7. 3D Grafikprogrammierung

7.1 Einführung

- Datenaufkommen:
  - z.B. 1 Mio. Pixel bei 20 Bilder/Sek. = 20 Mio. Pixel /Sekunde = 60 MB/Sek. (24-Bit/Pixel).
- 3D Grafik:
  - 3D Modell der virtuellen Welt → große Menge von Dreiecken.
  - Daten pro Dreieck: Eckkoordinaten, Materialeigenschaften, Textur.
  - Datenaufkommen, z.B. bis zu 500.000 Dreiecke bei modernen Spielen.
- Aufgaben der Bildgenerierung:
  - Projektion der Dreiecke ins Bild (Perspektive),
  - Verdeckungsberechnung im Bild,
  - Rasterisierung der Dreiecke,
  - Beleuchtungsberechnung,
  - Texturierung, ...

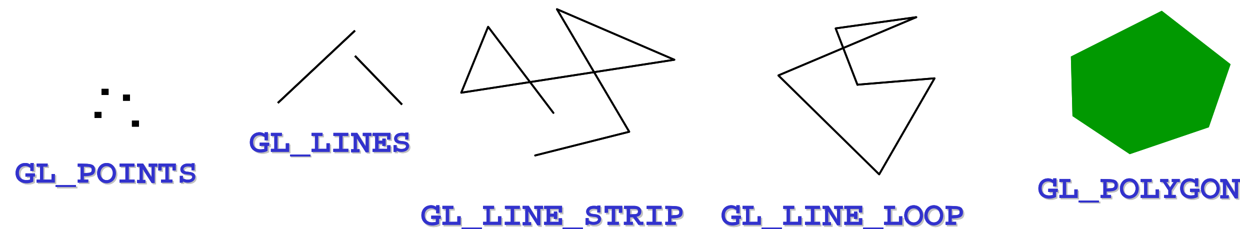
- Grafikhardware
  - Moore‘s law: Geschwindigkeit von CPUs verdoppelt sich ca. alle 18 Monate.
  - Geschwindigkeit von GPUs hat sich ca. alle 6 Monate verdoppelt.
  - z.B. ATI Radeon 9700 bis zu 100 Mio. Dreiecke/Sekunde.

7.2 Allgemeins zu OpenGL

- OpenGL = **Open** **G**raphics **L**ibrary von SGI (Silicon Graphics Inc).
- Informationen & Tutorials: www.opengl.org, nehe.gamedev.net, ...
- Industriestandard, aktuelle Version 2.0.
- 2D/3D Grafik-Bibliothek:
  - Low-Level API für direkten Hardware-Zugriff,
  - für verschiedenste Sprachen: C, C++, Java, .NET, ...
  - Abstraktion von Grafik-Hardware und Betriebssystem.
- Beschränkt auf einfache Grafikprimitive: Punkte, Linien, Polygone, ...
- Zustandsmaschine:
  - Operationen abhängig von Zustandsvariablen,
  - z.B. Zeichenattribute, Rendermodus, ...
- Keine Eingabeschnittstellen vorhanden.
- Zusatzbibliotheken:
  - GLU = OpenGL Utility Library: Darstellung komplexer Objekte (z.B. Kugeln, Zylinder).
  - GLUT = OpenGL Utility Toolkit: Funktionen zur Fenster- und Ereignisbehandlung.

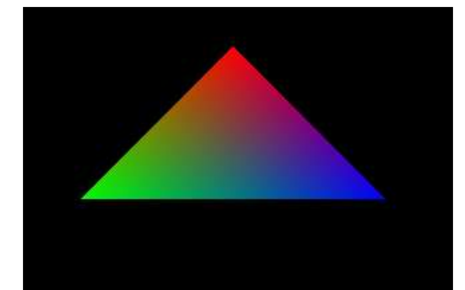
7.3 Grafikprimitve

- Klammerung von Raumpunkten durch: *glBegin(primType) & glEnd()*.
  - *primType:* best. den Typ des Grafik-Primitves u. den Verbund der folgenden Punkte (vertices).
  - Beispiele:



- Farbe eines Raumpunktes bestimmt durch:
  - Farbattribut: *glColor3ub(r,g,b)*, 3 Bytes bei 24 Bit Farbtiefe.
  - Transparenzwert (optional): *glColor4f(r,g,b,alpha)*, alpha: 0.0=transparent, 1.0=opak.
- Farbfüllmodus mit *glShadeModel(mode)*:
  - GL_SMOOTH: Farbverlauf (Standard), GL_FLAT: Primitv erhält Farbe eines Raumpunktes.
- Beispiel:

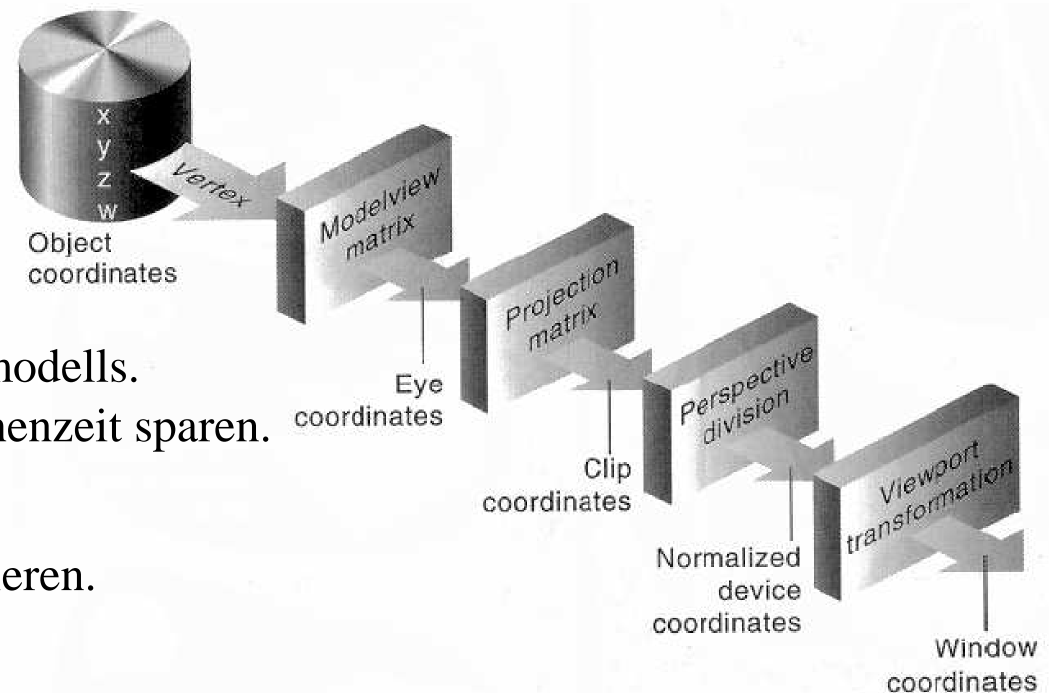
```
glShadeModel (GL_SMOOTH);
glBegin (GL_TRIANGLES);
  glColor3ub(255, 0, 0);    glVertex3f(0.0f, 1.0f, 0.0f);
  glColor3ub(0, 255, 0);    glVertex3f(-1.0f, 0.0f, 0.0f);
  glColor3ub(0, 0, 255);    glVertex3f(1.0f, 0.0f, 0.0f);
glEnd();
```



7.4 OpenGL Viewing Pipeline

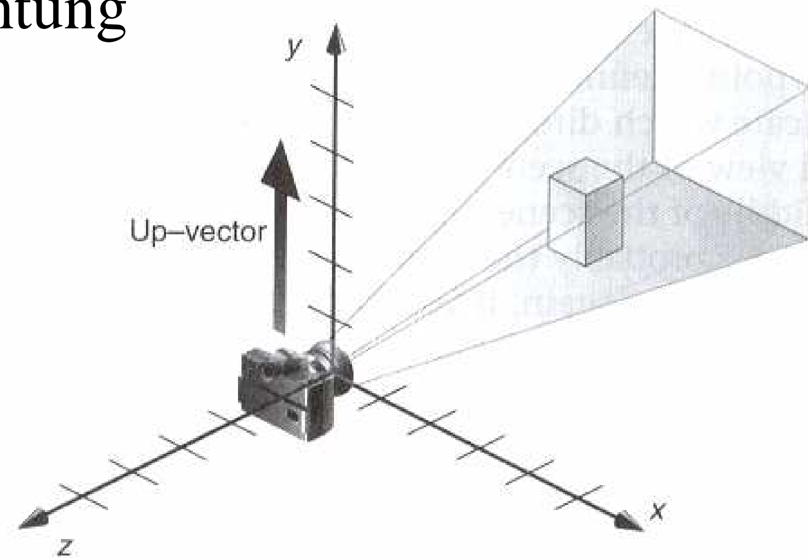
7.4.1 Überblick

- Objekte definiert aus Menge von Vektoren (Raumpunkte, vertices).
- Rendern: Umsetzung der 3D Szenerie auf 2D Bildschirm.
- Szenerie-Transformationen:
  - Viewing: Kameraposition,
  - Modeling: Objektmanipulation,
  - Projection: 3D-Sichtbarkeitsbereich.
- Clipping:
  - Verkleinern des zu rendernden Datenmodells.
  - unsichtbare Objekte auslassen → Rechenzeit sparen.
- Perspektivische Division:
  - Koordinaten des 2D Abbilds normalisieren.
  - auf den Wertebereich: -1.0 bis +1.0.
- Viewport-Transformation:
  - Skalierung des 2D Abbilds auf den Viewport,
  - sichtbarer Bereich (Größe & Position) auf dem Bildschirm.



7.4.2 Viewing Transformation

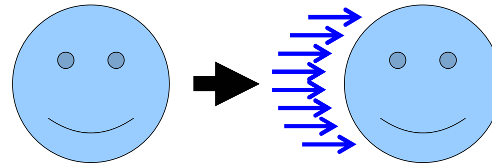
- Ändern von Position und Richtung der Kamera (view point).
- Relevante Matrix: Modelview-Matrix.
- Manipulation durch *gluLookAt()* Funktion.
- Parameter:
  - *eyeX, eyeY, eyeZ*: Position des "view point".
  - *centerX, centerY, centerZ*: Punkt i.d.R. im Zentrum der Szene, durch den die Blickrichtung ("line of sight") des view points spezifiziert wird.
  - *upX, upY, upZ*: Orientierungsvektor bestimmt wo oben in der Szenerie ist. Die Kamera wird so gedreht, dass der Vektor senkrecht über der Kamera steht.
- Beispiel: Standardposition und -ausrichtung
  - void gluLookAt (0.0, 0.0, 0.0,
    0.0, 0.0, -1.0,
    0.0, 1.0, 0.0
    );



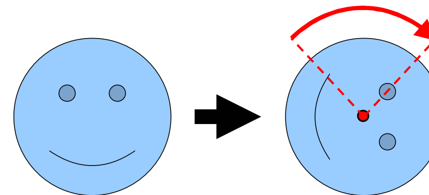
- Alternativ: Modeling Transformation entsprechend durchführen.

7.4.3 Modeling Transformation

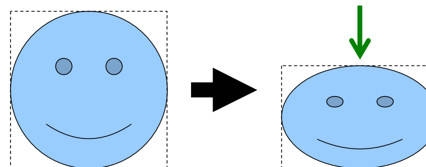
- Ändern von Position, Ausrichtung und Form der zu rendernden Objekte.
- Relevante Matrix: Modelview-Matrix.
- Translation (Verschiebung):
  - Befehl: *glTranslate3f(x,y,z)*,
  - Param.: Verschiebungsvektor.



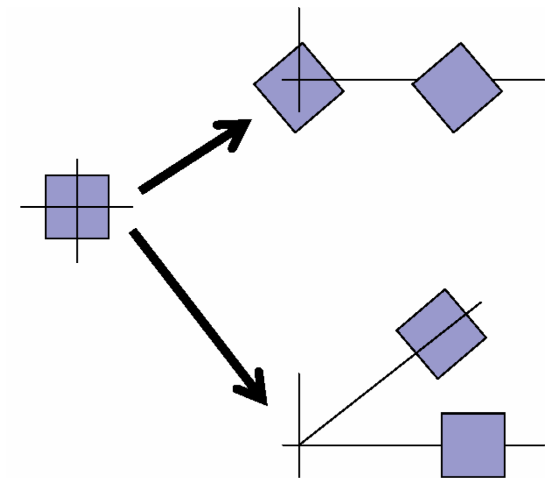
- Rotation:
  - Befehl: *glRotate3f(alpha,x,y,z)*,
  - Param.: Winkel & Drehachse (Ortsvektor).



