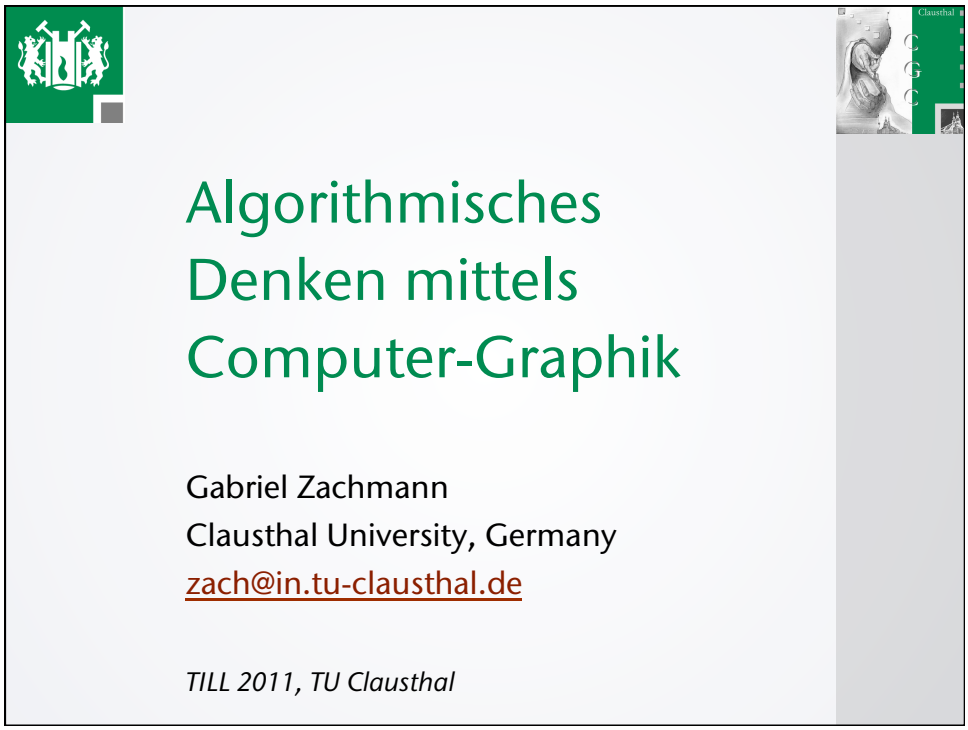
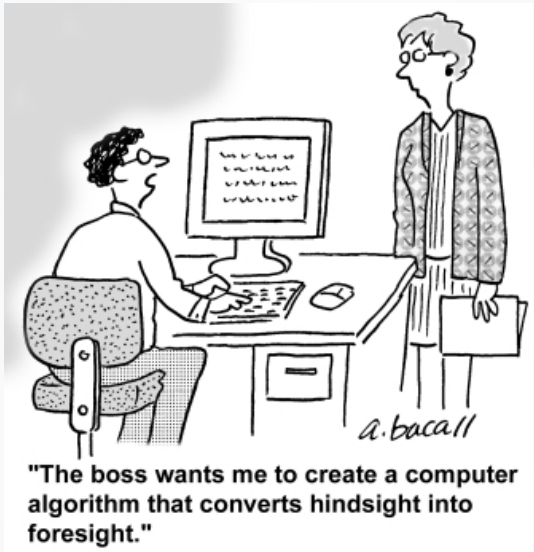


Algorithmisches Denken mittels Computer-Graphik

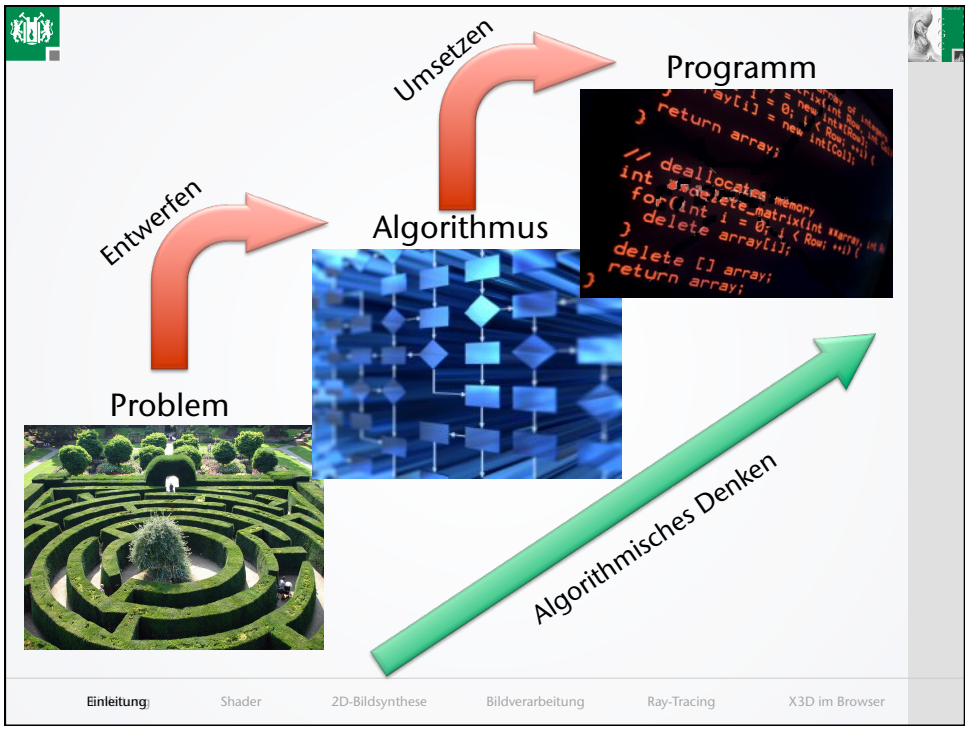
Gabriel Zachmann
Clausthal University, Germany
zach@in.tu-clausthal.de

TILL 2011, TU Clausthal

Was ist algorithmisches Denken?



"The boss wants me to create a computer algorithm that converts hindsight into foresight."



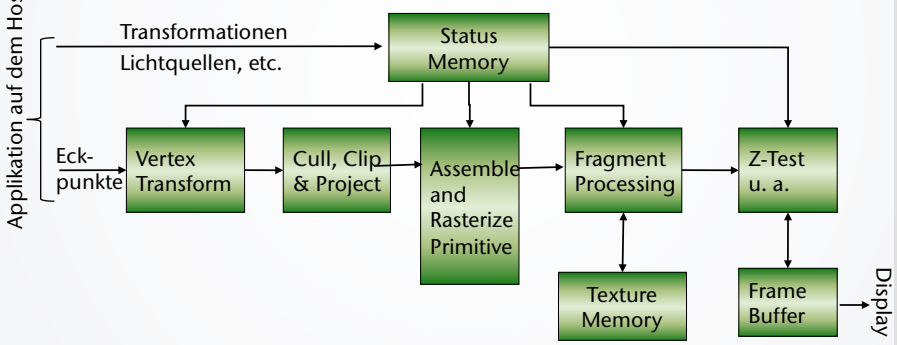
Übersicht

1. Einfache Programme in Form von Shaders
2. Einfache Bildverarbeitungseffekte
3. 2D-Bildsynthese
4. Ein einfacher Ray-Tracer
5. Ein Szenengraph-API (Game Engine)

