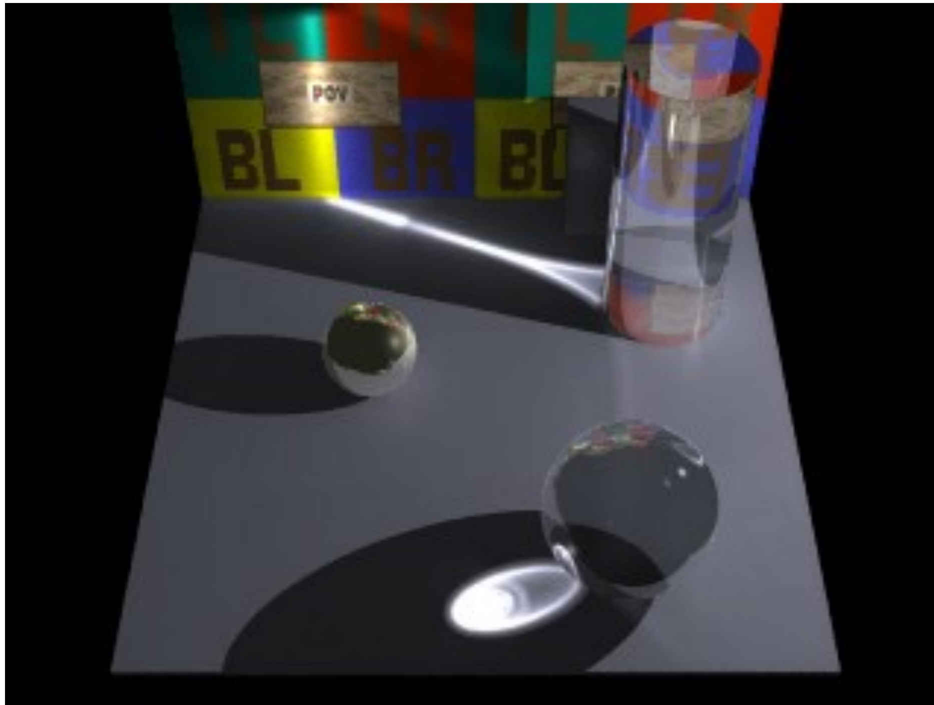
RUPRECHT-KARLS-
UNIVERSITÄT
HEIDELBERG



A Short Introduction to POV-Ray

Peter Fischer, ZITI, Heidelberg University

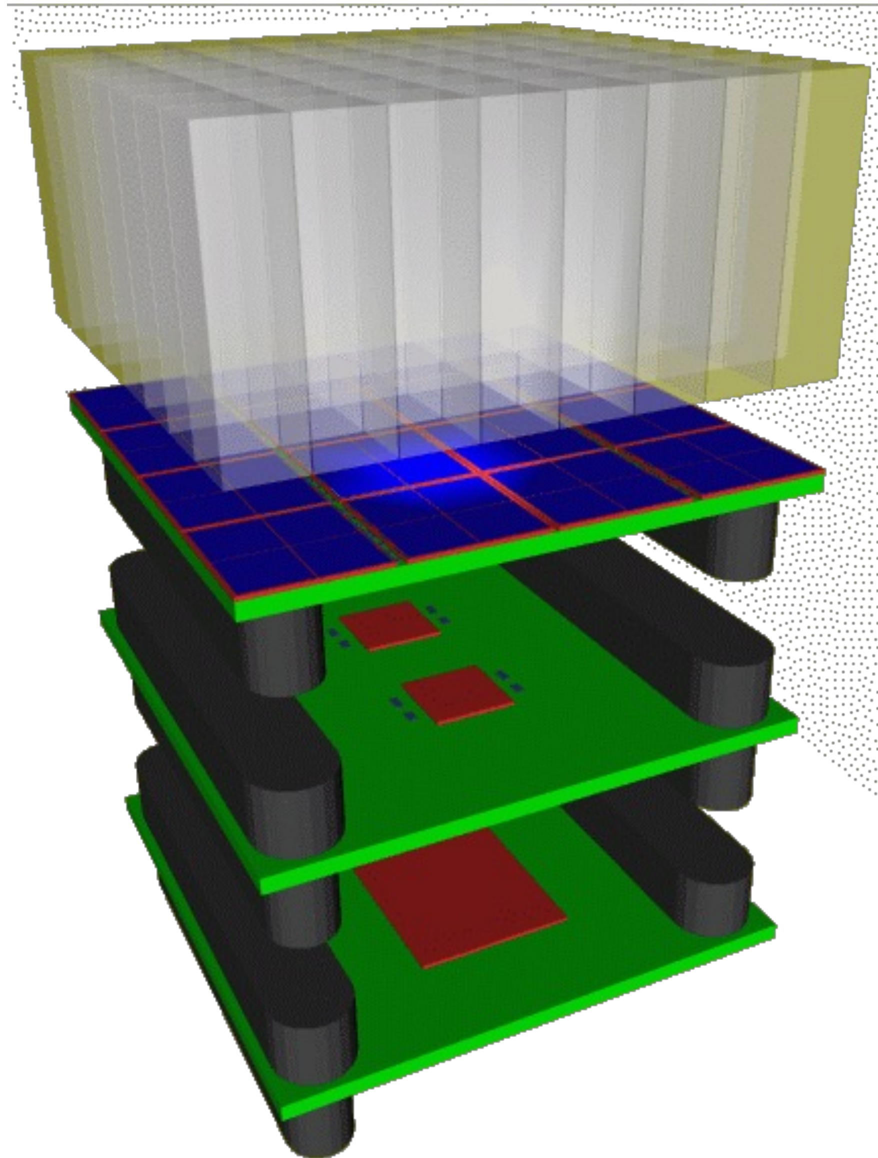
What is POV-Ray ?

- A software to produce 3D images
- Very easy to use (in my opinion)
  - Simple concept to generate complex shapes
  - Can define new objects easily
  - Can do mathematics & calculations & loops & ...
- Can obtain *very* high quality
  - Based on Ray Tracing
  - Many 3D textures
  - Many illumination schemes
- Open source standard – many examples available

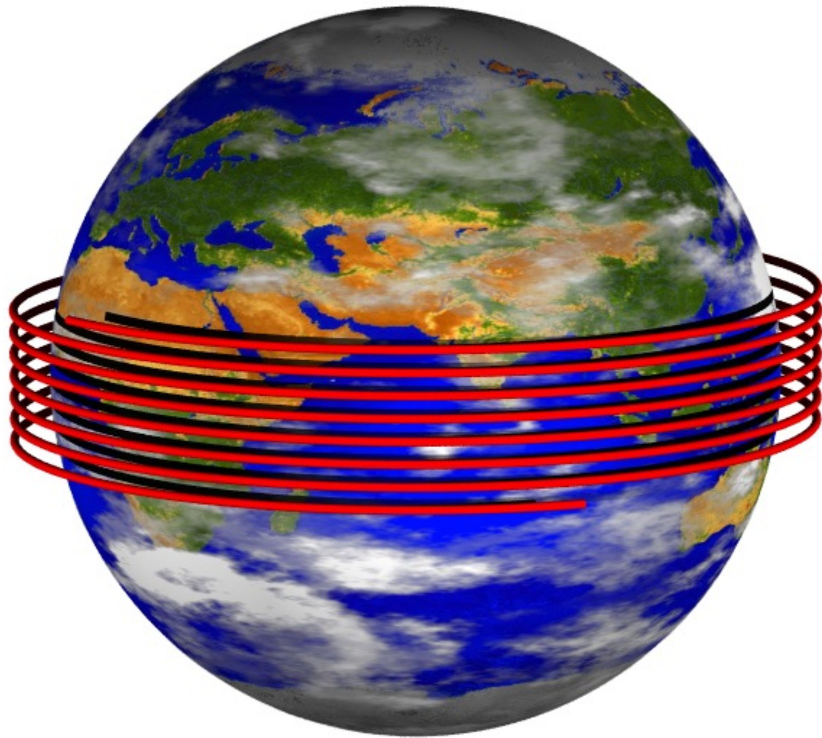
Why Use & Know About POV-Ray ?

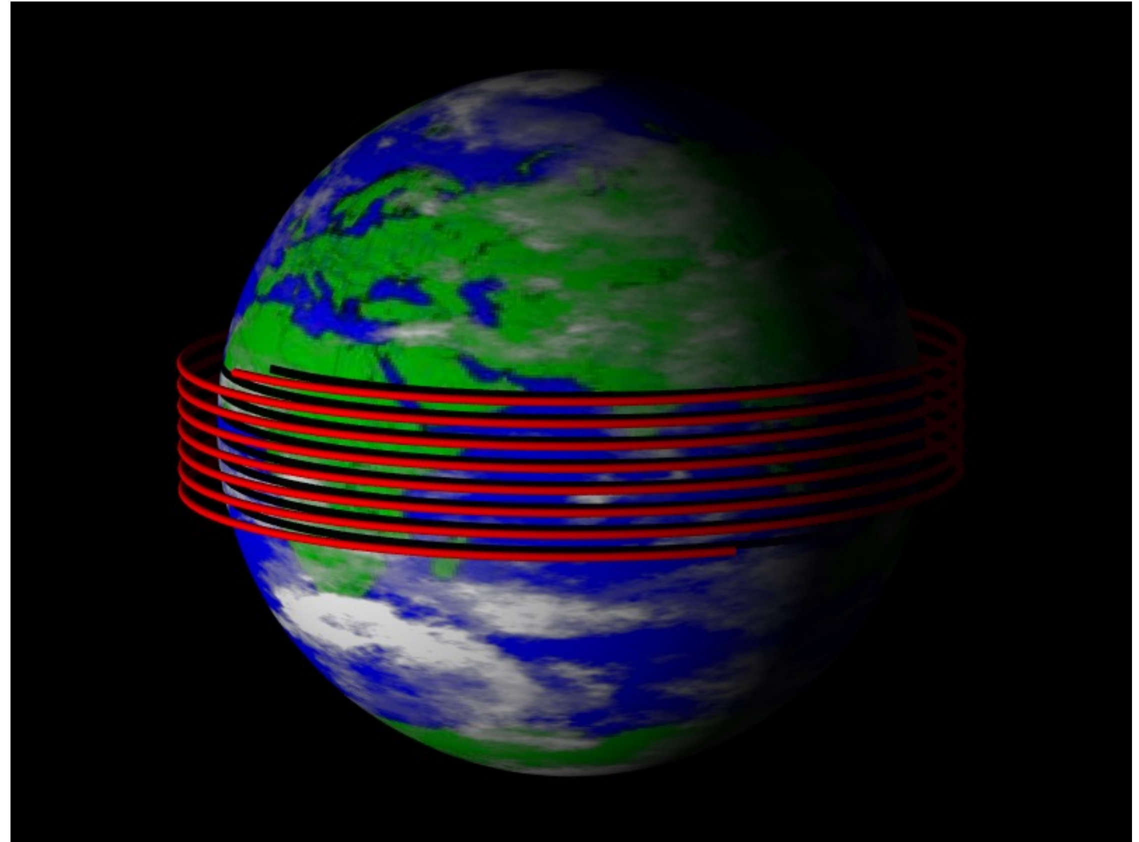
- 3D images are good to illustrate presentations or documents

Illustration of a Sensor Module



Another Example

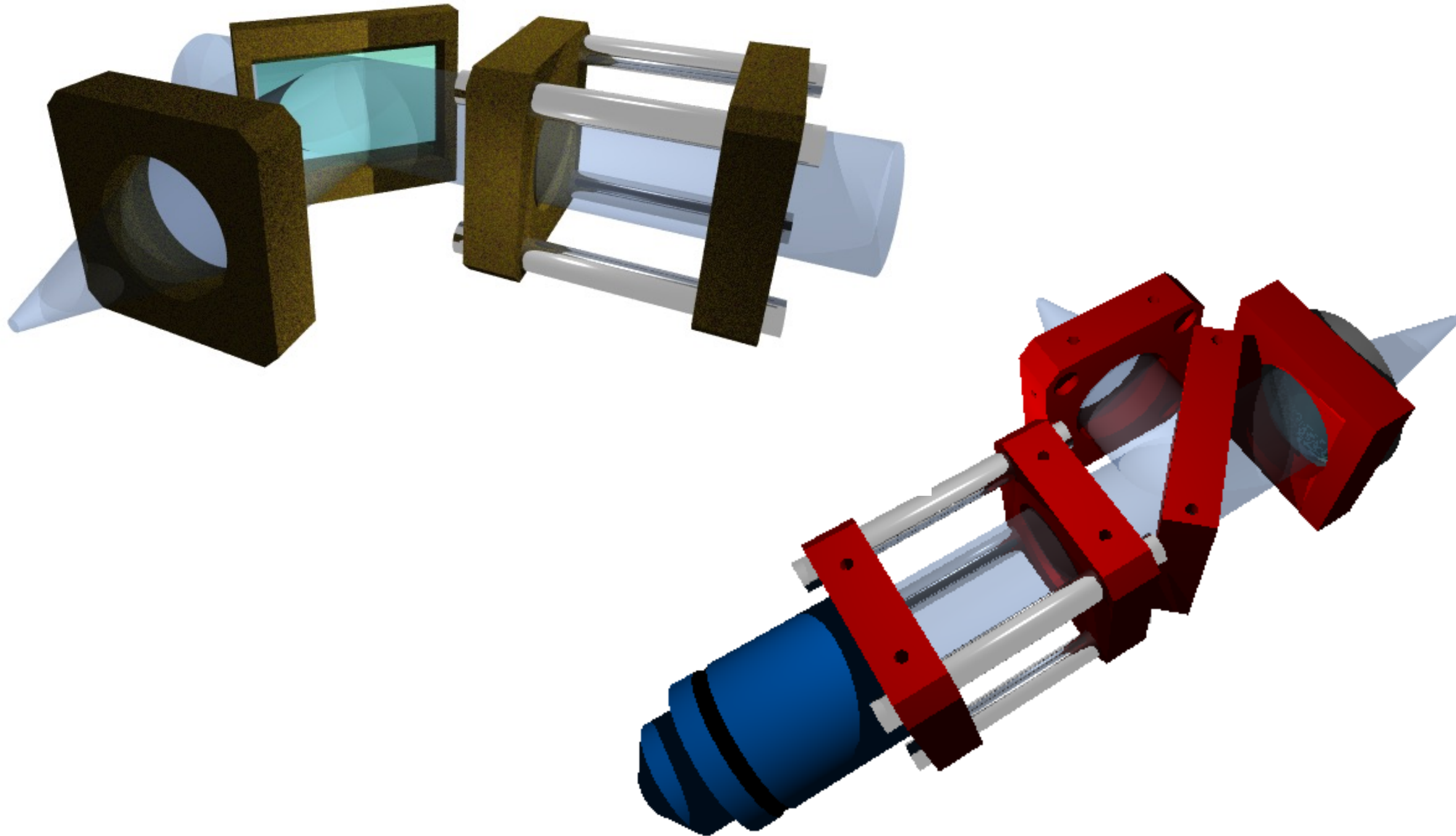




(Light path around earth in 1 second)

Illustration of your Experimental Setup

- Here: Microscope Parts



Another Example

- Taken from a PhD Thesis (N. Camus) on Optics

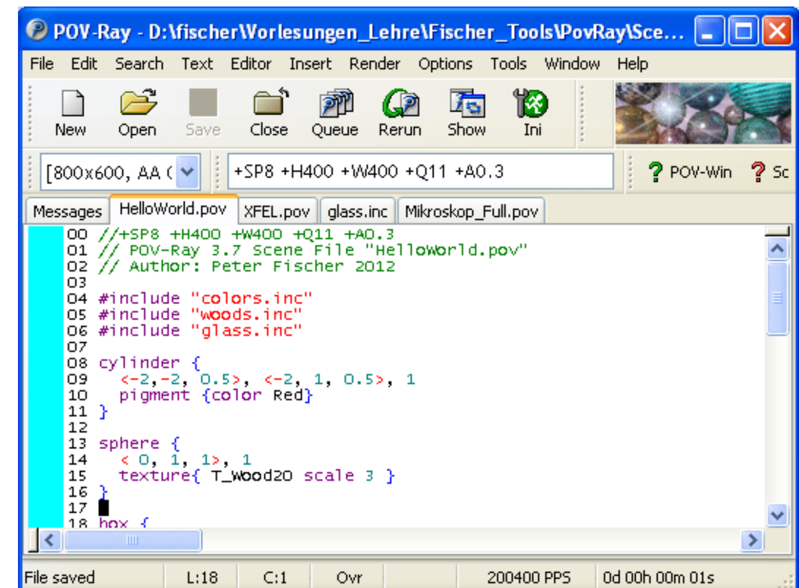
Ions detector
Helmholtz Coil
$\vec{E},\vec{B}$
Focussing mirror
pol. dir.
Collimating slits
Laser Pulse
Gas jet
Electrons detector
z
y
x

Where to get POV-Ray and help ?

