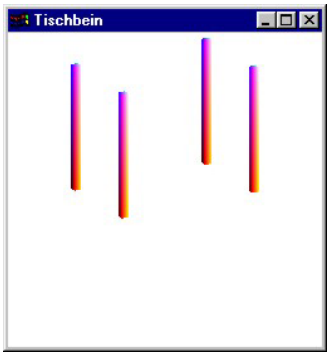


Abb. 9.6: 4 Tischbeine



9.1.3 Entkoppelung von Transformationen in einer Hierarchie

Nebenstehend ist der entsprechende Auszug aus dem OpenGL - Programm gelistet. Folgendes kann diesem Codebeispiel entnommen werden (Vergleichen Sie dazu die Transformationshierarchie in Abb 9.6):

- Die beiden Rotationen werden auf das gesamte Teilmodell "4 Tischbeine" angewendet.
- Jeder der 4 skalierten Quader wird durch eine Translation an die richtige Position verschoben. In y-Richtung wird um die halbe Tischbeinhöhe, nämlich 35 verschoben, damit das Teilmodell schon auf dem gedachten Boden in der Höhe y=0 steht.
- Die Funktion glPushMatrix() "rettet" die Drehung um die beiden Achsen auf den sogenannten ***Matrix - Stack***
- Die Funktion glPopMatrix() holt die zuletzt auf dem Matrix - Stack abgelegte ***Transformationsmatrix*** (s. Kap.10) zurück und macht sie wieder "wirksam".

```
glRotatef (20.0f, 1.0f, 0.0f, 0.0f );
glRotatef (30.0f, 0.0f, 1.0f, 0.0f );
glPushMatrix ( );
glTranslatef (-40.0f, 35.0f, 25.0f );
glScalef (0.04f,  0.7f,  0.04f );
Wuerfel ( );
glPopMatrix ( );
glPushMatrix ( );
glTranslatef (+40.0f, 35.0f, 25.0f );
glScalef (0.04f,  0.7f,  0.04f );
Wuerfel ( );
glPopMatrix ( );
glPushMatrix ( );
glTranslatef (+40.0f, 35.0f, -25.0f );
glScalef (0.04f,  0.7f,  0.04f );
Wuerfel ( );
glPopMatrix ( );
glPushMatrix ( );
glTranslatef (-40.0f, 35.0f, -25.0f );
glScalef (0.04f,  0.7f,  0.04f );
Wuerfel ( );
glPopMatrix ( );
```

- Ohne die Anwendung der beiden Funktionen glPushMatrix() und glPopMatrix() würden sich alle Translationen und Skalierungen "aufaddieren", so wie das ja schon beim einzelnen Tischbein der Fall war! In der Transformationshierarchie können Sie das Zusammenwirken mehrerer Transformationen daran ablesen, dass die Knoten untereinander stehen. Das Aufsetzen auf früheren Zuständen (Entkoppelung) erkennen Sie an der Anordnung der Knoten in waagrechter Richtung.

Nun soll der Tisch auch noch eine geeignete Deckplatte erhalten, oder wie sollte man sonst daran arbeiten? Wir verwenden nochmals den Würfel, skalieren ihn und verschieben ihn an die richtige Stelle. Der Graph für die Transformationshierarchie ändert sich nur dadurch, dass noch 3 weitere Knoten Translation, Skalierung und Würfel rechts angefügt werden. Entsprechend erweitert sich der Code wie folgt:

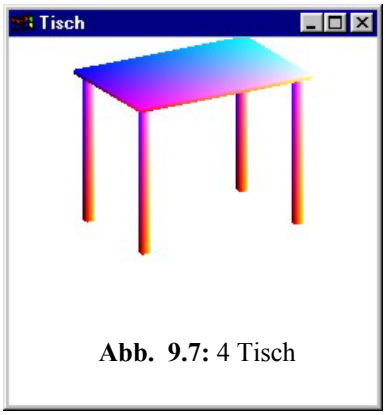


Abb. 9.7: 4 Tisch

```
glPushMatrix ( );
glTranslatef (0.0f, 71.0f, 0.0f );
glScalef (0.9f,  0.02f,  0.6f );
Wuerfel ( );
glPopMatrix ( );
```

Da wir nun entsprechend Abb. 8.1 im Kapitel 8 mehrere Tische erzeugen wollen, fassen wir den obigen Code, bis auf die beiden obersten Rotationen in der Funktion Tisch ( ) zusammen. Der Aufruf lautet nun:

```
glRotatef (20.0f, 1.0f, 0.0f, 0.0f );
glRotatef (30.0f, 0.0f, 1.0f, 0.0f );
Tisch ( );
```