- Skalierung:
  - Befehl: *glScale(x,y,z)*.
  - Param.: Skalierungsfaktoren.

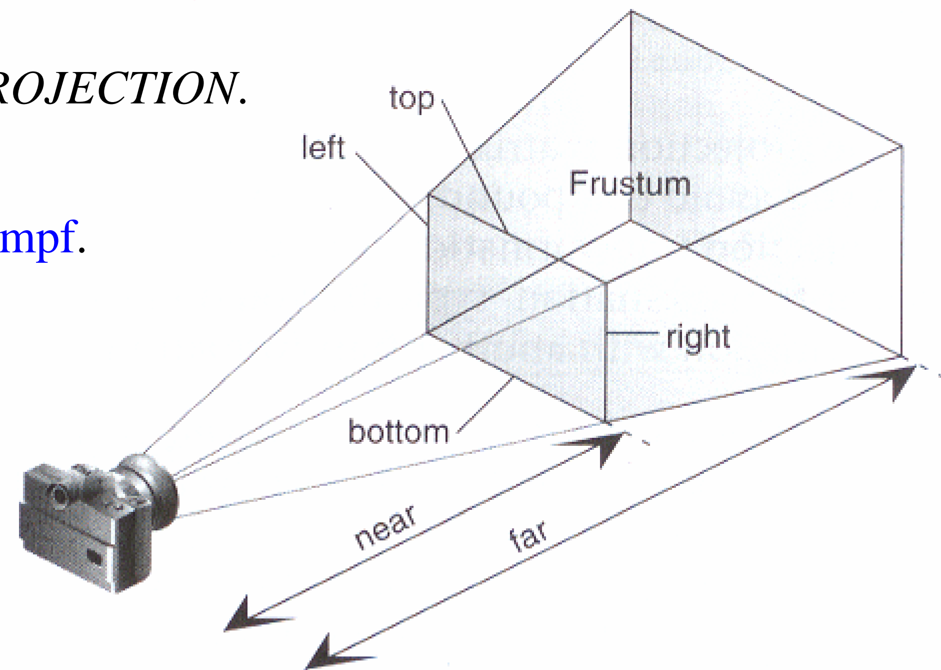


- Transformationen sind nicht kommutativ:
  - Beispiel: 2D Drehung um den Ursprung + Translation.
  - Fall 1 (oben): rotieren und anschließend verschieben.
  - Fall 2 (unten): verschieben und dann rotieren.

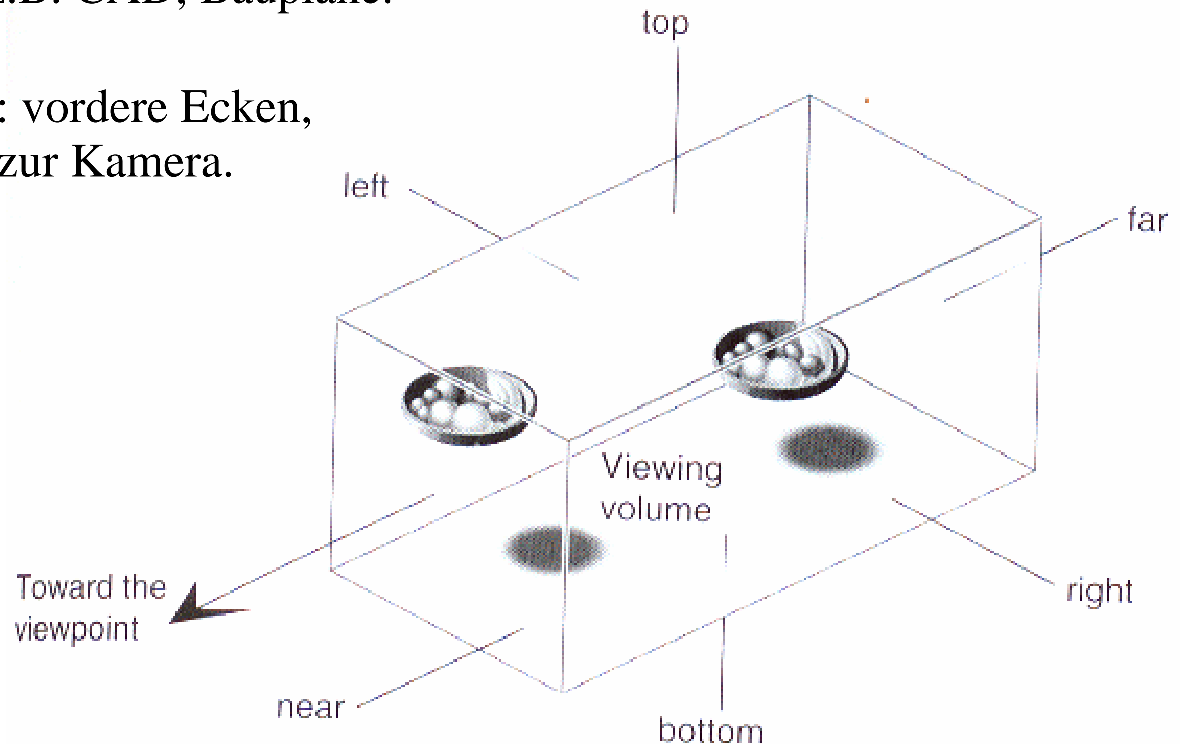


7.4.4 Projection Transformation

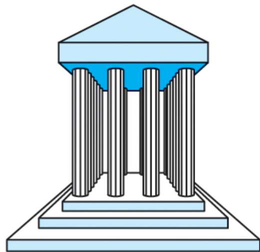
- Grösse, Form, Position und Projektions-Weise des Viewing Volumes.
- Viewing Volume:
  - Zusammenhängender Bereich des 3D-Koordinatensystems, der sichtbar dargestellt wird.
  - Nur Objekte(-Teile), die sich innerhalb dieses Bereichs befinden, werden gerendert.
  - Die Seiten des Viewing Volumes werden als "clipping planes" bezeichnet.
  - Durch die Form des Viewing Volume wird festgelegt, wie die im Viewing Volume enthaltenen Objekte auf das zu erstellende 2D-Bild projiziert werden.
- Relevante Matrix: Projection-Matrix.
  - Auswahl der Matrix mit *glMatrixMode()*.
  - Parameter: *GL_MODELVIEW* und *GL_PROJECTION*.
- Perspektivische Projektion:
  - Form des Viewing Volume: Pyramidenstumpf.
  - Charakteristik: Größe der Objekte nimmt mit der Entfernung zur Kamera ab.
  - Für Spiele und virtuelle Welten.
  - Fkt.: *glFrustum()*, Param.:
    left,right,bottom,top: vordere Ecken,
    near, far: Abstände zur Kamera.
  - Alternativ: *gluPerspective()*.



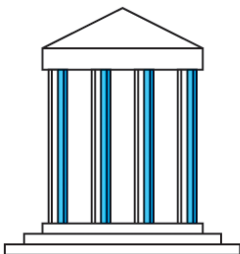
- Orthogonale Projektion:
  - Form des Viewing Volume: Quader.
  - Charakteristik: Größe der Objekte ändert sich nicht mit der Entfernung zur Kamera.
  - Anwendungsgebiete: Szenen, bei denen es wichtig ist, die tatsächlichen Grössen und Winkel der Objekte zu sehen. z.B. CAD, Baupläne.
  - Funktion: *glOrtho*
  - Param.: *left,right,bottom,top*: vordere Ecken,
    near, far: Abstände zur Kamera.



- Projektsbeispiele:
  - perspektivisch:

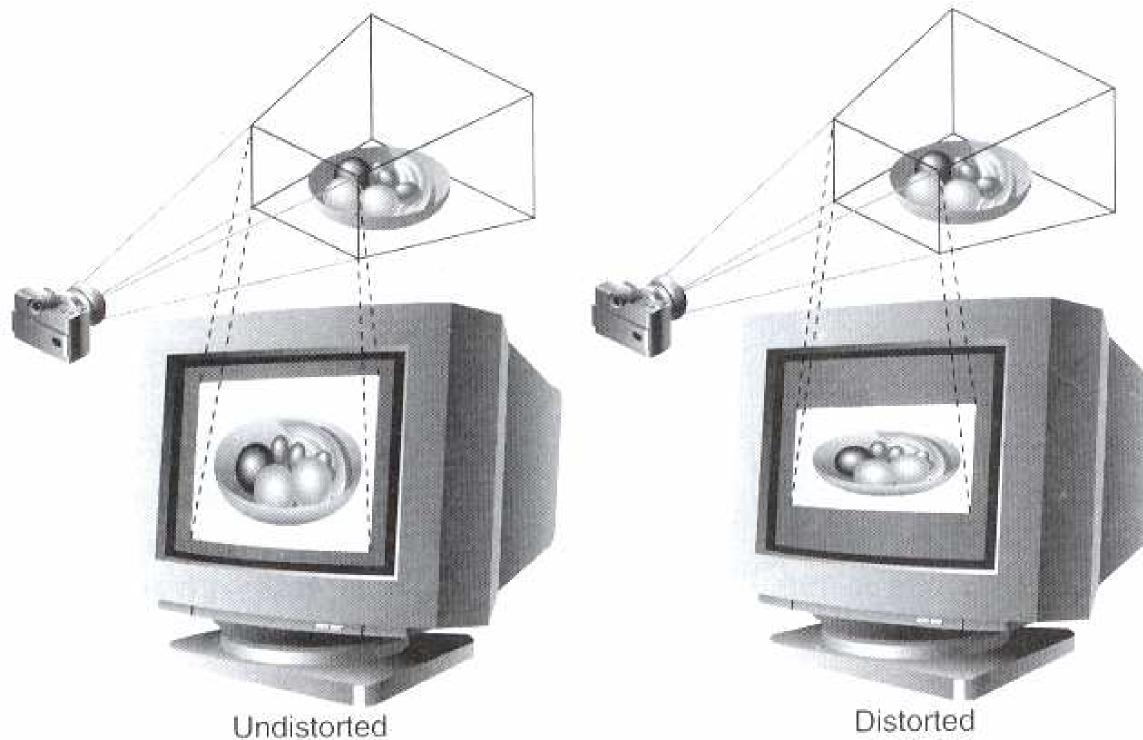


  - orthogonal:



7.4.5 Viewport Transformation

- Transformieren der Raumkoordinaten der zu rendernden Objekte.
- Relevante Matrix: Viewport-Matrix.
- Funktion: *glViewport (x,y,w,h)*: definiert Pixelrechteck innerhalb eines Betriebssystemfensters.
- Bem.: wird beim Viewport ein anderes Seitenverhältnis als beim Viewing Volume verwendet, so wird die Szene verzerrt:



7.4.6 Beispielfragment: rotes Dreieck

- Java mit freier Bibliothek *gl4java* (www.jausoft.com/gl4java.html).