- POVRAY is open source:
  - Main site: www.povray.org
  - Extensions: megapov.inetart.net
  - For Mac: megapov.inetart.net/povrayunofficial_mac
  - Wiki: wiki.povray.org/content/Main_Page
  - Tutorial: de.wikibooks.org/wiki/Raytracing_mit_POV-Ray
  - Tutorial: www.f-lohmueller.de/pov_tut/pov__ger.htm

- Windows / MAC versions comes with integrated editor

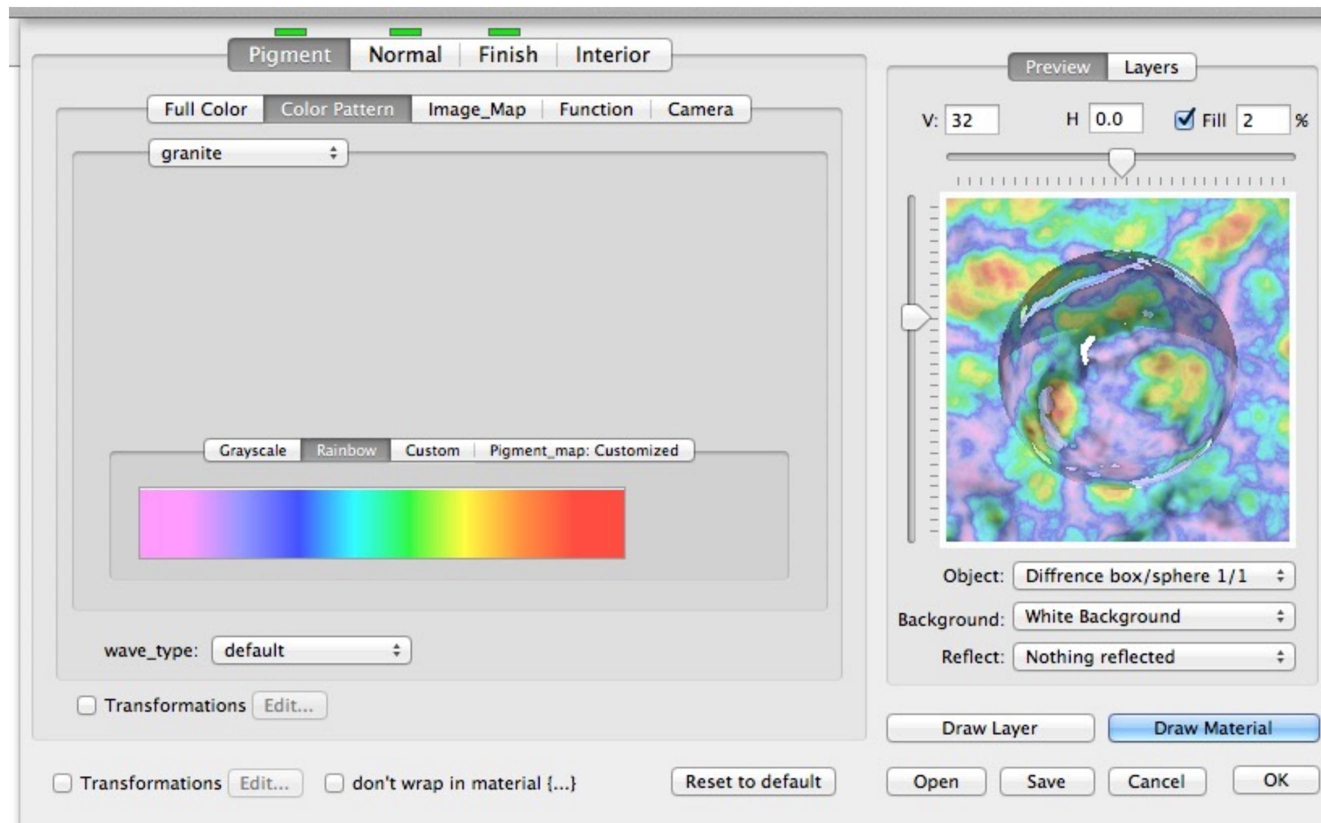
- Linux version is command line tool



RUPRECHT-KARLS-
UNIVERSITÄT
HEIDELBERG

POV-Ray for MAC

- Available since 2013. Not supported any more ☹
- 'Unofficial' version. Implements latest version 3.7
- Several additional useful dialogs:
  - Colour Editor, Texture Editor, Options Dialog, ...



RUPRECHT-KARLS-
UNIVERSITÄT
HEIDELBERG

Have a Look at this ***Phantastic*** Web-Site:

- The site www.f-lohmueller.de offers a *very wide* range of
  - Tutorials
  - Examples (with source codes)
  - Animations
  - Textures
  - Shapes
  - ... (much more)...
  - ... (much more)...

- Pages are available in German / English / French / Italian

- There are many tricky details explained on that site!

Go and see it!

RUPRECHT-KARLS-
UNIVERSITÄT
HEIDELBERG

Command Line Options (Linux, CIP Pool)

- Start for instance with

```
povray +Ifile.pov +H400 +W400 +SP8 +Q8 +A0.3 +P +fn
```

- Options shown are:

`+Ifile.pov`	: input file (do not forget '`+I`' !!!)
`+H400`	: image height in pixels
`+W400`	: image width in pixels
`+SP8`	: generate every 8th pixel first, then every 4th etc. (does not work in CIP Pool!)
`+Q8`	: quality: 1.., higher = better & slower (8=reflections...)
`+A0.3`	: anti-aliasing setting (slower)
`+P`	: pause after rendering (to admire the picture)
`+fn`	: output format = .png (default on Linux)

- To get more information:

`-H`	: show all options

RUPRECHT-KARLS-
UNIVERSITÄT
HEIDELBERG

Using a Configuration File

- Options can be put into a **`par.ini`** file.
- It can contain several *sections*. Example:

```
; POV-Ray configuration file example

      ; common options
+SP8  ; start with every 8th pixel
+Q8   ; quality is high
+A0.3 ; anti aliasing
+P    ; pause after rendering

[lo]  ; options in section '[lo]'
+W150 ; lo res image width
+H100 ; lo res image height

[hi]  ; another section
+W600 ; hi res image width
+H400 ; hi res image height
```

- Rendering using section **`hi`** in **`par.ini`** is started by:

`povray par[hi] +Ifile.pov`

RUPRECHT-KARLS-
UNIVERSITÄT
HEIDELBERG

A First Example

Red cylinder

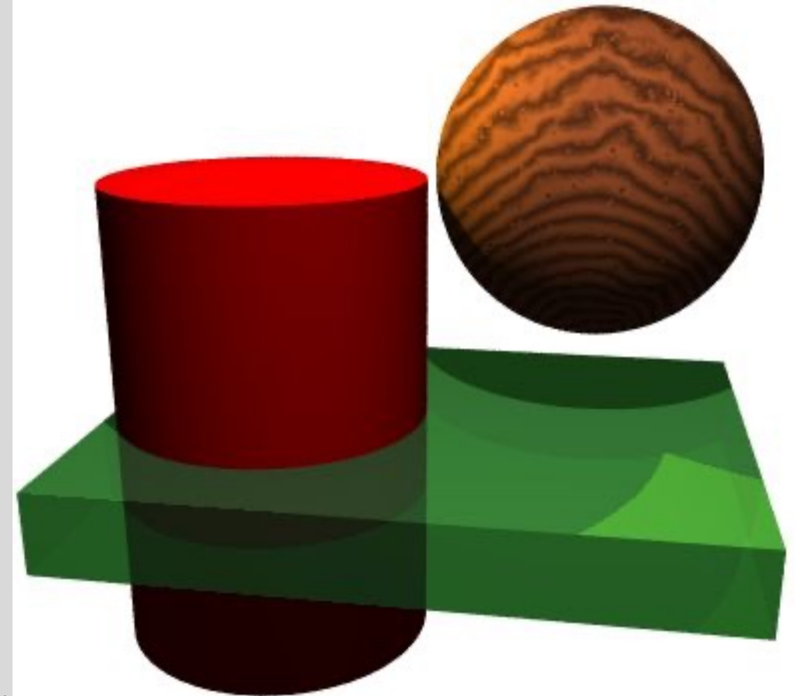
Wooden sphere

transparent box

illumination

Position of camera

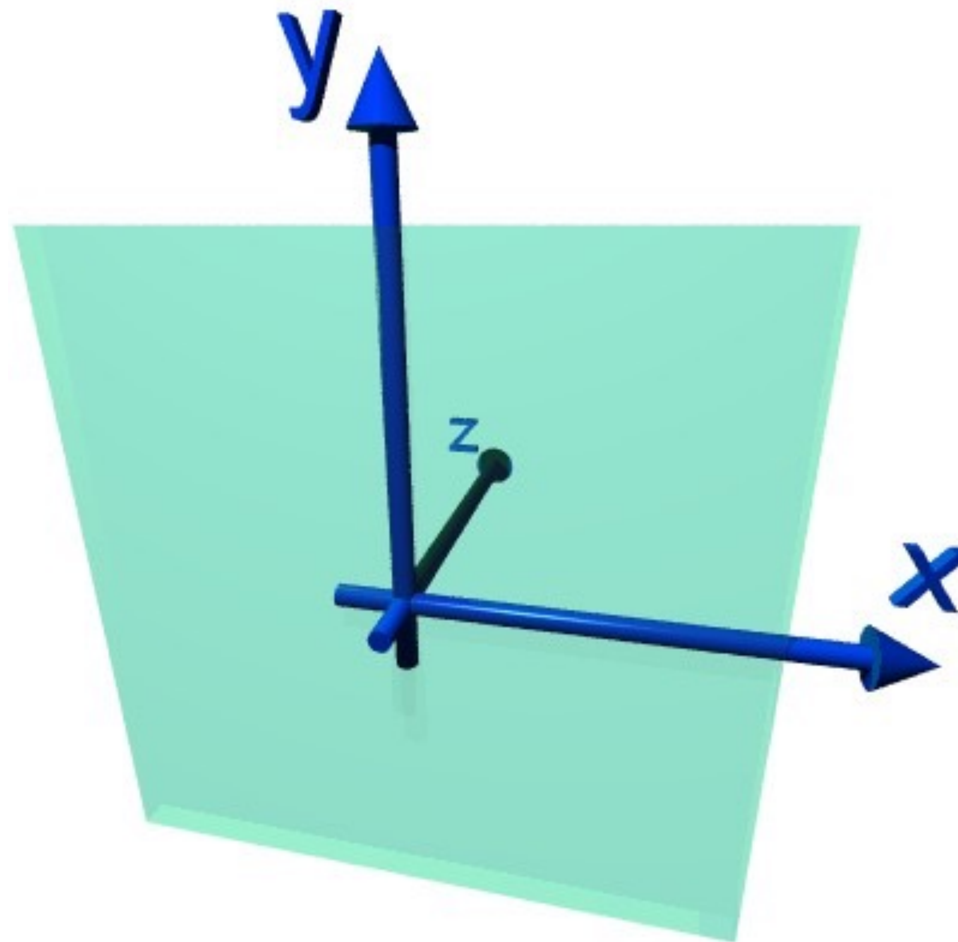
```
#include "colors.inc"
#include "woods.inc"
#include "glass.inc"
cylinder {
  <-2,-2,0.5>, <-2,1,0.5>, 1
  pigment {color Red}
}
sphere {
  <0,1,1>, 1
  texture{ T_Wood20 scale 3 }
}
box {
  <-3,-1,-1>, <1,-0.5, 2>
  pigment {Col_Glass_Winebottle}
}
background   { color White }
light_source {<0, 5, -3>   color White }
light_source {<-2, 2, 0.5> color Yellow }
camera {
  location <0, 2, -6>
  angle 50 right x
  look_at <-1, 0, 0>
}
```



HelloWorld.pov

The Coordinate System

- Remember: X and Y are like in Mathematics
- Positive Z is ‘**into** the screen’
  - note: This is a LEFT-handed coordinate system



Vectors, Directions & Rotations

- A vector (position or direction) is given by:

```
<cx,cy,cz>
```

- It can be multiplied with numbers (floats):

```
factor * <cx,cy,cz>
```

- Predefined vectors are the 3 axes:

```
x   (= <1,0,0>)
y   (= <0,1,0>)
z   (= <0,0,1>)
```

- Arithmetic expressions can be used everywhere:

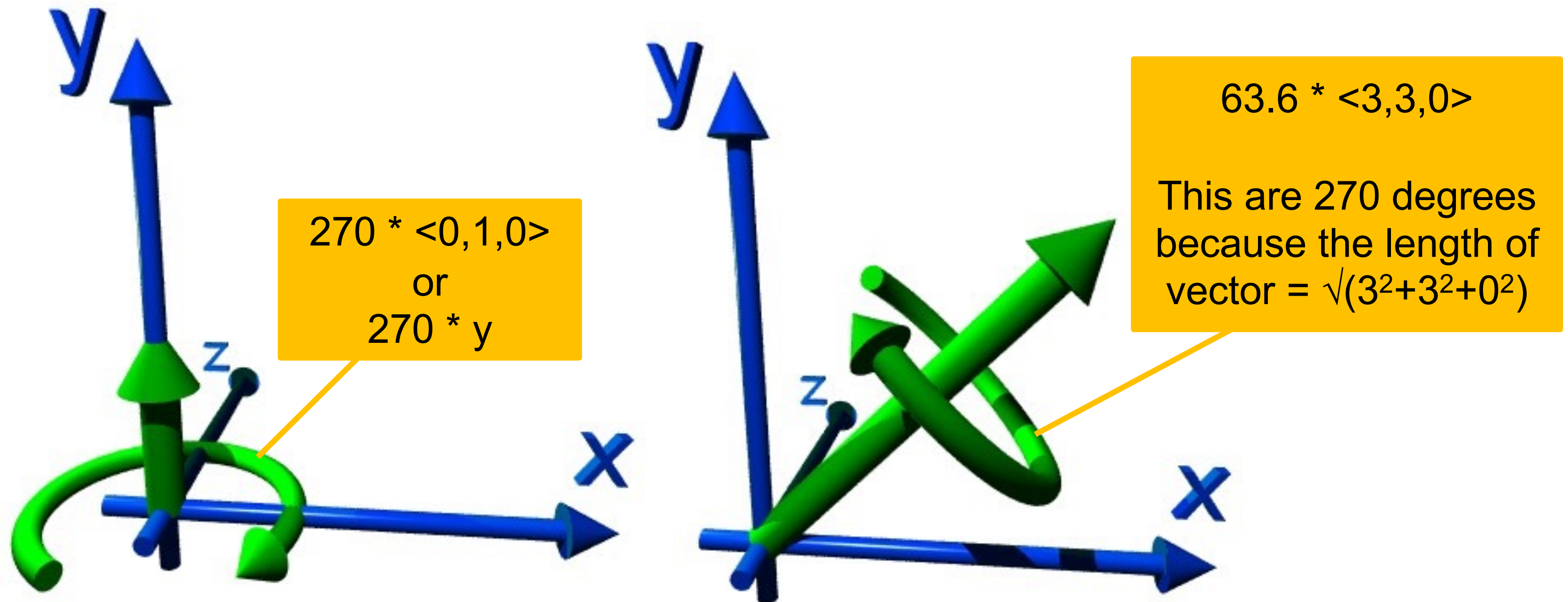
```
x + 3*y
<rand(rnd1) * sqrt(2), pi/2, log(3)>
```

