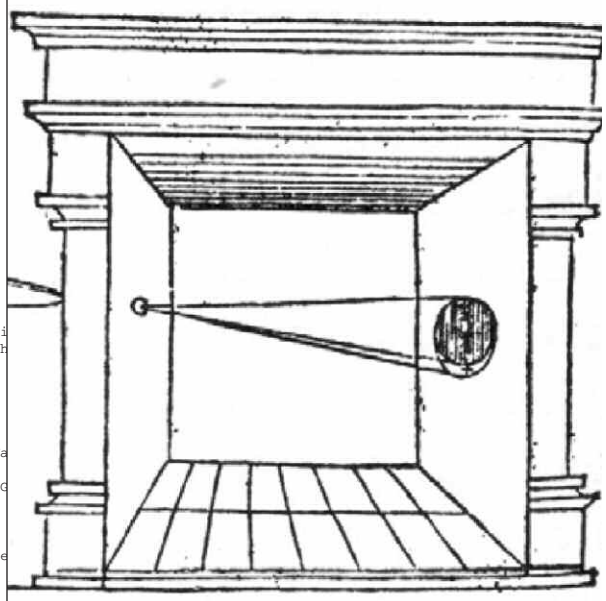
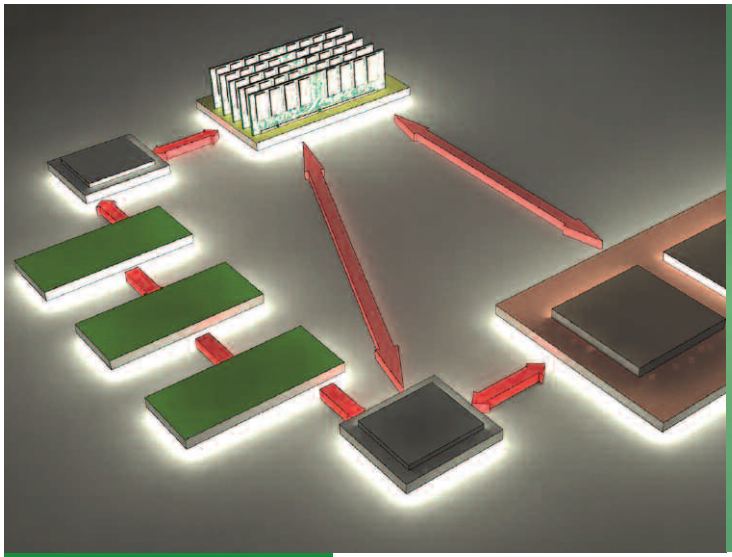
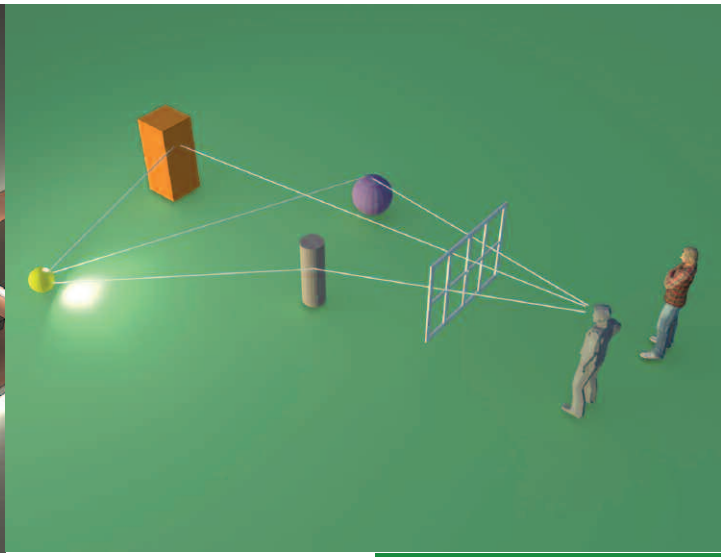
RENDER PROGRAMMA'S

```
            drawPhoton(rgb, gPoint);                        //Draw Photon
            shadowPhoton(ray);                              //Shadow Photon
            ray = reflect(ray,prevPoint);                   //Bounce the Photon
            raytrace(ray, gPoint);                          //Trace It to Next Location
            prevPoint = gPoint;
            bounces++;}
      }
   }

26 void storePhoton(int type, int id, float[] location, float[] direction, float[] energy){
      photons[type][id][numPhotons[type][id]][0] = location;  //Location
      photons[type][id][numPhotons[type][id]][1] = direction; //Direction
      photons[type][id][numPhotons[type][id]][2] = energy;    //Attenuated Energy (Color)
      numPhotons[type][id]++;
   }

27 void shadowPhoton(float[] ray){                            //Shadow Photons
      float[] shadow = {-0.25,-0.25,-0.25};
      float[] tPoint = gPoint;
      int tType = gType, tIndex = gIndex;                    //Save State
      float[] bumpedPoint = add3(gPoint,mul3c(ray,0.00001));  //Start Just Beyond Last Intersecti
      raytrace(ray, bumpedPoint);                            //Trace to Next Intersection (In Sh
      float[] shadowPoint = add3( mul3c(ray,gDist), bumpedPoint); //3D Point
      storePhoton(gType, gIndex, shadowPoint, ray, shadow);
      gPoint = tPoint; gType = tType; gIndex = tIndex;           //Restore State
   }

   float[] filterColor(float[] rgbIn, float r, float g, float b){ //e.g. White Light Hits Red Wa
      float[] rgbOut = {r,g,b};
      for (int c=0; c<3; c++) rgbOut[c] = min(rgbOut[c],rgbIn[c]); //Absorb Some Wavelengths (R,G
      return rgbOut;
   }

28 float[] getColor(float[] rgbIn, int type, int index){ //Specifies Material Color of Each Obje
      if      (type == 1 && index == 0) { return filterColor(rgbIn, 0.0, 1.0, 0.0);}
      else if (type == 1 && index == 2) { return filterColor(rgbIn, 1.0, 0.0, 0.0);}
      else                              { return filterColor(rgbIn, 1.0, 1.0, 1.0);}
   }
```



ISBN 978-90-8814-036-5 NUR-code 648

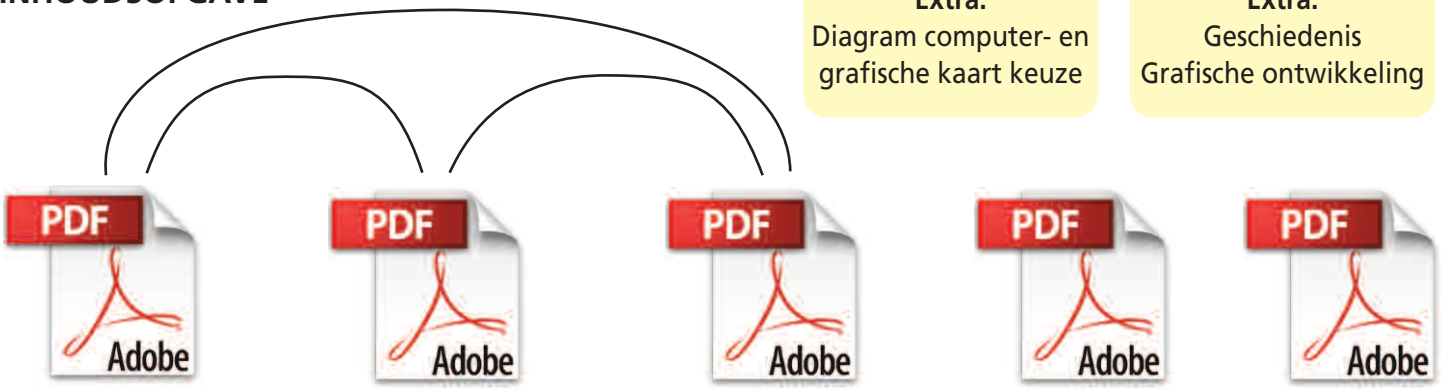




INHOUDSOPGAVE

Extra:
Diagram computer- en grafische kaart keuze

Extra:
Geschiedenis
Grafische ontwikkeling



render principes

programma codes

literatuur opgave met originele pdf's

NVIDIA LIJN

Werkstations

Render programma's

OPEN SOURCE
Real time Ray Tracing

Keim uit Japan

Keim ontwikkelde een Javascript renderprogramma dat binnen een beperkt aantal code regels een maximum aan resultaat laat zien. Waarbij de auteur vertelt dat z'n Javascript code nauwelijks onderdoet voor een Java variant. De coderegels met korte uitleg aan het einde zijn rood gekleurd. De auteur werkt ook met Flash en Action Script.