An der Darstellung ändert sich nichts.

```
glRotatef (90.0f, 1.0f, 0.0f, 0.0f );
glScalef (0.25f,  0.25f,  0.25f );

glPushMatrix ( );
glTranslatef (-60.0f, 0.0f, 80.0f );
Tisch ( );
glPopMatrix ( );

glPushMatrix ( );
glTranslatef (+60.0f, 0.0f, 80.0f );
Tisch ( );
glPopMatrix ( );

glPushMatrix ( );
glTranslatef (-180.0f, 0.0f, 0.0f );
glRotatef (-45.0f, 0.0f, 1.0f, 0.0f );
Tisch ( );
glPopMatrix ( );

glPushMatrix ( );
glTranslatef (+180.0f, 0.0f, 0.0f );
glRotatef (+45.0f, 0.0f, 1.0f, 0.0f );
Tisch ( );
glPopMatrix ( );

glPushMatrix ( );
glTranslatef (0.0f, 0.0f, -80.0f );
Tisch ( );
glPopMatrix ( );
```

Abb. 9.8: Tischgruppe von oben

Um die übrigen Tische hinzuzufügen und dies optisch kontrollieren zu können, verkleinern wir die Tische (Skalierung der Szene auf 1/4 der Größe in allen Richtungen) und sehen uns die Szene von oben an (Rotation um die x-Achse um 90 Grad). Darauf folgen die bereits bekannten Sequenzen Translation und Aufruf des Teilmodells, eingerahmt von Push und Pop. Das Ergebnis ist in Abb. 9.8 zu sehen. Die Transformationshierarchie ist in Abb. 9.9 dargestellt, wobei sich hinter "Tisch" die bereits bekannte Hierarchie verbirgt.

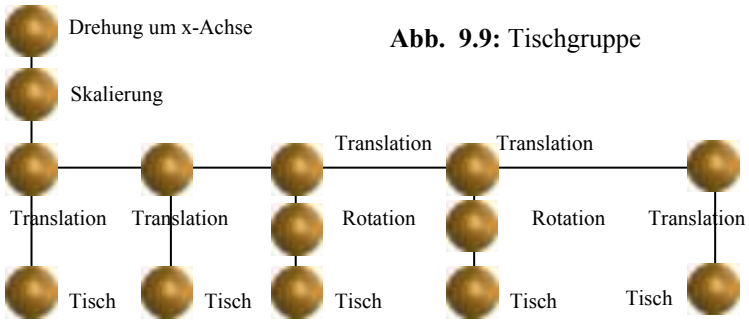


Abb. 9.9: Tischgruppe

9.1.4 Kamera

Skalierung und Rotation, die in Abb. 9.8 den Blick von oben auf die Tischgruppe ermöglichen, stellen logisch gesehen eine Sicht (View) auf das Modell (die Szene) dar. Sie sind ja eigentlich nicht zur Modelldefinition erforderlich. OpenGL stellt Funktionen zur Verfügung, die die Definition einer View und eines Frustums besser unterstützen. Die Anwendung derartiger Funktionen soll im folgenden demonstriert werden.

Abb. 9.10: Kamera von vorne

Werden die beiden erwähnten Rotationen aus dem Code entfernt, so erhält man eine Darstellung der Szene wie in Abbildung 9.10 dargestellt. Dies kommt dadurch zustande, dass im Programm bereits eine Kamera verwendet wurde, ohne dass wir darüber gesprochen haben. Die Einstellung dieser Kamera (***innere Kameraparameter***) beschreibt ein ***Frustum*** und wird über die Funktion **glOrtho** vorgenommen. "Ortho" soll aussagen, dass es sich bei der Projektion um eine ***orthographische Projektion*** (=senkrechte Parallelprojektion) handelt. Diese Projektionsart verzerrt im Gegensatz zu einer ***perspektivischen Projektion*** Objekte nicht. Sie entspricht einer unendlich großen Kamerabrennweite. Das Frustum hat dann die Form eines Quaders, dessen Abmessungen (xlinks, xrechts, yunten, yoben, zvorne, zhinten) sind. In unserem Fall lautet der Aufruf:

```
glOrtho ( -KL*2.5, +KL*2.5, -KL*2.5, +KL*2.5, -40*KL, +40*KL );
```

Durch dieses Frustum wird also ein Ausschnitt der Welt beschrieben, der einen Querschnitt von (5*KL)x(5*KL) Einheiten besitzt und damit die gesamte Szene umfassen kann. Die Tiefe des Frustums ist 80*KL. Für die Orientierung der Kamera wählen wir die Defaulteinstellung von OpenGL. (Orientierung in Richtung der negativen z-Achse des WKS).