- Trigonometric ***functions*** use (and deliver) ***radian***!

```
180 / 3.141 * atan(dy/dx)
```

Vector as Rotation Axis

- Vectors are also used for rotations:
  - The *direction* of the vector is the *rotation axis*
  - The *length* of the vector is the *angle* (in ***degrees***)
  - *left hand rotation sense* is used (for positive angle values)



Technicalities: Comments & Definitions

- POV-Ray is *caseSensiTIve*
- Comment *lines* start with //:

```
// comment
```

- Comment *blocks* are done as in C:

```
/* ... (can be multiple lines) ... */
```

- *Constants* or frequently used *expressions* are defined by:

```
#declare name = ... ;
```

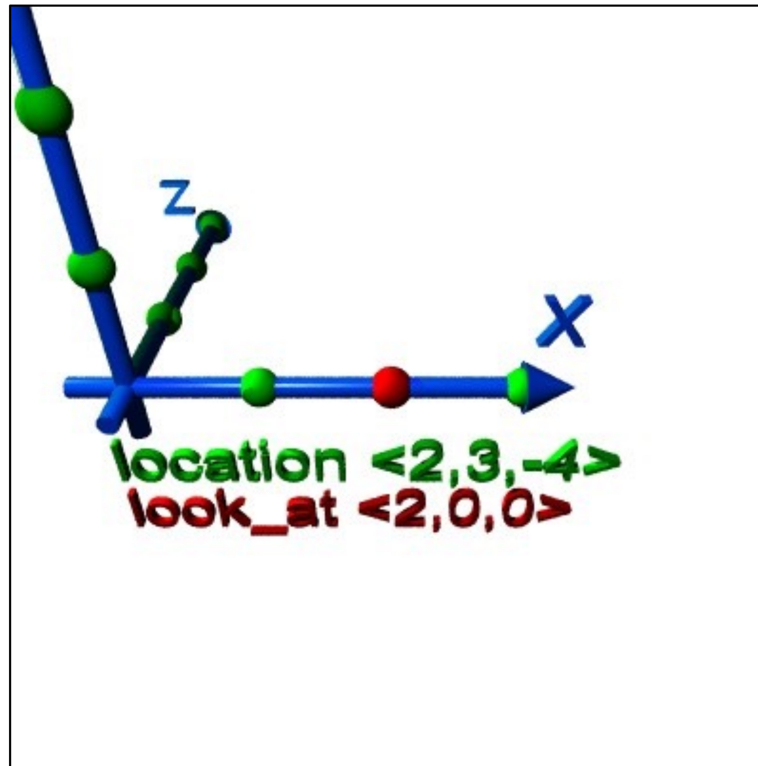
  - Note: #define does *not* work
  - A semicolon is required! (No Semicolon is required after `}`)

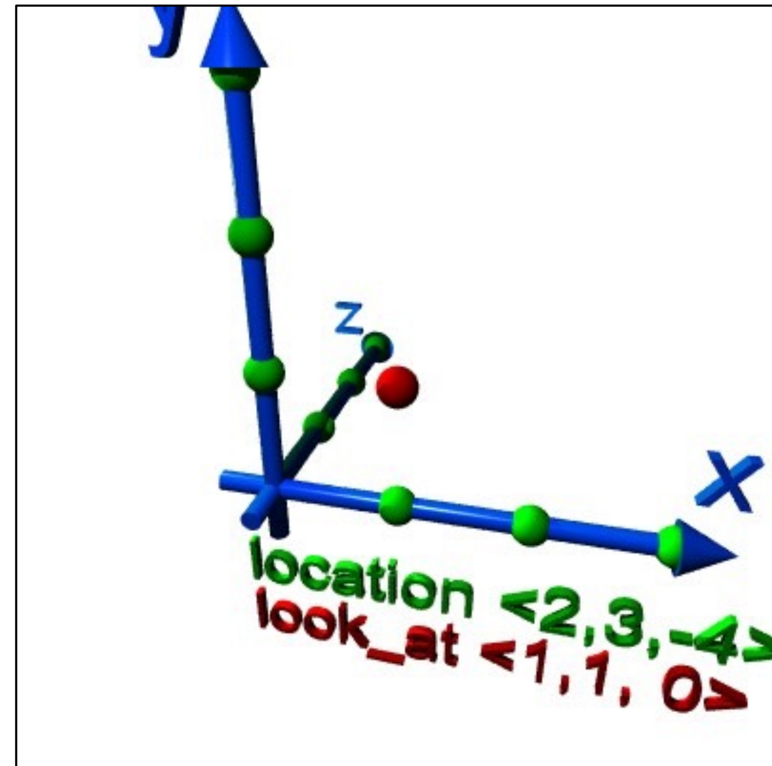
- *Parameters* can be passed by defining a 'macro':

```
#macro name (parameters,..)
   ...
#end
```

The Camera

- It has a position: `location <point>`
- The viewing direction can be set by: `look_at <point>`



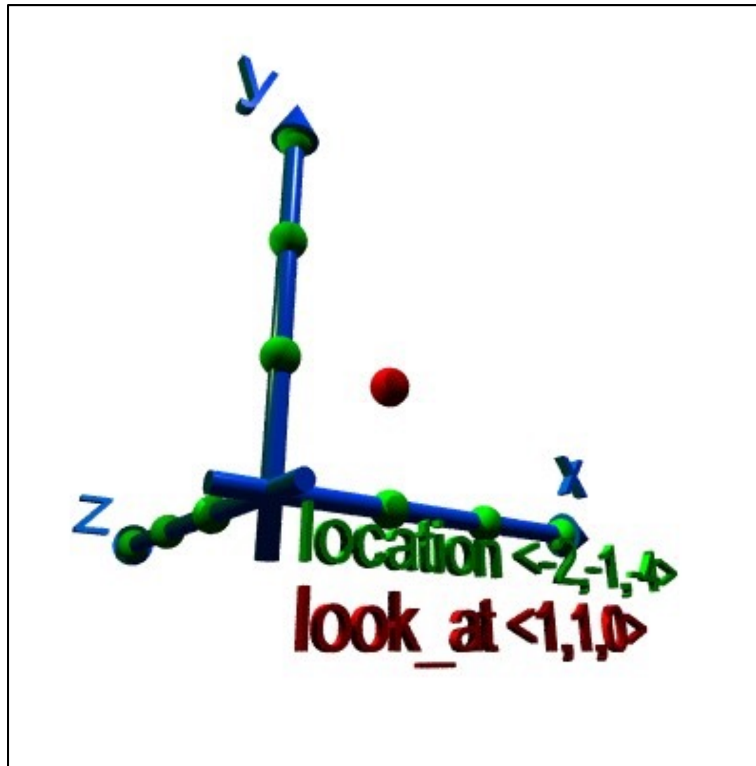


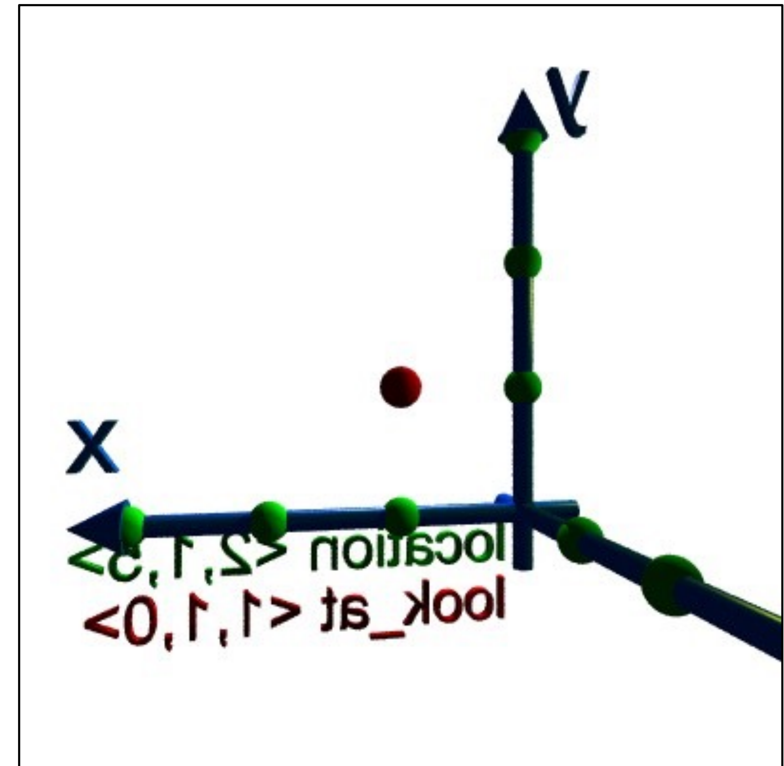
location <2,3,-4>
look_at <2,0,0> // red sphere

location <2,3,-4>
look at <1,1,0> // red sphere

More Camera Positions

- The two scenes look at the same (red) point <1,1,0> from different camera positions



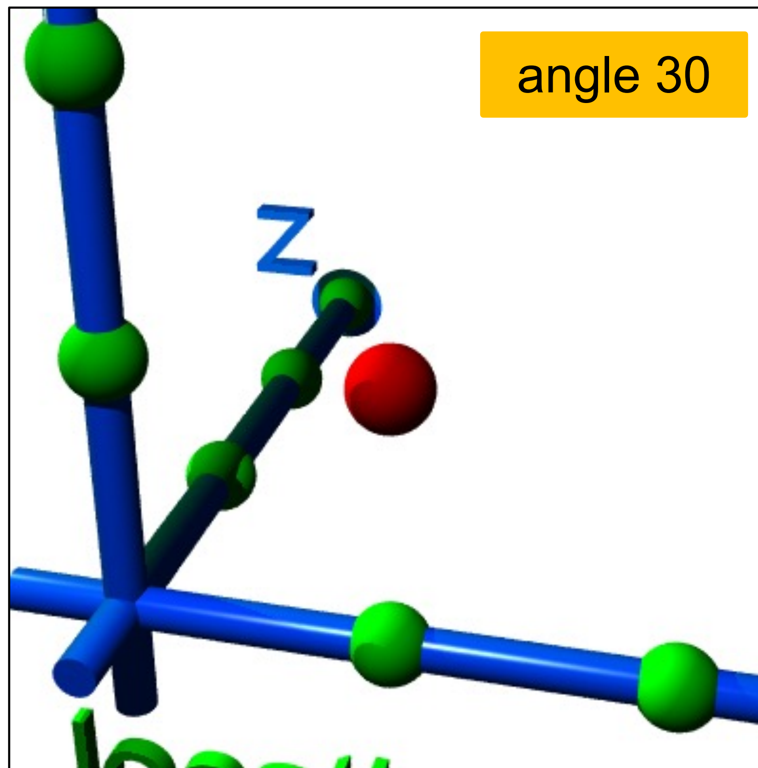


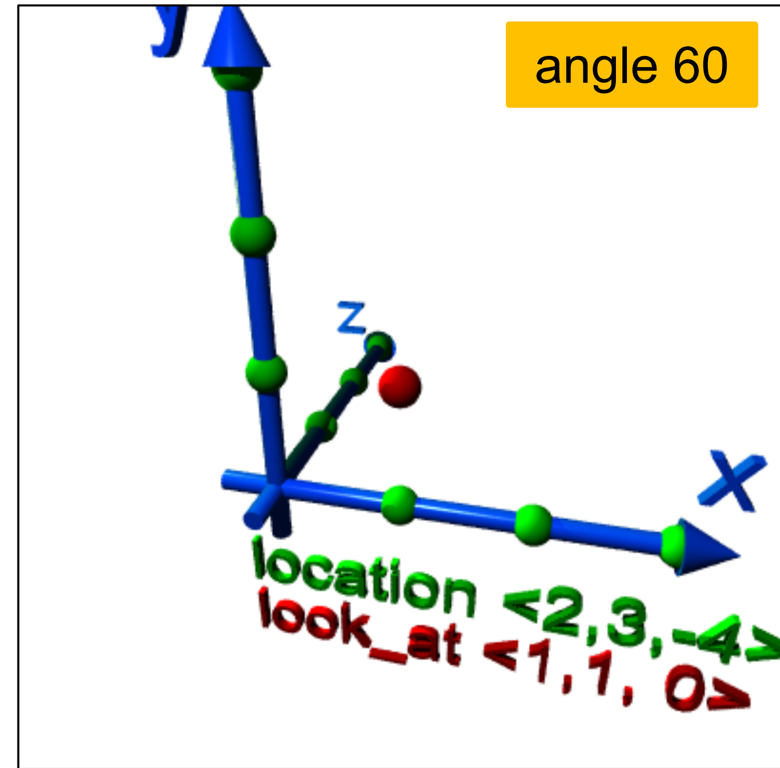
From below the x-z-ground plane, from left side.