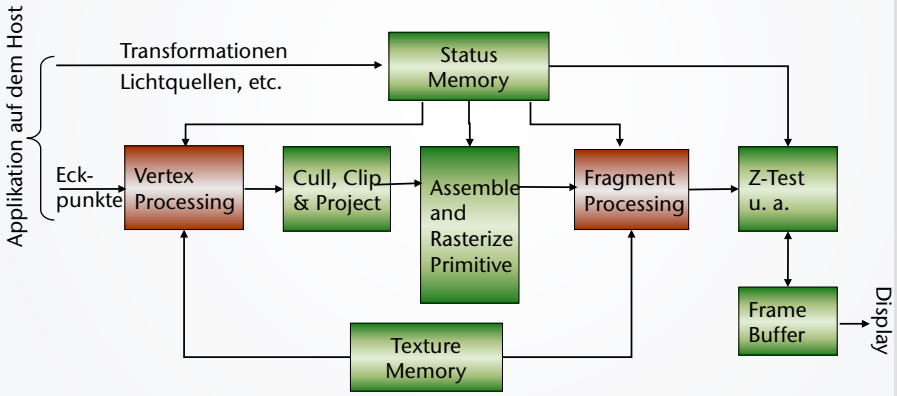
Einleitung Shader 2D-Bildsynthese Bildverarbeitung Ray-Tracing X3D im Browser

Einfache Shader-Programme

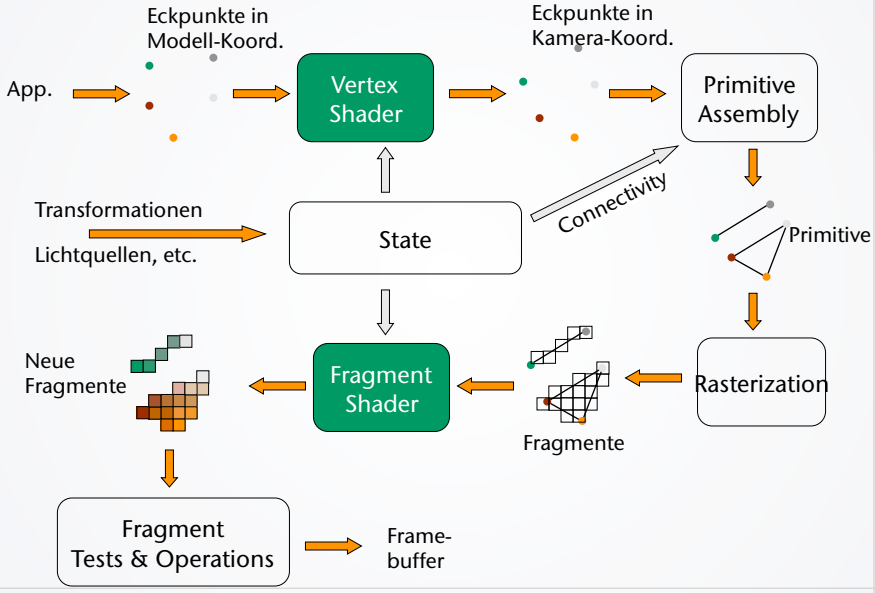
- Die fixed-function Pipeline der Graphikkarte:



- Die programmierbare Pipeline (heutzutage in jedem PC):



Abstraktere Übersicht der programmierbaren Pipeline



Die Shader-Sprache GLSL

- Fester Bestandteil seit OpenGL 2.0 (Oktober 2004)
- Plattform-unabhängig
- Gleiche Syntax für Vertex-Program und Shader-Program
- Rein prozedural (nicht object-orientiert, nicht funktional, ...)
- Im Wesentlichen ist die Syntax = ANSI C ohne Pointer
- Man hat als Programmierer nicht die volle Kontrolle über den Ablauf des Renderings, sondern programmiert "nur" noch 2 "Unterfunktionen", nämlich einen Vertex- und einen Fragment-Shader

- Datentypen: **`float, bool, int,`** Vektoren **`(vec3)`**, Matrizen **`(mat4)`**, Arrays (nur 1-dim. konstante Arrays)
- Die wichtigsten Variablen-Arten:
  - Konstanten und lokale Variablen wie in C (u.a. imperativen Sprachen)
  - **`uniform`**:
    - globale Variable, im Vertex- und Fragment-Shader bekannt, gleicher Wert in beiden Shadern, wird von der Applikation gesetzt, konstant während eines gesamten Primitives (Dreiecks)
  - **`varying`**:
    - wird vom Vertex-Shader gesetzt (pro Vertex) als Ausgabe,
    - wird vom Rasterizer interpoliert,
    - und vom Fragment-Shader gelesen (pro Fragment)

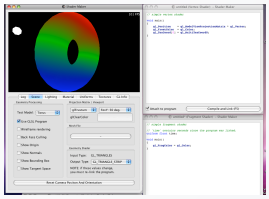
- Operatoren und vordefinierte Funktionen:
  - Wie in C, Java, etc.
  - Beispiele: **`* / + - == != && sin abs ...`**
  - Spezielle Operatoren / Funktionen für Vektoren und Matrizen, z.B.:

```
uniform mat3 myMatrix;          // muss von App. gesetzt werden
vec3 v1 = vec3(0.0, 0.0, 1.0); // Erzeugung eines Vektors
vec3 v2 = vec3(1.0, 0.0, 0.0);
float f = dot( v1, v2 );        // Skalarprodukt
vec3 u = myMatrix * v1;         // Matrix * Vektor
```

- Kontrollstrukturen: wie in C, Java, etc.
  - **`if ( bool expression ) { ... } else { ... }`**
  - **`for ( initialization; bool expression; loop expr ) { ... }`**

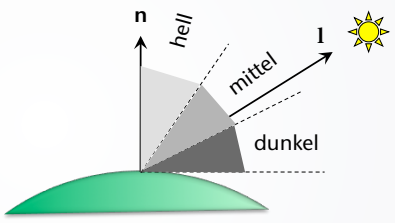
Der Shader-Editor

- Shader-Entwicklung passiert üblicherweise mit einer eigenen Programmierumgebung
- Übernimmt u.a. folgende Aufgaben:
  - Shader compilieren und linken
  - Geometrie laden und an OpenGL übergeben
  - Lichtquellen und ggf. Texturen setzen
  - Interaktion zwischen User und Szene
- Unser Shader-Editor "Shader Maker":
  - Cross-Platform (Mac, Linux, Windows)
  - Light-weight



Beispiel: Shader für Cartoon-artiges Shading

- Normalerweise gilt bei diffusen Oberflächen:
  rückgestrahltes Licht ~ cos(Winkel zw. Normale und Lichtvektor)
- Idee: nur einige
  wenige Stufen zulassen

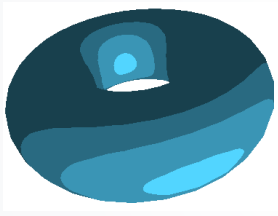


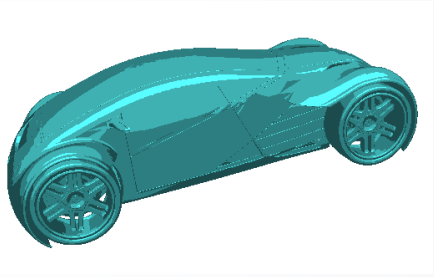
$$\text{Obj.farbe} = \text{Lichtfarbe} \cdot \begin{cases} 1.0 & \text{, falls } \mathbf{l}\cdot\mathbf{n} > 0.95 \\ 0.7 & \text{, falls } \mathbf{l}\cdot\mathbf{n} > 0.7 \\ 0.5 & \text{, falls } \mathbf{l}\cdot\mathbf{n} > 0.4 \\ 0.3 & \text{, sonst} \end{cases}$$