Nun soll die Kameraposition und -Orientierung (***äußere Kameraparameter***) also die ***view*** verändert werden. Hierfür können wir die Funktion **gluLookAt** verwenden. Sie besitzt 9 Parameter. Die Gruppe der ersten 3 definiert die (x,y,z)-Position der Kamera (COP). Die Gruppe der nächsten 3 Parameter definiert einen Punkt in der Szene auf den die Kamera gerichtet ist (***Center of Interest*** =CoI). Die 3 folgenden Parameter stellen einen Vektor dar, der die Aufwärtsrichtung der Kamera bezügliche des WKS festlegt. Hierzu ein kleines Beispiel: Stellen Sie sich die Beispielszene in der Realität vor: Ein Raum mit 5 Tischen. Die x- und z-Achsen liegen in der Bodenebene des Raumes. Die y-Achse weist nach oben. Um ein Normalphoto im Querformat zu erhalten, müsste die Kameraaufwärtsrichtung also z.B. durch (0.0, 1.0, 0.0) definiert werden. Soll die Aufnahme auf dem Kopf stehen lautet der Vektor (0.0, -1.0, 0.0), wollen Sie ein Photo im Hochformat, so sollte der Vektor z.B. (1.0, 0.0, 0.0) lauten. Die Funktion *gluLookAt* wird genau an der Stelle des Programms eingefügt, wo zuvor die Skalierung und die Rotation die view definierten.

Mit

```
gluLookAt (0.0, 100.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, -1.0 );   //von oben
```

erhalten wir eine Darstellung wie in Abb 9.11 gezeigt. Die Kamera befindet sich 100 Einheiten über der Tischgruppe.und blickt in den Ursprung des WKS.

gluLookAt (0.0, 180.0, -110.0, 0.0, 80.0, 80.0, 0.0, 1.0, 0.0); //von schräg oben

versetzt den Betrachter (Kamera) hinter den Einzeltisch (-110), führt eine Augenhöhe (180) ein, so dass die Szene von schräg oben aus der Gegenrichtung betrachtet werden kann (Abb. 9.12).

Abb. 9.11: Kamera von oben

Abb. 9.12: Kamera von schräg oben

Abb. 9.13: Weitwinkelobjektiv

Abb. 9.14: Normalobjektiv

In einem letzten Schritt soll die orthographische Kamera nun durch eine perspektivische ersetzt werden. Hierzu ersetzen wir die Funktion glOrtho durch **gluPerspective**. Durch diese wird ein (echtes) ***Frustum*** in Form eines Pyramidenstumpfs definiert. Die Funktion besitzt 4 Parameter. Diese sind der Reihe nach: Ein Winkel (Öffnung von der Spitze aus nach oben gemessen), das Seitenverhältnis (aspect ratio) des Pyramidenquerschnitts, den Abstand der vorderen Clipping - Ebene vom COP (ungleich =0!) und der Abstand der hinteren Clipping - Ebene vom COP.

gluPerspective (90.0, 1.0, 1.5*KL, 40*KL);

liefert Abb. 9.13, in Abb. 9.14 wurde ein kleinerer Öffnungswinkel (60 Grad) gewählt. Bei den beiden letzten Bildern wurde der Betrachterstandpunkt leicht erhöht. Deutlich sind die stärkeren perspektivischen Verzerrungen in Abb. 9.13 gegenüber 9.14 zu erkennen.

9.2. VRML

VRML (**V**irtual **R**eality **M**odeling **L**anguage) verwendet WEB Technologie und wurde entwickelt um 3D Graphik in das Worldwide Web einzubringen. VRML basiert auf der Open Inventor Klassenbibliothek. Die Beschreibung der Szene erfolgt in einem hierarchisch aufgebauten ***Szenegraph***. In der hier verwendeten Notation erscheinen untergeordnete Kindknoten rechts. So sind in dem Szenegraph der Abbildung 9.15 dem **Shape** - Knoten ein **Appearance** - Knoten und ein **Box** - Knoten untergeordnet. Eine Ebene unterhalb des Appearance - Knotens existiert ein Kindknoten **Material**. Das dabei definierte Objekt wird also durch sein Aussehen (appearance) und seine spezielle Form (box = Quader) definiert. Die einem Knoten zugeordneten Werte werden in so genannten Feldern (fields) definiert. Diese sind in den Szenegraphen hier nicht aufgeführt. Diese Felder können z.B. direkte Werte enthalten (im Beispiel Abb. 9.15: **size** mit den Abmessungen 0.04, 07 und 0.04 in x-, y- und z-Richtung) oder sie enthalten Knoten wie das Feld **material** den **Material** - Knoten.

```
#VRML V2.0 utf8
#  Szene01: Das Tischbein weiß
Shape {
  appearance Appearance {
    material Material { }
  }
  geometry Box {size 0.04 0.7 0.04}
}
```