GEGEVENS AUTEUR
© Keim, Tokyo, Japan

My wonderfl
http://wonderfl.net/user/keim_at_Si

Twitter
http://twitter.com/keim_at_si

JActionScriptors
http://blog.jactionscripters.com/author/keim/

Blog in Japanese
http://d.hatena.ne.jp/keim_at_Si/

Blog in English (low activity...)
http://keim-at-si.blogspot.com/

```
// Flash version here. http://wonderfl.net/c/horg

var size = 260;
var ambient = 0.1;
var diffusion = 0.7;
var specular = 0.6;
var power = 12;


var mainGraphics, canvas, graphics, imageData, pixels;
var focusZ = size, floorY = 100;
var initialDir = new Array(size * size), spheres = new Array();
var ldirX, ldirY, ldirZ;
var smap = new Array(512), fcol = new Array(1024);
var refs = [0, 1, 2, 3, 3];
var reff = [0xffffff, 0x7f7f7f, 0x3f3f3f, 0x1f1f1f, 0x1f1f1f];
var time, prevTime, mx, my, camX, camY, camZ, tcamX, tcamY, tcamZ;

function Material(col, amb, dif) {
  this.lmap = new Array(512);
  var r = (col>>16)&255, g = (col>>8)&255, b = col&255, a;
  for (i=0; i<512; i++) {
    a = ((i<256) ? amb : ((((i-256) * (dif - amb)) * 0.00390625) + amb)) * 2;
    if (a<1) this.lmap[i] = ((r*a)<<16)|((g*a)<<8)|((b*a)<<0);
    else {
      a = 2-a;
      this.lmap[i] = ((255-(255-r)*a)<<16)|((255-(255-g)*a)<<8)|((255-(255-b)*a)<<0);
    }
  }
}

function Sphere(cx, cy, cz, cr, omg, pha, r, mat) {
  this.cx = cx;
  this.cy = cy;
  this.cz = cz;
  this.cr = cr;
  this.omg = omg;
  this.pha = pha;
```

```
  this.r2 = r * r;
  this.mat = mat;
  this.update = function() {
    var ang = time * this.omg + this.pha;
    this.x = Math.cos(ang)*this.cr+this.cx;
    this.y = this.cy;
    this.z = Math.sin(ang)*this.cr+this.cz;
  }
}


function setup() {
  canvas = $('#canvas').mousemove(onMouseMove).get(0);
  graphics = canvas.getContext('2d');
  imageData = graphics.createImageData(size, size);
  pixels = imageData.data;

  var i, j, idx, l, s, hs = (size - 1) * 0.5;
  for (j=0,idx=0; j<size; j++) for (i=0; i<size; i++) {
    l = 1/Math.sqrt((i - hs)*(i - hs) + (j - hs)*(j - hs) + focusZ*focusZ);
    initialDir[idx] = (i-hs) * l; idx++;
    initialDir[idx] = (j-hs) * l; idx++;
    initialDir[idx] = focusZ * l; idx++;
  }
  for (i=0; i<512; i++) {
    s = (i<256) ? 64 : ((Math.pow((i-256) * 0.00390625, power) * (power + 2) * specular * 0.15915494309189534 * 192 +
64)>>>0);
    if (s > 255) s = 255;
    smap[i] = (0x10101 * s) >>> 0;
    s = (i<256) ? (255-i) : (i-256);
    fcol[i] = (0x10101 * (s - (s>>3) + 31)) >>> 0;
    fcol[i+512] = (0x10101 * ((s>>2) - (s>>5) + 31)) >>> 0;
  }
  spheres.push(new Sphere(100, 40, 600, 200, 0.3, 1.0, 60, new Material(0x8080ff,ambient,diffusion)));
  spheres.push(new Sphere(  0, 50, 300, 100, 0.8, 0.8, 50, new Material(0x80ff80,ambient,diffusion)));
  spheres.push(new Sphere( 50, 60, 200, 200, 0.6, 2.0, 40, new Material(0xff8080,ambient,diffusion)));
  spheres.push(new Sphere(-50, 70, 500, 300, 0.4, 1.4, 30, new Material(0xc0c080,ambient,diffusion)));
  spheres.push(new Sphere(-90, 30, 600, 400, 0.2, 1.5, 70, new Material(0xc080c0,ambient,diffusion)));
  spheres.push(new Sphere( 70, 80, 400, 100, 0.7, 1.2, 20, new Material(0x80c0c0,ambient,diffusion)));
  camX = camY = camZ = tcamX = tcamY = tcamZ = 0;

  mx = my = time = 0;
  prevTime = (new Date()).getTime();
  setInterval(draw, 33);
}


function draw() {
  var i, j, k, l, idx, t, tmin, n, s, hit, ln, pixel, a, kmax, pidx,
    ox, oy, oz, dx, dy, dz, nx, ny, nz, dsx, dsy, dsz, B, C, D;

  t = (new Date()).getTime();
  time += (t - prevTime) * 0.001;
  prevTime = t;

  ldirX = -Math.cos(time*0.6)*100;
  ldirY = -Math.sin(time*1.1)*25-100;
  ldirZ = -Math.sin(time*0.9)*100+100;
  l = 1/Math.sqrt(ldirX*ldirX + ldirY*ldirY + ldirZ*ldirZ);
  ldirX *= l;
  ldirY *= l;
  ldirZ *= l;

  tcamX = mx * 400;
  tcamY = my * 150 - 50;
```

```
tcamZ = my * 400 - 200;
camX += (tcamX - camX) * 0.02;
camY += (tcamY - camY) * 0.02;
camZ += (tcamZ - camZ) * 0.02;

kmax = spheres.length;
for (k=0; k<kmax; k++) spheres[k].update();

for (j=0,idx=0,pidx=0; j<size; j++) {
  for (i=0; i<size; i++) {
    ox = camX;
    oy = camY;
    oz = camZ;
    dx = initialDir[idx]; idx++;
    dy = initialDir[idx]; idx++;
    dz = initialDir[idx]; idx++;

    pixel = 0;
    for (l=1; l<5; l++) {
      tmin = 99999;
      hit = null;
      for (k=0; k<kmax; k++) {
        s = spheres[k];
        dsx = ox - s.x;
        dsy = oy - s.y;
        dsz = oz - s.z;
        B = dsx * dx + dsy * dy + dsz * dz;
        C = dsx * dsx + dsy * dsy + dsz * dsz - s.r2;
        D = B * B - C;
        if (D > 0) {
          t = - B - Math.sqrt(D);
          if ((t > 0) && (t < tmin)) {
            tmin = t;
            hit = s;
          }
        }
      }

      if (hit) {
        ox += dx * tmin;
        oy += dy * tmin;
        oz += dz * tmin;
        nx = ox - hit.x;
        ny = oy - hit.y;
        nz = oz - hit.z;
        n = 1 / Math.sqrt(nx*nx + ny*ny + nz*nz);
        nx *= n;
        ny *= n;
        nz *= n;
        n = -(nx*dx + ny*dy + nz*dz) * 2;
        dx += nx * n;
        dy += ny * n;
        dz += nz * n;
        ln = (((ldirX * nx + ldirY * ny + ldirZ * nz) * 256) + 256) >>> 0;
        a = hit.mat.lmap[ln];
        a >>= refs[l];
        a &= reff[l];
        pixel += a;
      } else {
        if (dy < 0) {
          ln = (((ldirX * dx + ldirY * dy + ldirZ * dz) * 256) + 256) >>> 0;
          a = smap[ln];
          ln = l - 1;
          a >>= refs[ln];
          a &= reff[ln];
```

```
          pixel += a;
          break;
        } else {
          tmin = (floorY-oy)/dy;
          ox += dx * tmin;
          oy += dy * tmin;
          oz += dz * tmin;
          dy = -dy;
          ln = (dy * 256 + ( ( ( (((ox>>>0)+(oz>>>0))>>7) + (((ox>>>0)-(oz>>>0))>>7) ) &1)<<9) +256) >>> 0;
          a = fcol[ln];
          a >>= refs[l];
          a &= reff[l];
          pixel += a;
        }
      }
    }

    pixels[pidx] = (pixel>>16)&255; pidx++;
    pixels[pidx] = (pixel>>8)&255; pidx++;
    pixels[pidx] = (pixel)&255; pidx++;
    pixels[pidx] = 255; pidx++;
  }
 }

  graphics.putImageData(imageData, 0, 0);
}


function onMouseMove(e) {
  mx = (e.clientX - 232.5) * 0.00390625;
  my = (232.5 - e.clientY) * 0.00390625;
}


$(setup);
```

Verdere benodigdheden & uitleg

jquery.js file, style.css en de html pagina index.
In de index.html pagina wordt gebruik gemaakt van de HTML5 instructie:

```
<canvas id='canvas' width="192" height="192"/>

<script type="text/javascript"
src="jquery.js"></script>
<script type="text/javascript"
src="index.js"></script>
```

jQuery UI is the official jQuery User Interface framework. jQuery UI provides abstractions for low-level interactions and animation, advanced effects and high-level, themeable widgets, built on to...

```
this.lmap = new Array ( 512 );
```
nieuwe Array 512 aangemaakt

```
function Sphere(cx, cy, cz, cr, omg, pha, r, mat) {
```
start met de bollen (spheres)

```
  ldirX = -Math.cos(time*0.6)*100;
  ldirY = -Math.sin(time*1.1)*25-100;
  ldirZ = -Math.sin(time*0.9)*100+100;
  l = 1/Math.sqrt(ldirX*ldirX + ldirY*ldirY + ldirZ*ldirZ);
```

berekening X van cos (tijd * 0.6) * 100 =
cos (60 * tijd)
Bij de Y de Z wordt dat met andere waarden gedaan met de sinus functie.

Vervolgens:

$$\frac{1}{\sqrt{(ldirX^2 + ldirY^2 + ldirZ^2)}}$$

```
    hit = null;
    for ( k=0; k < kmax; k++ ) {
      s = spheres [ k ];
```

```
dsx = ox - s.x;
dsy = oy - s.y;
dsz = oz - s.z;

B = dsx * dx + dsy * dy + dsz * dz;
C = dsx * dsx + dsy * dsy + dsz * dsz - s.r2;
D = B * B - C;

if (D > 0) {
  t = - B - Math.sqrt(D);
  if ((t > 0) && (t < tmin)) {
    tmin = t;
    hit = s;
```

Hieruit volgt de bekende ABC formule
Eerst worden de verschillende kwadraten uitgerekend
= B Dan die met de kwadraten = D

Deze vergelijking voldoet aan de algemene vierkantsvergelijking

$f(x) = ax^2 + bx + c$

http://nl.wikipedia.org/wiki/Vierkantsvergelijking

Met de bekend veronderstelde oplossing van de vierkantsvergelijking (abc formule) zijn er een aantal mogelijkheden:

$D = a^2 (x1 - x2)^2 = b^2 - 4\,ac$

waarbij x1 en x2 complexe wortels van de oplossing zijn.
* Als D > 0 dan zijn er twee verschillende reële oplossingen
* Als D = 0 dan zijn er twee gelijke reële oplossingen x1 = x2, de wortel = - b / 2a
* Als D < 0 geen oplossingen, de straal raakt de bol niet maar schiet er voorbij.

```
if (hit) {
```

Als er een (hit) is dan wordt ondermeer de vector grootte in het 3D vlak berekend = n.

Centileo met 25 miljoen driekhoeken en 8 GB aan textures.

A Mathematica Optimization Problem door Xah Lee

Optimizing a Ray Trace Code

http://xahlee.info/UnixResource_dir/writ/Mathematica_optimization.html

In een dicussie in november 2008 daagde Dr Jon Harrop mij uit om een optimalisatie probleem op te lossen of te verbeteren.
De discussiegroep waarin dat verscheen was “comp.lang.python”. Over en weer werden weddenschappen afgesloten en de code hieronder moet bewijzen hoeveel en waar de code kon worden versneld. Thomas M Hermann

bood de betere code aan. De code voor Mathematica 6 uitleg onderstaand.

```
A Mathematica Optimization Problem
http://xahlee.info/UnixResource_dir/writ/Mathematica_optimization.html


delta = Sqrt[$MachineEpsilon];
RaySphere[o_, d_, c_, r_] := Block[{v, b, disc, t1, t2}, v = c - o;
    b = v.d;
    disc = Sqrt[b^2 - v.v + r^2];
    t2 = b + disc;
    If[Im[disc]   0 || t2   0,   , t1 = b - disc;
      If[t1 > 0, t1, t2]]]

Intersect[o_, d_][{lambda_, n_}, Sphere[c_, r_]] :=
  Block[{lambda2 = RaySphere[o, d, c, r]},
    If[lambda2   lambda, {lambda, n}, {lambda2,
        Normalize[o + lambda2 d - c]}]]

Intersect[o_, d_][{lambda_, n_}, Bound[c_, r_, s_]] :=
  Block[{lambda2 = RaySphere[o, d, c, r]},
    If[lambda2   lambda, {lambda, n}, Fold[Intersect[o, d], {lambda, n},
s]]]

neglight = N@Normalize[{1, 3, -2}];
nohit = {  , {0, 0, 0}};

RayTrace[o_, d_, scene_] :=
  Block[{lambda, n, g, p}, {lambda, n} = Intersect[o, d][nohit, scene];
    If[lambda ==   , 0, g = n.neglight;
```

```
      If[g   0,
        0, {lambda, n} =
          Intersect[o + lambda d + delta n, neglight][nohit, scene];
        If[lambda <   , 0, g]]]]

Create[level_, c_, r_] :=
  Block[{obj = Sphere[c, r]},
    If[level == 1, obj,
      Block[{a = 3*r/Sqrt[12], Aux},
        Aux[x1_, z1_] := Create[level - 1, c + {x1, a, z1}, 0.5 r];
        Bound[c,
          3 r, {obj, Aux[-a, -a], Aux[a, -a], Aux[-a, a], Aux[a, a]}]]]]

scene = Create[1, {0, -1, 4}, 1];
Main[level_, n_, ss_] :=
  Block[{scene = Create[level, {0, -1, 4}, 1]},
    Table[Sum[
          RayTrace[{0, 0, 0},
            N@Normalize[{(x + s/ss/ss)/n - 1/2, (y + Mod[s, ss]/ss)/n -
1/2,
                  1}], scene], {s, 0, ss^2 - 1}]/ss^2, {y, 0, n - 1}, {x,
0,
        n - 1}]]

AbsoluteTiming[Export["image.pgm", Graphics@Raster@Main[9, 512, 4]]]
```

De gecorrigeerde en verbeterde Mathematica code kunt u ophalen van deze pagina: http://xahlee.info/UnixResource_dir/writ/Mathematica_optimization.html

Het programma heeft de volgende hoofdfuncties

- RaySphere
- Intersect
- Ray Trace
- Create
- Main

De Main lus roept “Create” op en voert de verkregen uitkomst naar Ray Trace. De oproepen van Create zijn recursive en voert in principe een lange lijst van herhalende elementen op. Elk element verschilt van parameter.

Ray Trace roept Intersect 2 keer op en bestaat uit 2 formules, waarvan er één wederom recursive is. Beiden roepen RaySphere één keer op.