From ‘behind’

Camera: Viewing Angle

- viewing angle `angle <value>` sets
  - small value: ‘Tele’ (low distortion)
  - large value: ‘Wide angle’
  - very large value: ‘Fish Eye’ (significant bending of straight lines)

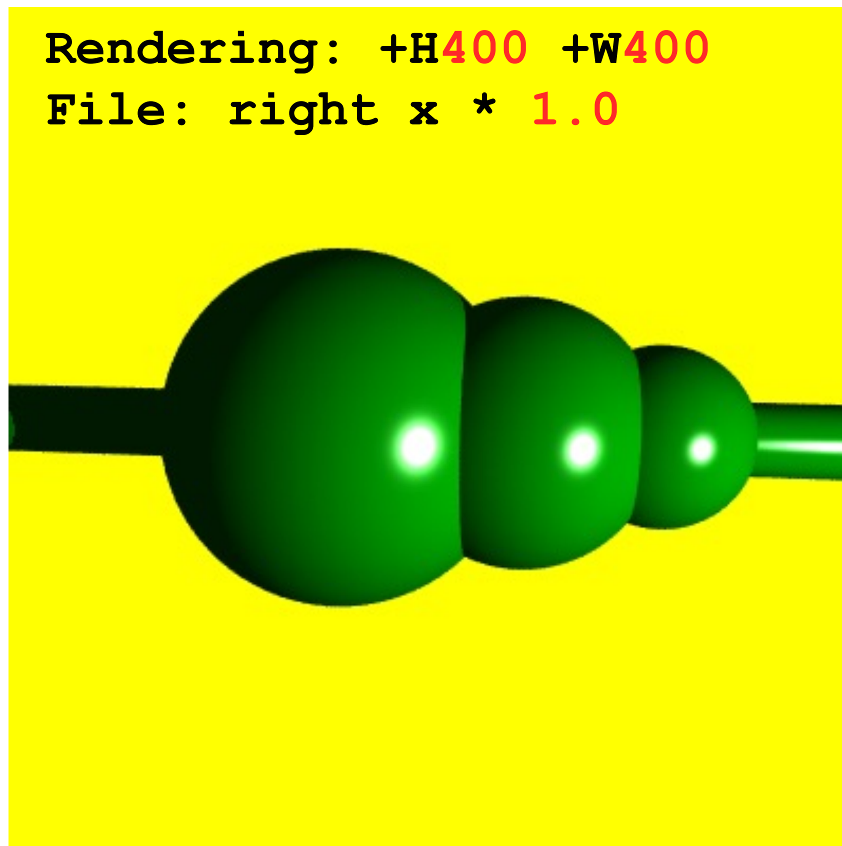


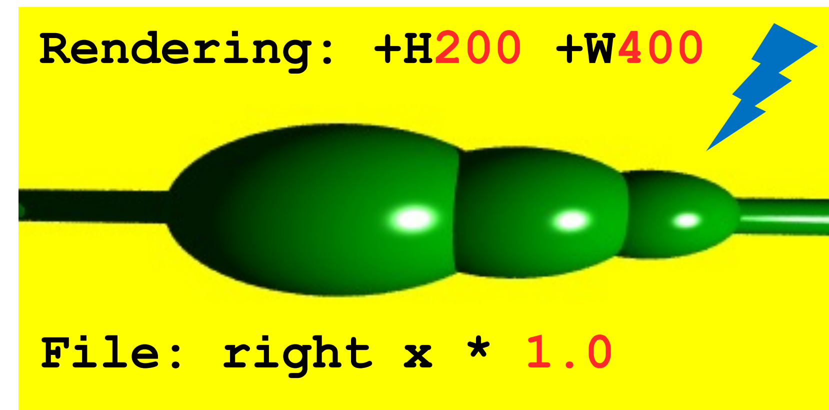


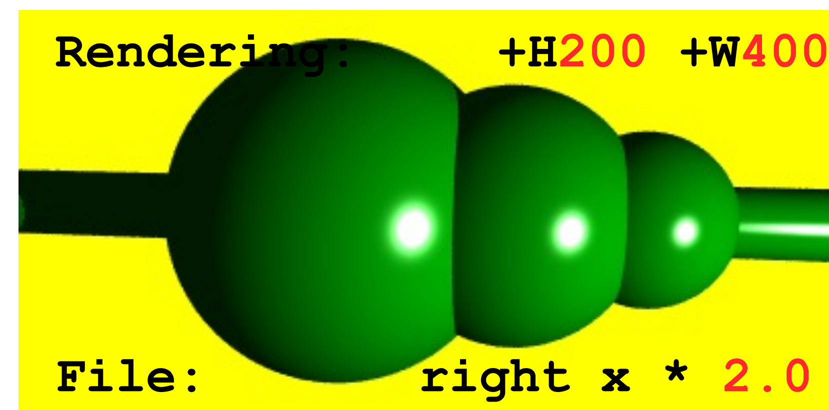
Camera: Aspect Ratio

- A nasty detail is that we must tell the renderer the aspect ratio (width / height) of the image to avoid distortion
- This ratio is best set to the rendering command values:

```
right x * image_width / image_height
```







RUPRECHT-KARLS-
UNIVERSITÄT
HEIDELBERG

Camera: All together

- A typical camera command is

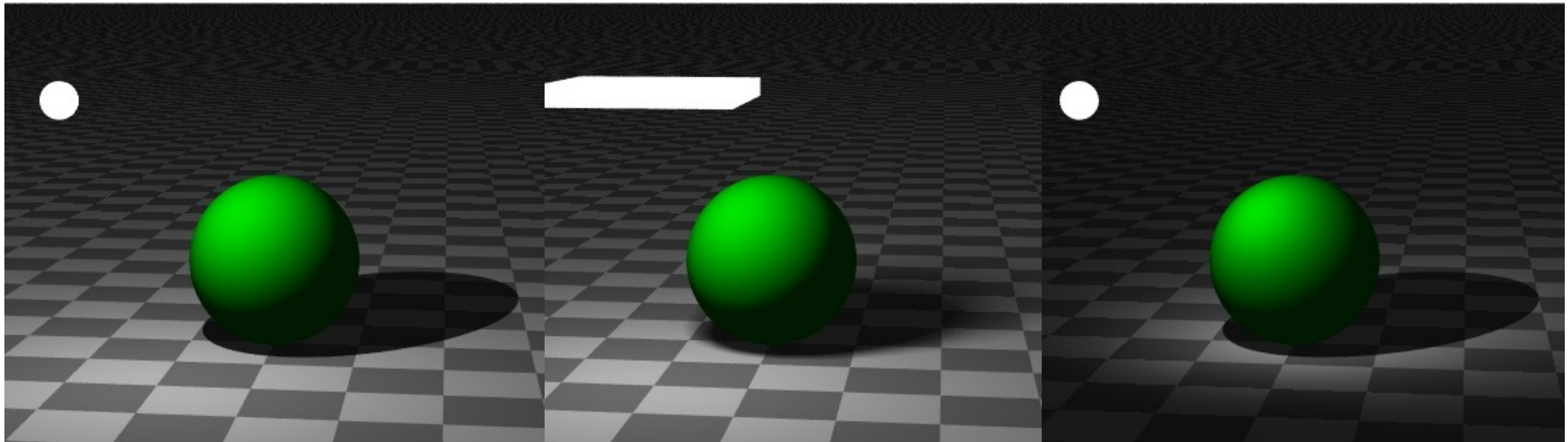
```
camera {
   location <1, 1, -6>
   look_at  <0, 1, 0>
   angle 50
   right x * image_width / image_height
}
```

- The default camera (if no camera is defined) is

```
camera {
  perspective
  location <0,0,0>
  direction <0,0,1>
  right 4.0 / 3.0 * x
  up y
  sky <0,1,0>
}
```

Light Sources

- They can be *point sources*, *area sources* or *spotlights, ...*



```
light_source {
  <position>
  color ...
}
```

```
light_source {
 <position>
  area_light
  // parameters..
}
```

```
light_source {
 <position>
  spotlight
  // parameters..
}
```

- Use **`background {color xxx}`** to color the 'sky'
- Use **`global_settings {ambient_light`** ***`color`*** **`}`** to get diffuse light on all surfaces. (More light with ...**`10*<1,0,0>`**)

Light Sources

- Light Sources can have color:
  - Image shows a **white** sphere. Look at sphere color & shadows!

Lights are normally invisible. I added the spheres to show where they are..

No green light here: Plane is red

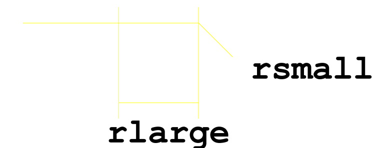
(Note)

- The order of commands in light source definitions matters, some combinations give syntax errors…

```
light_source {
  <-1,2,-2>
  color White
  spotlight              // ERROR if used before
  point_at <0,0,0>       // 'color' line!!!
  radius 50.0
  falloff 200.0
  tightness 5
}
```

Simple Objects

- Simple objects are:
  - `sphere { <location>, radius }`
  - `box { <corner>, <opposite_corner> }`
  - `cylinder { <p1>, <p2>, radius }`
  - `cone { <p1>, r1, <p2>, r2 }`
  - `plane { <normal>, dist_origin }`
  - `torus { rlarge, rsmall }        // in x-y plane`



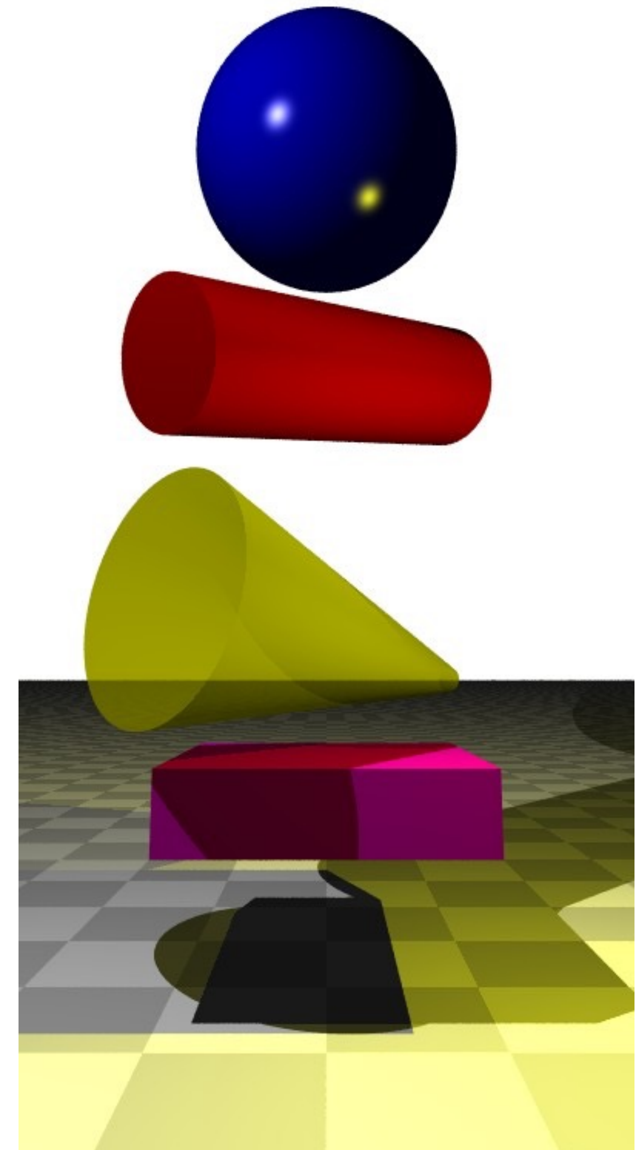
- They can be *colored* (surface & *volume*!) using
  - `pigment {color rgb <r, g, b>}`

    or just
  - `pigment {color <r, g, b>}`
- *Transparency* can be added by a 4th parameter
  - `pigment {color rgbt <r, g, b, t>}`
- Predefined colors from "`colors.inc`" are **R**ed, **W**hite,..

Example

```
#include "colors.inc"

sphere { < 0, 5, 0>, 1
  pigment {color Blue}
  finish { phong 0.9 phong_size 60 }
}
cylinder { <-1, 3, -1>, <1, 3.5, 1>, 0.5
  pigment {color <1,0,0>}
}
cone {
  <-1, 1.5, -1>, 0.8, <1, 1, 1>,  0.1
  pigment {color rgbt <1,1,0,0.5>}
}
box { <-1,0,-1>, <1, 0.5, 1>
  pigment {color Magenta}
}
plane { y,-1
  pigment {checker Gray80, White}
}

background   { color White }
light_source {<-2,5,-3>  color White }
light_source {<2,2,-1.5> color Yellow }

camera {
  location <0, 1, -6>
  angle 40 right x * 400 / 800
  look_at <0,2,0>
}
```

background = color where NO object is



Exercise 1

- Enter the minimal POV-Ray file

```
sphere {
 <0,0,3>, 1
 pigment { color <1,0,0> }
}
light_source { <1,1,1> color <1,1,1> }
```

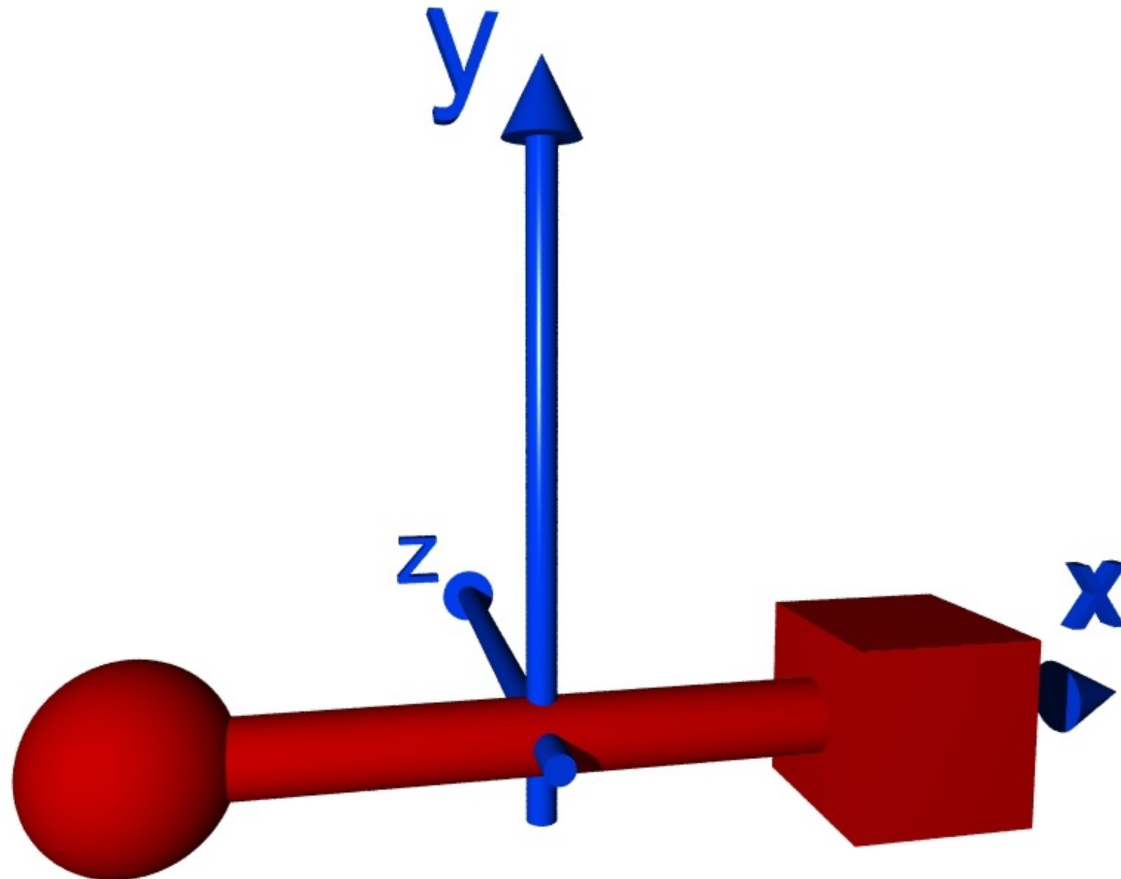
- Render it using

```
povray +IPOV_Mini +H300 +W400 +P
```

- Then
  - Use `#include "colors.inc“` and use White, Red,...
  - Change the sphere to a box
  - Change the `background`
  - Add a camera (otherwise it is at <0,0,0>) and move it around
  - … play around ...