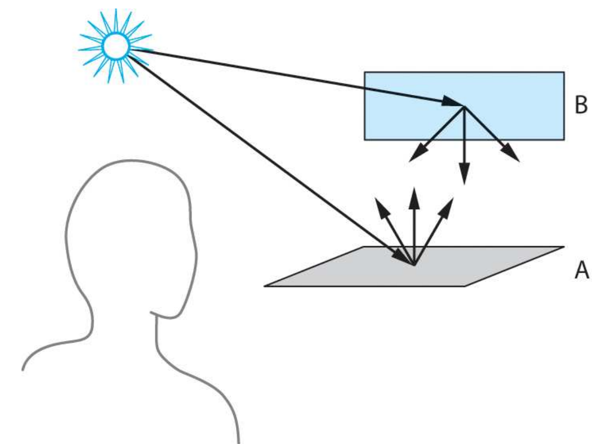
```
public class NeHeCanvas extends GLAnimCanvas implements KeyListener, MouseListener {
public void init() {
    gl.glClearColor(0.0f, 0.0f, 0.0f, 0.0f);                // background color = black
    gl.glClearDepth(1.0);                                   // clear value for depth buffer
    gl.glEnable(GL_DEPTH_TEST);                             // enables depth testing
    gl.glDepthFunc(GL_LEQUAL);                              // condition for a pixel to be drawn
}
public void reshape(int width, int height) {
        gl.glViewport(0, 0, width, height);                 // set viewport
        gl.glMatrixMode(GL_PROJECTION); gl.glLoadIdentity(); // reset projection matrix
        gl.glFrustum(-1.0f, 1.0f, -1.0f, +1.0f, 1.0f, 100.0f) // projection
        gl.glMatrixMode(GL_MODELVIEW); gl.glLoadIdentity();// reset the modelview matrix
}
public void DrawScene() {
        gl.glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);   // clear screen & depth buf
        gl.glLoadIdentity();                                // reset the view
        gl.glTranslatef(-0.5f,0.0f,-2.0f);                  // move
        gl.glBegin ( GL_TRIANGLES );                        // red triangle
            gl.glColor3f(1.0f, 0.0f, 0.0f);
            gl.glVertex3f(-0.5f,-0.5f, 0.5f); gl.glVertex3f( 0.5f,-0.5f, 0.5f); gl.glVertex3f( 0.5f, 0.5f, 0.5f);
        gl.glEnd ();
}}
```


Virtuelle Präsenz, Winter 04/05 © Verteilet Systeme, Universität Ulm, M. Schöttner

7.5 Beleuchtung in OpenGL

7.5.1 Einführung

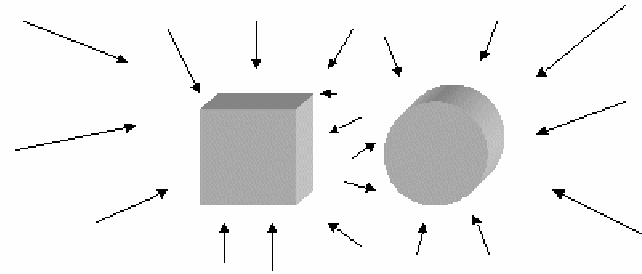
- Pixelfarbe bestimmt sich aus der Summe der Beiträge aller Lichtquellen:
  - Position und Eigenschaften der Lichtquellen,
  - Reflexionseigenschaften der Oberflächen,
  - Kameraposition ebenfalls relevant.
- Beleuchtungsmodelle:
  - lokal: nur direkten Lichteinfall auf Oberflächen berücksichtigen (OpenGL),
  - global: zusätzl. Mehrfachreflexionen, Lichtbrechung & Schatten berücksichtigen (RayTracer).
- Lichtquellen:
  - unsichtbar,
  - acht Stück vorhanden,
  - müssen explizit eingeschaltet werden,
  - Position durch Transformation änderbar.



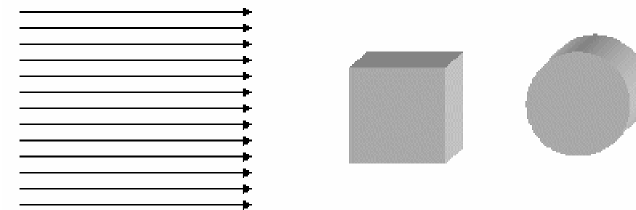
- Programmierschritte:
  - Lichtquellen: Position, Eigenschaften, einschalten,
  - Objekte: Normalenvektoren und Materialparameter definieren.
  - Beleuchtungsmodell konfigurieren (optional), z.B. Grundhelligkeit der Szene.

7.5.2 Lichtquellenmodelle

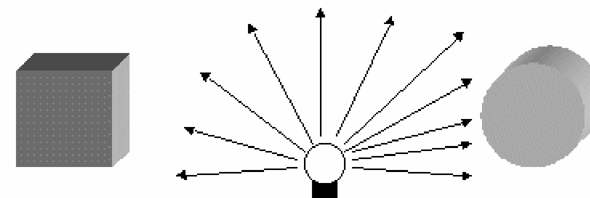
- Ambientes Licht:
  - Grundhelligkeit der Szene,
  - Ursprung und Richtung sind nicht definiert,
  - gleichmässig verteilt mit konstanter Intensität.



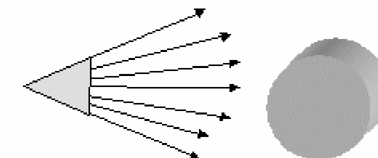
- Gerichtetes Licht:
  - definierte Richtung,
  - Ursprung liegt in unendlicher, Entfernung, daher parallele Lichtstrahlen,
  - Einfallswinkel und Intensität sind konstant.



- Punktlicht:
  - Lichtquellle mit fester Position in der Szene,
  - gleichmässige Emission in alle Richtungen,
  - Intensität nimmt mit der Entfernung ab,
  - unterschiedliche Einfallswinkel für verschiedene Punkte der Szene.

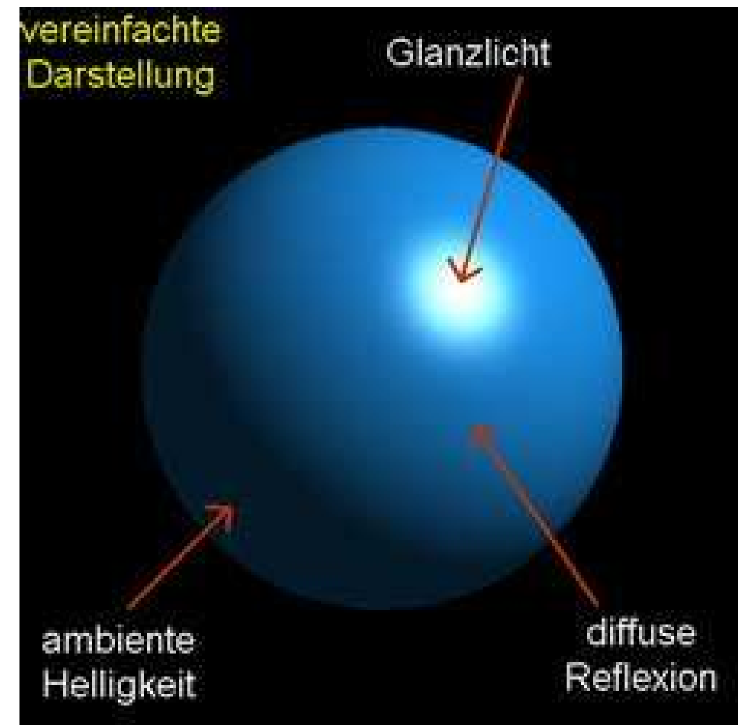


- Strahler (Spot Light):
  - Emission erfolgt nur innerhalb eines Lichtkegels (Öffnungswinkel, Richtung, Intensitätsabfall zum Kegelrand),
  - Intensität nimmt mit der Entfernung ab.



7.5.3 Farbanteile einer Lichtquelle

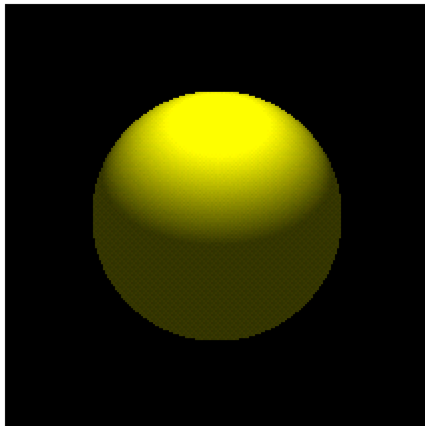
- Jede Lichtquelle besitzt RGBA-Werte für ambientes, diffuses und glänzendes Licht:
  - ambiente Helligkeit (GL_AMBIENT): verstärkt die Hintergrundbeleuchtung der Szene,
  - diffuse Relexion (GL_DIFFUSE): bestimmt den Lichtanteil, der beim Auftreffen auf eine Oberfläche in **alle** Richtungen reflektiert wird.
  - glänzende Reflexion (GL_SPECULAR): best. den Lichtanteil, der beim Auftreffen auf eine Oberfläche in **eine bestimmte** Richtung reflekiert wird.
- Hinweis:
  - Um eine realistische Beleuchtung zu erreichen, sollte GL_AMBIENT den gleichen Wert wie GL_DIFFUSE haben.



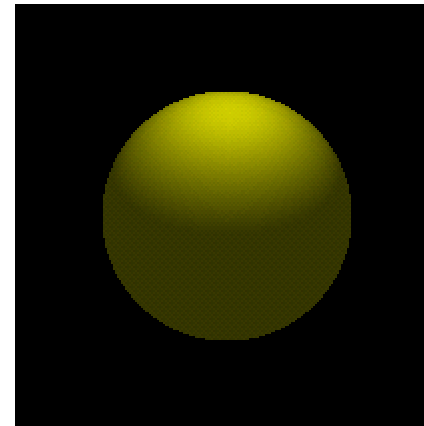
 © Verteilet Systeme, Unversität Ulm, M. Schöttner

7.5.4 Dämpfung

- In der realen Welt schwächen Partikel in der Luft Lichtquellen ab.
- Deshalb erhalten Lichtquellen einen Dämpfungsfaktor:
  - Beitrag einer Lichtquelle zur Beleuchtung wird mit diesem Dämpfungsfaktor multipliziert.
  - Dämpfungsfaktor wächst mit der Entfernung zur Lichtquelle (konstant, linear, quadrat.),
  - Alle Beiträge einer Lichtquelle werden gleichermassen gedämpft.
- Besonderheiten:
  - der Dämpfungsfaktor gilt nicht für gerichtete Lichtquellen,
  - globales ambientes Licht und Eigenleuchtkraft eines Objekts werden nie gedämpft.
- Ohne Dämpfung:

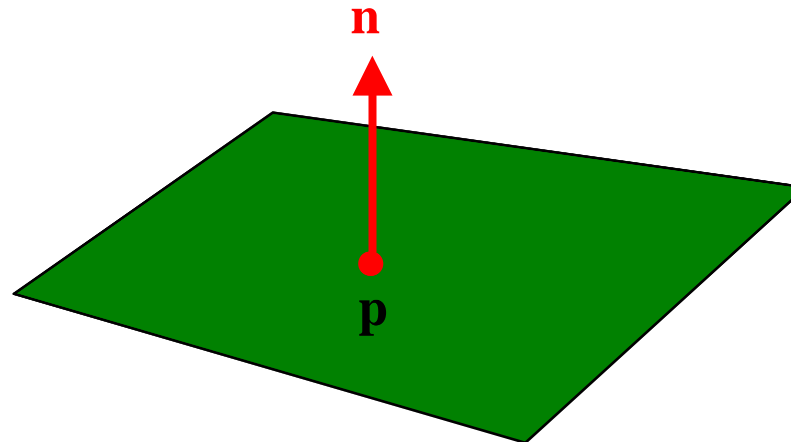


- Mit Dämpfung:



7.5.5 Normalenvektoren

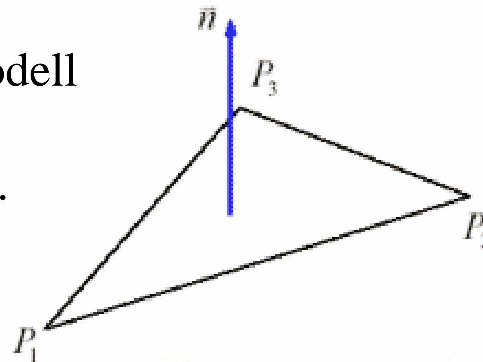
- Normalenvektor best. die Orientierung einer Fläche zu einer Lichtquelle.
- Notwendig um Einfallswinkel von Lichtstrahlen zu bestimmen.
- Eigenschaften von Normalenvektoren:
  - sie stehen senkrecht auf ihrer zugehörigen Fläche,
  - sie zeigen zur Außenseite der Fläche,
  - ihre Länge ist 1.0.