De hoofdlus in het render programma is de Intersect functie en RaySphere. Volgens de ontwerper wordt 99,9 % van de renderingstijd in deze hoofdlus zoet gebracht.
De Main [ ] functie roept Create op. Create heeft 3 parameters: level, c en r. Het level is een geheel getal voor het aantal spheres in het model. De input daarvan heeft c en r als hele getallen. In Mathematica betekent dat er met ‘echte’ wiskunde wordt gerekend. Waardoor er met een ‘oneindige’ nauwkeurigheid, indien gewenst, zou kunnen worden gewerkt. Door c en r om te zetten naar een drijvende komma maakt direct een snelheidsverbetering zichtbaar van 0,3 x ten opzichte van het origineel. Bij RaySphere dient niet gecontroleerd te wor-

den of de discriminant een imaginair deel heeft maar eenvoudig òf het argument van de wortel negatief is.
De **RaySphere** code is dus aan vervanging toe. Op de website kunt u de volledige tekst en de nieuwe code vinden. Het resultaat in tijdwinst ten opzichte van het orgineel is op 0,2x snelheidswinst uitgekomen. Ook verbetering aan andere coderegels komen ter sprake om de snelheid nog verder op te voeren. Hier ziet u in het 'klein' wat bij grote renderingsprogramma's ook continue dient te gebeuren: controleren en zo mogelijk aanpassen van formules om tijdwinst te kunnen boeken.

In **Intersect** wordt met twee recursive lussen gewerkt, dat kan worden verbeterd door er een uit te halen.

Zie ook:
http://www.ffconsultancy.com/languages/ray_tracer/
Flying Frog Consultancy
Ray tracer language comparison

Waarin vijf progressieve geoptimaliseerde versies van de Ray Tracer wordt besproken die in de diverse C++, Java, OCaml en SML zijn geschreven.

http://www.ffconsultancy.com/languages/ray_tracer/verbosity.html
Code onderdelen van de verschillende programmeer omgevingen.

http://www.ffconsultancy.com/languages/ray_tracer/performance.html
Ray Tracer Language comparison
Er worden veel problemen aangehaald om een echt vergelijk te kunnen maken tussen diverse programmeertalen.

Daarbij zijn er enkele hoofdpunten die voor Ray Tracing in het algemeen gelden:
- floating point rekensnelheid wordt beïnvloed door de Ray-Sphere intersectie routine.
- de bestemming en afkomst wordt beïnvloed door de snelle recycling van de vector waarden.
- beschrijvende of Tree Data structuren worden beïnvloed door de scene intersectie routine.

Daaruit kan een voorlopige conclusie worden getrokken:
De 'Garbage collection' is niet langzaam.
De snelste is die met programmeertaal C++ Statische type informatie maakt goede optimalisatie mogelijk. Met OCaml, C++ en Standaard ML worden snellere programma's gemaakt. Daarmee komen C++ (64- en 32 bit) en OCaml als beste uit deze test. Ook het soort compiler maakt deel uit van een sneller eindresultaat. De beste compilers zijn Stalin en MLton.

Ofschoon **Java** met een aantal onderdelen uit deze Benchmark kan wedijveren, is het toch 2 tot 3x zo langzaam dan de C++ codes. Java heeft natuurlijk een groot aantal voordelen, waardoor sommige programmeurs toch voor Java kiezen ondanks de mindere snelheidsprestaties. De auteur van deze programmeer code adviseert echter het gebruik van C++ en dat studie van C++ in de toekomst voordelen op zal leveren.

De uiteindelijke snelheidsverschillen zijn:

Language: Compiler	**Compile time**
SML: SML/NJ	0.23s
OCaml: ocamlopt	0.41s
C++: g++	0.71s
Java: JDK 1.5	0.86s
Lisp: SBCL	1.1s
SML: MLton	3.1s
Scheme: Stalin	166s

This work was inspired by the excellent book "An Introduction to Ray Tracing" by Andrew S. Glassner.

http://semanticvector.blogspot.nl/2008/08/f-for-scientists-misses-boat-on.html
F# voor Scientists Misses the Boat On Mathematica Performance. augustus 2008.

Hoe wiskundigen en vakgenoten een routine beoordelen. De eerste testmeting kwam uit op 26 seconden. Met een speciale translater 7,5 seconde. Maar de schrijver toont aan dat de programmeercode niet juist is geïnterpreteerd en tovert een fraaie 5,2 seconde uit de hoge hoed. Ook voor renderingscodes en programma's geldt dat de gebruikte code en corresponderende wiskunde gigantische renderingstijd verschillen kan opleveren.

Open Source

Toxic Renderer

Toxic is a 'physically correct' global illumination renderer aiming to produce photorealistic images and animations.

Toxic's goal is to provide artists with a free, powerful rendering tool which is actively maintained, developed and extended.

Toxic is an open source project, available on SourceForge, and licensed under the GPL license. That means:

* toxic is free for commercial and non-commercial use.
* toxic sources are freely available, modifiable and redistributable.
* toxic sources may be used in any open source project licensed under the GPL license.

Modern renderprogramma met subcodes titels:

utilities.inl
utilities.h
utilities.cpp
triboxoverlap.h
triboxoverlap.cpp
TODO.txt
thinlenscamera.inl
surfacesamplerfactory.inl
surfacesamplerfactory.h
thinlenscamera.h
surfacesamplerfactory.cpp
surfacebasis.inl
surfacebasis.h
surfacebasis.cpp
stratifiedsurfacesampler.inl
stratifiedsurfacesampler.h
stratifiedsurfacesampler.cpp
statistics.inl
statistics.h
statistics.cpp
square.inl
square.h
square.cpp
sphere.inl
sphere.h
sphere.cpp
shadingdata.inl
thinlenscamera.cpp
settings.cpp
SConscript
scene.inl
scene.h
scene.cpp
ring.inl
ring.h
ring.cpp
renderer.inl
renderer.h
renderer.cpp
regularsurfacesampler.h
regularsurfacesampler.inl
regularsurfacesampler.cpp
ray.inl
ray.h
ray.cpp
randomsurfacesampler.inl
randomsurfacesampler.h
randomsurfacesampler.cpp
projects
pointlight.inl
pointlight.h
pointlight.cpp
plane.inl
plane.h
plane.cpp
pinholecamera.inl
pinholecamera.h
pinholecamera.cpp
photonmap.inl
photonmap.h
photonmap.cpp
photon.inl
photon.h
perfectspecularbrdf.inl
perfectspecularbrdf.h
octree.inl
perfectspecularbrdf.cpp
octree.h
octree.cpp
meshbuilder.inl
meshbuilder.h
meshbuilder.cpp
mesh.inl
mesh.h
mesh.cpp
map2.inl
map2.h
map2.cpp
lambertianedf.inl
lambertianedf.h
lambertianedf.cpp
lambertianbrdf.inl
lambertianbrdf.h
lambertianbrdf.cpp
itexture.inl
photon.cpp
itexture.h
isurfaceshader.inl
isurfaceshader.h
isurfaceshader.cpp
isurfacesampler.inl
isurfacesampler.h
isurfacesampler.cpp
iphotontracer.inl
iphotontracer.h
iphotontracer.cpp
iobject.inl
iobject.h
iobject.cpp
intersecttriangle.h
intersecttriangle.cpp
intersectiondata.inl
intersectiondata.cpp
imagetexture.inl
imagetexture.h
imagetexture.cpp
ilight.inl
ilight.h
ilight.cpp
iedf.inl
iedf.h
iedf.cpp
icamera.inl
icamera.h
icamera.cpp
iboundedobject.inl
iboundedobject.h
iboundedobject.cpp
ibdf.inl
ibdf.h
ibdf.cpp
iarealight.inl
iarealight.h
iarealight.cpp
hit.inl
hit.h
hit.cpp
globals.h
globalphotontracer.inl
globalphotontracer.h
globalphotontracer.cpp
framebuffer.inl
framebuffer.cpp
dummyobject.inl
directionallight.h
directionallight.cpp
color3.inl
context.cpp
directionallight.inl
itexture.cpp
cube.inl
cube.h
collection.inl
context.inl
context.h
cube.cpp
constanttexture.h
constanttexture.cpp
dummyobject.h
dummyobject.cpp
bsdf.inl
color3.h
constanttexture.inl
framebuffer.h
intersectiondata.h
basicsurfaceshader.h
basis.h
basicsurfaceshader.inl
causticsphotontracer.cpp
bsdf.cpp
collection.h
basis.inl
basicsurfaceshader.cpp
causticsphotontracer.inl
bsdf.h
collection.cpp
causticsphotontracer.h
basis.cpp
color3.cpp
shadingdata.h
shadingdata.cpp
settings.inl
settings.h