Exercise 2

- Create the following scene (just the red stuff, no blue axes)
  - The centre of the sphere is at `<-2,0,0>`
  - put the camera at `<-1,2,-5>`

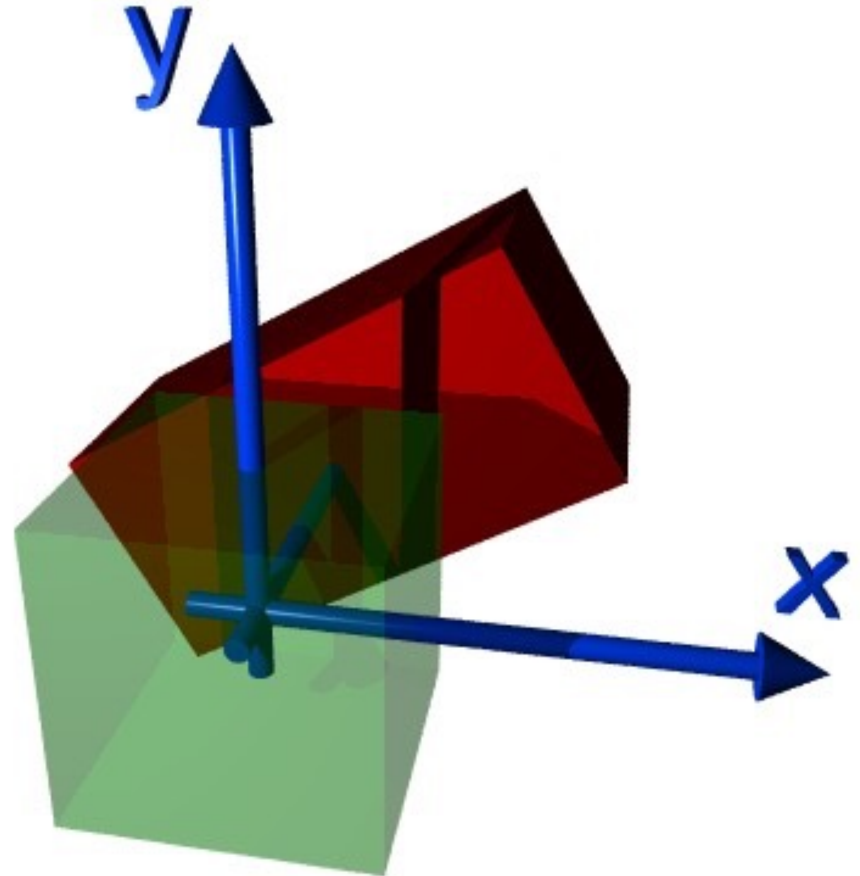


Moving and Rotating Things

- Objects can be moved, scaled & rotated:
  - **translate <shift_vector>**
  - **scale <scalex, scaley, scalez>** (or **scale val**)
  - **rotate <vector>** (vector defines direction & angle)

- Example:

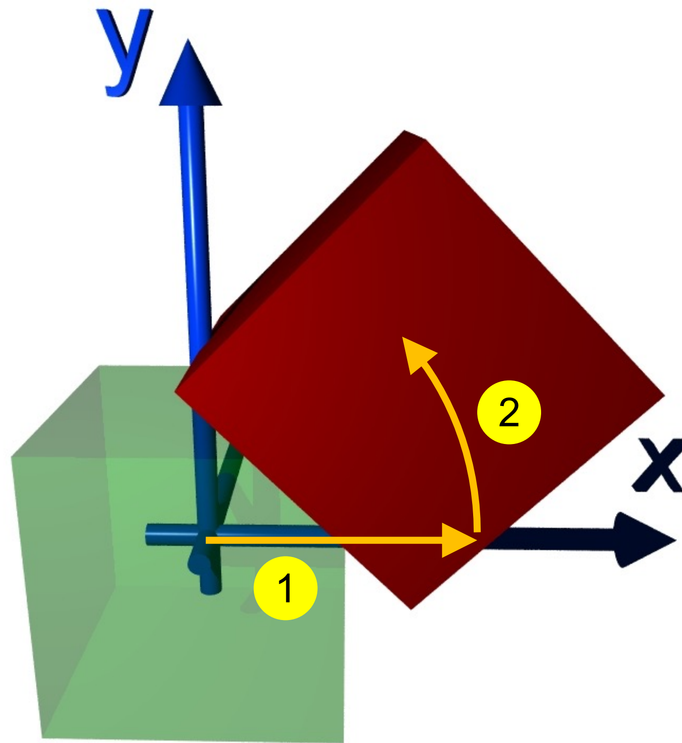
```
box {
  <-1,-1,-1>, <1,1,1>
  pigment {color Red}
  scale <2,1,1>
  rotate 30*z
  translate 3*z
}
```

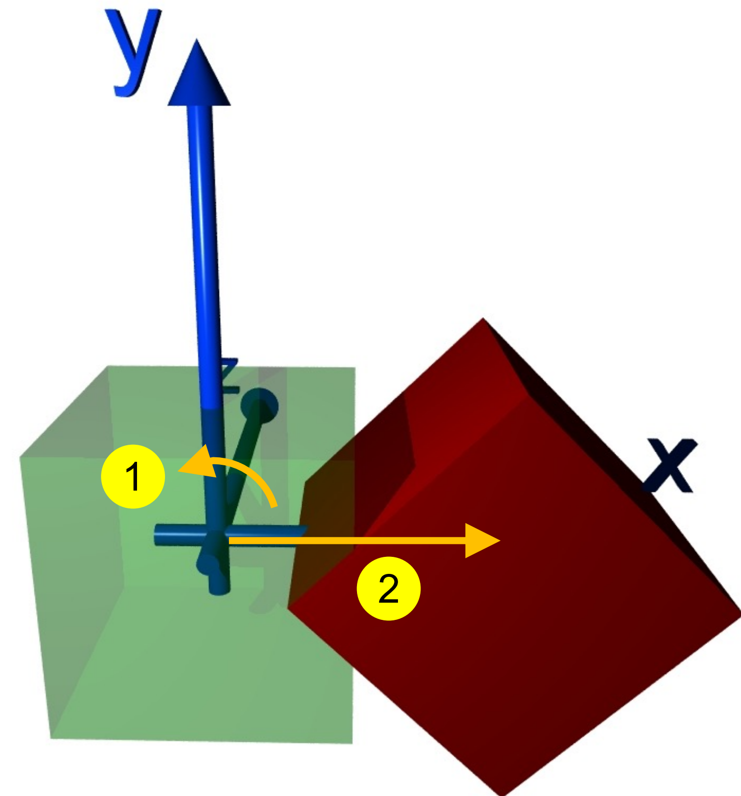


Rotations & Translations are *not* Commutative!

```
box {
   <-1,-1,-1>, <1,1,1>
   translate 2*x
   rotate    45*z
}
```

```
box {
   <-1,-1,-1>, <1,1,1>
   rotate    45*z
   translate 2*x
}
```





RUPRECHT-KARLS-
UNIVERSITÄT
HEIDELBERG

(Exercise 3 – not so easy…some have solved this!)

- The painting ‘Waterfall’ from M.C.Escher contains two geometric figures
- Draw the left one, which consists of 3 rotated cubes!

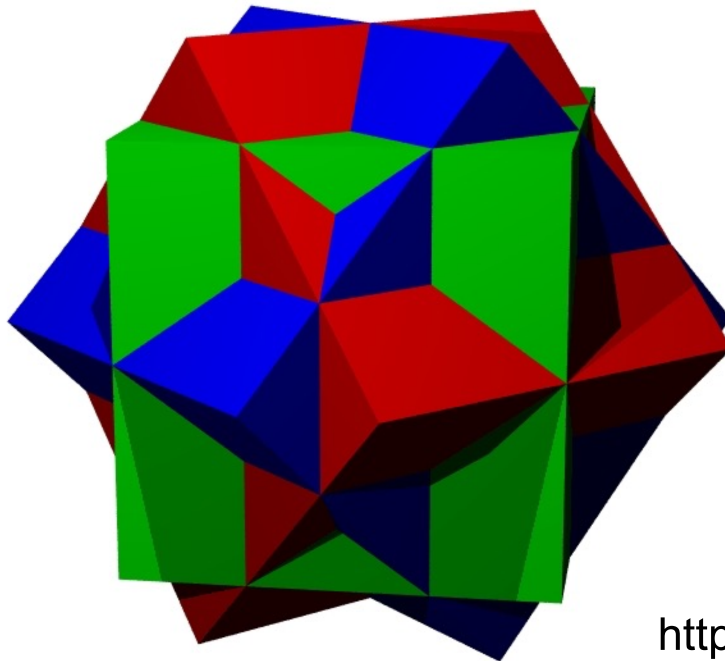
Picture removed. See web site of university:

Gestaltung von Webseiten

Die Universität Heidelberg sieht sich in letzter Zeit vermehrt hohen Schadensersatzforderungen ausgesetzt, die Urheber oder von diesen beauftragte Kanzleien aufgrund von Urheberrechts- und sonstigen Schutzrechtsverletzungen geltend machen. Dies betrifft insbesondere die unerlaubte Veröffentlichung von lizenzpflichtigen Fotos und Kartenausschnitten auf Webseiten der Universität. Die rechtlichen Möglichkeiten, gegen solche finanziellen Ansprüche vorzugehen, sind sehr beschränkt und die Forderungen damit im Regelfall zu erfüllen.

Das Hochladen eines urheberrechtlich geschützten Werkes, wie einer Fotografie oder eines Kartenausschnitts, auf einer Webseite stellt eine öffentliche Zugänglichmachung des Werkes dar, die grundsätzlich nur mit Genehmigung des Urhebers zulässig ist. Aus dem Umstand, dass das verwendete Werk möglicherweise bereits im Internet veröffentlicht wurde und ggfls. ohne weiteres kopiert werden konnte, kann nicht auf die Genehmigung des Urhebers zur weiteren Verwendung geschlossen werden.

Wird ein urheberrechtlich geschütztes Werk ohne Zustimmung des Urhebers auf einer Internetseite öffentlich zugänglich gemacht, liegt in der Regel eine Urheberrechtsverletzung vor. In diesem Fall hat der Urheber Ansprüche auf Beseitigung der Beeinträchtigung, Unterlassung sowie Schadensersatz. Die Schadensersatzforderungen belaufen sich immer auf mehrere hundert Euro. Diese Mittel müssen aus dem Aversum der Einrichtung aufgebracht werden, von der die Verletzungshandlung ausging. Zusätzlich werden für die Zukunft Unterlassungserklärungen gefordert, die für den Fall einer Wiederholung nochmals deutlich höhere Strafzahlungen vorsehen.

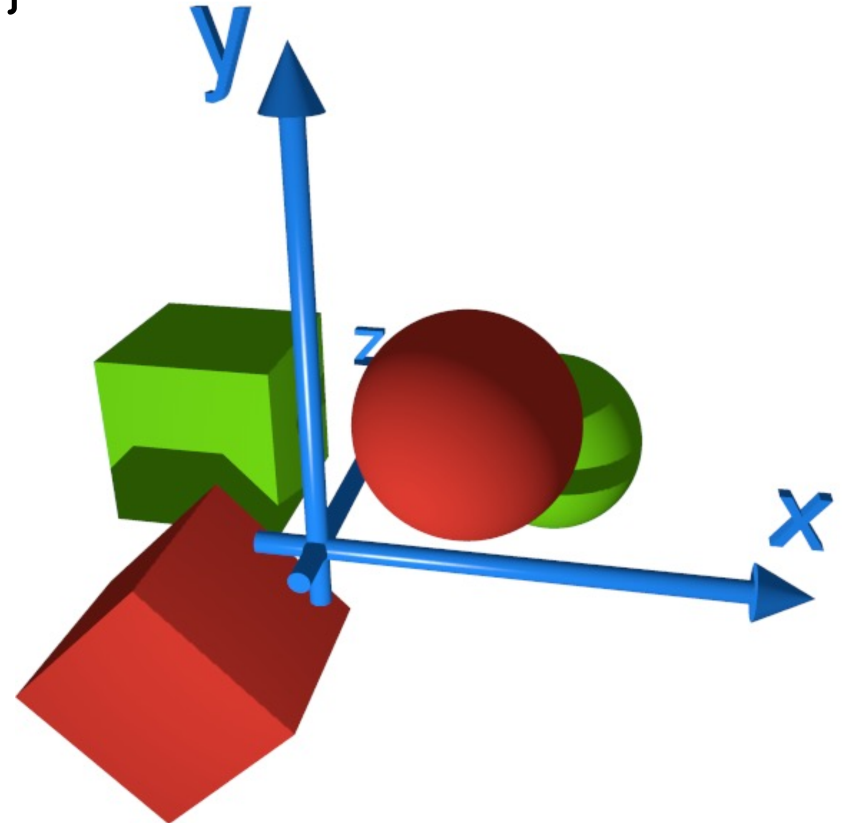


https://en.wikipedia.org/wiki/Waterfall_(M._C._Escher)

Instantiating and Merging Objects

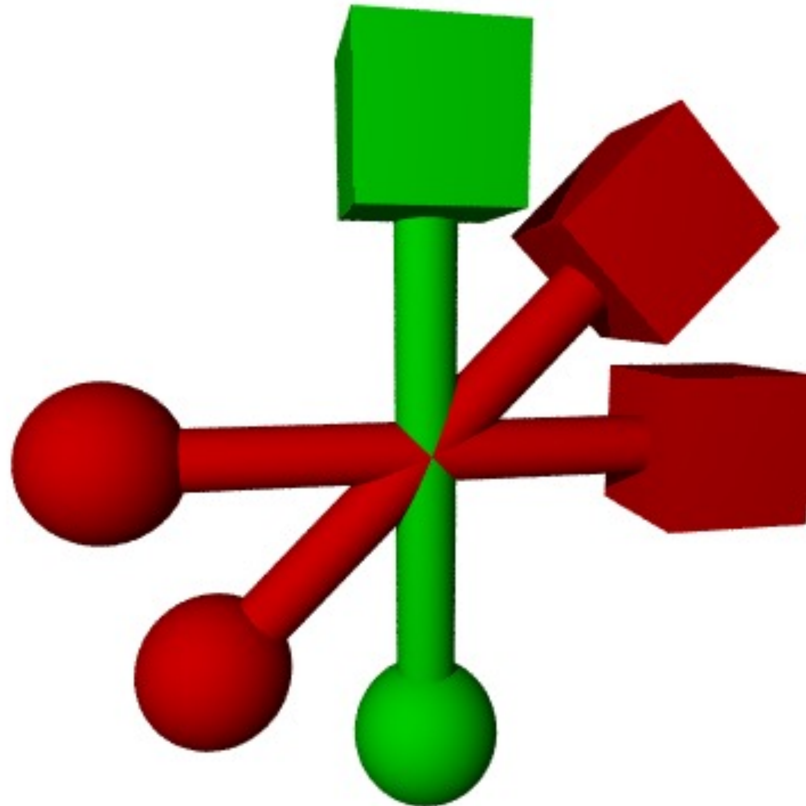
- Several Objects can be grouped with
  - `union { objects … transformations … pigment …}`
- Objects can be predefined by `#declare name = …;`
- A `#declare`d object can be instantiated with
  - `object{ name … pigment …}`

```
#declare H = 0.5;
#declare CUBE = box {<-H,-H,-H>,<H,H,H>}

#declare TWO = union {
  object {CUBE translate -x}
  sphere {x, 0.5}
  pigment {color Red}
  scale 1.5
}

object {TWO rotate 45 * z}
object {TWO translate 2*z
        pigment {color Green}
       }
...
```