- Das Shader-Programm (im Wesentlichen):

```
vec3 l = normalize( gl_LightSource[0].position.xyz );
vec3 n = normalize( normal );
float cos_winkel = dot( l, n );
float c;
if (cos_winkel > 0.95)
    c = 1.0;
else if (intensity > 0.7)
    c = 0.7;
else if (intensity > 0.4)
    c = 0.5;
else
    c = 0.3;
gl_FragColor = c * vec4( color, 1.0);
```


- Resultat:

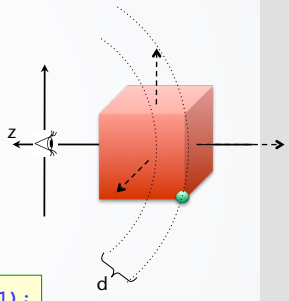




Beispiel: Shader zur Erzeugung einer Tiefenkarte

- Idee: färbe die Eckpunkte mit der "Tiefe" des jew. Eckpunktes
- Der "Algorithmus" (= Definition der Tiefe):
  - Tiefe zählt immer vom Auge aus → transformiere Eckpunkte ins Kamera-koord.system:

```
posInCS = gl_ModelViewMatrix * gl_Vertex;
```



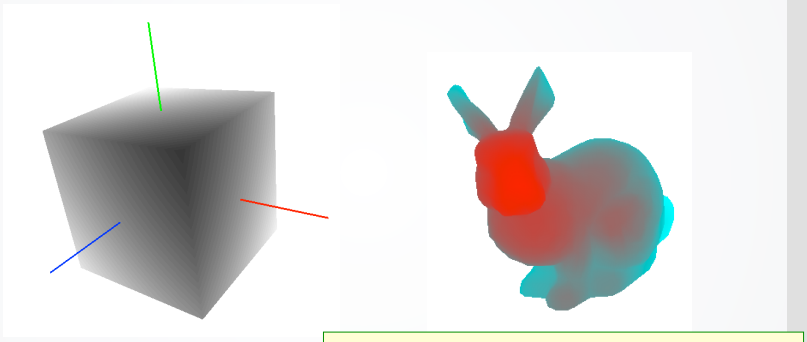
  - Beziehe Entfernung vom Auge auf die Entfernung des Obj.mittelpunktes (wir kennen die Ausdehnung des Obj. nicht):

```
centerInCS = gl_ModelViewMatrix * vec4(0,0,0,1);
dist = length(posInCS) - length(centerInCS);
```

  - Das Resultat als Grauwert setzen:

```
gl_FrontColor = vec4( dist, dist, dist, 1 );
```


- Resultat:

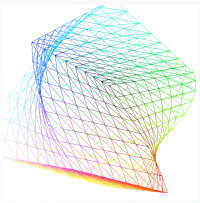


```
gl_FrontColor = vec4( 1.0-dist, dist, dist, 1 );
```

- Weitere Beispiele:

Farbe = Normale
Vertex-Shader: 1 Zeile
Fragment-Shader: 3 Zeilen

Volles Beleuchtungsmodell pro Pixel
Vertex-Shader: 4 Zeilen
Fragment-Shader: 9 Zeilen



Verschraubung
Vertex-Shader: 5 Zeilen
Fragment-Shader: 1 Zeile

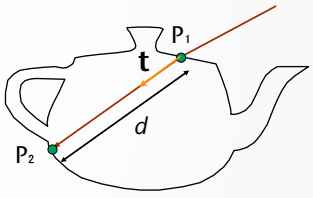
Ersetze die y- und z-Koord. jedes Eckpunktes durch

$$\begin{pmatrix} y' \\ z' \end{pmatrix} = \begin{pmatrix} \cos(t) & \sin(t) \\ -\sin(t) & \cos(t) \end{pmatrix} \begin{pmatrix} y \\ z \end{pmatrix}$$

wobei $t = f \cdot x$

Ausblick

- Brechung von Lichtstrahlen:

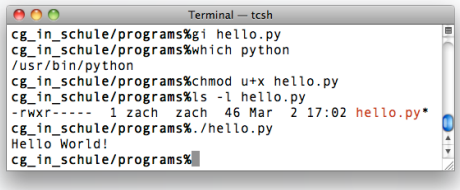


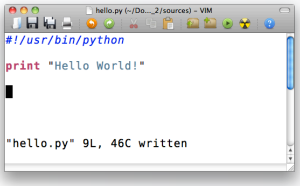
Einfache 2D-Bildsynthese mit Python

- Einige der Vorteile von Python:
  - Skript-Sprache
    → sehr schnelle Write-~~Compile~~-Debug-Zyklen
  - Inzwischen relativ weit verbreitet (z.B. eine der 3 "offiziellen" Sprachen von Google)
  - Very-high-level-Sprache → Konzentration auf die Algorithmen
  - Ist unter Linux & Mac vorinstalliert

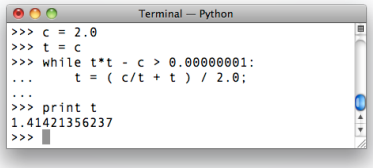
- Hier das Hello-World-Programm von Python:

- Keine Einarbeitung in ein IDE nötig:





- Zum schnellen Ausprobieren gibt es eine interaktive Python-Shell, mit Kommandozeilen-History und Kommandozeilen-Editor



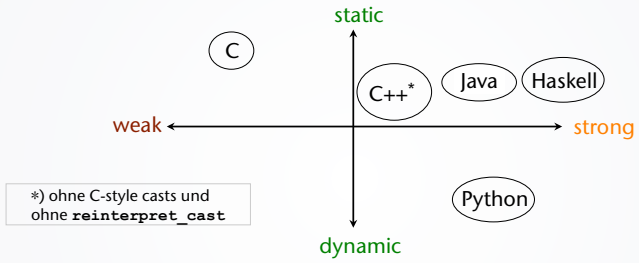
Ein kurzer Überblick über Python

- Ein kleiner, bemerkenswerter syntaktischer Unterschied: Blocks werden durch Einrücktiefe begrenzt, nicht durch Klammerung!

```
for i in range(20):
    if i%3 == 0:
        print i
        if i%5 == 0:
            print "Bingo!"
    print "---"
```

```
for (i = 0; i < 20; i++)
{
    if (i%3 == 0)
    {
        printf("%d\n", i);
        if (i%5 == 0)
            printf("Bingo!\n");
    }
    printf("---\n");
}
```


- Python ist dynamisch typisiert (und stark sowieso):



- Konsequenzen:
  - Der Typ steht beim Wert im Speicher, nicht in der Symboltabelle des Compilers
  - Generic Programming bzw. Polymorphie wird (fast) trivial
  - Variablen müssen vom Programmierer *nicht* deklariert werden

Exkurs: ein Beispiel aus einer statisch typisierten Sprache (Ada) ...

```
...
declare
  vertical_veloc_sensor: float;
  horizontal_veloc_sensor: float;
  vertical_veloc_bias: integer;
  horizontal_veloc_bias: integer;
  ...
begin
  declare
    pragma suppress(numeric_error, horizontal_veloc_bias);
  begin
    sensor_get( vertical_veloc_sensor );
    sensor_get( horizontal_veloc_sensor );
    vertical_veloc_bias := integer( vertical_veloc_sensor );
    horizontal_veloc_bias := integer(horizontal_veloc_sensor);
    ...
  exception
    when numeric_error => calculate_vertical_veloc();
    when others => use_irs1();
  end;
end irs2;
```


... und das Resultat

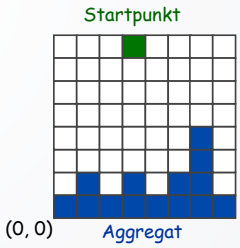
Start der ersten Ariane-5, 1996

Weitere Features von Python

- Datentypen:
  - Die üblichen Verdächtigen
  - Plus: höhere Datenstrukturen, z.B. Listen, Sets, und Dictionaries
- Kontrollstrukturen:
  - Dito (die Syntax sieht leicht anders aus, z.B. keine Klammern nötig)
  - Plus: Erweiterungen, um über die höheren Datenstrukturen bequem iterieren zu können
- Objektorientiertes Programmieren
- ...