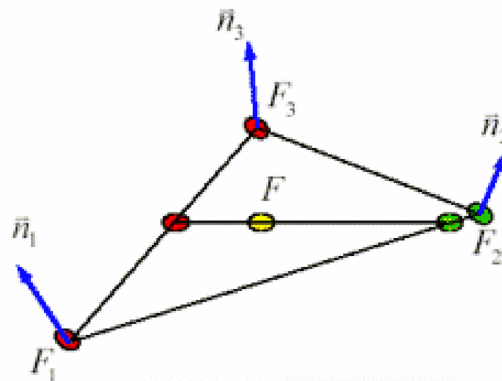
Virtuelle Präsenz, Winter 04/05 © Verteilet Systeme, Universität Ulm, M. Schöttner

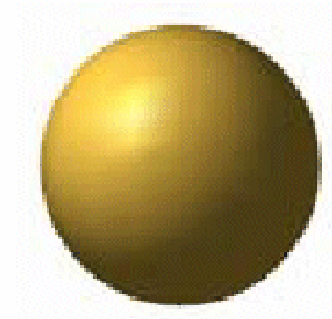
7.5.6 Schattierung

- Shading:
  - pixelweise Berechnung der Farbe sichtbarer Oberflächen an Hand eines Beleuchtungsmodells,
  - lokales Beleuchtungsmodell (schnelle Berechnung), verwendet in OpenGL,
  - vernachlässigt wesentliche Effekte (z.B. Spiegelung, Brechung).
- Flat-Shading:
  - Oberflächennormale und Beleuchtungsmodell bestimmen Farbe der Fläche.
  - Farbe „konstant" über die gesamte Fläche.
  - Ein Normalenvektor pro Fläche,
  - Veraltete Technik.



- Gouraud-Shading:
  - Normale an den Eckpunkten und Beleuchtungsmodell bestimmen Farbe an den Eckpunkten.
  - In der Fläche werden dann die Farben der Eckpunkte linear interpoliert.





7.5.7 Oberflächenmodelle

- Erscheinungsbild eines Objekts hängt wesentlich davon ab, wie es auf das einfallende Licht reagiert → Materialeigenschaften:
  - Reflexion der RGBA-Anteile von ambientem, diffusem & glänzendem Licht (Phong Modell),
  - Konzentration von reflektiertem glänzendem Licht & Eigenleuchten.
- Diffuse Reflexion: maßgeblich für die Farbe in der man ein Obj. sieht
  - abhängig vom Einfallswinkel des Lichts,
  - beeinflusst von der Farbe des einfallenden Lichts,
  - aber unabhängig vom Standpunkt des Beobachters.
- Ambiente Reflexion: beeinflusst die Gesamtfarbe eines Objektes
  - wird von diffuser Reflexion normalerweise überstrahlt,
  - am besten erkennbar, wo Obj. nicht direkt von Lichtquellen angestrahlt wird,
  - beeinflusst vom globalen ambienten Licht und dem ambienten Anteil jeder Lichtquelle,
  - Ambiente und diffuse Reflexion von Objekten der realen Welt ist normalerweise gleich.
- Beispiel: steigende diffuse Reflexion:

Virtuelle Präsenz, Winter 04/05 © Verteilet Systeme, Unversität Ulm, M. Schöttner

- Glänzende Reflexion: erzeugt Glanzpunkte
  - abhängig vom Standpunkt des Beobachters,
  - Konzentration des reflektierten Lichts steuerbar.
  - Einfluss des Materials auf die reflektierten RGBA-Werte definierbar.
  - Beispiel: steigender Glanzlichtexponent (Glanzpunkte werden kleiner & heller)

- Eigenleuchten:
  - für jedes Objekt kann eine Eigenhelligkeit definiert werden,
  - z.B. zum Zeichnen von "Lampen"-Objektenfür Lichtquellen,
  - Objekte leuchten scheinbar, fungieren aber nicht als zusätzliche Lichtquelle und erhellen somit ihre Umgebung nicht.
  - Beispiel: ohne Eigenleuchten mit Eigenleuchten

- Bem.: Lichtfarben != Materialfarben
  - Eine rote Kugel erscheint unter grünem Licht schwarz.
  - Das rote Material absorbiert das grüne Licht komplett und reflektiert nichts.

7.5.8 Beispielfragment: beleuchtetes Dreieck

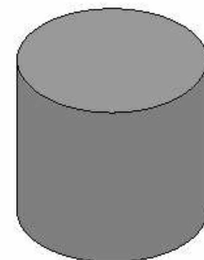
```
public class NeHeCanvas extends GLAnimCanvas implements KeyListener, MouseListener {
public void init() {
    gl.glLightfv (GL_LIGHT0, GL_POSITION, light0_pos);
    gl.glLightfv (GL_LIGHT0, GL_AMBIENT,  ambient0);
    gl.glLightfv (GL_LIGHT0, GL_DIFFUSE,  diffuse0);
    gl.glLightfv (GL_LIGHT0, GL_SPECULAR, specular0);
    gl.glLightfv (GL_LIGHT0, GL_SPOT_DIRECTION, spotdir0);
    gl.glLightfv (GL_LIGHT0, GL_SPOT_CUTOFF, spotcut0);
     gl.glEnable (GL_LIGHTING);    gl.glEnable (GL_LIGHT0);
}
public void DrawGLScene() {
    gl.glMaterialfv (GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE, ambient_diffuse);
    gl.glMaterialfv (GL_FRONT_AND_BACK, GL_SPECULAR, specular);
    gl.glMaterialf (GL_FRONT_AND_BACK, GL_SHININESS, 20);

    gl.glBegin ( GL_TRIANGLES );
        gl.glColor3f(1.0f, 0.0f, 0.0f);
        gl.glNormal3f( 0.0f, 0.0f, 1.0f);
        gl.glVertex3f(-0.5f,-0.5f, 0.5f);   gl.glVertex3f( 0.5f,-0.5f, 0.5f);   gl.glVertex3f( 0.5f, 0.5f, 0.5f);
    gl.glEnd ();
}}
```


Virtuelle Präsenz, Winter 04/05 © Verteilet Systeme, Universität Ulm, M. Schöttner

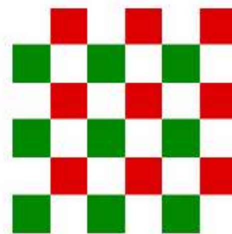
7.6 Texturen

7.6.1 Grundidee & Anwendung

- Darstellung realistischer Objekte nur über Dreiecke ist zu aufwendig.
- Grundidee: Texturen (Rastergrafik) auf Gittermodell „aufkleben“.
- Detaillierte Darstellung bei geringer geometrischer Komplexität.
- Anwendungen:
  - Simulation von Materialien,
  - Light Maps (Darstellung von Licht),
  - Shadow Maps (Darstellung von Schatten),
  - Bumb Mapping (Oberflächenbeschaffenheit),
  - Environment Maps (Spiegelung der Umgebung).
- Beispiel:



+

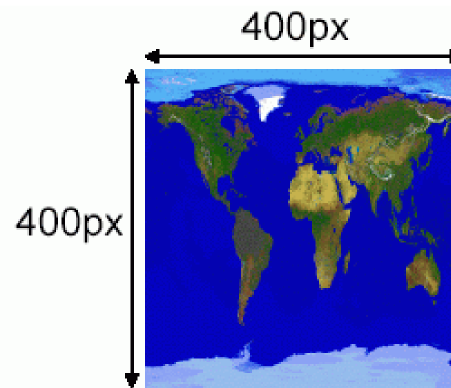


=

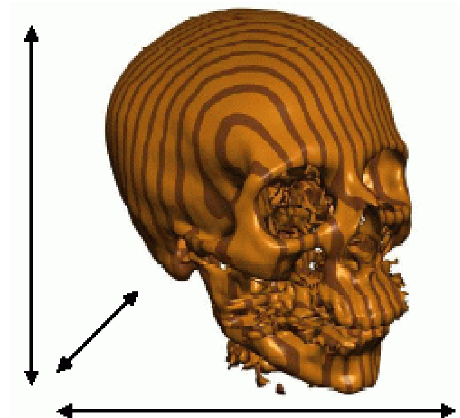
7.6.2 Erzeugen von Texturen

- Eindimensional:

- Zweidimensional:
  - am meisten verwendet.
  - Fkt.: *glTexImage2D()*.



- Dreidimensional:
  - mehrere Schichten von 2D Texturen,
  - für medizinische Anwendungen,
  - hoher Ressourcenbedarf.



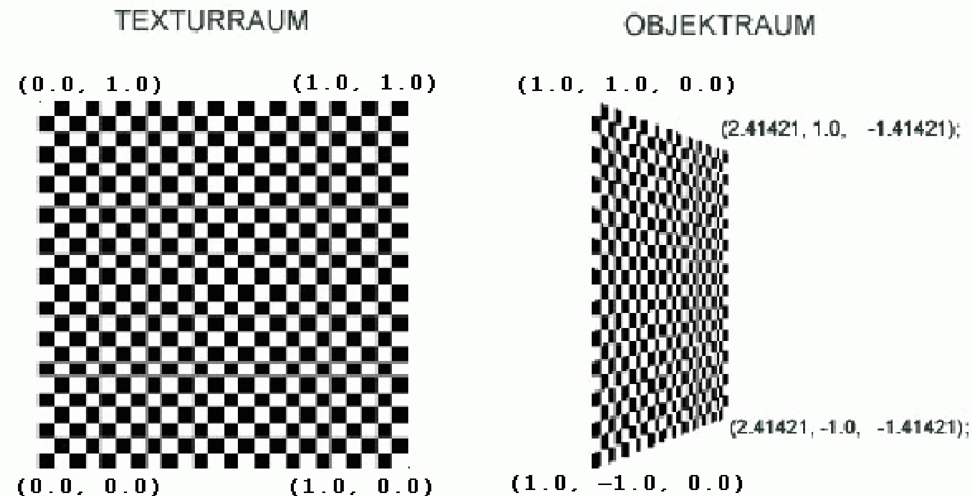
- Erzeugung von Texturen:
  - Rastergrafik erzeugen: selbstgeneriert oder durch Laden einer Grafikdatei,
  - Textur anlegen: Aufruf *glTexImage2D* (Zweierpotenz- Abmessungen),
  - Gff. müssen Bilder vorab skaliert werden → *gluScaleImage()*.

7.6.3 Textur-Objekte

- Bisher wurden Texturen im Speicher angelegt mit *glTexImage2D()*.
- Damit mehrere Texturen verwendet werden könnnen, müssen diese an benannte Texturobjekte gebunden werden.
- Schritt-1: IDs für Texturobjekte generieren
  - *glGenTextures(int, int[ ])*,
  - erzeugt *N* IDs.
- Schritt-2: Texturobjekt erzeugen bzw. aktivieren
  - *glBindTextures(GL_TEXTURE_2D, int ID)*,
  - Aufruf mit unbenutztem Namen:
    - neues Texturobjekt mit übergebenem Namen erzeugen,
    - dieses Objekt als aktives Texturobjekt festlegen,
    - nachfolgend erzeugte Textur wird automatisch diesem aktiven Texturobjekt zugewiesen.
  - Aufruf mit belegtem Namen:
    - assoziiertes Texturobjekt wird aktiviert.
- Texturobjekte können mit *glDeleteTextures()* gelöscht werden (Namen werden hierbei nicht gelöscht).

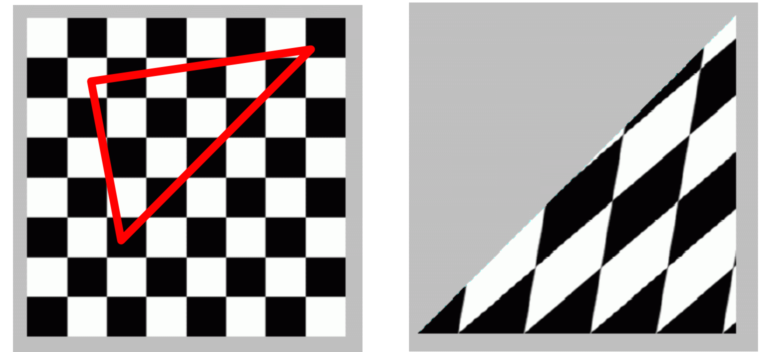
7.6.4 Textur-Koordinaten

- Mapping: Zuordnen von Textur- zu Vertexkoordinaten.
  - Vertexkoordinaten: Koordinaten in der Szene.
  - Texturkoordinaten: Koordinaten innerhalb der Textur
    - Bereich: 0.0 – 1.0,
    - Pixel als Texel bez.



- Zuweisung von Text- zu Vertexkoordinaten mit *glTexCoord()*.