Exercise 4

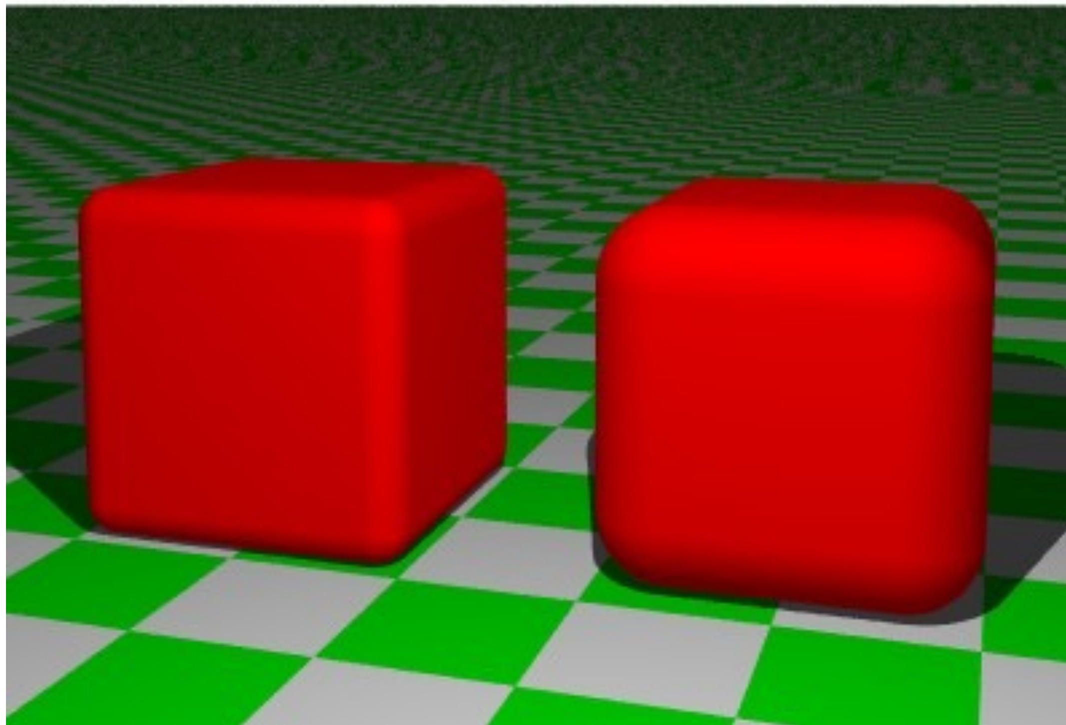
- Create the following scene using the object from Exercise 2:

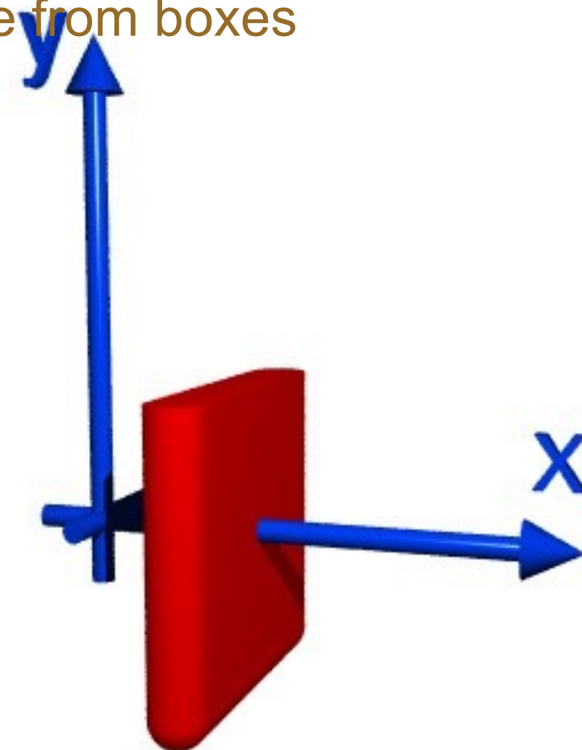


- Define first a `union` containing sphere, cylinder and cube
- Instantiate three of these with appropriate shifts / rotates
- Change the pigment of one part to Green

(Exercise 6)

- Make a rounded cube by adding spheres to the corners and cylinders to the edges
  - You *may* keep (*inner) cube size* and *corner radius* variable using
    `#macro CUBE (D,R) ..code using D and R.. #end`
  - Hint: You can assemble the cube from 4 face-objects (as shown on the right) and two 'covers' made from boxes





COMPLICATED OBJECTS

RUPRECHT-KARLS-
UNIVERSITÄT
HEIDELBERG

Making Complex Shapes, Cutting Things

- *Constructive Solid Geometry* (CSG) allows to construct complex shapes from simple shapes.

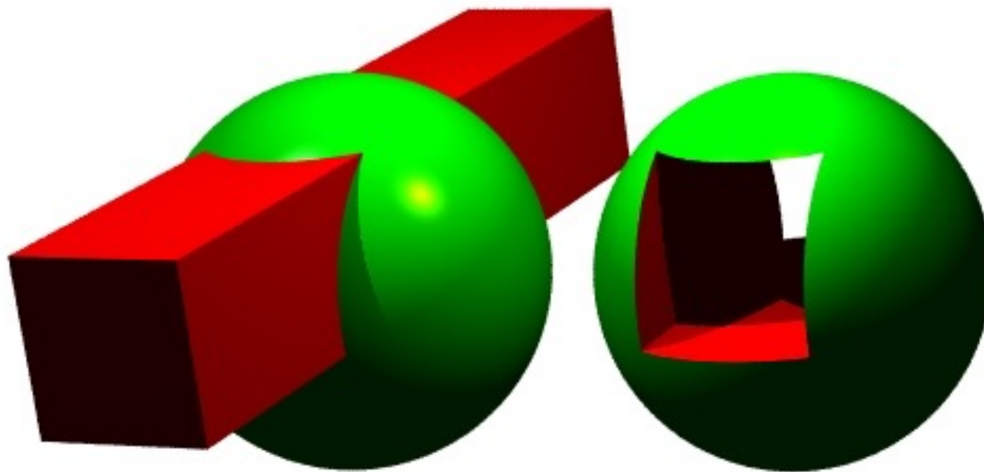
- Commands to *cut* objects are:
  - Volume covered by *both* objects (A **and** B):
    ```
    intersection { objA objsB }   //several Bs ok
    ```
  - Volume covered by A and *not* B (A **and !**B):
    ```
    difference   { objA objsB }   //several Bs ok
    ```

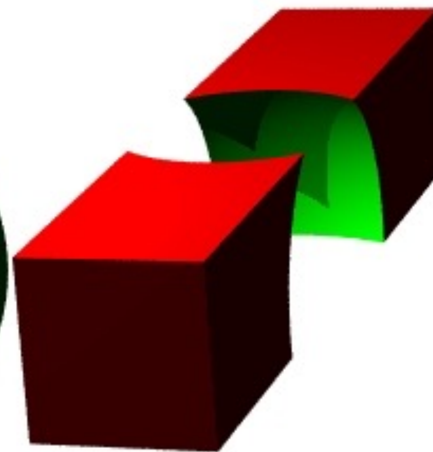
- We also have
  - Volume covered by (A or B), inner faces stay (just group objects, e.g. to translate them together)
    ```
    union        { obj1 obj2 }
    ```
  - Volume covered by (A or B), inner faces are removed (useful for transparent objects)
    ```
    merge        { obj1 obj2 }
    ```

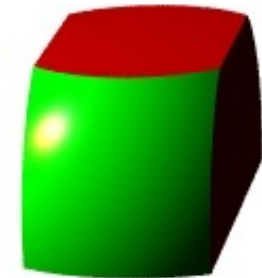
Examples for CSG

S:

B:







```
union {
  object {S}
  object {B}
}
```

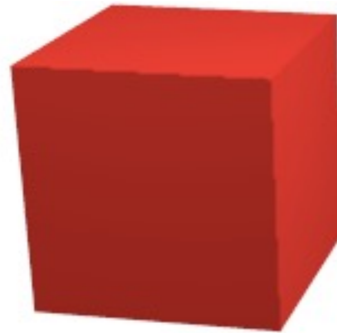
```
difference {
  object {S}
  object {B}
}
```

```
difference {
  object {B}
  object {S}
}
```

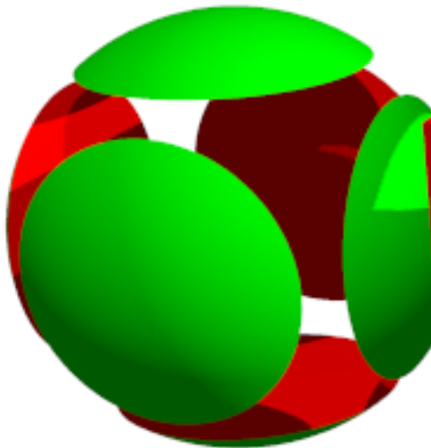
```
intersection {
  object {B}
  object {S}
}
```

More Examples for CSG

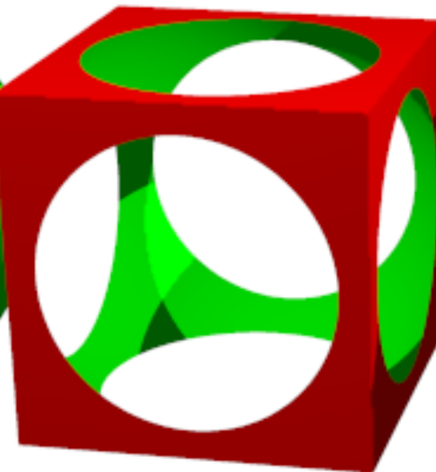
B:



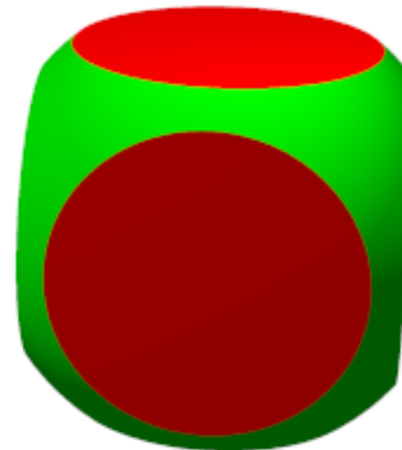
S:



```
difference {
  object {S}
  object {B}
}
```

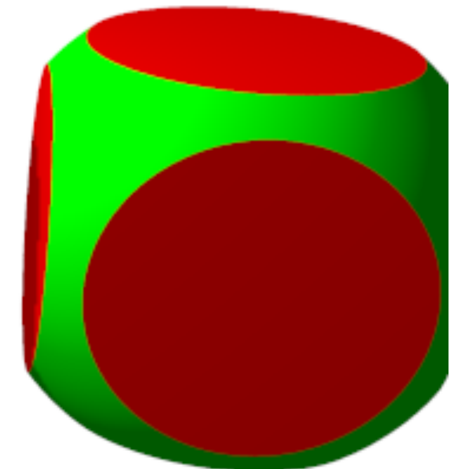


```
difference {
  object {B}
  object {S}
}
```



```
difference {
  object {B}
  object {S inverse}
}
```

'**inverse**' exchanges inside and outside

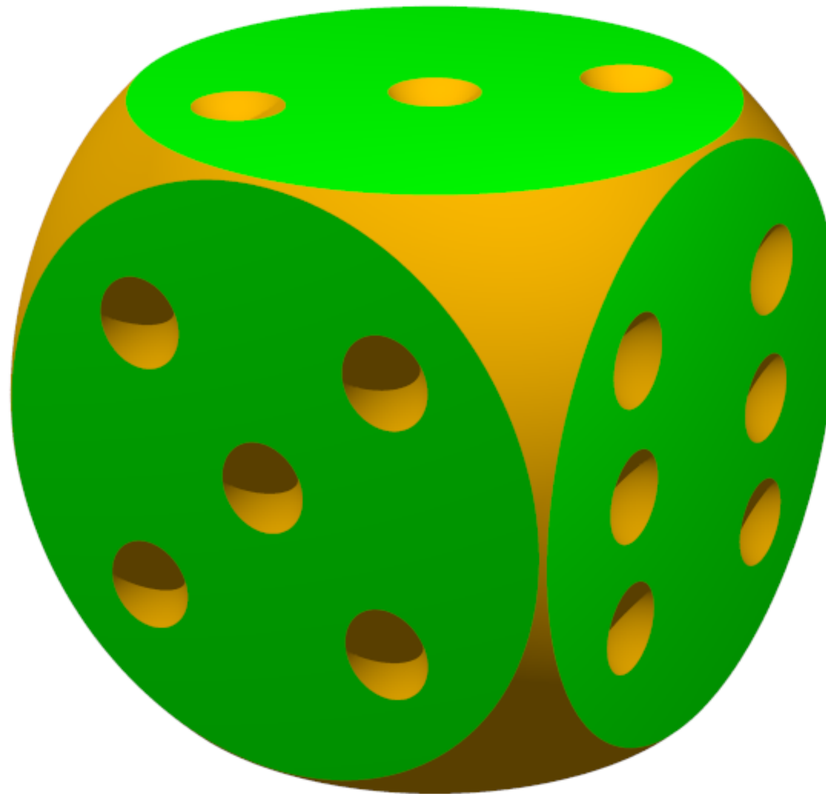


```
intersection {
  object {B}
  object {S}
}
```

RUPRECHT-KARLS-
UNIVERSITÄT
HEIDELBERG

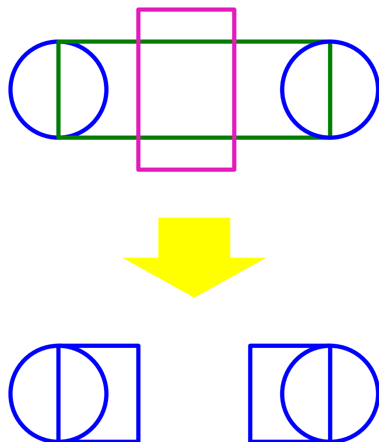
Exercise 7: Dice

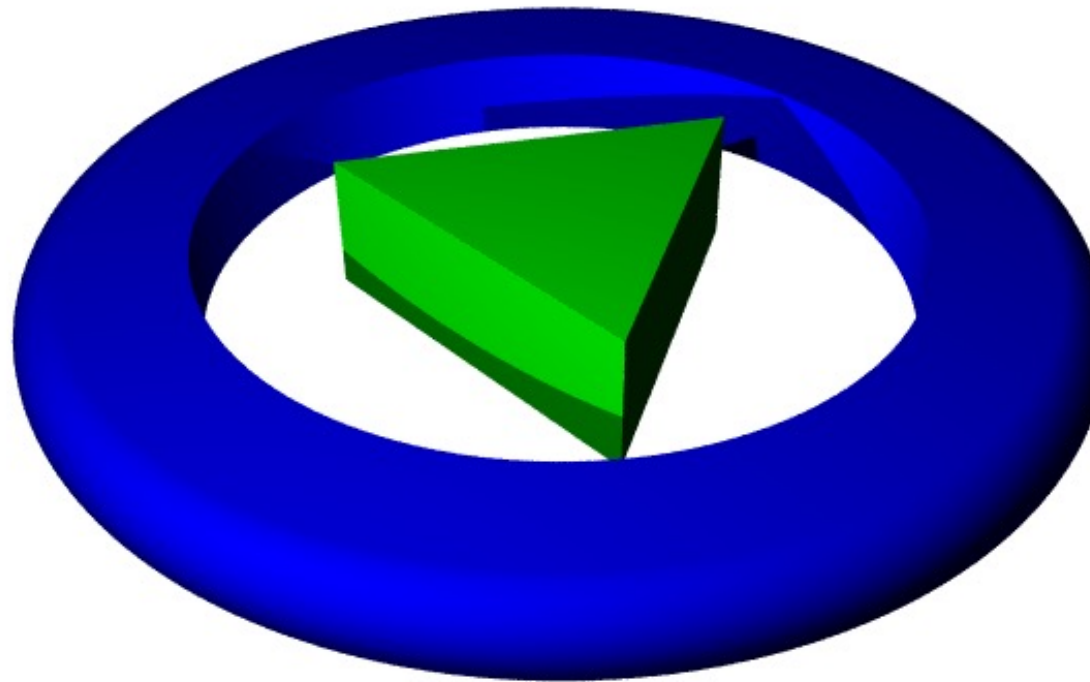
- Make a (Game) Dice
  - This is the difference of a cube with the inverse of a large sphere and several small spheres