Ray-Triangle Intersection Test Routines - GI renderer

```
/*
      toxic - A Global Illumination Renderer
      Copyright (C) 2003-2004 Francois Beaune
      Copyright (C) 2004 The toxic Project
      Contact: http://www.toxicengine.org

      This file is part of toxic.

      toxic is free software; you can redistribute it and/or modify
      it under the terms of the GNU General Public License as published by
      the Free Software Foundation; either version 2 of the License, or
      (at your option) any later version.

      toxic is distributed in the hope that it will be useful,
      but WITHOUT ANY WARRANTY; without even the implied warranty of
      MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
```

```
      GNU General Public License for more details.

      You should have received a copy of the GNU General Public License
      along with toxic; if not, write to the Free Software
      Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
*/

/* Ray-Triangle Intersection Test Routines          */
/* Different optimizations of my and Ben Trumbore's */
/* code from journals of graphics tools (JGT)       */
/* http://www.acm.org/jgt/                          */
/* by Tomas Moller, May 2000                        */

#include "intersecttriangle.h"      // include first
#include "ray.h"

#include <cmath>

using namespace sheep;
using namespace toxic;

const Real EPSILON = 0.000001;

// The original jgt code.
bool toxic::intersect_triangle(const Ray &ray,
                                          const Vector3 &vert0,
                                          const Vector3 &vert1,
                                          const Vector3 &vert2,
                                          Real *t, Real *u, Real *v)
{
      // Find vectors for two edges sharing vert0.
      const Vector3 edge1 = vert1 - vert0;
      const Vector3 edge2 = vert2 - vert0;

      // Begin calculating determinant - also used to calculate U parameter.
      const Vector3 pvec = ray.m_direction ^ edge2;

      // If determinant is near zero, ray lies in plane of triangle.
      const Real det = edge1 * pvec;
//    if(det > -EPSILON && det < EPSILON)
//          return false;
      if(det == 0.0)
            return false;

      const Real inv_det = 1.0 / det;

      // Calculate distance from vert0 to ray origin.
      const Vector3 tvec = ray.m_origin - vert0;

      // Calculate U parameter and test bounds.
      *u = (tvec * pvec) * inv_det;
      if(*u < 0.0 || *u > 1.0)
            return false;

      // Prepare to test V parameter.
      const Vector3 qvec = tvec ^ edge1;

      // Calculate V parameter and test bounds.
      *v = (ray.m_direction * qvec) * inv_det;
      if(*v < 0.0 || *u + *v > 1.0)
            return false;

      // Calculate t, ray intersects triangle.
      *t = (edge2 * qvec) * inv_det;
```

```
	return true;
}

// Code rewritten to do tests on the sign of the determinant.
// The division is at the end in the code.
bool toxic::intersect_triangle1(const Ray &ray,
                                                const Vector3 &vert0,
                                                const Vector3 &vert1,
                                                const Vector3 &vert2,
                                                Real *t, Real *u, Real *v)
{
	// Find vectors for two edges sharing vert0.
	const Vector3 edge1 = vert1 - vert0;
	const Vector3 edge2 = vert2 - vert0;

	// Begin calculating determinant - also used to calculate U parameter.
	const Vector3 pvec = ray.m_direction ^ edge2;

	// If determinant is near zero, ray lies in plane of triangle.
	const Real det = edge1 * pvec;

	Vector3 qvec;

	if(det > EPSILON) {
		// Calculate distance from vert0 to ray origin.
		const Vector3 tvec = ray.m_origin - vert0;

		// Calculate U parameter and test bounds.
		*u = tvec * pvec;
		if(*u < 0.0 || *u > det)
			return false;

		// Prepare to test V parameter.
		qvec = tvec ^ edge1;

		// Calculate V parameter and test bounds.
		*v = ray.m_direction * qvec;
		if(*v < 0.0 || *u + *v > det)
			return false;
	}
	else if(det < -EPSILON) {
		// Calculate distance from vert0 to ray origin.
		const Vector3 tvec = ray.m_origin - vert0;

		// Calculate U parameter and test bounds.
		*u = tvec * pvec;
		if(*u > 0.0 || *u < det)
			return false;

		// Prepare to test V parameter.
		qvec = tvec ^ edge1;

		// Calculate V parameter and test bounds.
		*v = ray.m_direction * qvec;
		if(*v > 0.0 || *u + *v < det)
			return false;
	}
	else return false;  // ray is parallel to the plane of the triangle

	const Real inv_det = 1.0 / det;

	// Calculate t, ray intersects triangle.
	*t = (edge2 * qvec) * inv_det;
	(*u) *= inv_det;
	(*v) *= inv_det;
```

```
	return true;
}

// Code rewritten to do tests on the sign of the determinant.
// The division is before the test of the sign of the determinant.
bool toxic::intersect_triangle2(const Ray &ray,
                                                const Vector3 &vert0,
                                                const Vector3 &vert1,
                                                const Vector3 &vert2,
                                                Real *t, Real *u, Real *v)
{
	// Find vectors for two edges sharing vert0.
	const Vector3 edge1 = vert1 - vert0;
	const Vector3 edge2 = vert2 - vert0;

	// Begin calculating determinant - also used to calculate U parameter.
	const Vector3 pvec = ray.m_direction ^ edge2;

	// If determinant is near zero, ray lies in plane of triangle.
	const Real det = edge1 * pvec;

	// Calculate distance from vert0 to ray origin.
	const Vector3 tvec = ray.m_origin - vert0;

	const Real inv_det = 1.0 / det;

	Vector3 qvec;

	if(det > EPSILON) {
		// Calculate U parameter and test bounds.
		*u = tvec * pvec;
		if(*u < 0.0 || *u > det)
			return false;

		// Prepare to test V parameter.
		qvec = tvec ^ edge1;

		// Calculate V parameter and test bounds.
		*v = ray.m_direction * qvec;
		if(*v < 0.0 || *u + *v > det)
			return false;
	}
	else if(det < -EPSILON) {
		// Calculate U parameter and test bounds.
		*u = tvec * pvec;
		if(*u > 0.0 || *u < det)
			return false;

		// Prepare to test V parameter.
		qvec = tvec ^ edge1;

		// Calculate V parameter and test bounds.
		*v = ray.m_direction * qvec;
		if(*v > 0.0 || *u + *v < det)
			return false;
	}
	else return false;  // ray is parallel to the plane of the triangle

	// Calculate t, ray intersects triangle.
	*t = (edge2 * qvec) * inv_det;
	(*u) *= inv_det;
	(*v) *= inv_det;

	return true;
```

```
}

// Code rewritten to do tests on the sign of the determinant.
// The division is before the test of the sign of the determinant and
// one cross product has been moved out from the if-else if-else.
bool toxic::intersect_triangle3(const Ray &ray,
                                                const Vector3 &vert0,
                                                const Vector3 &vert1,
                                                const Vector3 &vert2,
                                                Real *t, Real *u, Real *v)
{
      // Find vectors for two edges sharing vert0.
      const Vector3 edge1 = vert1 - vert0;
      const Vector3 edge2 = vert2 - vert0;

      // Begin calculating determinant - also used to calculate U parameter.
      const Vector3 pvec = ray.m_direction ^ edge2;

      // If determinant is near zero, ray lies in plane of triangle.
      const Real det = edge1 * pvec;

      // Calculate distance from vert0 to ray origin.
      const Vector3 tvec = ray.m_origin - vert0;

      const Real inv_det = 1.0 / det;

      const Vector3 qvec = tvec ^ edge1;

      if(det > EPSILON) {
            // Calculate U parameter and test bounds.
            *u = tvec * pvec;
            if(*u < 0.0 || *u > det)
                  return false;

            // Calculate V parameter and test bounds.
            *v = ray.m_direction * qvec;
            if(*v < 0.0 || *u + *v > det)
                  return false;
      }
      else if(det < -EPSILON) {
            // Calculate U parameter and test bounds.
            *u = tvec * pvec;
            if(*u > 0.0 || *u < det)
                  return false;

            // Calculate V parameter and test bounds.
            *v = ray.m_direction * qvec;
            if(*v > 0.0 || *u + *v < det)
                  return false;
      }
      else return false;  // ray is parallel to the plane of the triangle

      // Calculate t, ray intersects triangle.
      *t = (edge2 * qvec) * inv_det;
      (*u) *= inv_det;
      (*v) *= inv_det;

      return true;
}
```

Pseudocode voor het eerste Ray Tracing principe
U kunt deze code omwerken voor bv. C++ of het op Java gebaseerde Processing dat een Open Source programma is, onafhankelijk van het platform.

```
for ( int j = 0;
        j < beeldHoogte; ++j ) {
   for ( int i = 0;
        i < beeldBreedte; ++i ) {
        // bereken de richting van de eerste lichtstraal
        Ray primStraal;
        berekenPrimStraal ( i, j, &primStraal );
        // richt de eerste lichtstraal naar de 3D scene en ga op zoek naar het trefpunt met een object
 Point pTref;
 Normal nTref;
 float minAfstand = oneindig;
 Object object = NULL;

for (int k = 0; k < objects.grootte(); ++k) {
     if (Trefpunt(objects[k], primStraal, &pTref, &nTref)) {
        float afstand = Afstand (oogPositie, pTref);
        if (afstand < minAfstand) { object = objects[i];
           minAfstand = afstand;  // update min distance {
          }
        }
    }
// als object 0 is dan is het een schaduw straal, bereken verlichting
if (object != NULL) {
    schaduwStraal.richting = lichtPositie - pTref;
     bool Schaduw = false;
    for (int k = 0; k < objects.size(); ++k) {
        if (Intersect (objects[k], schaduwStraal)) {
           ligtInSchaduw = true; break;
          }
       }
    }
    if   (! ligtInSchaduw) // niet in de schaduw, dan pixel inkleuren
         pixels[i] [j] = object -> color * licht.helderheid;
      else
         pixels[i] [j] = 0;
   }
 }
```

De kracht en eenvoud van een Ray Tracing programma zoals hierboven blijkt duidelijk uit het geringe aantal programma regels. Samen met een bescheiden interface en een extra 3D bibliotheek voor de scene komen we aan enkele honderden coderegels om de meest simpele 3D modellen te kunnen renderen.

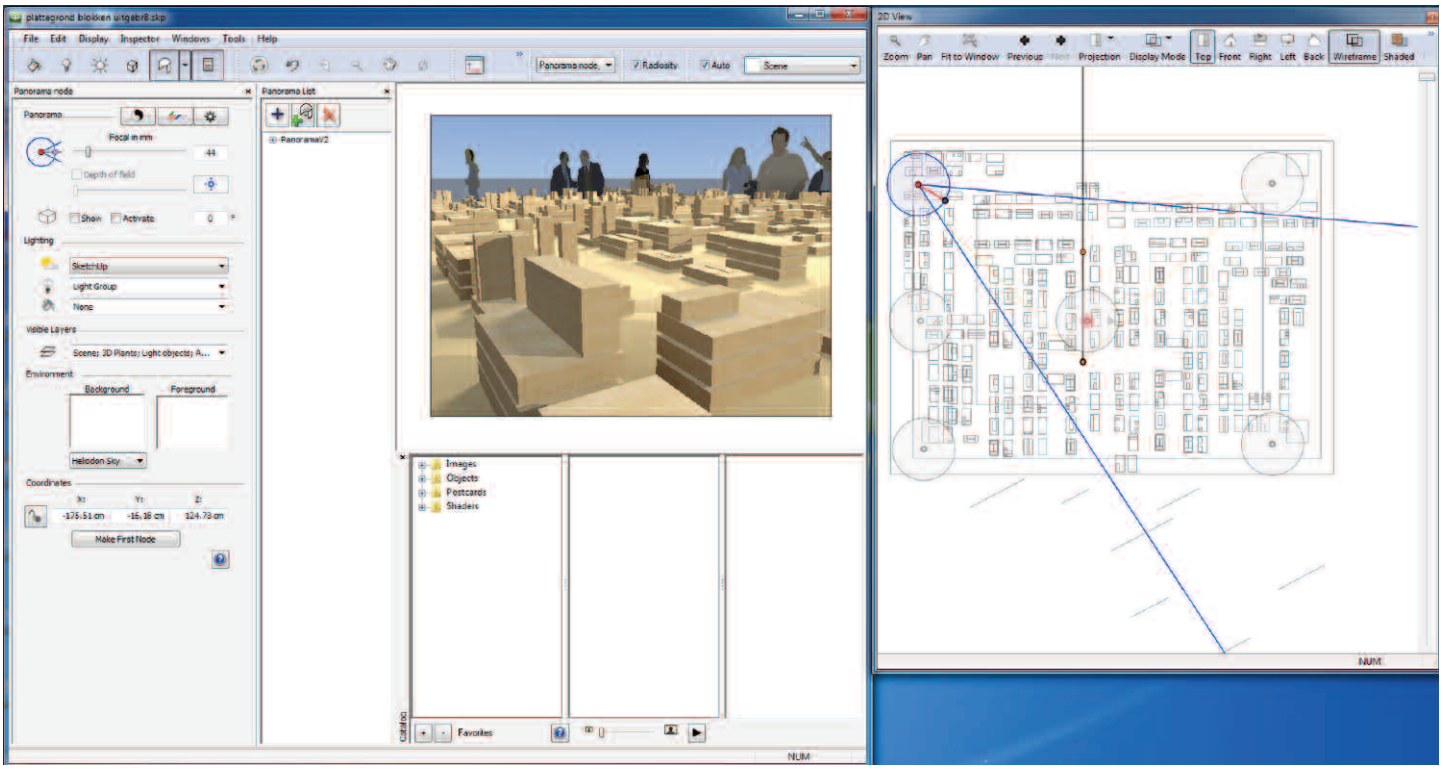
Pseudo code regels voor een basis renderprogramma

```
for (int j = 0; j < beeldHoogte; ++j) {
   for (int i = 0; i < beeldBreedte; ++i) {
   // compute primary ray direction
   Ray primRay;
   computePrimRay(i, j, &primRay);
   // shoot primary ray in the scene and search
      for intersection
   Point pHit;
   Normal nHit;
   float minDist = INFINITY;
   Object object = NULL;
   for (int k = 0; k < objects.size(); ++k){
     if (Intersect(objects[k], primRay, &pHit, &nHit)){
   float distance = Distance(eyePosition, pHit);
     if (distance < minDistance) { object = objects[i];
       minDistance = distance; // update min distance
        }
      }
   }
   if (object != NULL) {
   // compute illumination Ray shadowRay;
   shadowRay.direction = lightPosition - pHit;
   bool isShadow = false;
   for (int k = 0; k < objects.size(); ++k) {
     if (Intersect(objects[k], shadowRay)) {
       isInShadow = true;
       break;
```

```
        }
      }
    }
    if (!isInShadow)
     pixels[i][j] = object->color * light.brightness;
    else pixels[i][j] = 0;
    }
}
```

$$L_o = L_e + \int_\Omega L_i \cdot f_r \cdot \cos\theta \cdot d\omega$$

Algemene renderformule.