Fraktales Wachstum in Python

- Es muss nicht immer Mandelbrot sein ...
- Beispiele für fraktales Wachstum:
  - Anlagerung von Rußteilchen an Wänden und Kaminen
  - Korallenwachstum, u.v.m.
- Der simple Ansatz: Monte-Carlo-Simulation
  1. Erzeuge einen Partikel am *Startpunkt*
  2. Der Partikel wandert zufällig durch das 2-D Gitter bis
     - er einen anderen Partikel berührt ⇒ dann wird er dort angelagert
     - er das Gitter wieder verlässt
  3. Gehe wieder zu 1.

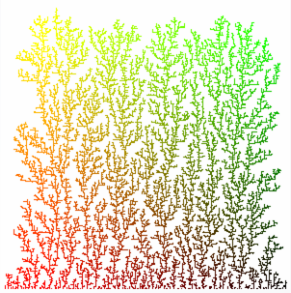


Der wesentliche Teil des Programms

```
done = False
while not done:

   x = random.randint( 0, N )              # init new particle
   y = launch                              # at the top line ofthe grid

   # perform random walk, until outside grid, or sticking to the plant
   while (x < N - 2) and (x > 1) and (y < N - 2) and (y > 1):
      r = random.random()
      if r < 0.25:                         # make 1 random step
          x -= 1                           # left, right, up, or down
      elif r < 0.50:
          x += 1
      elif r < 0.65:
          y += 1
      else:
          y -= 1

      # check whether particle touches the "plant"
      if grid[x-1][y] or grid[x+1][y] or grid[x][y-1] or grid[x][y+1]:

          grid[x][y] = True                # occupy grid cell
          im.putpixel( (x,y), (1,1,1) )    # draw new particle
          if y > launch:                   # height of plant > grid
              done = True                  # -> stop growth
          break                            # start with new particle
```

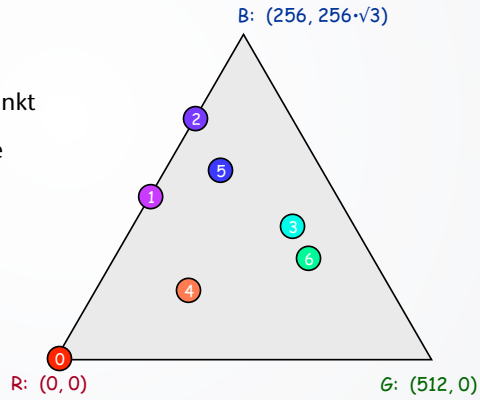

Ausblick

- Mit Farben experimentieren
- Höhere Datenstrukturen verwenden, z.B. Paare (x,y) für die Koordinaten der Partikel
- Das Gitter als Klasse implementieren
  - Ausgabebild → Instanzvariable
  - Methoden z.B.: Konstruktor, isInGrid, touchesPlant, ...
- Bewegung der Partikel ändern, z.B.:
  - Vom Rand zum Zentrum
  - Bestehende Struktur wirkt wie "Magnet"



Ein einfaches iteriertes Funktionensystem

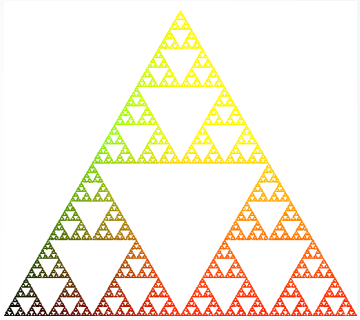
- Ein Spiel in einem gleichseitigem Dreieck, dessen Ecken rot (R), grün (G) und blau (B) eingefärbt sind
- Starte bei Punkt R
- Wiederhole:
  - Wähle zufällig einen Eckpunkt
  - Gehe die Hälfte der Strecke zwischen momentanem Standpunkt und dem ausgewählten Eckpunkt
  - Male dort einen Punkt



Das Programm

```
im = Image.new("RGB", (512,512), (256,256,256) )

x = 0.0                          # starte beim Eckpunkt "Rot"
y = 0.0

for i in range( 0, 100000 ):     # Pfad des Punktes

    r = random.random()          # zufällige Ecke des Dreiecks
    if r < 0.333:
        ex = 0.0
        ey = 0.0
    elif r < 0.6667:
        ex = 512.0
        ey = 0.0
    else:
        ex = 256.0
        ey = 443.4

    x = ( ex + x ) / 2.0         # halber Weg zwischen x und e
    y = ( ey + y ) / 2.0

    im.putpixel ( (int(x), int(y)), (int(x), int(y), 0) )

im.show()
```

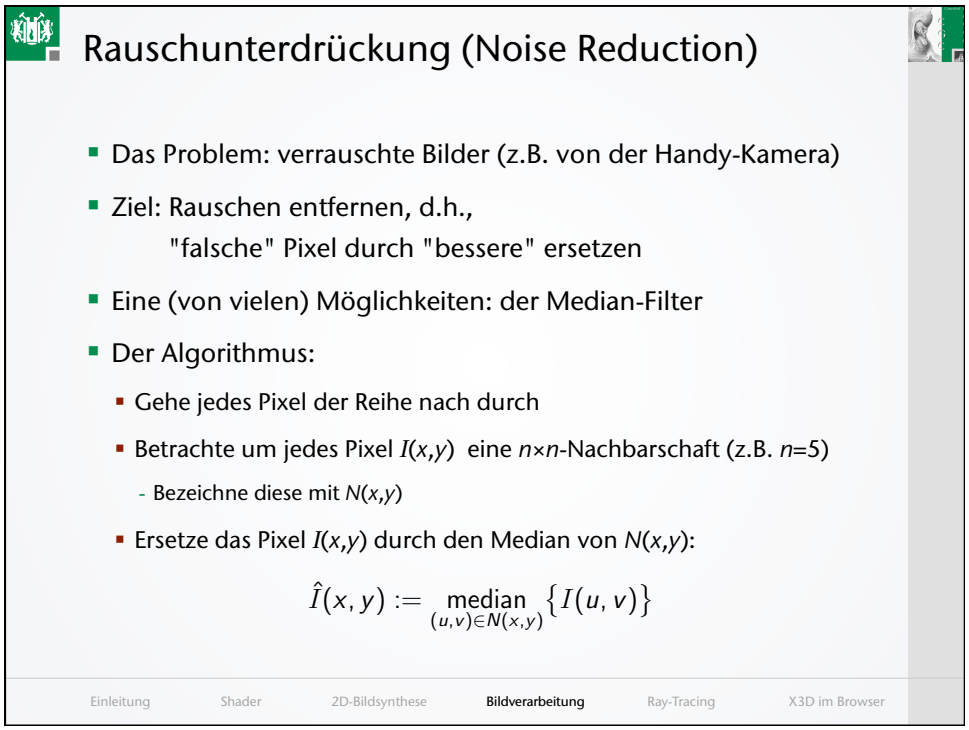
Das Ergebnis und Ausblick



- Das Ergebnisbild:
- Weiterführende Experimente:
  - Was passiert, wenn man einen zufälligen Punkt im Inneren des ursprünglichen RGB-Dreiecks als Startpunkt nimmt?
  - Was passiert, wenn man nicht immer genau den Mittelwert zwischen aktuellem Punkt und Ecke als nächsten Punkt wählt?
  - Klasse für 2D-Punkte einführen
  - Film mit animierten Ecken des Dreiecks