Exercise 8

- Create the following scene





- The blue ring is a flat cylinder merged with a torus, cut by a (higher) central cylinder
- The green triangle is the intersection of 5 planes

  (One of the 3 vertical sides is `plane{x,0.5 rotate 120*y}`)

RUPRECHT-KARLS-
UNIVERSITÄT
HEIDELBERG

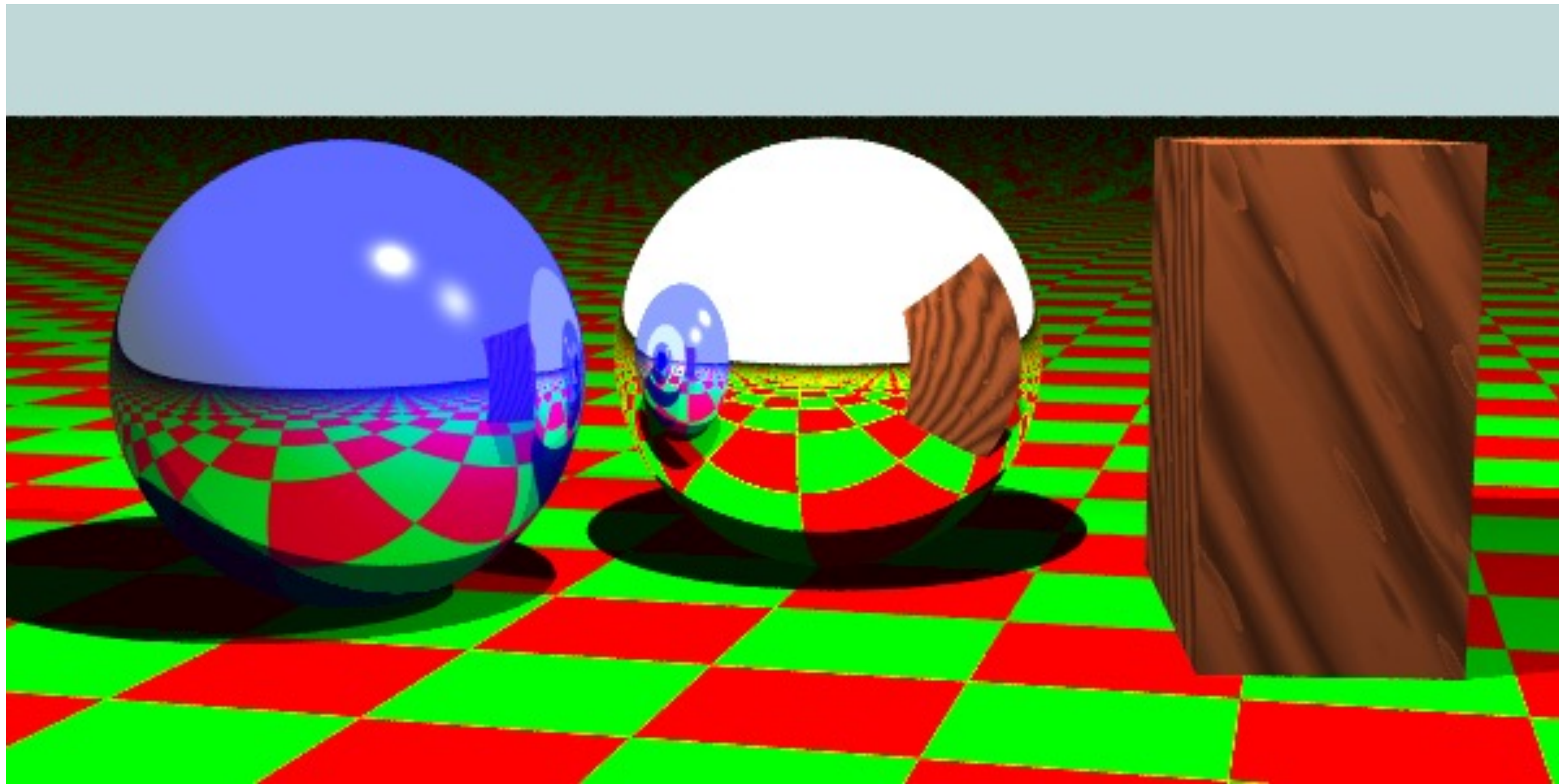
Textures

- Textures define the surface & *volume* appearance
- They consist of, for instance

```
texture {
  pigment {color ...} // color, transparency
  finish {            // surface properties
    ambient 0...1     // emitted light
    diffuse 0...1     // % of light reflected in a diffuse way
    reflection 0...1  // % of light reflected in a specular way
    phong  0...1      // intensity of highlights
    phong_size 60     // size of highlights
    ...               // many more (see documentation!)
  }
  normal {bumps 2 scale 0.5} // surface roughness
}
```

RUPRECHT-KARLS-
UNIVERSITÄT
HEIDELBERG

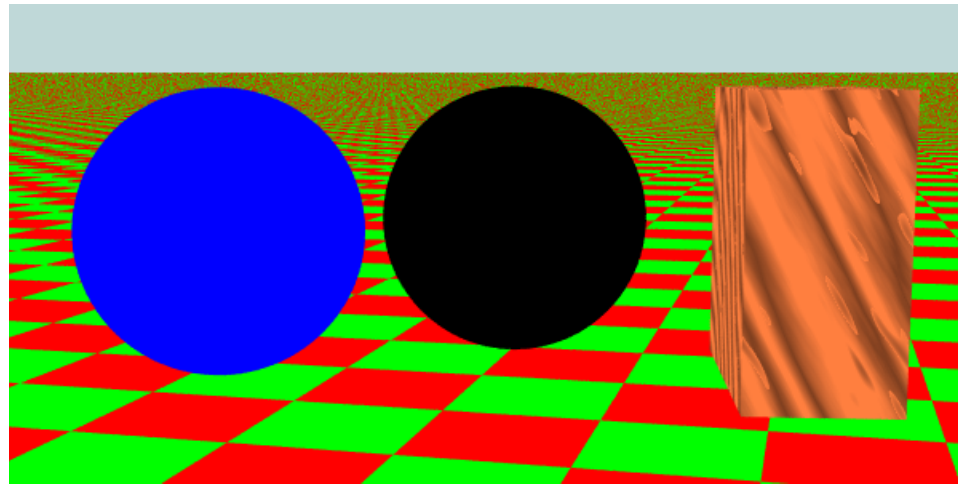
Example for Surface Properties



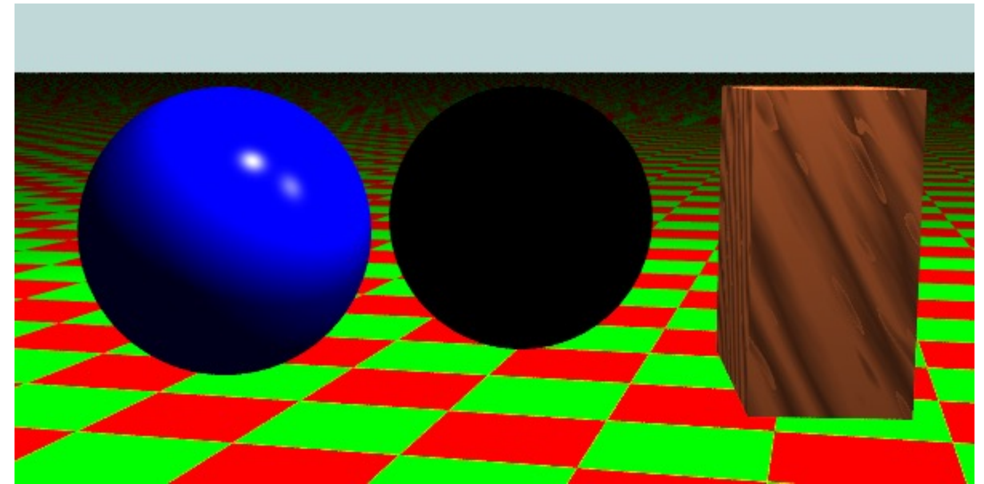
```
#include "colors.inc" #include "woods.inc"
plane  {y, 0 pigment { checker Green, Red rotate 10*y} } // note that checker is rotated!
sphere { y, 1 finish {reflection 1.5} }                  // fully reflective sphere
sphere { y-2*x-z, 1 pigment {color Blue} finish {reflection 0.5 phong 0.5 phong_size 80} }
box { <1.5,0,-2>, <2.5,2,0> texture{ T_Wood20 scale 5 rotate 30*z+80*x} }
background   { color LightBlue }
light_source {<0, 5, -3> color 2*White } light_source {<4, 5, -3> color White }
camera { location <0.5, 2.1, -10> look_at <-0.2,0.8,0> angle 40 right 2*x }
```

(Rendering Quality)

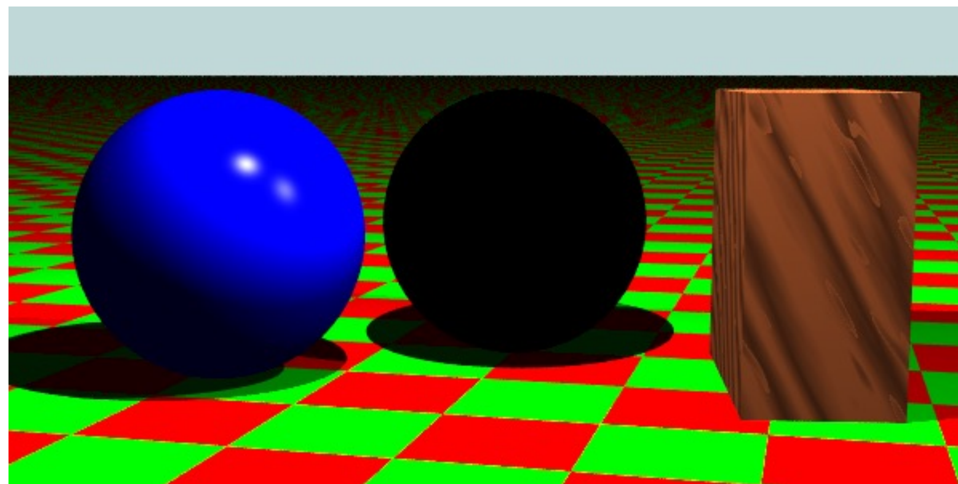
- The previous scene rendered with different +Q levels:



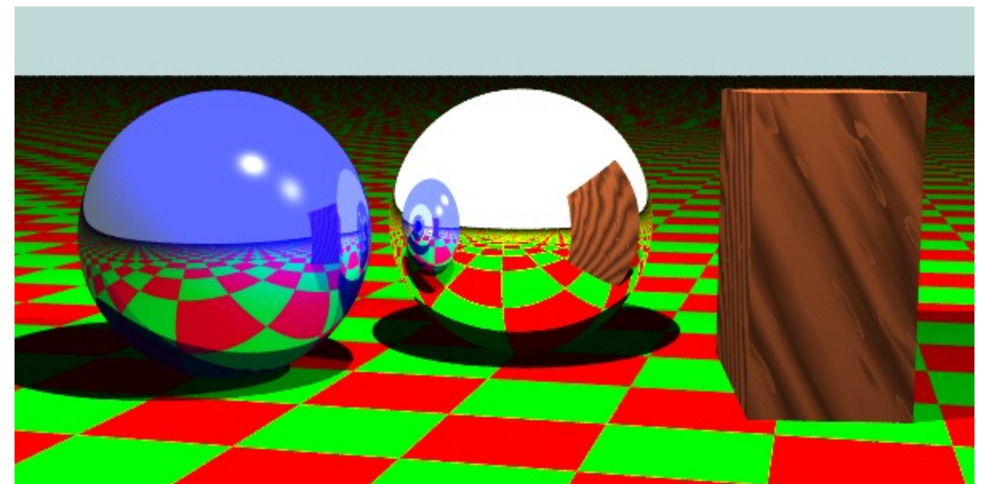
+Q1 only shapes



+Q3 surfaces, no shadows



+Q7 shadows, no reflections



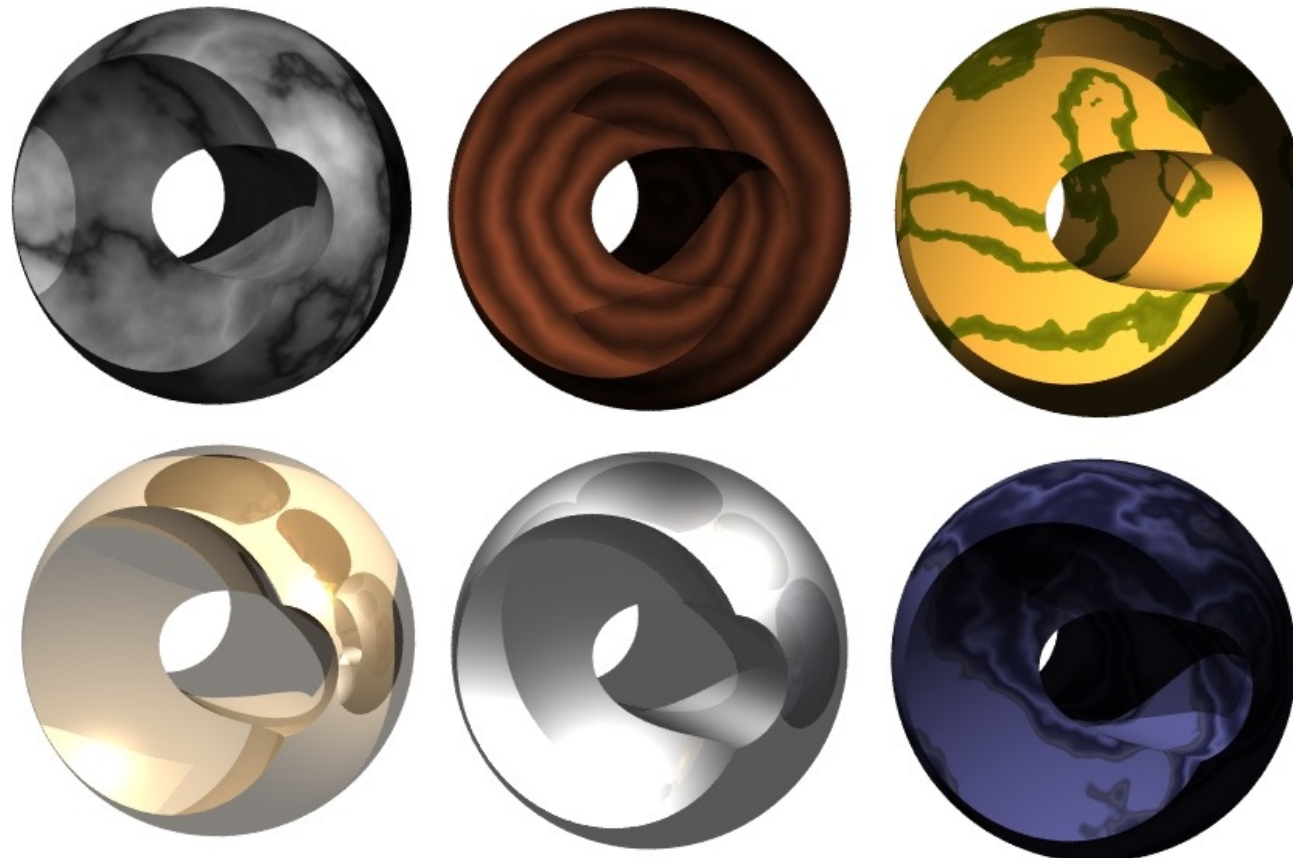
+Q8 reflections

Predefined Texture

- Many textures are available in the include files

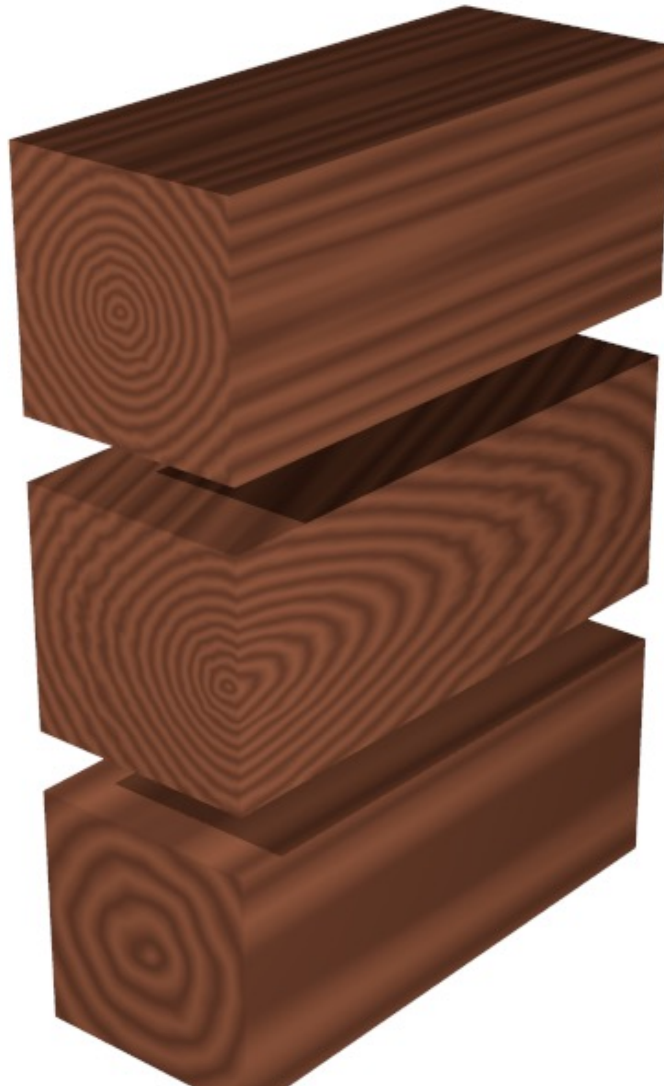
```
#include "stones.inc"
#include "woods.inc"   etc.
```