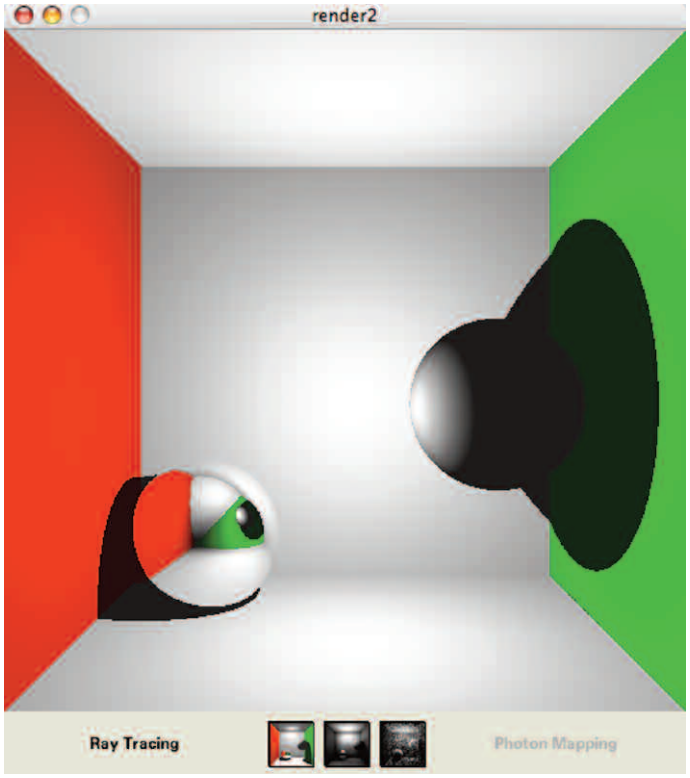


Commercieel render programma Artlantis Studio. Links de diverse menu's, in het midden de Preview die bijzonder snel reageert en rechts het 2D aanzicht. Daar zijn diverse hotspots aangemaakt voor koppeling naar panorama voor iPad en iPhone of Androïd smart phones met iVisit.

Ray Tracing & Photon Mapping
Grant Schindler, 2007

met werkend voorbeeld op internetsite
http://www.cc.gatech.edu/~phlosoft/photon/

Het doel was om op de meest eenvoudige manier Ray Tracing met Photon Mapping te combineren met een Cornell Box voorbeeld. De broncode is bestemd voor educatieve doeleinden. Het is geschreven in Processing. Indien u de gratis Processing download (http://www.processing.org) dan kunt u ervaren hoe dit render programma werkt. En zelf veranderingen aanbrengen en direct het resultaat daarvan beoordelen.



Cornell Box. Ray Tracing en of Photon Mapping met Processing in een Mac OSX Intel. Render WERKT IN PROCESSING 1.5 Mac OSX 10.4.11

```
//Ray Tracing & Photon Mapping
//Grant Schindler, 2007

http://www.cc.gatech.edu/~phlosoft/photon/

// ------------------- Scene Description --------------
int szImg = 512;                //Image Size
int nrTypes = 2;                //2 Object Types (Sphere = 0, Plane
= 1)
int{ } nrObjects = {2,5};       //2 Spheres, 5 Planes
float gAmbient = 0.1;           //Ambient Lighting
float{ } gOrigin = {0.0,0.0,0.0};  //World Origin for Convenient
Re-Use Below (Constant)
float{ } Light = {0.0,1.2,3.75};   //Point Light-Source Position
float{ }{ } spheres = {{1.0,0.0,4.0,0.5},{-0.6,-1.0,4.5,0.5}};
//Sphere Center & Radius
float{ }{ } planes  = {{0, 1.5},{1, -1.5},{0, -1.5},{1, 1.5},{2,5.0}};
//Plane Axis & Distance-to-Origin

// ----- Photon Mapping -----
int nrPhotons = 1000;           //Number of Photons Emitted
int nrBounces = 3;              //Number of Times Each Photon
Bounces
boolean lightPhotons = true;    //Enable Photon Lighting?
float sqRadius = 0.7;           //Photon Integration Area
(Squared for Efficiency)
float exposure = 50.0;          //Number of Photons Integrated
at Brightest Pixel
int{ }{ } numPhotons = {{0,0},{0,0,0,0,0}};          //Photon
Count for Each Scene Object
float{ }{ }{ }{ }{ } photons = new float[2][5][5000][3][3]; //Allocated
Memory for Per-Object Photon Info

// ----- Raytracing Globals -----
boolean gIntersect = false;     //For Latest Raytracing Call...
Was Anything Intersected by the Ray?
int gType;                      //... Type of the Intersected Object
(Sphere or Plane)
int gIndex;                     //... Index of the Intersected Object
(Which Sphere/Plane Was It?)
float gSqDist, gDist = -1.0;    //... Distance from Ray Origin to
Intersection
float{ } gPoint = {0.0, 0.0, 0.0}; //... Point At Which the Ray In-
tersected the Object
```

```
//------------------------------------------------------------------
//Ray-Geometry Intersections  ---------------------------------
//------------------------------------------------------------------

void raySphere(int idx, float{ } r, float{ } o) //Ray-Sphere Inter-
section: r=Ray Direction, o=Ray Origin
{
  float{ } s = sub3(spheres[idx],o);  //s=Sphere Center Translated
into Coordinate Frame of Ray Origin
  float radius = spheres[idx][3];    //radius=Sphere Radius

  //Intersection of Sphere and Line     =      Quadratic Function
of Distance
  float A = dot3(r,r);                    // Remember This From High
School? :
  float B = -2.0 * dot3(s,r);             //    A x^2 +     B x +
C  = 0
  float C = dot3(s,s) - sq(radius);         // (r'r)x^2 - (2s'r)x + (s's -
radius^2) = 0
  float D = B*B - 4*A*C;                  // Precompute Discriminant

  if (D > 0.0){                           //Solution Exists only if sqrt(D) is
Real (not Imaginary)
    float sign = (C < -0.00001) ? 1 : -1;    //Ray Originates Inside
Sphere If C < 0
    float lDist = (-B + sign*sqrt(D))/(2*A); //Solve Quadratic
Equation for Distance to Intersection
    checkDistance(lDist,0,idx);}           //Is This Closest Intersec-
tion So Far?
}

void rayPlane(int idx, float{ } r, float{ } o){ //Ray-Plane Intersec-
tion
  int axis = (int) planes[idx][0];          //Determine Orientation of
Axis-Aligned Plane
  if (r[axis] != 0.0){                    //Parallel Ray -> No Intersec-
tion
    float lDist = (planes[idx][1] - o[axis]) / r[axis]; //Solve Linear
Equation (rx = p-o)
```

```
    checkDistance(lDist,1,idx);}
}

void rayObject(int type, int idx, float[ ] r, float[ ] o){
  if (type == 0) raySphere(idx,r,o); else rayPlane(idx,r,o);
}

void checkDistance(float lDist, int p, int i){
  if (lDist < gDist && lDist > 0.0){ //Closest Intersection So Far in
Forward Direction of Ray?
    gType = p; gIndex = i; gDist = lDist; gIntersect = true;} //Save
Intersection in Global State
}

//------------------------------------------------------------------------
// Lighting --------------------------------------------------------------
//------------------------------------------------------------------------

float lightDiffuse(float[ ] N, float[ ] P){
  //Diffuse Lighting at Point P with Surface Normal N

  float[ ] L = normalize3( sub3(Light,P) );
  //Light Vector (Point to Light)

  return dot3(N,L);
   //Dot Product = cos (Light-to-Surface-Normal Angle)
}

float[ ] sphereNormal(int idx, float[ ] P){
  return normalize3(sub3(P,spheres[idx]));
  //Surface Normal (Center to Point)

}

float[ ] planeNormal(int idx, float[ ] P, float[ ] O){
  int axis = (int) planes[idx][0];
  float [ ] N = {0.0,0.0,0.0};
  N[axis] = O[axis] - planes[idx][1];      //Vector From Surface to
Light
  return normalize3(N);
}

float[ ] surfaceNormal(int type, int index, float[ ] P, float[ ] Ins-
ide){
  if (type == 0) {return sphereNormal(index,P);}
  else           {return planeNormal(index,P,Inside);}
}

float lightObject(int type, int idx, float[ ] P, float lightAmbient){
  float i = lightDiffuse( surfaceNormal(type, idx, P, Light) , P );
  return min(1.0, max(i, lightAmbient));   //Add in Ambient Light
by Constraining Min Value
}

//------------------------------------------------------------------------
// Raytracing ------------------------------------------------------------
//------------------------------------------------------------------------

void raytrace(float[ ] ray, float[ ] origin)
{
  gIntersect = false; //No Intersections Along This Ray Yet
  gDist = 999999.9;   //Maximum Distance to Any Object

  for (int t = 0; t < nrTypes; t++)
    for (int i = 0; i < nrObjects[t]; i++)
      rayObject(t,i,ray,origin);
}

float[ ] computePixelColor(float x, float y){
  float[ ] rgb = {0.0,0.0,0.0};
  float[ ] ray = {  x/szImg - 0.5 ,       //Convert Pixels to Image
Plane Coordinates
                -(y/szImg - 0.5), 1.0}; //Focal Length = 1.0
  raytrace(ray, gOrigin);                //Raytrace!!! - Intersected Ob-
jects are Stored in Global State

  if (gIntersect){                       //Intersection
    gPoint = mul3c(ray,gDist);           //3D Point of Intersection

    if (gType == 0 && gIndex == 1){      //Mirror Surface on This
Specific Object
      ray = reflect(ray,gOrigin);        //Reflect Ray Off the Surface
      raytrace(ray, gPoint);             //Follow the Reflected Ray
      if (gIntersect){ gPoint = add3( mul3c(ray,gDist), gPoint); }}
//3D Point of Intersection

    if (lightPhotons){                   //Lighting via Photon Mapping
      rgb = gatherPhotons(gPoint,gType,gIndex);}
    else{                                //Lighting via Standard Illumination
Model (Diffuse + Ambient)
      int tType = gType, tIndex = gIndex;//Remember Intersected
Object
      float i = gAmbient;                //If in Shadow, Use Ambient
Color of Original Object
      raytrace( sub3(gPoint,Light) , Light);  //Raytrace from Light
to Object
      if (tType == gType && tIndex == gIndex) //Ray from Light-
>Object Hits Object First?
        i = lightObject(gType, gIndex, gPoint, gAmbient); //Not In
Shadow - Compute Lighting
      rgb[0]=i; rgb[1]=i; rgb[2]=i;
      rgb = getColor(rgb,tType,tIndex);}
  }
  return rgb;
}

float[ ] reflect(float[ ] ray, float[ ] fromPoint){        //Reflect Ray
  float[ ] N = surfaceNormal(gType, gIndex, gPoint, fromPoint);
```