Einfache Bildverarbeitung mit Python

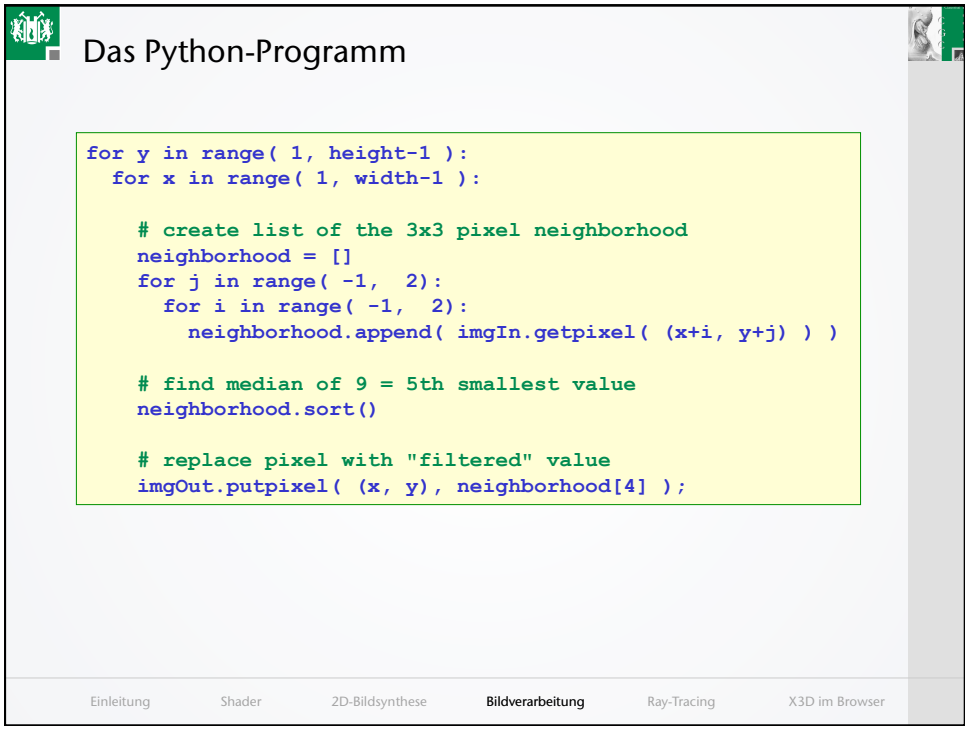
- Vereinfachung (zunächst): nur Grauwertbilder (= 1 Float/Pixel)
- Bild := ein 2D-Array von Pixeln $I(x, y) \in [0, 1]$
  oder $I(x, y) \in [0, 255]$
- Später dann einfach auf R-, G-, und B-Kanal anwenden



Rauschunterdrückung (Noise Reduction)

- Das Problem: verrauschte Bilder (z.B. von der Handy-Kamera)
- Ziel: Rauschen entfernen, d.h.,
  "falsche" Pixel durch "bessere" ersetzen
- Eine (von vielen) Möglichkeiten: der Median-Filter
- Der Algorithmus:
  - Gehe jedes Pixel der Reihe nach durch
  - Betrachte um jedes Pixel $I(x,y)$ eine $n \times n$-Nachbarschaft (z.B. $n=5$)
    - Bezeichne diese mit $N(x,y)$
  - Ersetze das Pixel $I(x,y)$ durch den Median von $N(x,y)$:

$$\hat{I}(x,y) := \underset{(u,v)\in N(x,y)}{\text{median}} \{I(u,v)\}$$



Das Python-Programm

```
for y in range( 1, height-1 ):
  for x in range( 1, width-1 ):

    # create list of the 3x3 pixel neighborhood
    neighborhood = []
    for j in range( -1,  2):
      for i in range( -1,  2):
        neighborhood.append( imgIn.getpixel( (x+i, y+j) ) )

    # find median of 9 = 5th smallest value
    neighborhood.sort()

    # replace pixel with "filtered" value
    imgOut.putpixel( (x, y), neighborhood[4] );
```

Resultate

Gimp-Filter = Random Hurl

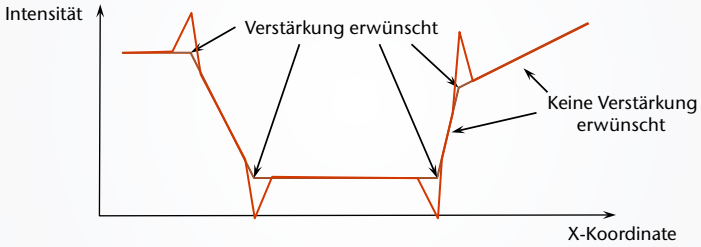
Median-Filter

Weiterführende Arbeiten

- Eine eigene Bild-Klasse schreiben, die einen generischen Faltungsoperator anbietet
  - Lässt sich in Python sehr leicht machen, da Funktionen vollwertige Objekte sind, die an andere Funktionen übergeben werden können
- Auf RGB-Bilder erweitern (einfach unabhängig auf jeden Kanal)
- Andere Filter ausprobieren
- Einen Filter mehrfach anwenden
- Bzgl. Performance optimieren

Sharpening (Schärfung) eines Bildes

- Betrachte zunächst nur eine Zeile eines Bildes
- Ziel:



- Das tut genau die zweite Ableitung!
- Wie berechnet man Ableitungen von einer diskreten Funktion?
  - Steigung der Funktion ist

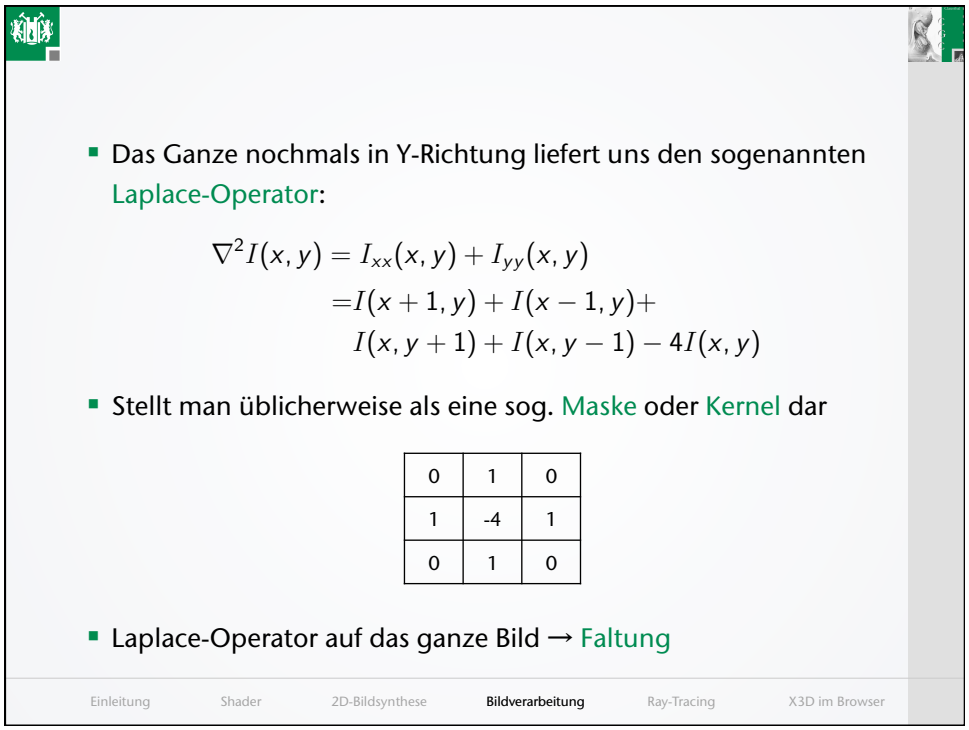
$$I'(x) = I_x(x) \approx \frac{I(x+h) - I(x)}{h}$$