- For an overview, see for instance http://texlib.povray.org



(Rotating and Scaling Textures)

- Textures can be rotated and scaled ***inside*** the object:

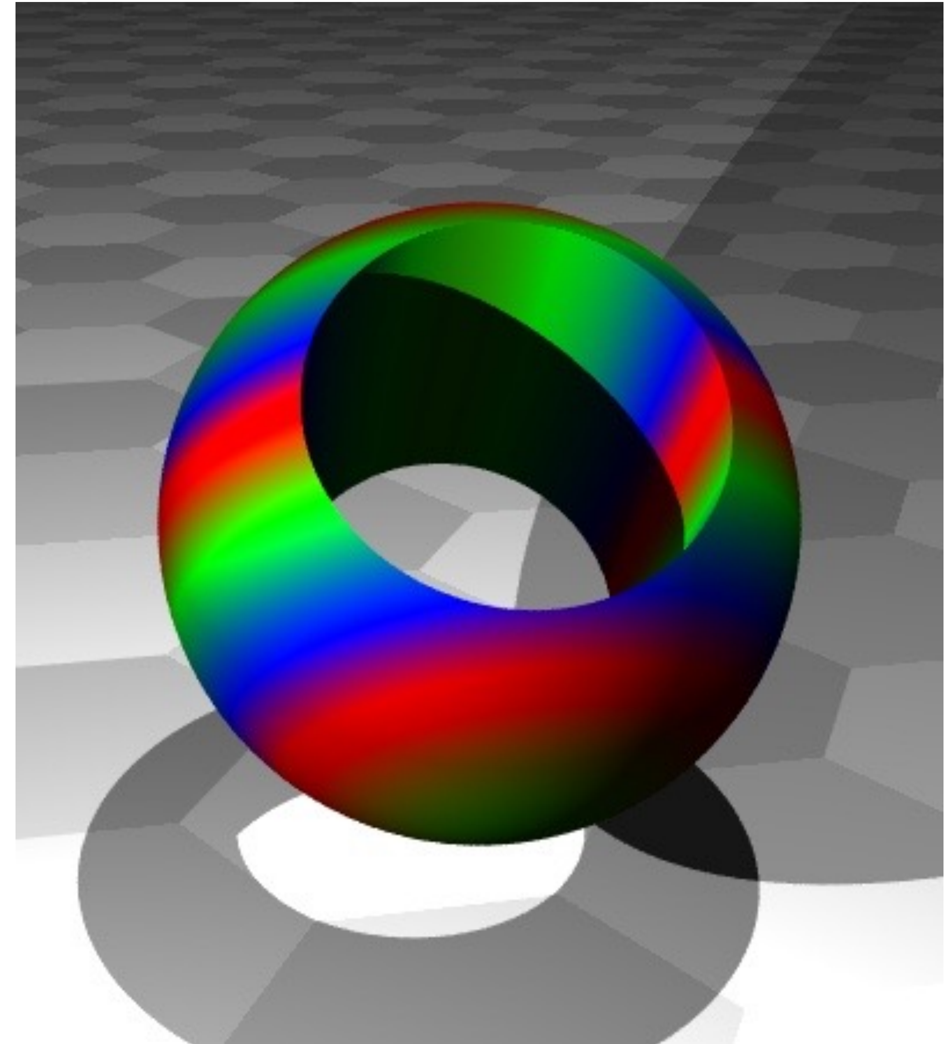


```
#declare C = box {
  <-0.4,-0.4,-1>,<0.4,0.4,1>
}
#declare T = DMFWood3;

object { C
  texture {T}
  translate y
}

object { C
  texture {T rotate -20*y}
}

object { C
  texture {T scale 3}
  translate -y
}
```

(Self Made Volume Textures)

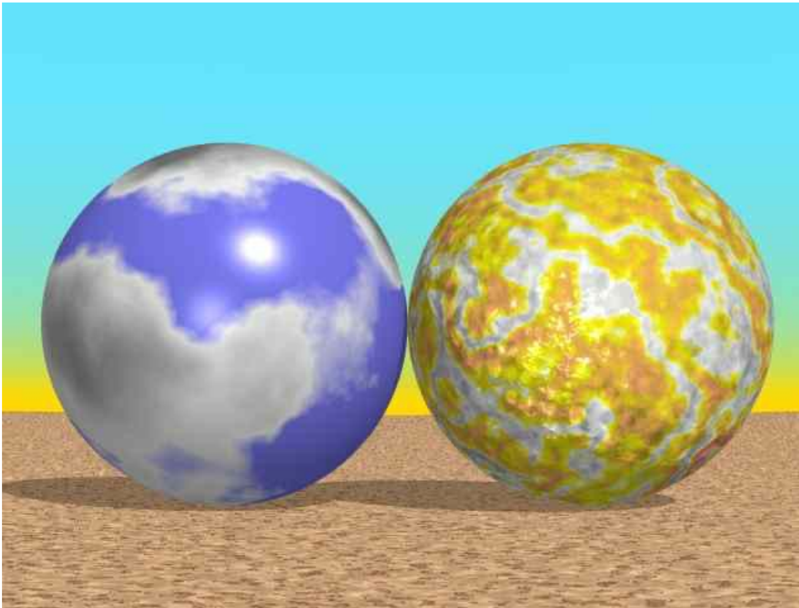
- There are many options to define (volume) textures.
- Just one example: Layered colours

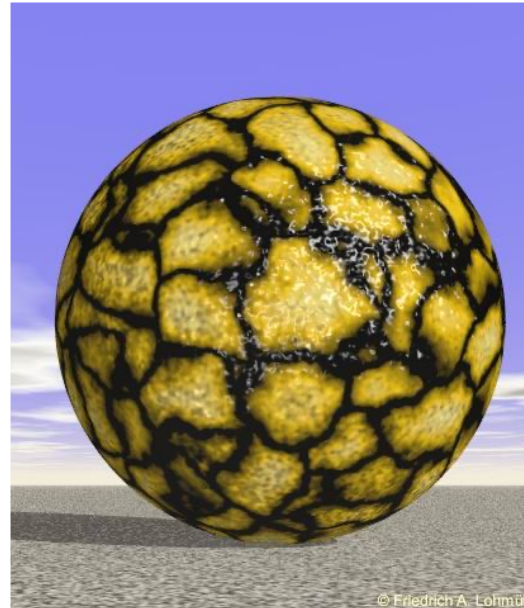
```
#include "colors.inc"
plane {y,-2 pigment{hexagon Gray80 White Gray90}}
difference {
  sphere { <0,0,0>, 1 }
  cylinder {-2 * y, 2*y, 0.6 rotate -45*x}
  pigment {
    gradient <0,1,1>  // orientation
    color_map {       // colors in range 0...1
      [0.1 color Red]   [0.3  color Blue]
      [0.7 color Green] [1.0  color Red]
    }
    scale 0.8
  }
}
background   { color White }
light_source {<0, 5, -3> color White }
light_source {<-2, 0.5, -2> color 1.5*White }
camera {
  location <-1.0, 2.0, -5> look_at <0,0,0>
  angle 30 right x*image_width/image_height
}
```

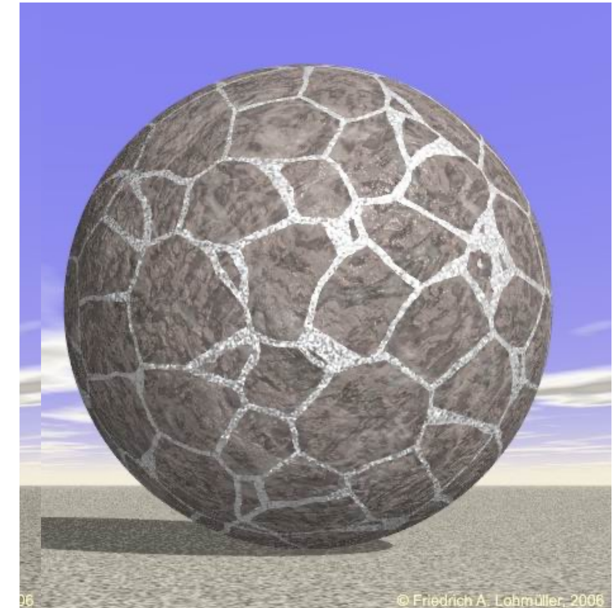


(More Stuff: Bump Maps)

- The surface normal (used to calculate how light is reflected) can be disturbed with various pattern. This gives spectacular ‘bumpy’ surfaces.
  - Many examples at www.f-lohmueller.de/pov_tut/tex/tex_000d.htm







Programming

Just two examples:

- Conditional blocks:

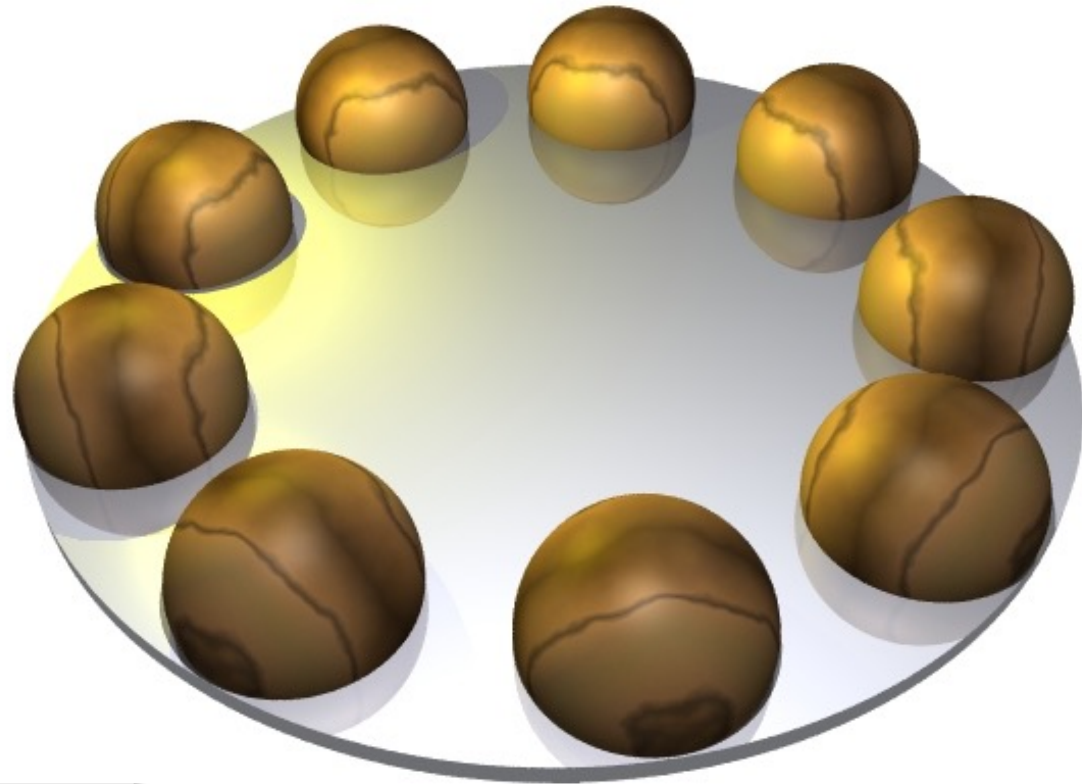
```
#if (VERSION=1)   // NOTE: only one '='
   ...
#end
```

- Loops

```
#declare angle = 0;
#while(angle < 360)
   ...
   #declare angle = angle + 60;
#end
```

Programming Example

```
#include "textures.inc"

cylinder {-0.2*y, -0.1*y, 2.5
  texture {Silver_Metal}
}
#declare S = sphere {2*x, 0.5
  texture {EMBWood1}
}

#declare phi = 0;
#while(phi < 360)
  object { S rotate phi * y}
  #declare phi = phi + 40;
#end

background   { color White }
light_source {<0, 5, -3> color White }
light_source {<-2,2,0.5> color Yellow }

camera {
  location <0, 4, -5>
  look_at  <0,-0.5,0>
  angle 50 right 6/4*x
}
```



Animations

- Several renderings can be done in a batch job
- The value of `clock` is incremented in each frame from a start to an end value in predefined steps
- Rendering creates a series of images
- The sequence of images can be merged to a movie

- See Demo, or for instance
  - http://www.alzinger.de/cms5/robert/raytracing/raytracing-video.html
  - http://www.alzinger.de/cms5/robert/raytracing/marble-machine-in-povray.html

Common Problems

- No light
  - Put a light source at the camera position (no nice illumination, but light everywhere)
  - use `global_settings {ambient_light color}`
  - add `finish {ambient…) to shapes`

- Bad camera
  - Make sure you are not *inside* a shape
  - Move camera far away to start with
  - Use large `angle`
  - Use `look_at`

Other Tools

- OpenScenegraph
  - Free tool
  - Graphic editor
  - Animation and Collision detection

- Blender
  - Graphics editor, very powerful, animation, games, characters

RUPRECHT-KARLS-
UNIVERSITÄT
HEIDELBERG

An alternative to POV-Ray (?): VRML

- VRML = Virtual Reality Modeling Language

- Is a 3D Scene description language similar to POV-Ray
- Can be rendered *in real time*
  - Viewer for instance from http://www.instantreality.org
  - Cortona3D browser plugin is not free any more
- User interaction is possible

- Much less powerful graphics, but interactive

- VRML has not really become a standard.
  Plugin installation not standard
- Seems to die out… (replaced by X3D and others)