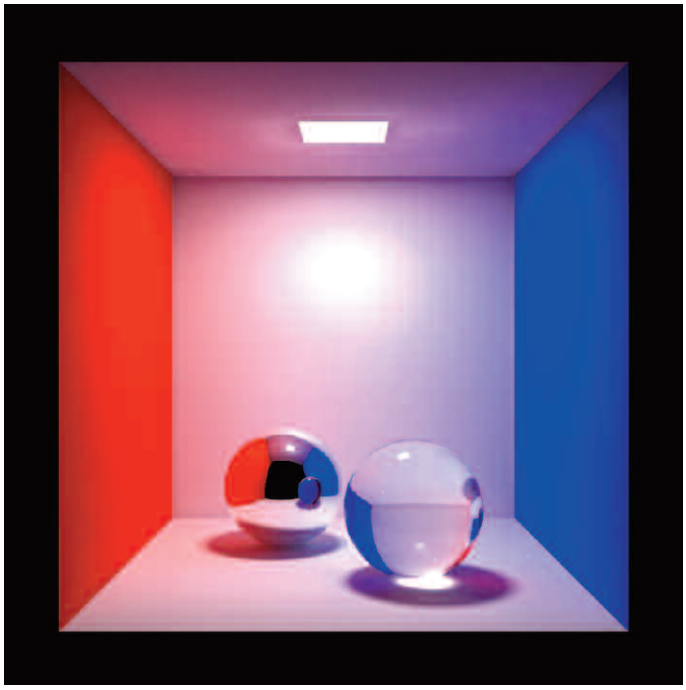
```
//Surface Normal
  return normalize3(sub3(ray, mul3c(N,(2 * dot3(ray,N)))));
//Approximation to Reflection
}

//----------------------------------------------------------------------
//Photon Mapping ------------------------------------------------------
//----------------------------------------------------------------------
float[ ] gatherPhotons(float[ ] p, int type, int id){
  float[ ] energy = {0.0,0.0,0.0};
  float[ ] N = surfaceNormal(type, id, p, gOrigin);
//Surface Normal at Current Point
  for (int i = 0; i < numPhotons[type][id]; i++){                  //Pho-
tons Which Hit Current Object
    if (gatedSqDist3(p,photons[type][id][i][0],sqRadius)){
//Is Photon Close to Point?
      float weight = max(0.0, -dot3(N, photons[type][id][i][1] ));
//Single Photon Diffuse Lighting
      weight *= (1.0 - sqrt(gSqDist)) / exposure;
//Weight by Photon-Point Distance
      energy = add3(energy, mul3c(photons[type][id][i][2],
weight)); //Add Photon's Energy to Total
   }}
  return energy;
}

void emitPhotons(){
  randomSeed(0);               //Ensure Same Photons Each Time
  for (int t = 0; t < nrTypes; t++)     //Initialize Photon Count to
Zero for Each Object
    for (int i = 0; i < nrObjects[t]; i++)
      numPhotons[t][i] = 0;

  for (int i = 0; i < (view3D ? nrPhotons * 3.0 : nrPhotons); i++){
//Draw 3x Photons For Usability
    int bounces = 1;
    float[ ] rgb = {1.0,1.0,1.0};   //Initial Photon Color is White
    float[ ] ray = normalize3( rand3(1.0) );  //Randomize Direction
of Photon Emission
    float[ ] prevPoint = Light;     //Emit From Point Light Source

    //Spread Out Light Source, But Don't Allow Photons Outside
Room/Inside Sphere
    while (prevPoint[1] >= Light[1]){ prevPoint = add3(Light,
mul3c(normalize3(rand3(1.0)), 0.75));}
    if (abs(prevPoint[0]) > 1.5 || abs(prevPoint[1]) > 1.2 ||
       gatedSqDist3(prevPoint,spheres[0],spheres[0][3]*sphe-
res[0][3])) bounces = nrBounces+1;

    raytrace(ray, prevPoint);          //Trace the Photon's Path

    while (gIntersect && bounces <= nrBounces){   //Intersection
With New Object
       gPoint = add3( mul3c(ray,gDist), prevPoint);  //3D Point of
Intersection
       rgb = mul3c (getColor(rgb,gType,gIndex), 1.0/sqrt(boun-
ces));
       storePhoton(gType, gIndex, gPoint, ray, rgb);  //Store Pho-
ton Info
       drawPhoton(rgb, gPoint);               //Draw Photon
       shadowPhoton(ray);                     //Shadow Photon
       ray = reflect(ray,prevPoint);          //Bounce the Photon
       raytrace(ray, gPoint);               //Trace It to Next Location
       prevPoint = gPoint;
       bounces++;}
  }
}

void storePhoton(int type, int id, float[ ] location, float[ ] direc-
tion, float[ ] energy){
  photons[type][id][numPhotons[type][id]][0] = location;  //Loca-
tion
  photons[type][id][numPhotons[type][id]][1] = direction; //Direc-
tion
  photons[type][id][numPhotons[type][id]][2] = energy;    //Atte-
nuated Energy (Color)
  numPhotons[type][id]++;
}

void shadowPhoton(float[ ] ray){                          //Shadow
Photons
  float[ ] shadow = {-0.25,-0.25,-0.25};
  float[ ] tPoint = gPoint;
  int tType = gType, tIndex = gIndex;                     //Save
State
  float[ ] bumpedPoint = add3(gPoint,mul3c(ray,0.00001));
//Start Just Beyond Last Intersection
  raytrace(ray, bumpedPoint);                             //Trace to
Next Intersection (In Shadow)
  float[ ] shadowPoint = add3( mul3c(ray,gDist), bumpedPoint);
//3D Point
  storePhoton(gType, gIndex, shadowPoint, ray, shadow);
  gPoint = tPoint; gType = tType; gIndex = tIndex;          //Res-
tore State
}

float[ ] filterColor(float[ ] rgbIn, float r, float g, float b){ //e.g.
White Light Hits Red Wall
  float[ ] rgbOut = {r,g,b};
  for (int c=0; c<3; c++) rgbOut[c] = min(rgbOut[c],rgbIn[c]);
//Absorb Some Wavelengths (R,G,B)
  return rgbOut;
}

float[ ] getColor(float[ ] rgbIn, int type, int index){ //Specifies
Material Color of Each Object
  if     (type == 1 && index == 0) { return filterColor(rgbIn, 0.0,
1.0, 0.0);}
  else if (type == 1 && index == 2) { return filterColor(rgbIn, 1.0,
0.0, 0.0);}
  else                        { return filterColor(rgbIn, 1.0, 1.0, 1.0);}
}

//----------------------------------------------------------------------
//Vector Operations ---------------------------------------------------
//----------------------------------------------------------------------

float[ ] normalize3(float[ ] v){       //Normalize 3-Vector
  float L = sqrt(dot3(v,v));
  return mul3c(v, 1.0/L);
}

float[ ] sub3(float[ ] a, float[ ] b){   //Subtract 3-Vectors
  float[ ] result = {a[0] - b[0], a[1] - b[1], a[2] - b[2]};
  return result;
}

float[ ] add3(float[ ] a, float[ ] b){   //Add 3-Vectors
  float[ ] result = {a[0] + b[0], a[1] + b[1], a[2] + b[2]};
  return result;
}

float[ ] mul3c(float[ ] a, float c){    //Multiply 3-Vector with Scalar
  float[ ] result = {c*a[0], c*a[1], c*a[2]};
  return result;
}

float dot3(float[ ] a, float[ ] b){     //Dot Product 3-Vectors
  return a[0] * b[0] + a[1] * b[1] + a[2] * b[2];
}

float[ ] rand3(float s){              //Random 3-Vector
  float[ ] rand = {random(-s,s),random(-s,s),random(-s,s)};
  return rand;
```

```
}

boolean gatedSqDist3(float{ } a, float{ } b, float sqradius){
//Gated Squared Distance
  float c = a[0] - b[0];    //Efficient When Determining if Thou-
sands of Points
  float d = c*c;   //Are Within a Radius of a Point (and Most Are
Not!)
  if (d > sqradius) return false; //Gate 1 - If this dimension alone
is larger than
  c = a[1] - b[1];          //  the search radius, no need to continue
  d += c*c;
  if (d > sqradius) return false; //Gate 2
  c = a[2] - b[2];
  d += c*c;
  if (d > sqradius) return false; //Gate 3
  gSqDist = d;      return true ; //Store Squared Distance Itself in
Global State
}

//----------------------------------------------------------------------
// User Interaction and Display ----------------------------------
//----------------------------------------------------------------------
boolean empty = true, view3D = false; //Stop Drawing, Switch
Views
PFont font; PImage img1, img2, img3;  //Fonts, Images
int pRow, pCol, pIteration, pMax;     //Pixel Rendering Order
boolean odd(int x) {return x % 2 != 0;}

void setup(){
  size(szImg,szImg + 48, JAVA2D);
  frameRate(9999);
  font = loadFont("Helvetica-Bold-12.vlw");
  emitPhotons();
  resetRender();
  drawInterface();
}

void draw(){
  if (view3D){
    if (empty){
      stroke(0); fill(0); rect(0,0,szImg-1,szImg-1); //Black Out Dra-
wing Area
      emitPhotons(); empty = false; frameRate(10);}} //Emit &
Draw Photons
  else{
    if (empty) render(); else frameRate(10);} //Only Draw if Image
Not Fully Rendered
}

void drawInterface() {
  stroke(221,221,204); fill(221,221,204); rect(0,szImg,szImg,48);
//Fill Background with Page Color
  img1=loadImage("1_32.png"); img2=loadImage("2_32.png");
img3=loadImage("3_32.png"); //Load Images

  textFont(font); //Display Text
  if (!view3D) {fill(0); img3.filter(GRAY);} else fill(160);
  text("Ray Tracing", 64, szImg + 28);
  if (lightPhotons || view3D) {fill(0); img1.filter(GRAY);} else
fill(160);
  text("Photon Mapping", 368, szImg + 28);
  if (!lightPhotons || view3D) img2.filter(GRAY);

  stroke(0); fill(255);  //Draw Buttons with Icons
  rect(198,519,33,33); image(img1,199,520);
  rect(240,519,33,33); image(img2,241,520);
  rect(282,519,33,33); image(img3,283,520);
}

void render(){ //Render Several Lines of Pixels at Once Before
Drawing
  int x,y,iterations = 0;
  float{ } rgb = {0.0,0.0,0.0};

  while (iterations < (mouseDragging ? 1024 : max(pMax, 256)
)){

    //Render Pixels Out of Order With Increasing Resolution:
2x2, 4x4, 16x16... 512x512
    if (pCol >= pMax) {pRow++; pCol = 0;
      if (pRow >= pMax) {pIteration++; pRow = 0; pMax =
int(pow(2,pIteration));}}
    boolean pNeedsDrawing = (pIteration == 1 || odd(pRow) ||
(!odd(pRow) && odd(pCol)));
    x = pCol * (szImg/pMax); y = pRow * (szImg/pMax);
    pCol++;

    if (pNeedsDrawing){
      iterations++;
      rgb = mul3c( computePixelColor(x,y), 255.0);              //All
the Magic Happens in Here!
      stroke(rgb[0],rgb[1],rgb[2]); fill(rgb[0],rgb[1],rgb[2]);  //Stroke
& Fill
      rect(x,y,(szImg/pMax)-1,(szImg/pMax)-1);}                 //Draw
the Possibly Enlarged Pixel
  }
  if (pRow == szImg-1) {empty = false;}
}

void resetRender(){ //Reset Rendering Variables
  pRow=0; pCol=0; pIteration=1; pMax=2; frameRate(9999);
  empty=true; if (lightPhotons && !view3D) emitPhotons();}

void drawPhoton(float{ } rgb, float{ } p){          //Photon Visuali-
zation
  if (view3D && p[2] > 0.0){                    //Only Draw if In Front
of Camera
    int x = (szImg/2) + (int)(szImg *  p[0]/p[2]); //Project 3D Points
into Scene
    int y = (szImg/2) + (int)(szImg * -p[1]/p[2]); //Don't Draw Outs-
ide Image
    if (y <= szImg) {stroke(255.0*rgb[0],255.0*rgb[1],255.0*rgb[2]);
point(x,y);}}
}

//----------------------------------------------------------------------
//Mouse and Keyboard Interaction ------------------------------
//----------------------------------------------------------------------
int prevMouseX = -9999, prevMouseY = -9999, sphereIndex = -
1;
float s = 130.0; //Arbitary Constant Through Experimentation
boolean mouseDragging = false;
void mouseReleased() {prevMouseX = -9999; prevMouseY = -
9999; mouseDragging = false;}
void keyPressed() {switchToMode(key,9999);}

void mousePressed(){
  sphereIndex = 2; //Click Spheres
  float{ } mouse3 = {(mouseX - szImg/2)/s, -(mouseY -
szImg/2)/s, 0.5*(spheres[0][2] + spheres[1][2])};
  if (gatedSqDist3(mouse3,spheres[0],spheres[0][3])) sphereIn-
dex = 0;
  else if (gatedSqDist3(mouse3,spheres[1],spheres[1][3])) spher-
eIndex = 1;
  if (mouseY > szImg) switchToMode('0', mouseX); //Click But-
tons
}

void mouseDragged(){
  if (prevMouseX > -9999 && sphereIndex > -1){
    if (sphereIndex < nrObjects[0]){ //Drag Sphere
      spheres[sphereIndex][0] += (mouseX - prevMouseX)/s;
      spheres[sphereIndex][1] -= (mouseY - prevMouseY)/s;}
```

```
    else{ //Drag Light
      Light[0] += (mouseX - prevMouseX)/s; Light[0] =
constrain(Light[0],-1.4,1.4);
      Light[1] -= (mouseY - prevMouseY)/s; Light[1] =
constrain(Light[1],-0.4,1.2);}
    resetRender();}
 prevMouseX = mouseX; prevMouseY = mouseY; mouseDrag-
ging = true;
}

void switchToMode(char i, int x){ // Switch Between Raytracing,
Photon Mapping Views
  if      (i=='1' || x<230) {view3D = false; lightPhotons = false;
resetRender(); drawInterface();}
  else if (i=='2' || x<283) {view3D = false; lightPhotons = true;
resetRender(); drawInterface();}
  else if (i=='3' || x<513) {view3D = true; resetRender(); drawIn-
terface();}
}
```