  - Kleinstes h in einem Bild ist 1 Pixel, also

$$I_x(x) = I(x+1) - I(x)$$

  - Ableitungsvorschrift nochmals auf I_x anwenden und symmetrisch um x machen liefert zweite Ableitung

$$I_{xx}(x) = I(x+1) - 2I(x) + I(x-1)$$



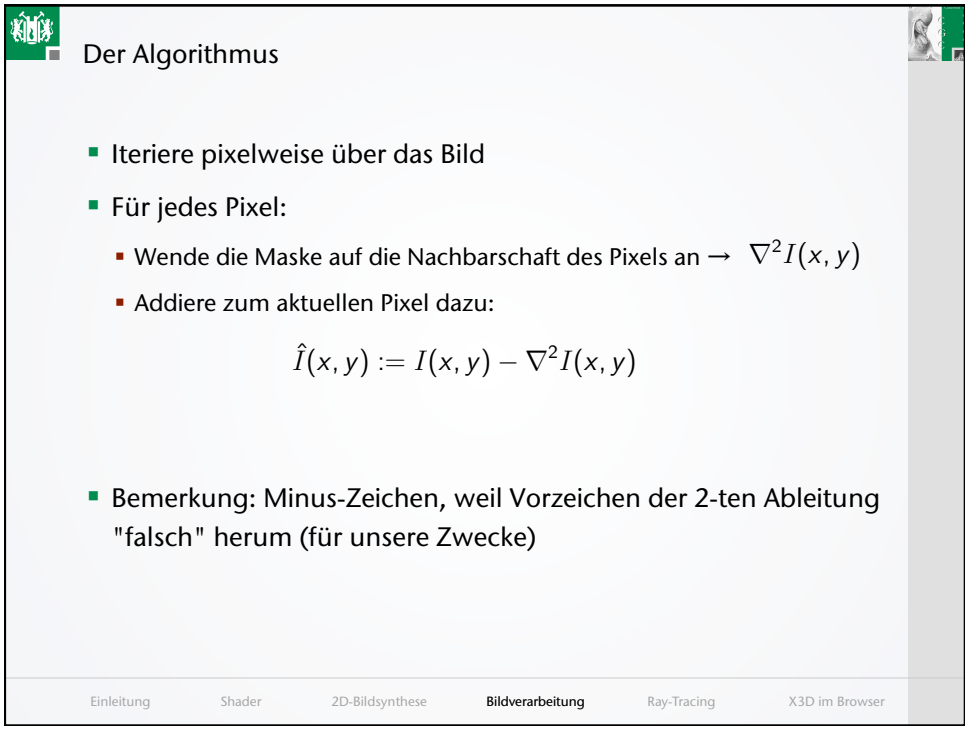
- Das Ganze nochmals in Y-Richtung liefert uns den sogenannten Laplace-Operator:

$$\begin{aligned}\nabla^2 I(x,y) &= I_{xx}(x,y) + I_{yy}(x,y)\\ &= I(x+1,y) + I(x-1,y) +\\ &\quad I(x,y+1) + I(x,y-1) - 4I(x,y)\end{aligned}$$

- Stellt man üblicherweise als eine sog. Maske oder Kernel dar

0	1	0
1	-4	1
0	1	0

- Laplace-Operator auf das ganze Bild → Faltung



Der Algorithmus

- Iteriere pixelweise über das Bild
- Für jedes Pixel:
  - Wende die Maske auf die Nachbarschaft des Pixels an → $\nabla^2 I(x,y)$
  - Addiere zum aktuellen Pixel dazu:

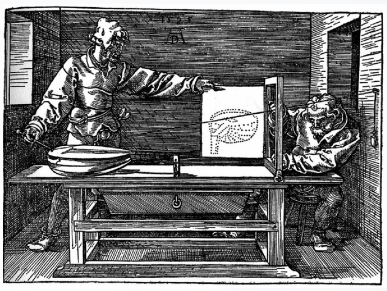
$$\hat{I}(x,y) := I(x,y) - \nabla^2 I(x,y)$$

- Bemerkung: Minus-Zeichen, weil Vorzeichen der 2-ten Ableitung "falsch" herum (für unsere Zwecke)

Resultate

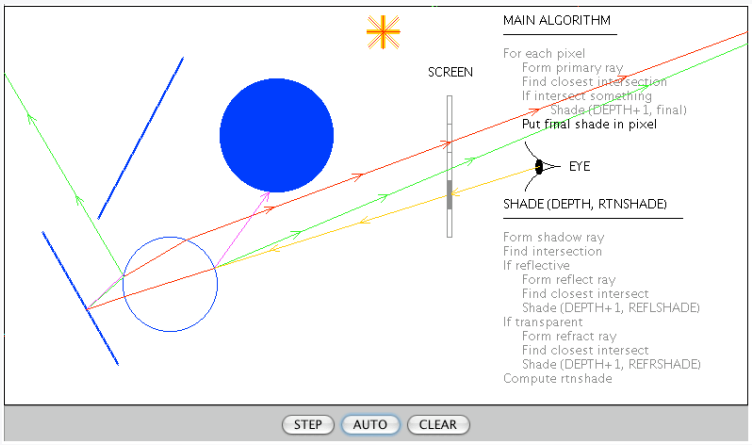
Einfaches Ray-Tracing

- Das Grundprinzip: Strahlverfolgung
- Der Algorithmus:
  1. Schieße Strahlen vom Augpunkt aus durch die Pixel in die Szene
  2. Falls der Strahl mehr als ein Objekt schneidet, betrachte nur den ersten Schnittpunkt



  3. Schieße weitere Strahlen von dort zu allen Lichtquellen (Schattenstrahlen)
  4. Treffen diese Schattenstrahlen auf ein Objekt, so liegt der betrachtete Flächenpunkt im Schatten. Andernfalls werte das Beleuchtungsmodell aus
  5. Ist das getroffene Objekt spiegelnd, dann schieße einen reflektierten Strahl in die Szene → Rekursion
  6. Ist das Objekt transparent, verfolge zusätzlich einen gebrochenen Strahl → Rek.