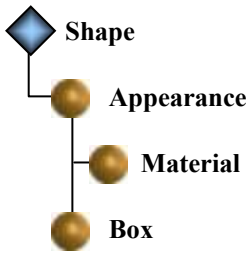


Abb. 9.15: Tischbein

Um einen ersten Eindruck zu vermitteln wie eine 3D - Szene in VRML beschrieben werden kann, soll noch einmal das in Kapitel 9.1 verwendete Beispiel dienen.

Die Konstruktion der Tischgruppe soll wieder bottom - up erfolgen. Die Definition des Tischbeins in den richtigen Abmessungen erfolgt in einem nahezu minimalen VRML - file. Der Szenegraph (Abb. 9.15) enthält lediglich einen **Shape** - Knoten (**shape** = Form). Das Feld **geometry** enthält einen Quader (= **Box**) mit den gewünschten Abmessungen. Eine Farbe ist nicht definiert. Der vordefinierte Standardwert ist ein helles grau (0.8 0.8 0.8).

Soll das Tischbein die Farbe ROT erhalten, wird das Feld **diffuseColor** des Materialknotens wie folgt gesetzt: **material Material { diffuseColor 1.0 0.0 0.0 }.**

```
#VRML V2.0 utf8
#  4 rote Tischbeine

Transform  {
 children [
   Transform {
     translation   -0.4 0.35 0.25
     children
     DEF Tischbein Shape {
       appearance Appearance {
         material Material { diffuseColor 1.0 0.0 0.0}
       }
       geometry Box {size 0.04 0.7 0.04}
     }
   }

 Transform {
  translation  0.4 0.35 0.25
  children
  USE Tischbein
 }

 Transform {
  translation  0.4 0.35 -0.25
  children
   USE Tischbein
 }

 Transform {
   translation -0.4 0.35 -0.25
   children
   USE Tischbein
 }

 ]
}
```

Im nächsten Schritt sollen nun 4 Tischbeine in geeigneten Positionen angeordnet werden. Hierzu werden 4 Translationen benötigt, die in jeweils einem **Tranform** - Knoten definiert werden (s. **translation - Feld** nebenstehend).

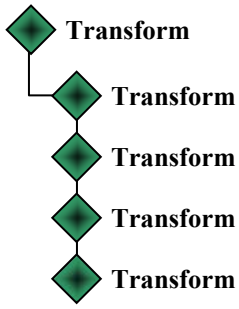


Abb. 9.16: 4 Tischbeine

Der zugehörige Szenegraph ist in Abb. 9.16 dargestellt. Es ist nicht erforderlich das (jeweils gleich aussehende) Tischbein in jedem **Transform** - Knoten erneut zu definieren. Die Definition (**DEF**) erfolgt einmalig im ersten Knoten und wird mit einem Namen (hier: "Tischbein") versehen. In den darauf folgenden Knoten kann nun mit Hilfe von **USE** auf diese Definition zurückgegriffen werden (Exemplare von "Tischbein"). Der oberste **Transform** - Knoten hält die untergeordneten 4 **Transform** - Knoten auf einer Ebene mit Hilfe von **children [...]** zusammen.

DEF und **USE** können nur verwendet werden, wenn alle Exemplare identisches Aussehen besitzen, in unserem Fall, wenn sie in der Größe und Farbe übereinstimmen. Im Folgenden soll gezeigt werden, wie vorgegangen werden kann, wenn diese Voraussetzung nicht gegeben ist. In unserem Beispiel sollen die Tischbeine verschiedene Farben erhalten. VRML stellt für solche Fälle den sog. Prototyp (**PROTO** - Knoten) zur Verfügung. Dort werden Felder festgelegt, deren Werte erst beim Aufruf des Prototyps definiert werden. Hier im

Beispiel ist dies das Feld für die Farbe, das den Namen "tischbeinFarbe" und den Standardwert ROT erhält:

PROTO Tischbein [field SFColor tischbeinFarbe 1.0 0.0 0.0] #Default-Wert

```
#VRML V2.0 utf8
#  Einzeltisch:

PROTO Tischbein   [
        field SFColor tischbeinFarbe 1.0 0.0 0.0
]{
Shape {
appearance Appearance {
material Material { diffuseColor IS tischbeinFarbe }
}
geometry Box {size 0.04 0.7 0.04 }
}
}
```