Verkorte uitleg

Ray Tracing gebeurt met slechts 1 straal per pixel in de 3D scene. Daarmee is het mogelijk om een enkele reflectie van de spiegelende bol te maken. Een enkele schaduwstraal bepaald of het object door de lichtbron direct wordt aangestraald of niet. Voor het berekenen van de Photon Map worden 1000 photonen berekend, die elk 3x kunnen weerkaatsen vanuit de lichtbron. Voor elk berekend photon wordt ook het corresponderende schaduw photon berekend. Photon Mapping wordt geactiveerd door eenvoudig de licht volgorde voor Ray Tracing te vervangen door Photon verzameling. Er wordt geen kd-tree toegepast, de Photons zijn geïntegreerd in het vastgestelde gebied.
De Ray Tracing code (150 regels) en de Photon Mapper (75 regels) vormen de hoofdzaken in dit programma. Totaal zijn er ongeveer 275 programma regels waarbij de gebruikers interface

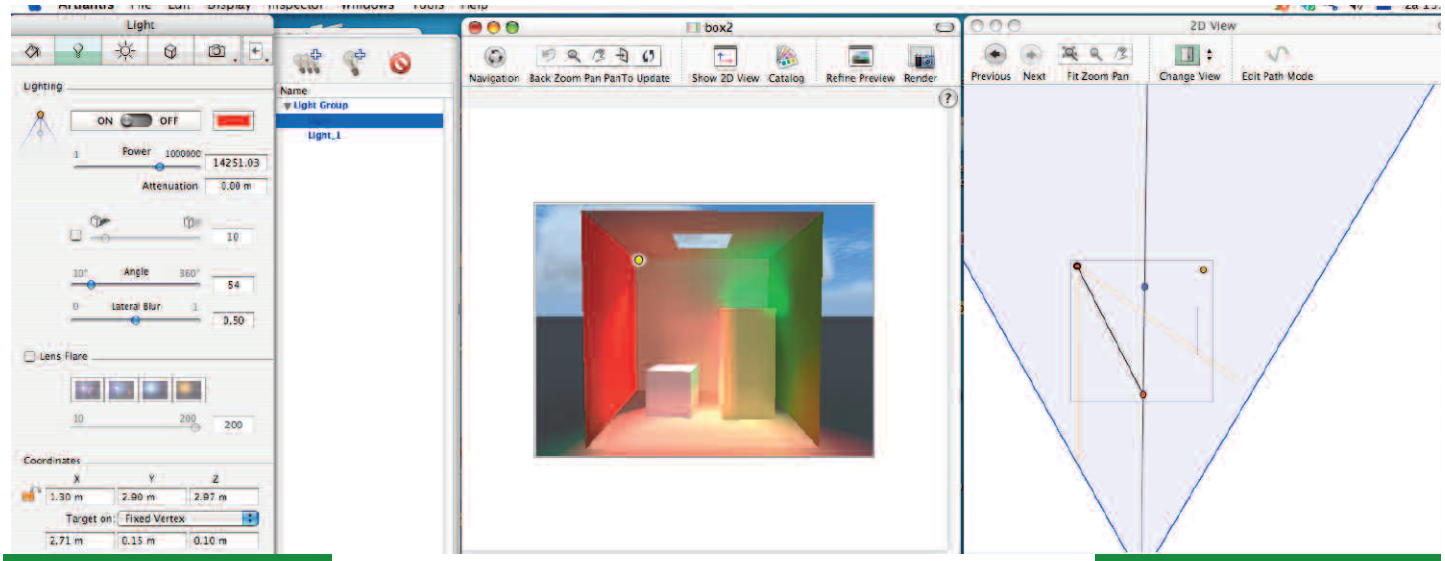


is opgenomen. Er worden geen externe libraries in Processing toegepast het programma is dus zo klaar voor gebruik. De complete en originele broncode is op de eerder genoemde website te downloaden (24 KB ZIP bestand). Omdat Processing in 2011 de regels heeft aangepast en nu van OpenGL gebruik maakt, kan het nodig zijn om enkele instructies te veranderen om het te laten werken.

Grant Schindler is Research Scientist, Georgia Institute of Technology, College of Computing. Zijn onderzoeks interesses liggen bij computer graphics en programmeren.
http://www.cc.gatech.edu/~phlosoft/

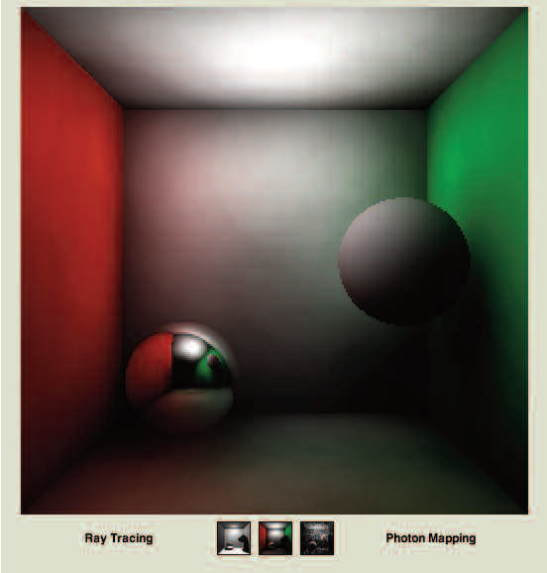
Op de volgende pagina's een uitleg van de diverse coderegels in dit programma.

Cornell Box in Artlantis Studio Render programma. In het midden de Preview, rechts corresponderende 2D View en links de diverse menu items.



Ray Tracing en Photon Mapping
met
Processing

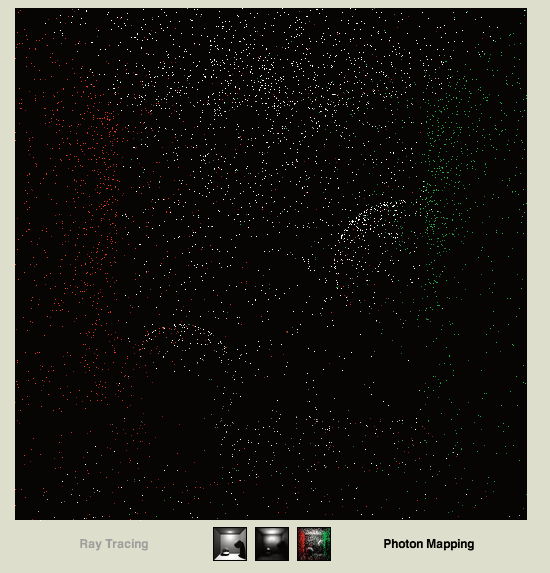
Verbetering van de natuurgetrouwheid met Ray Tracing kan worden bereikt door het simuleren van individuele Photons die van de lichtbron komen en in de 3d scene heen en weer worden gereflecteerd. De photonen in dit programma worden maximaal 3x gereflecteerd waardoor redelijk goede kleurweergave en zachte schaduwen (zie hieronder) worden verkregen. Bekijk wat er gebeurt met de schaduw indien u de bollen heen en weer beweegt. 1. Standaard Ray Tracing. 2. Ray Tracing met Photon Mapping. 3. Weergave van de Photon Map zelf. Gebruik de toetsen 1, 2 of 3 op het toetsenbord om de keuze te maken.



Boven: Ray Tracing.

Links het resultaat van Photon Mapping.