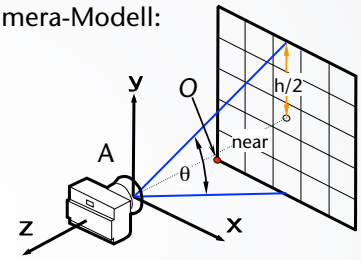
Interaktive Demo



http://www.siggraph.org/education/materials/HyperGraph/raytrace/rt_java/raytrace.html

- Die Main-Loop und das einfache Kamera-Modell:

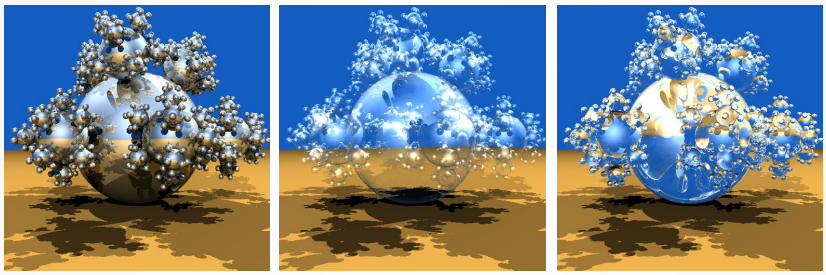
```
for ( i = 0; i < height; i ++ )
  for ( j = 0; j < width; j ++ )
    ray.from = A
    t = (i/height - 0.5) * h
    s = (j/width - 0.5)  * w
    ray.at = O + s·x + t·y
    ray.dir = ray.at - ray.from;
    color = trace( 0, ray );
    putPixel( x, y, color );
```



$$\frac{h}{2} = \text{near} \cdot \tan\frac{\theta}{2}$$

$$O = A - \text{near}\cdot\mathbf{z} - \frac{w}{2}\mathbf{x} - \frac{h}{2}\mathbf{y}$$

- Schon mit sehr einfachen Geometrien kann man sehr ansprechende Bilder erstellen:



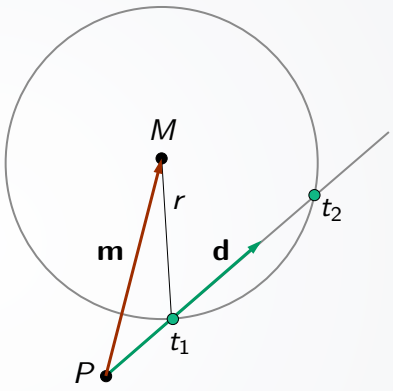
Schnitt Strahl—Kugel

- Strahl ist gegeben durch Aufpunkt P und Richtung **d**
- Die geometrische Methode:

$$\|t\cdot\mathbf{d}-\mathbf{m}\| = r$$

$$(t\cdot\mathbf{d}-\mathbf{m})^2 = r^2$$

$$t^2 - 2t\cdot\mathbf{md} + \mathbf{m}^2 - r^2 = 0$$

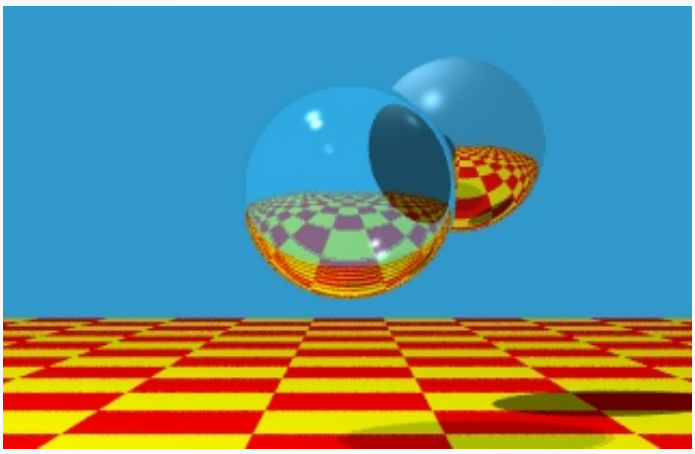


- Es gibt noch andere Ansätze ...

- Der Algorithmus, mit kleinen Optimierungen:

```
berechne m²−r²
berechne b = m·d
if  m²−r²  >= 0        // Blickpunkt ausserhalb Kugel
    and b <= 0 :       // und sieht von Kugel weg
then
     return "kein Schnittpunkt"
setze d = b² − m² + r²
if d < 0:
    return "kein Schnittpunkt"
if m²−r² > ε :
    return t1 = b − √d  // enter; t1 is > 0
else:
    return t2 = b + √d  // leave; t2 is > 0   (t1<0)
```


- Es ist so einfach, daß alle Ray-Tracer Kugeln als Primitiv haben ;-)



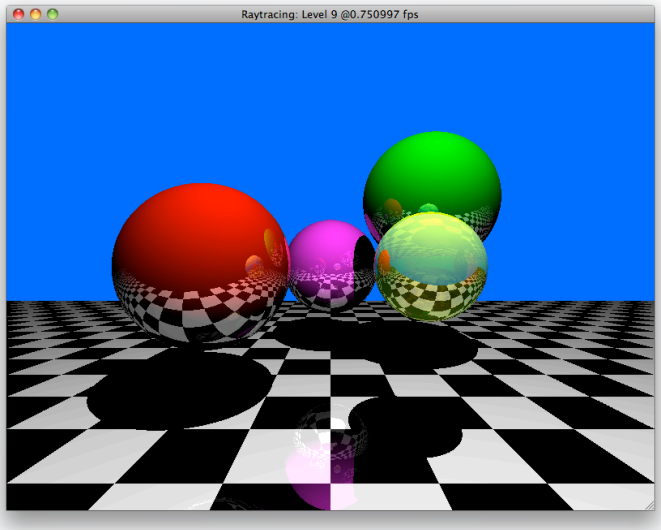
Die vollständige Ray-Tracing-Routine

```
traceRay( ray ):
  hit = intersect( ray )
  if no hit:
    return no color
  reflected_ray = reflect( ray, hit )
  reflected_color = traceRay( reflected_ray )
  refracted_ray = refract( ray, hit )
  refracted_color = traceRay( refracted_ray )
  for each lightsource[i]:
    shadow_ray = compShadowRay( hit, lightsource[i] )
    if intersect(shadow_ray):
      light_color[i] = 0
  overall_color = shade( hit,
                         reflected_color,
                         refracted_color,
                         light_color )
  return overall_color
```

hit ist eine Datenstruktur, die alle Infos über einen Schnitt zwischen Strahl und Szene enthält, u.a. Schnittpunkt, Objekt, Normale, ...

Wertet die Beleuchtungsgleichung für das getroffene Obj aus

Demo

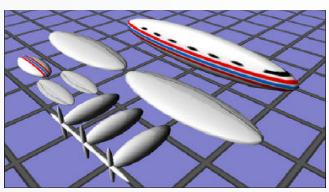


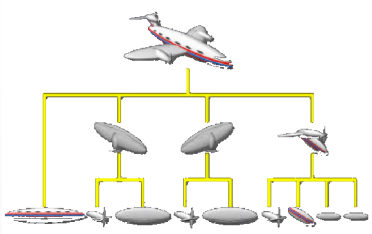
Weiterführende Arbeiten

- Unsere Methode:
  - Framework zur Verfügung stellen
  - Einen Teil des Ray-Tracers weglassen → Aufgabe für die Studenten
- Andere Objekte (z.B. Zylinder oder Dreiecke)
- Kamerafahrten (Z.B. linear zwischen Punkten interpolieren)
- Etwas komplexere Beleuchtungsmodelle (z.B. mit spekularer Reflexion)
- Etwas komplexeres Brechungsgesetz (Fresnel-Term)
- Adaptive Abbruchkriterien bei der Strahlverfolgung
- Optimierungen (z.B. schnellere Intersect-Routine für die Schattenstrahlen)