Der gesamte Prototyp - Knoten, der am Anfang des files eingefügt wurde, sieht dann wie nebenstehend aus.
Der Aufruf des Prototyps findet wie die Verwendung von **USE** im **Transform** - Knoten statt:

```
Transform {
   translation  0.4 0.35 0.25
   children
     Tischbein {tischbeinFarbe
                0.0 1.0 0.0
     }
}
```

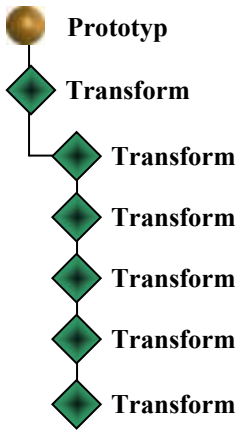


Abb. 9.17: Einzeltisch

Abbildung 9.17 zeigt nun den Szenegraph des kompletten Tisches. Durch Hinzufügen der Tischplatte ist ein weiterer **Transform** - Knoten hinzu gekommen. Dieser muss untergeordnet einen **Shape** - Knoten enthalten, da die Abmessungen des Quaders im Prototyp nicht parametrisiert wurden. 4 der **Transform** - Knoten enthalten Aufrufe des Prototyps.

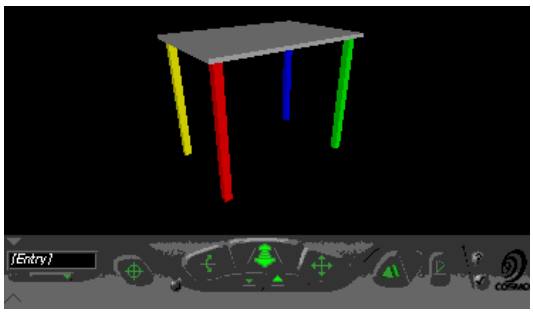


Abb. 9.18: Tisch

Abb. 9.18. zeigt den Tisch (gedreht und vergrößert) wie er durch einen Browser (***retained mode***) dargestellt wird. Das 4. Bein ist (da es im Original blau ist) über dem schwarzen Hintergrund im Druck kaum sichtbar. Beachten Sie die perspektivische Verzerrung.

Es bleiben nun noch 2 Aufgaben zu erledigen. Erstens müssen 5 gleiche Tische in der wie im vorigen Kapitel gezeigten Anordnung positioniert werden. In einem (hier) letzten Schritt sollen verschiedene Views auf die Tischgruppe definiert werden.
Der zuvor definierte Einzeltisch wurde in dem VRML - file "Einzeltisch.wrl" abgespeichert. Auf diesen file kann nun ein link (über das **url** - Feld eines **Inline** - Knotens) definiert werden. Wie die Bezeichnung "url" vermuten lässt kann statt eines lokalen files auch eine beliebige Adresse im Internet stehen.

```
Transform     {
        translation -0.6 0 0.8
        children  [Inline {
          url ["Einzeltisch.wrl"]
        }]
}
```

```
#VRML V2.0 utf8

#  Tischgruppe mit views und Hintergrundfarbe

Background      {
        skyColor            [1 1 1 ]
}

Viewpoint {
        position  0 0 3
        orientation  0 0 1 0
        fieldOfView 0.785398
        description "Von vorne"
}

Viewpoint {
        position  0 10 0
        orientation 1 0 0 -1.571
        fieldOfView 0.785398
        description "Von oben"
}

Viewpoint {
        position  0 0 -10
        orientation  0 1 0  3.14159265359
        fieldOfView  0.785398
        description "Einzeltisch hinten"
}

.....
```

Nebenstehend wurden dem VRML-file nun noch 4 verschiedene Views hinzugefügt. Auf die genaue Bedeutung der Felder im **Viewpoint** - Knoten wird hier nicht eingegangen. Browser erlauben in der Regel einen Wechsel zwischen den unterschiedlichen Ansichten.

Wegen der Druckdarstellung wurde die Hintergrundfarbe weiss gewählt.

Die Ergebnisse der 4 views sind in der Abbildung 9.19 dargestellt. Es wurden dabei jeweils Ausschnitte aus dem Browserfenster gebildet. Betrachterposition und Blickrichtung wurden jedoch nicht geändert.

von vorne

von oben

von vorne (vom Einzeltisch aus)

Abb. 9.19: verschiedene views