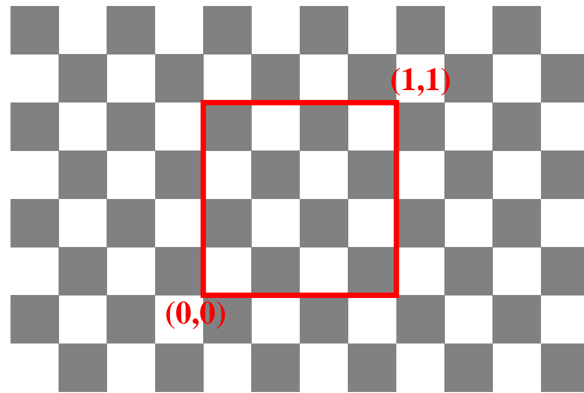
```
glBegin(GL_TRIANGLES);
  glTexCoord2f(0.3, 0.3); glVertex3f(0.0, 0.0);
  glTexCoord2f(0.9, 0.9); glVertex3f(0.0, 1.0);
  glTexCoord2f(0.2, 0.8); glVertex3f(1.0, 1.0);
glEnd();
```



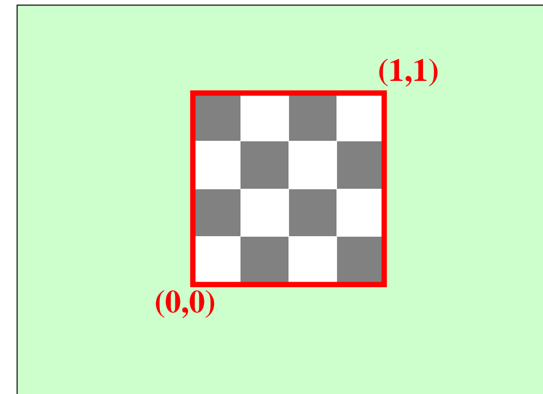
Virtuelle Präsenz, Winter 04/05 © Verteilet Systeme, Universität Ulm, M. Schöttner

Wrapping

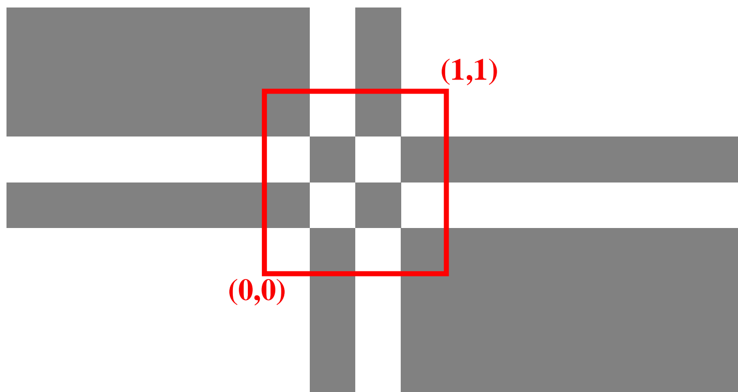
- Texturen sind aus Speichergründen i.d.R. kompakt.
- Was passiert, wenn Texel ausserhalb Textur adressiert wird?
- Fkt. *glTexParameter()* def., wie Textur für diesen Fall vergrößert wird:
  - Repeat:



  - Border Color:

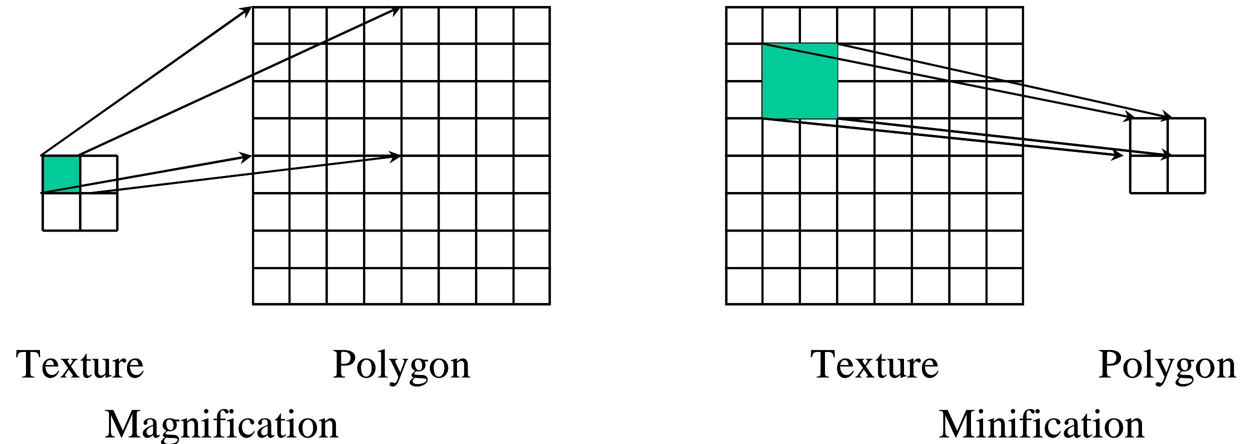


  - Clamp: z.B. Schnitt durch Holz

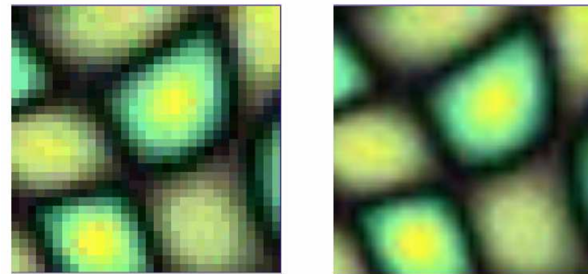


Filtering

- Minification: mehr als ein Texel überdeckt einen Pixel.
- Magnification: mehr als ein Pixel überdeckt einen Texel.



- Filterarten für beide Fälle einstellbar, mit *glTexParameter()*:
  - GL_NEAREST: nächstliegenden Texel verwenden (schärfer, Klötzcheneffekt),
  - GL_LINEAR: linearer Filter mittelt aus umgebenden Texeln (weicher, aber unschärfer).
- Beispiel: Magnification (nearest & linear):



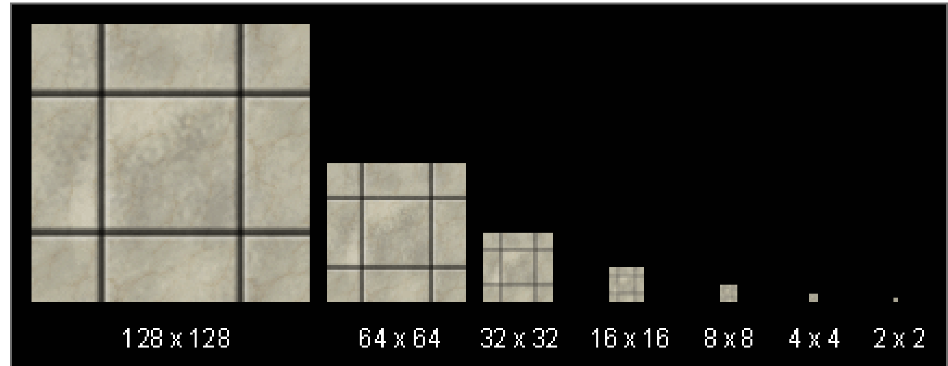
Virtuelle Präsenz, Winter 04/05 © Verteilet Systeme, Unversität Ulm, M. Schöttner

7.6.5 Textur-Beispielfragment

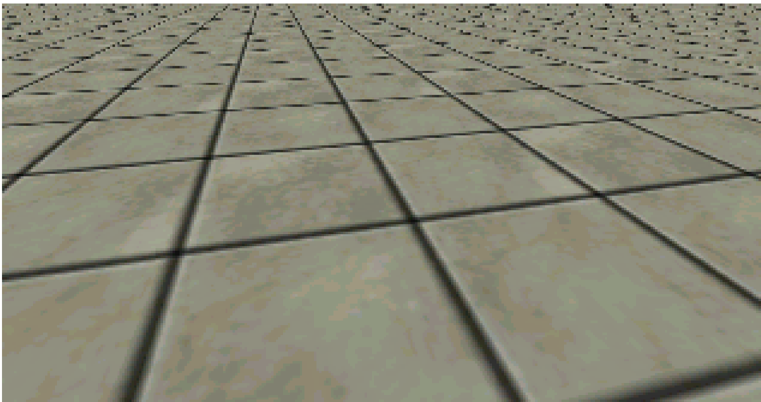
```
public class NeHeCanvas extends GLAnimCanvas implements KeyListener, MouseListener {
    int[] texture = new int[1];                                   // texture name
public void init() {
    gl.glGenTextures(1, texture);                                 // generate name
    gl.glBindTexture(GL_TEXTURE_2D, texture[0]);                  // create texture object
    gl.glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    gl.glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    gl.glTexImage2D(GL_TEXTURE_2D,                                // generate texture
                    0, 3,                                         // detail level & #color components
                    image_width,  image_height,                   // image dimensions
                    0,                                            // border width (here no border)
                    GL_RGB,                                       // pixel format
                    GL_UNSIGNED_BYTE,                             // pixel data type format
                    texels);                                      // ptr. to pixel data
}
public void DrawGLScene() {
    gl.glBindTexture(GL_TEXTURE_2D, texture[0]);                  // select texture
    gl.glBegin(GL_QUADS);
        gl.glTexCoord2f(0.0f, 0.0f);                              // set texture coordinates
        gl.glVertex3f(-1.0f, -1.0f, 1.0f);
        // …
    gl.glEnd();
}
```

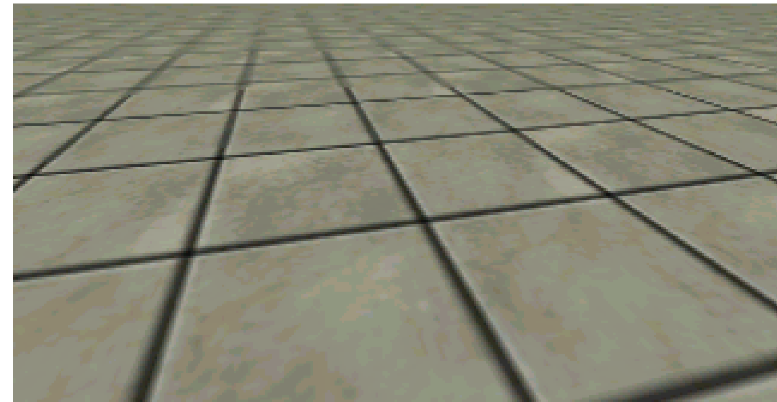
7.6.6 MipMaps: Multiple Levels of Detail

- Entfernt sich ein texturiertes Objekt, so muss Textur verkleinert werden.
- Rundungsfehler führen hierbei zu Artefakten und Verzerrungen.
- Lösung: Texture Map →
  - für mehrere vorgefilterte Bilder,
  - mit verschiedenen Auflösungen,
  - Param. „level“ in *glTexImage2D()*.



- Ergebnis: ohne MIP-mapping mit MIP-mapping



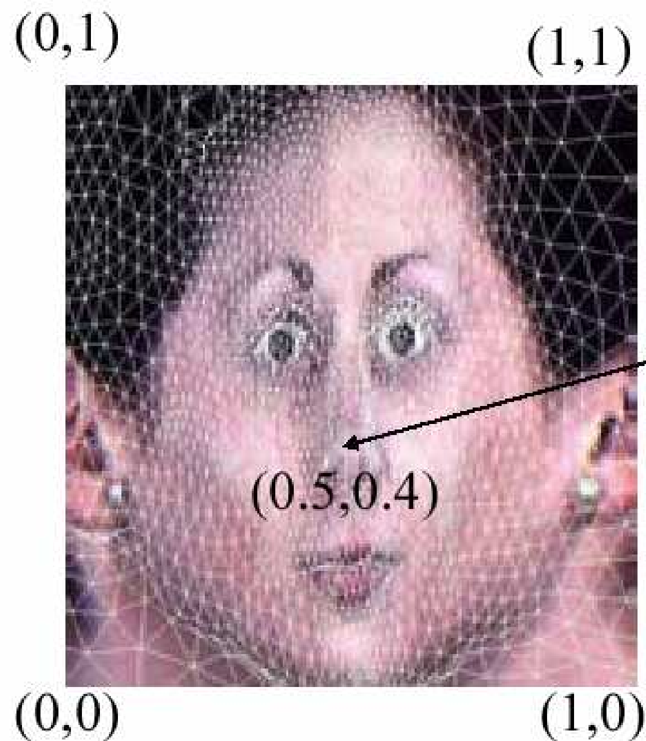


- Bem.: Gleiche Idee für 3D Daten
  - Level of Detail: einfachere Geometrien für weiter entfernte Objekte.