Szenengraphen im Browser

- Alle zukünftigen Browser werden polygonale 3D-Geometrie rendern können ("WebGL-enabled browsers")
  - WebGL = OpenGL-Bindings für Javascript
  - Chrome 9 (Mac, Linux, Windows) kann es schon jetzt
  - Firefox 4 beta 7 (Mac, Linux, Windows) und aufwärts
  - Webkit (= Safari beta) nightly build (Mac und Windows)
- Szenengraph = hierarchische Organisation einer virtuellen Szene





Der Standard X3D

- Ein ISO-Standard, basiert auf XML und VRML
- Erlaubt die Beschreibung von Geometrie & Verhalten(!)
  - Beides steckt im selben Szenengraphen
- Browser können XML — und somit X3D — laden und intern speichern
- Der Clou:
  - Eigene in der Webseite enthaltene Javascript-Programme können auf den X3D-Szenengraphen zugreifen und sogar verändern (animieren, auf Clicks reagieren, etc.)
  - Ein in Javascript geschriebener Renderer kann somit den X3D-Szenengraphen mittels WebGL rendern
  - Solch ein Renderer kommt z.B. von X3DOM (www.x3dom.org)

Ein "Hello World" in X3D / HTML5

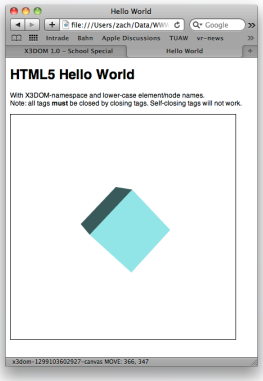
```
<!DOCTYPE html>
<html >
  <head>
    <link rel="stylesheet" type="text/css" href="x3dom.css"/>
    <script type="text/javascript" src="x3dom.js"></script>
  </head>
<body>
  <x3d>
  <scene>
    <viewpoint position='0 0 10'></viewpoint>
    <shape>
      <appearance>
        <material diffuseColor='0.603 0.894 0.909'></material>
      </appearance>
      <box DEF='box' ></box>
    </shape>
  </scene>
  </x3d>
</body>
</html>
```

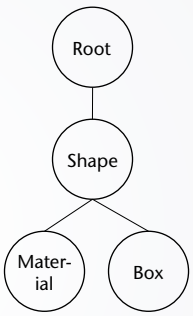
Dieses Javascript enthält den Renderer! (kommt von www.x3dom.org)

Material, das auf die Geometrie angewandt wird

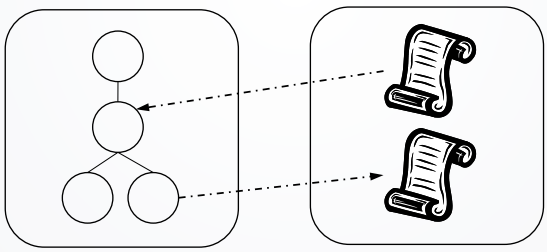
Die Geometrie

- Der zugehörige Szenengraph:
- Das Aussehen im Browser:



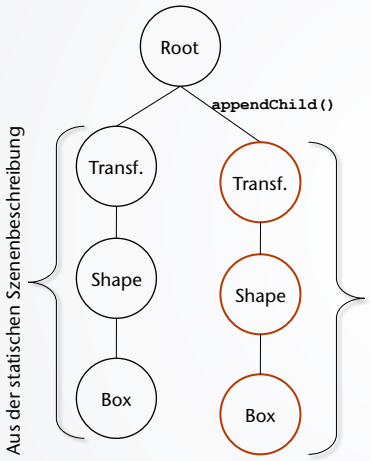


- Der X3DOM-Ansatz vereint deskriptive mit programmatischer Szenenbeschreibung
- Modifikation der Szene mit Hilfe von Javascript-Programmen
- Javascript-Funktionen können durch spezielle Knoten (z.B. "TouchSensor") im X3D-Szenengraph getriggert werden



X3D-Scenegraph Javascript-Funktionen

Beispiel für Javascript → X3D-Szene



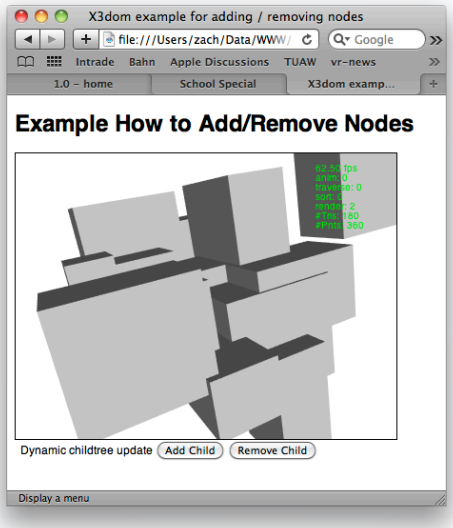

```
<scene>
  <transform id="root" translation="0 0 0">
    <shape>
      <box ></box>
    </shape>
  </transform>
</scene>
<script type="text/javascript">
function addNode()
{
  [...]
  var t = document.createElement('Transform');
  t.setAttribute("translation", x+" "+y+" "+z );
  var s = document.createElement('Shape');
  t.appendChild(s);
  var b = document.createElement('Box');
  s.appendChild(b);
  var ot = document.getElementById('root');
  ot.appendChild(t);
  return false;
};
</script>
<input type="button" value="Add Child"
       onclick="window.addNode();" />
```

Statische Szenenbeschr.

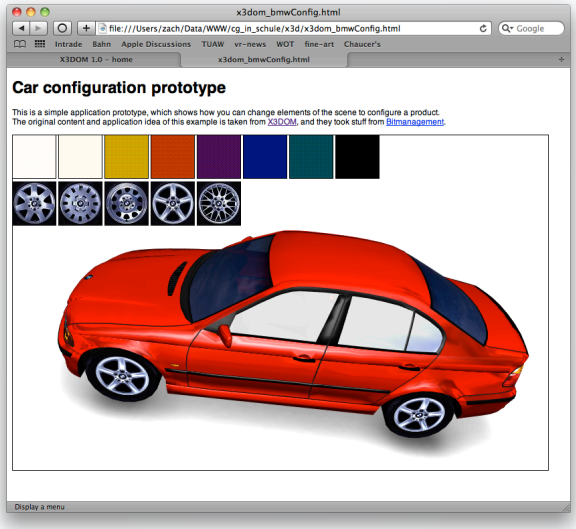
Neuen Teil-Szenen-graphen erzeugen

Einhängen

HTML-Button triggert JS



Ein sinnvolleres Beispiel



http://www.x3dom.org/

Beispiel für den Zugang zu weltweitem Kulturerbe

http://3d-coform.eu/x3dom/index.html

Weiterführende Literatur & Dokumentation

- Zu X3D (bzw. VRML, den Vorgänger):
  - Als Buch: "X3D: Extensible 3D Graphics for Web Authors"
  - Die offizielle X3D-Spezifikation: http://www.web3d.org/x3d/specifications/
  - Weitere Informationen: http://doc.instantreality.org/documentation

- Step-by-Step-Tutorials zu X3D im Browser:
  - http://www.x3dom.org/school/ (auf Deutsch)
  - http://www.x3dom.org/?page_id=482
- Weitere Beispiele:
  - http://www.x3dom.org/?page_id=5
  - http://www.x3dom.org/iX/ (auf Deutsch)
  - http://www.x3dom.org/?p=*

Vielen Dank für Ihre Aufmerksamkeit!

Die einzige, wichtige URL:

http://zach.in.tu-clausthal.de/cg_in_schule/

(Folien, weitere Beispiele, Programme, Frameworks, Links, etc.)