7.6.7 Dreiecksnetze

- Aufteilung der Textur auf viele Dreiecke
- Modellierungswerkzeug liefert: Unterteilung und Texturkoordinaten.
- Beispiel:

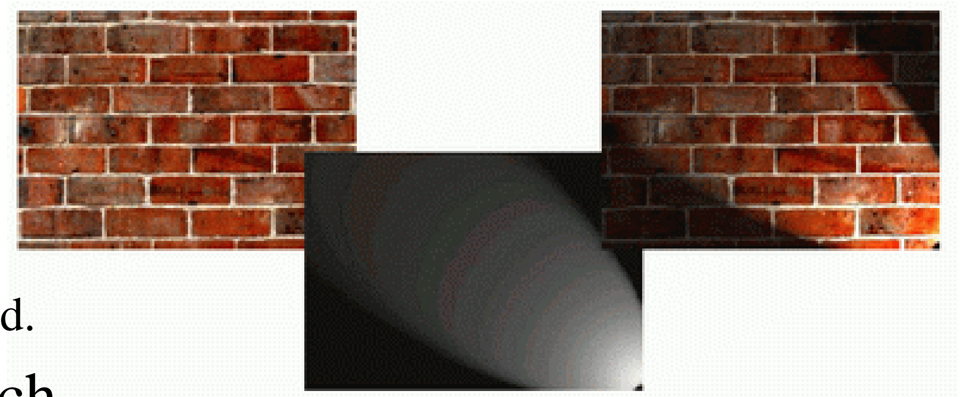
```
glTexCoord2f(0.5,0.4);
glNormal3f(...);
glVertex3f(...);
```



Virtuelle Präsenz, Winter 04/05 © Verteilet Systeme, Unversität Ulm, M. Schöttner

7.6.8 Light Maps

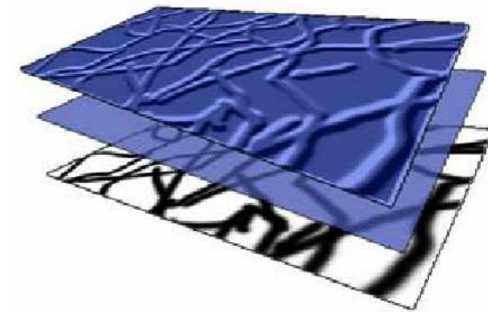
- Beleuchtung vermeiden → hoher Rechenaufwand.
- Für nicht bewegliche Lichtquellen gedacht.
- Alternative durch Ligthtmapping:
  - verwendet zwei Texturen,
  - eine Textur ist das Bild, welches angezeigt werden soll,
  - die andere Textur ist ein Graustufenbild (Lightmap genannt), die angibt, mit welcher Intensität die Pixel der ersten Textur dargestellt werden.
  - Bedeutung der Lightmap-Pixel: schwarz: dunkel, weiß: in Orginalfarbe darstellen.
- Vorgehensweise:
  - zunächst die „normale“ Textur zeichnen,
  - dann Lightmap an gleicher Stelle rendern,
  - für die Lightmap muss Blending aktiviert werden, wodurch diese transparent wird und mit der unterliegenden Textur kombiniert wird.

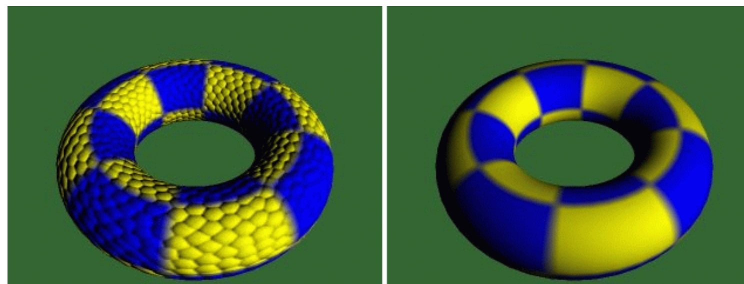


- Bem.: Lichtbewegungen können auch durch Lightmapverschiebungen realisiert werden.

7.6.9 Bump Mapping

- Reines Texturmapping erzeugt glatte Oberflächen.
- Idee beim Bump Mapping:
  - Beleuchtung aktivieren,
  - modifiziere Normalenvektoren.
- Hierdurch ändert sich die Reflexion von einfallendem Licht.
- Texture Map: enthält Oberflächengeometrie (einfarbig, Texel sind Höhenwerte).





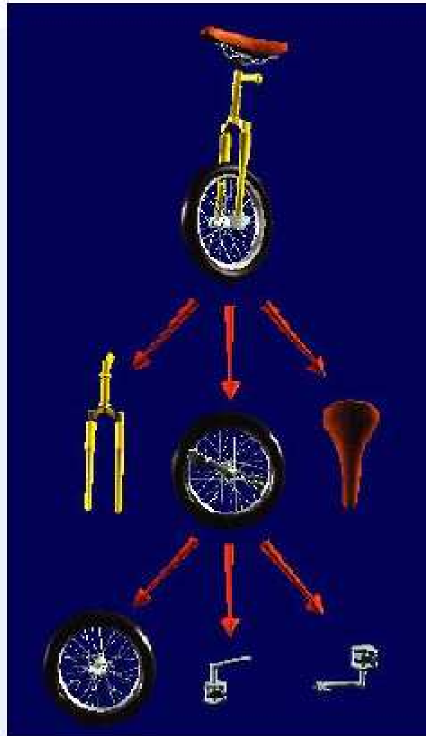
Virtuelle Präsenz, Winter 04/05 © Verteilet Systeme, Unversität Ulm, M. Schöttner

7.7 Szenengraphen

7.7.1 Allgemeines

- OpenGL und DirectX sind low-level 3D Grafikschnittstellen.
- Szenengraph:
  - Gerichteter, azyklischer Graph zur Beschreibung einer darzustellenden 3D Szene.
  - Zur Laufzeit werden die Knoten nach vordefiniertem Schema traversiert.
  - Knoten in der Hierarchie: Informationen für alle Kindknoten, oder zur Selektion einzelner Kinder enthalten.
  - Blattknoten für elementare Informationen.
- Wichtige Implementierungen:
  - Open Inventor 2.0 (aufbauend auf OpenGL), Entwicklung von VRML (für WWW) – 1994.
  - Java3D (SUN, 1998), OpenSG (OpenSource Scenegraph, ZGDV Darmstadt, 2001).
- Vorteile:
  - Kompakte hierarchische Beschreibung von Szenen.
  - Eigenschaften, Attribute und Transformationen für ganze Teilgraphen → Algorithmen müssen nicht immer alle Elemente einer Szene besuchen.
  - Teilgraphen können mehrfach genutzt werden → Objekte wieder verwendbar.
  - Optionale Graphtransformationen ermöglichen effiziente Struktur für Algorithmen.
  - Neben der Beschreibung der Szenerie ist oft auch das dynamische Verhalten integriert.

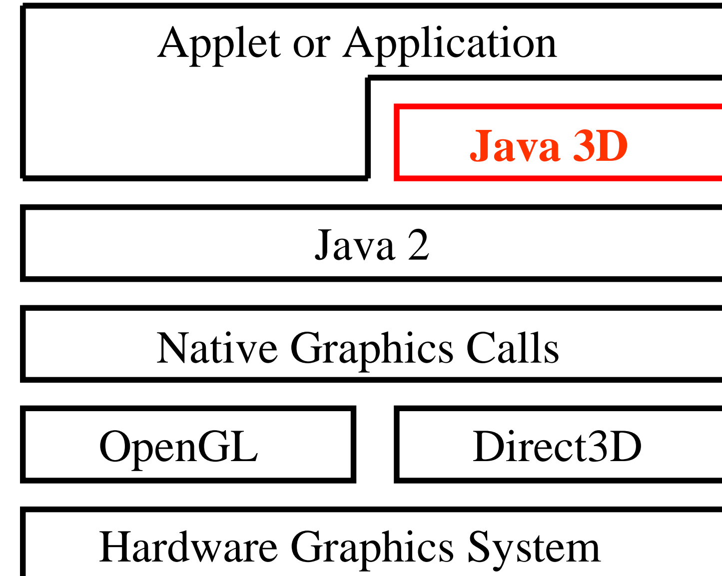
7.7.2 Beispiel



- Fahrrad
  - Translation
    - Sitz
      - Sitz Attrib.
      - Sitz Geo.
    - Translation
      - Gabel
        - Gabel Attrib.
        - Gabel Geo.
        - Translation
          - Rotation
            - Rad
              - Rad Attrib.
              - Rad Geo.
              - Translation
                - Pedal links
                  - Ped. Attrib.
                  - Ped. Geo.
              - Translation
                - Pedal rechts
                  - Ped. Attrib.
                  - Ped. Geo.

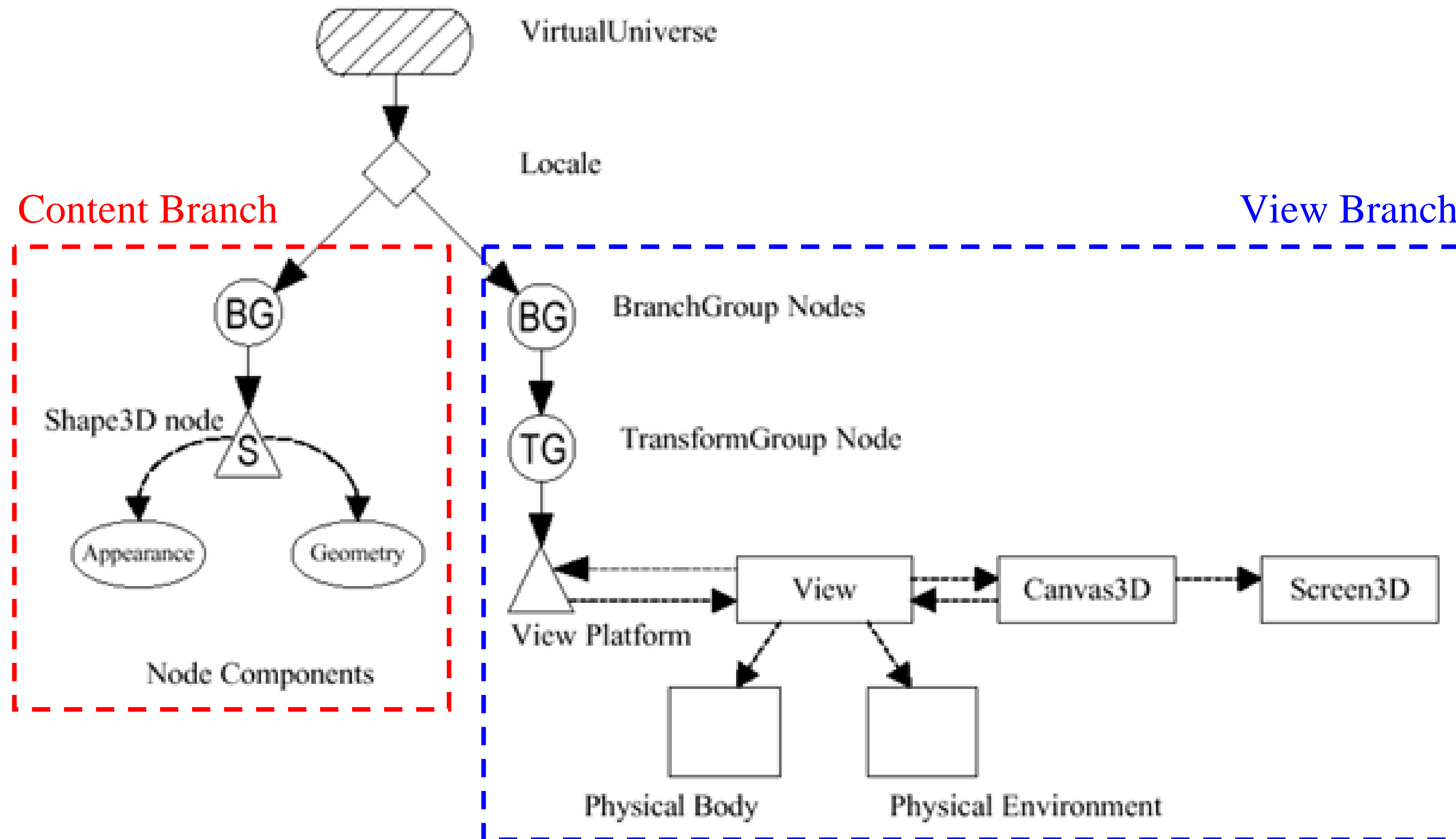
7.7.3 Java 3D

- Java-API zur Erstellung, Darstellung und Animation von 3D Szenen.
  - objektorientierte Schnittstelle,
  - Objekte werden in einem Szenengraph verwaltet,
  - Szenengraph enthält komplette Beschreibung einer Szene, einschließlich geometrischer Daten, Attribut- und Darstellungsinformationen.
- Optionale Erweiterung für JDK (akt. Version 1.3.1).
- Zugriff auf Grafik-Hardware über OpenGL & Direct3D.
- Infos im Web unter:
  - SUN: http://java.sun.com/products/java-media/3D,
  - Community: www.j3d.org.



Szenengraph

- Content Branch: Modell (Objekte, Materialien, Lichter, Animationen).
- View Branch: Steuerung der Ausgabe.



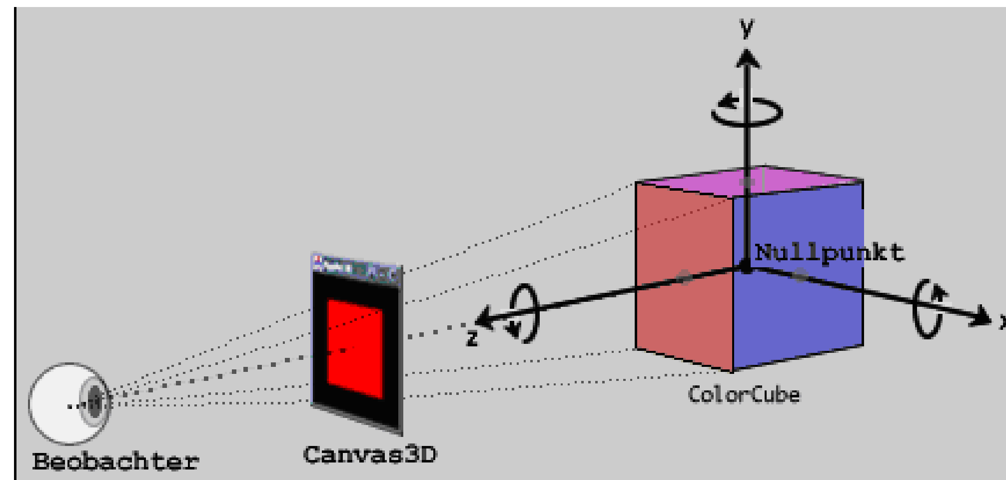
- VirtualUniverse: Menge von 3D Objekten.
- Locale:
  - Unterteilung der 3D Objekte eines VirtualUniverse,
  - Einhängen des lokalen Koordinatensystems.
  - Ursprung für alle Szenengraphobjekte.
- Branch Group Knoten: Container
- View Platform
  - Position des Benutzers.
  - Bewegung durch Universum → Modifikation der übergeordneten Transform Group.
- View
  - Generiert die Sicht auf das Universum in Abhängigkeit von Benutzerbeschreibung.
    - Physical Body: Charakteristik des Kopfs (z.B. Augenabstand) nur spezielle Virtual Reality Umgebung relevant (z.B. Head tracking), sonst Default Werte verwenden.
    - Phyiscal Environment: Beschreibung der physikalischen Umgebung (z.B Anzahl Tracking Sensoren) → Default Werte vorhanden.
- Canvas3D: Fenster für 3D Darstellung.
- Screen3D: physikalische Bildschirmeigenschaften.

Beispiel: SimpleUniverse

- Default *View Branch* verwenden → nur Inhalt programmieren.

```
SimpleUniverse univ = new SimpleUniverse(canvas);
univ.getViewingPlatform().setNominalViewingTransform( );
```

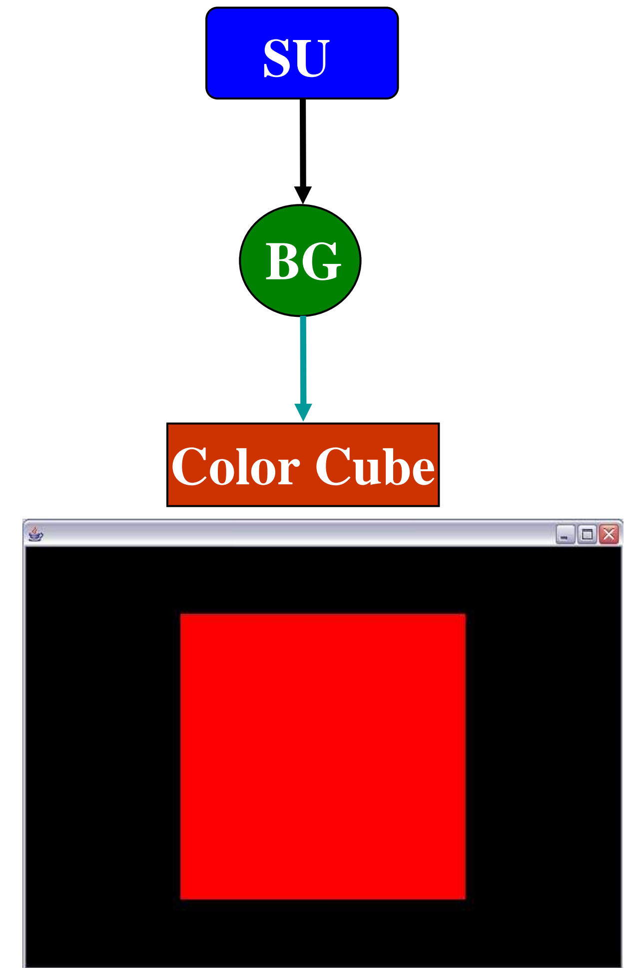
- Nullpunkt ist der Mittelpunkt von Canvas3D:
  - Y-Achse wesit nach oben,
  - X-Achse weist nach rechts,
  - Z-Achse zeigt auf den Beobachter.



- setNominalViewingTransform():
  - plaziert die Instanz von Canvas3D zwischen Beobachter und Nullpunkt auf der Z-Achse.
  - es kann exakt einen Kubus mit den Koordinaten (-1,0,0) und (1,0,0) dargestellt werden.
  -

• Programmtext:

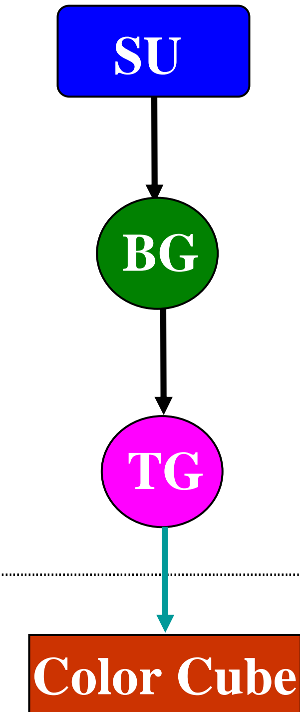
```
public static void main( String[] args ) {
        Frame frame = new Frame( );
        frame.setSize( 640, 480 );
        frame.setLayout( new BorderLayout( ) );

        Canvas3D canvas = new Canvas3D( null );
        frame.add( "Center", canvas );

        SimpleUniverse univ = new SimpleUniverse(canvas);
        univ.getViewingPlatform().setNominalViewingTransform( );

        BranchGroup scene = createSceneGraph( );
        univ.addBranchGraph( scene );

        frame.show( );
}

public BranchGroup createSceneGraph( ) {
        BranchGroup branch = new BranchGroup( );

        ColorCube my_Cube = new ColorCube( 0.4 );
        branch.addChild( my_Cube );
        return branch;
}
```

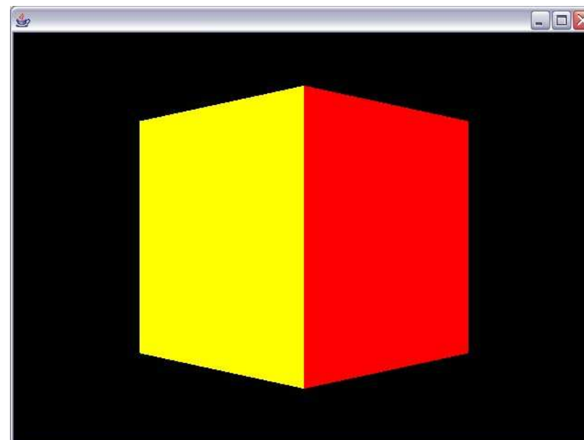


Transformationen

- TransformGroup: Translation, Rotation & Skalierung v. Objektgruppen:

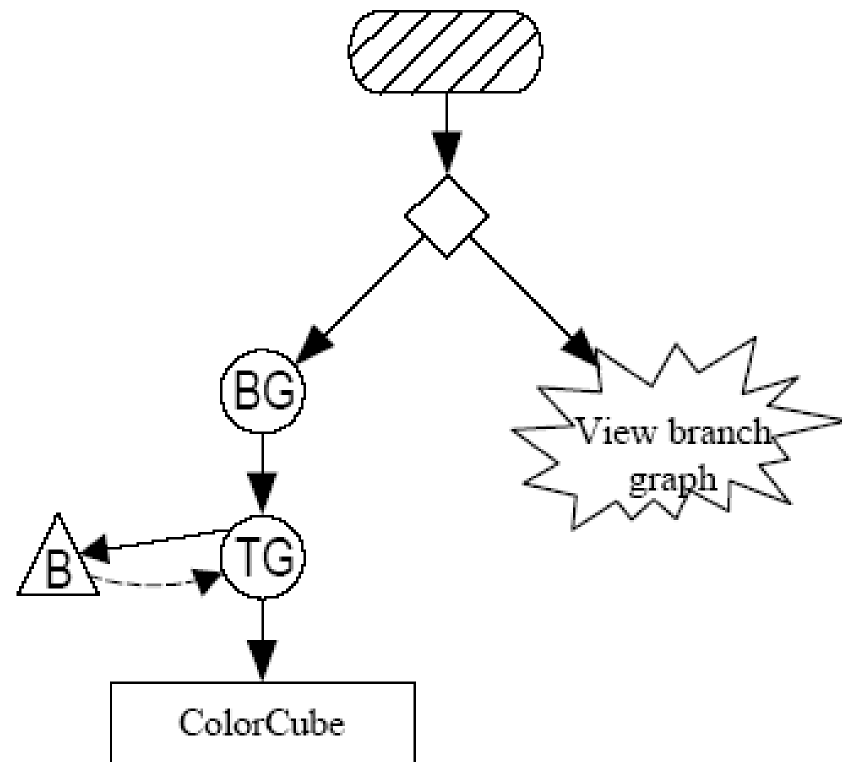
```
public BranchGroup createSceneGraph( ) {
        BranchGroup branch = new BranchGroup( );
        Transform3D rotate = new Transform3D( );

        rotate.rotY( Math.PI / 4.0d );
        TransformGroup trans = new TransformGroup(rotate);
        branch.addChild( trans );

        trans.add( new ColorCube( 0.4 ) );
        return branch;
}
```





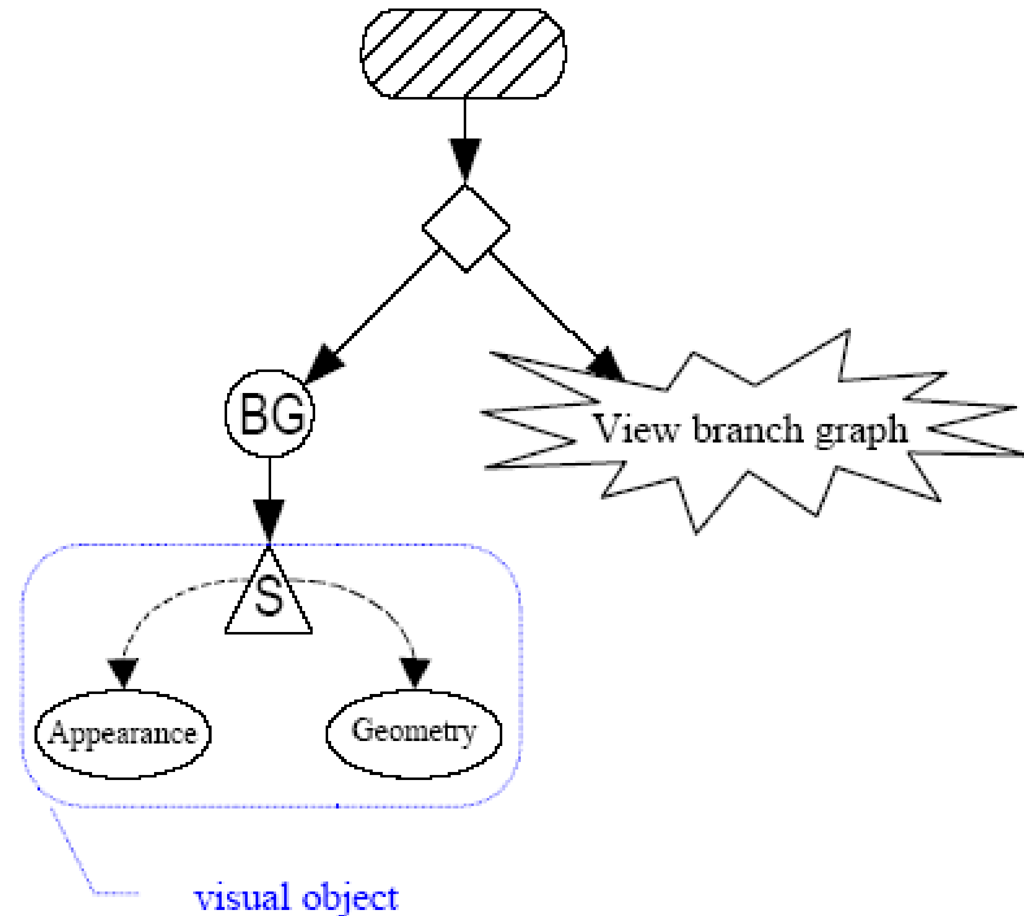
Behaviours

- Erlaubt dynamische Änderungen im Szenengraph zu beschreiben.
- Behaviours: assozierte Knoten für Animationen & Interaktionen.
  - Animation: Verändern von Eigenschaften über die Zeit.
  - Interaktion: Verändern von Eigenschaften in Reaktion auf Benutzeraktionen.
- Programmierer legt Wakeup-Bedingungen fest:
  - WakeupOnViewPlatformEntry,
  - WakeupOnCollisionEntry,
  - WakeupOnElapsedTime,
  - …
- Scheduling Region:
  - def. Aktivierungsradius eines Behaviors,
  - nur wenn ViewPlatform Akt.radius diesen schneidet, werden Wakeup-Bedingungen geprüft.



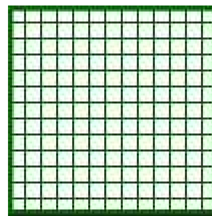
Frei definierte 3D Objekte

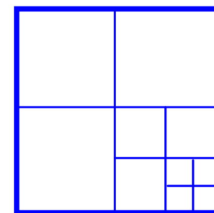
- Realisiert durch die Klasse *Shape3*.
- Geometry:
  - Form und Struktur,
  - definiert über Vertices.
- Appearance:
  - Farbe,
  - Shading,
  - Texturen,
  - Transparenz, …
- Beleuchtung:
  - Phong Modell,
  - wie bei OpenGL.
- 3D Objekte aus Datei ladbar:
  - VRML,
  - OBJ (Alias|Wavefront object),
  - LW3D (Lightwave 3D scene),
  - und weitere. . .

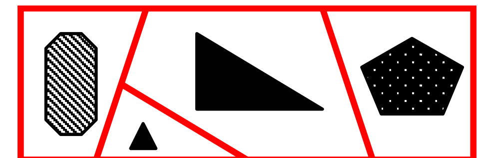


7.7.4 Optimierungen

- Szenengraphen beinhalten i.d.R. sehr viele Objekte:
  - es ist teuer immer den ganzen Graph abzuwandern,
  - sowohl beim Rendern, als auch für Kollisionstests ...
  - Avatar hält sich immer in einem bestimmten Raum auf.
- Räumlich-funktionale Aufteilung:
  - Objekte werden nach ihrer räumlichen Struktur hierarchisch geordnet,
  - z.B. liegt ein Stuhl in einem Haus im Graph unter dem Hausknoten,
  - Kindobjekte nur berücksichtigen, wenn das Haus sichtbar ist,
  - evt. im letzten Fall auch nicht, falls nicht durch Fenster hindurch gesehen werden kann.
- Raumaufteilung:
  - uniform: z.B. **Gittermethode**,
  - flexible Ansätze: z.B. **Quadbäume** Aufteilung bis: Zelle leer, Objekt vollst. enthalten, oder ein Schwellwert erreicht ist.
  - **Binary Space Partitioning** (BSP-Trees):
    - Aufteilung mit Trennebenen beliebiger Orientierung,
    - Ziele: ausgeglichener Baum, mögl. wenig geteilte Objekte.
  - Ist ein Knoten nicht zu berücksichtigen, so sind auch seine Kindknoten irrelevant.
  - Aber: bei beweglichen Objekten muss Raumaufteilung u.U. oft angepasst werden.

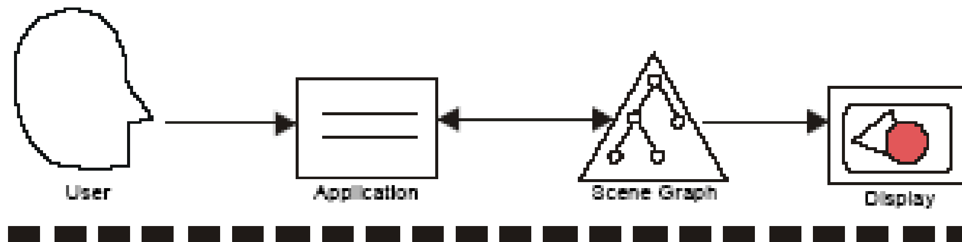


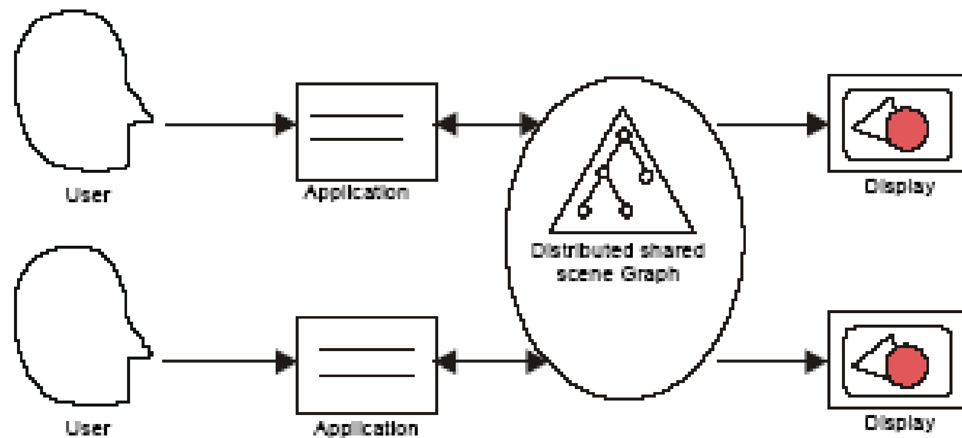


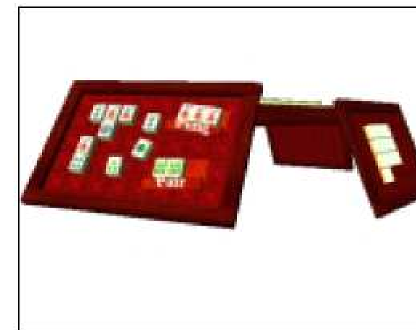


7.7.5 Verteilte Szenengraphen

- Mehrere Benutzer verwenden einen gemeinsamen Szenengraphen:
  - Jeder Teilnehmer hat seine individuelle Kameraposition,
  - Anwendung läuft auf jedem Knoten,
  - Szenengraph repliziert.





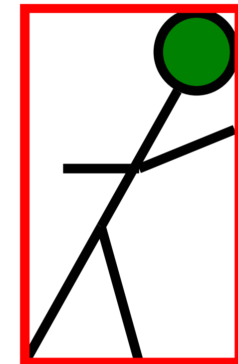


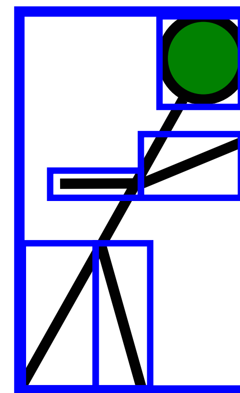
- Beispiele: Distributed Open Inventor, Avocado, Repo-3D, …

- Konsistenz - Ausgangssituation:
  - Graph komplett oder partiell repliziert,
  - überwiegend Leseoperationen → unproblematisch,
  - Schreibzugriffe auf Knotenvariablen (z.B. Transformationsmatrizen), und seltener auch strukturelle Änderungen (Knoten hinzufügen oder löschen).
- Avocado:
  - GMD, H. Tramberend, 1999-2003,
  - Framework für verteilte VR-Anwendungen,
  - verteilter Szenengraph mit gegebenen C++ Objekten,
  - Zustandsänderungen werden ereignisbasiert propagiert:
    - Instanzvariablen streng gekapselt,
    - Zugriff nur über Methoden: *get* und *put*,
    - hierdurch kann Oberklasse Konsistenz impl.
  - Bem.: Objekte an unterschiedl. Adressen auf jedem Knoten → (De-)Serialisierung & IDs notwendig.
- Alternative: seitenbasierter DSM
  - Beispiel: Plurix (siehe später),
  - Replikation: automatisch durch DSM,
  - Updates: werden beim Neuzeichnen automatisch angefordert,
  - Filterung: nur darzustellende Objekte werden über das Netz geladen.

7.8 Kollisionserkennung

- Ziel: Erkennen von Berührungen oder Durchdringen von Objekten.
- Anwendungsbeispiele:
  - stelle fest, ob ein Flugkörper ein Ziel trifft,
  - bestimme Punkte, an denen sich ein Verhalten ändern sollte.
  - prüfe, ob ein Spieler oder Charaktere eine Wand oder ein Hindernis treffen.
- Raumaufteilungsverfahren reduzieren #Objekte für Test.
- Probl.: Tests bei komplexen Objekten immer noch aufwendig.
- Lösung: **Hüllkörper**
  - Hüllkörper (z.B. Kugel → 1 Test) reduzieren die geom. Komplexität,
  - schneller Tests liefert eine näherungsweise Entscheidung,
  - bei vermuteter Kollision erfolgt genauer Test.
  - evt. **Hierarchie von Hüllkörpern**.





Virtuelle Präsenz, Winter 04/05 © Verteilet Systeme, Universität Ulm, M. Schöttner

7.9 Literatur

OpenGL (offizielle Seite)
www.opengl.org

OpenGL Tutorials
http://nehe.gamedev.net/

Multimedia
Abteilung Medieninformatik, Universität Ulm, http://medien.informatik.uni-ulm.de.

Interaktive Computergrafik
Marc Stamminger, Vorlesung, Universität Erlangen, orientiert sich an OpenGL, http://www9.informatik.uni-erlangen.de/Teaching/SS2003/InCG.

Herausforderungen an moderne Szenengraphsysteme am Beispiel OpenSG
Dirk Reiners, Informatik Spektrum, Band 27, Heft 6, Dezember 2004.

Distributed Open Inventor: A Practical Approach to Distributed 3D Graphics
Gerd Hesina, Dieter Schmalstieg, Anton Fuhrmann, and Werner Purgathofer, ACM Symposium on Virtual Reality Software and Technology, 1999.

Avocado: A Distributed Virtual Reality Framework
H. Tramberend, IEEE Virtual Reality, Texas, USA, 1999.

Spieleprogrammierung
Vorlesung WS04/05, Elisabeth Andre, Universität Augsburg.