Rechts de Photon Map.




```
//Ray Tracing & Photon Mapping
//Grant Schindler, 2007, Processing - Java

// http://www.cc.gatech.edu/~phlosoft/photon/

1 // ----- Scene Description -----
int szImg = 512;                  //Image Size
int nrTypes = 2;                  //2 Object Types (Sphere = 0, Plane = 1)
int[] nrObjects = {2,5};          //2 Spheres, 5 Planes
float gAmbient = 0.1;             //Ambient Lighting
float[] gOrigin = {0.0,0.0,0.0};  //World Origin for Convenient Re-Use Below (Constant)
float[] Light = {0.0,1.2,3.75};   //Point Light-Source Position
float[][] spheres = {{1.0,0.0,4.0,0.5},{-0.6,-1.0,4.5,0.5}};         //Sphere Center & Radius
float[][] planes  = {{0, 1.5},{1, -1.5},{0, -1.5},{1, 1.5},{2,5.0}}; //Plane Axis & Distance-to-
Origin

2 // ----- Photon Mapping -----
int nrPhotons = 1000;             //Number of Photons Emitted
int nrBounces = 3;                //Number of Times Each Photon Bounces
boolean lightPhotons = true;      //Enable Photon Lighting?
float sqRadius = 0.7;             //Photon Integration Area (Squared for Efficiency)
```

```
float exposure = 50.0;            //Number of Photons Integrated at Brightest Pixel
int[][] numPhotons = {{0,0},{0,0,0,0,0}};              //Photon Count for Each Scene Object
float[][][][][] photons = new float[2][5][5000][3][3]; //Allocated Memory for Per-Object
Photon Info

// ----- Raytracing Globals -----
boolean gIntersect = false;       //For Latest Raytracing Call... Was Anything Intersected
by the Ray?
int gType;                        //... Type of the Intersected Object (Sphere or Plane)
int gIndex;                       //... Index of the Intersected Object (Which Sphere/Plane
Was It?)
float gSqDist, gDist = -1.0;      //... Distance from Ray Origin to Intersection
float[] gPoint = {0.0, 0.0, 0.0}; //... Point At Which the Ray Intersected the Object

//-------------------------------------------------------------------------------------
//Ray-Geometry Intersections  ----------------------------------------------------------
//-------------------------------------------------------------------------------------

void raySphere(int idx, float[] r, float[] o) //Ray-Sphere Intersection: r=Ray Direction,
o=Ray Origin
{
  float[] s = sub3(spheres[idx],o);  //s=Sphere Center Translated into Coordinate Frame of
Ray Origin
  float radius = spheres[idx][3];    //radius=Sphere Radius

  //Intersection of Sphere and Line     =       Quadratic Function of Distance
  float A = dot3(r,r);                       // Remember This From High School? :
  float B = -2.0 * dot3(s,r);                //    A x^2 +     B x +               C  = 0
  float C = dot3(s,s) - sq(radius);          // (r'r)x^2 - (2s'r)x + (s's - radius^2) = 0
  float D = B*B - 4*A*C;                     // Precompute Discriminant

  if (D > 0.0){                  //Solution Exists only if sqrt(D) is Real (not Imaginary)
    float sign = (C < -0.00001) ? 1 : -1;    //Ray Originates Inside Sphere If C < 0
    float lDist = (-B + sign*sqrt(D))/(2*A); //Solve Quadratic Equat. Distance Intersect.
    checkDistance(lDist,0,idx);}             //Is This Closest Intersection So Far?
}

void rayPlane(int idx, float[] r, float[] o){ //Ray-Plane Intersection
  int axis = (int) planes[idx][0];            //Determine Orientation of Axis-Aligned Plane
  if (r[axis] != 0.0){                        //Parallel Ray -> No Intersection
    float lDist = (planes[idx][1] - o[axis]) / r[axis]; //Solve Linear Equation (rx = p-o)
    checkDistance(lDist,1,idx);}
}

void rayObject(int type, int idx, float[] r, float[] o){
  if (type == 0) raySphere(idx,r,o); else rayPlane(idx,r,o);
}

void checkDistance(float lDist, int p, int i){
  if (lDist < gDist && lDist > 0.0){
                                  //Closest Intersection So Far in Forward DirectionRay?
    gType = p; gIndex = i; gDist = lDist; gIntersect = true;}
                                                  //Save Intersection in Global State
}

//-------------------------------------------------------------------------------------
// Lighting ----------------------------------------------------------------------------
//-------------------------------------------------------------------------------------

float lightDiffuse(float[] N, float[] P){
                                   //Diffuse Lighting at Point P with Surface Normal N
  float[] L = normalize3( sub3(Light,P) );     //Light Vector (Point to Light)
  return dot3(N,L);                  //Dot Product = cos (Light-to-Surface-Normal Angle)
}

float[] sphereNormal(int idx, float[] P){
  return normalize3(sub3(P,spheres[idx])); //Surface Normal (Center to Point)
}

float[] planeNormal(int idx, float[] P, float[] O){
  int axis = (int) planes[idx][0];
  float [] N = {0.0,0.0,0.0};
```

```
  N[axis] = O[axis] - planes[idx][1];      //Vector From Surface to Light
  return normalize3(N);
}

float[] surfaceNormal(int type, int index, float[] P, float[] Inside){
  if (type == 0) {return sphereNormal(index,P);}
  else           {return planeNormal(index,P,Inside);}
}

float lightObject(int type, int idx, float[] P, float lightAmbient){
  float i = lightDiffuse( surfaceNormal(type, idx, P, Light) , P );
  return min(1.0, max(i, lightAmbient));   //Add in Ambient Light by Constraining Min Value
}

//-------------------------------------------------------------------------------
// Raytracing -------------------------------------------------------------------
//-------------------------------------------------------------------------------

void raytrace(float[] ray, float[] origin)
{
  gIntersect = false; //No Intersections Along This Ray Yet
  gDist = 999999.9;   //Maximum Distance to Any Object

  for (int t = 0; t < nrTypes; t++)
    for (int i = 0; i < nrObjects[t]; i++)
      rayObject(t,i,ray,origin);
}

float[] computePixelColor(float x, float y){
  float[] rgb = {0.0,0.0,0.0};
  float[] ray = {  x/szImg - 0.5 ,        //Convert Pixels to Image Plane Coordinates
                -(y/szImg - 0.5), 1.0}; //Focal Length = 1.0
  raytrace(ray, gOrigin);           //Raytrace!!! - Intersected Objects are Stored in Global State

  if (gIntersect){                        //Intersection
    gPoint = mul3c(ray,gDist);            //3D Point of Intersection

    if (gType == 0 && gIndex == 1){       //Mirror Surface on This Specific Object
      ray = reflect(ray,gOrigin);         //Reflect Ray Off the Surface
      raytrace(ray, gPoint);              //Follow the Reflected Ray
      if (gIntersect){ gPoint = add3( mul3c(ray,gDist), gPoint); }} //3D Point of Intersection

    if (lightPhotons){                    //Lighting via Photon Mapping
      rgb = gatherPhotons(gPoint,gType,gIndex);}

    else{                           //Lighting via Standard Illumination Model (Diffuse + Ambient)
      int tType = gType, tIndex = gIndex;//Remember Intersected Object
      float i = gAmbient;               //If in Shadow, Use Ambient Color of Original Object
      raytrace( sub3(gPoint,Light) , Light);  //Raytrace from Light to Object
      if (tType == gType && tIndex == gIndex) //Ray from Light->Object Hits Object First?
        i = lightObject(gType, gIndex, gPoint, gAmbient); //Not In Shadow - Compute Lighting
      rgb[0]=i; rgb[1]=i; rgb[2]=i;
      rgb = getColor(rgb,tType,tIndex);}
  }
  return rgb;
}

float[] reflect(float[] ray, float[] fromPoint){                //Reflect Ray
  float[] N = surfaceNormal(gType, gIndex, gPoint, fromPoint);  //Surface Normal
  return normalize3(sub3(ray, mul3c(N,(2 * dot3(ray,N)))));     //Approximation to Reflection
}

//-------------------------------------------------------------------------------
//Photon Mapping ---------------------------------------------------------------
//-------------------------------------------------------------------------------

float[] gatherPhotons(float[] p, int type, int id){
  float[] energy = {0.0,0.0,0.0};
  float[] N = surfaceNormal(type, id, p, gOrigin);              //Surface Normal at Current Point
  for (int i = 0; i < numPhotons[type][id]; i++){               //Photons Which Hit Current Object
    if (gatedSqDist3(p,photons[type][id][i][0],sqRadius)){      //Is Photon Close to Point?
      float weight = max(0.0, -dot3(N, photons[type][id][i][1] )); //Single Photon Diffuse Lighting
```

```
      weight *= (1.0 - sqrt(gSqDist)) / exposure;              //Weight by Photon-Point Distance
      energy = add3(energy, mul3c(photons[type][id][i][2], weight)); //Add Photon's Energy to Total
   }}
  return energy;
}

22 void emitPhotons(){
  randomSeed(0);                                  //Ensure Same Photons Each Time
  for (int t = 0; t < nrTypes; t++)               //Initialize Photon Count to Zero for Each Object
    for (int i = 0; i < nrObjects[t]; i++)
      numPhotons[t][i] = 0;

23  for (int i = 0; i < (view3D ? nrPhotons * 3.0 : nrPhotons); i++){
                                                  //Draw 3x Photons For Usability
    int bounces = 1;
    float[] rgb = {1.0,1.0,1.0};                  //Initial Photon Color is White
    float[] ray = normalize3( rand3(1.0) );       //Randomize Direction of Photon Emission
    float[] prevPoint = Light;                    //Emit From Point Light Source

24 //Spread Out Light Source, But Don't Allow Photons Outside Room/Inside Sphere
// while (prevPoint[1] >= Light[1]){ prevPoint = add3(Light, mul3c(normalize3(rand3(1.0)), 0.75));}

    if (abs(prevPoint[0]) > 1.5 || abs(prevPoint[1]) > 1.2 ||
        gatedSqDist3(prevPoint,spheres[0],spheres[0][3]*spheres[0][3])) bounces = nrBounces+1;

25    raytrace(ray, prevPoint);                                 //Trace the Photon's Path

    while (gIntersect && bounces <= nrBounces){               //Intersection With New Object
        gPoint = add3( mul3c(ray,gDist), prevPoint);          //3D Point of Intersection
        rgb = mul3c (getColor(rgb,gType,gIndex), 1.0/sqrt(bounces));
        storePhoton(gType, gIndex, gPoint, ray, rgb);         //Store Photon Info
        drawPhoton(rgb, gPoint);                              //Draw Photon
        shadowPhoton(ray);                                    //Shadow Photon
        ray = reflect(ray,prevPoint);                         //Bounce the Photon
        raytrace(ray, gPoint);                                //Trace It to Next Location
        prevPoint = gPoint;
        bounces++;}
  }
}

26 void storePhoton(int type, int id, float[] location, float[] direction, float[] energy){
  photons[type][id][numPhotons[type][id]][0] = location;  //Location
  photons[type][id][numPhotons[type][id]][1] = direction; //Direction
  photons[type][id][numPhotons[type][id]][2] = energy;    //Attenuated Energy (Color)
  numPhotons[type][id]++;
}

27 void shadowPhoton(float[] ray){                          //Shadow Photons
  float[] shadow = {-0.25,-0.25,-0.25};
  float[] tPoint = gPoint;
  int tType = gType, tIndex = gIndex;                      //Save State
  float[] bumpedPoint = add3(gPoint,mul3c(ray,0.00001));   //Start Just Beyond Last Intersection
  raytrace(ray, bumpedPoint);                              //Trace to Next Intersection (In Shadow)
  float[] shadowPoint = add3( mul3c(ray,gDist), bumpedPoint); //3D Point
  storePhoton(gType, gIndex, shadowPoint, ray, shadow);
  gPoint = tPoint; gType = tType; gIndex = tIndex;            //Restore State
}

float[] filterColor(float[] rgbIn, float r, float g, float b){ //e.g. White Light Hits Red Wall
  float[] rgbOut = {r,g,b};
  for (int c=0; c<3; c++) rgbOut[c] = min(rgbOut[c],rgbIn[c]); //Absorb Some Wavelengths (R,G,B)
  return rgbOut;
}

28 float[] getColor(float[] rgbIn, int type, int index){ //Specifies Material Color of Each Object
  if      (type == 1 && index == 0) { return filterColor(rgbIn, 0.0, 1.0, 0.0);}
  else if (type == 1 && index == 2) { return filterColor(rgbIn, 1.0, 0.0, 0.0);}
  else                              { return filterColor(rgbIn, 1.0, 1.0, 1.0);}
}
```

```
//------------------------------------------------------------------------------------------
//Vector Operations -------------------------------------------------------------------------
//------------------------------------------------------------------------------------------

float[] normalize3(float[] v){        //Normalize 3-Vector
  float L = sqrt(dot3(v,v));
  return mul3c(v, 1.0/L);
}

float[] sub3(float[] a, float[] b){   //Subtract 3-Vectors
  float[] result = {a[0] - b[0], a[1] - b[1], a[2] - b[2]};
  return result;
}

float[] add3(float[] a, float[] b){   //Add 3-Vectors
  float[] result = {a[0] + b[0], a[1] + b[1], a[2] + b[2]};
  return result;
}

float[] mul3c(float[] a, float c){    //Multiply 3-Vector with Scalar
  float[] result = {c*a[0], c*a[1], c*a[2]};
  return result;
}

float dot3(float[] a, float[] b){     //Dot Product 3-Vectors
  return a[0] * b[0] + a[1] * b[1] + a[2] * b[2];
}

float[] rand3(float s){               //Random 3-Vector
  float[] rand = {random(-s,s),random(-s,s),random(-s,s)};
  return rand;
}

boolean gatedSqDist3(float[] a, float[] b, float sqradius){ //Gated Squared Distance
  float c = a[0] - b[0];               //Efficient When Determining if Thousands of Points
  float d = c*c;                       //Are Within a Radius of a Point (and Most Are Not!)
  if (d > sqradius) return false;     //Gate 1 - If this dimension alone is larger than
  c = a[1] - b[1];                    //         the search radius, no need to continue
  d += c*c;
  if (d > sqradius) return false;      //Gate 2
  c = a[2] - b[2];
  d += c*c;
  if (d > sqradius) return false;      //Gate 3
  gSqDist = d;      return true ;      //Store Squared Distance Itself in Global State
}

//------------------------------------------------------------------------------------------
// User Interaction and Display -------------------------------------------------------------
//------------------------------------------------------------------------------------------
boolean empty = true, view3D = false;    //Stop Drawing, Switch Views
PFont font; PImage img1, img2, img3;     //Fonts, Images
int pRow, pCol, pIteration, pMax;        //Pixel Rendering Order
boolean odd(int x) {return x % 2 != 0;}

void setup(){
  size(szImg,szImg + 48, JAVA2D);
  frameRate(9999);
  font = loadFont("Helvetica-Bold-12.vlw");
  emitPhotons();
  resetRender();
  drawInterface();
}

void draw(){
  if (view3D){
    if (empty){
      stroke(0); fill(0); rect(0,0,szImg-1,szImg-1);      //Black Out Drawing Area
      emitPhotons(); empty = false; frameRate(10);}}      //Emit & Draw Photons
  else{
    if (empty) render(); else frameRate(10);}        //Only Draw if Image Not Fully Rendered
}
```

```
void drawInterface() {
  stroke(221,221,204); fill(221,221,204); rect(0,szImg,szImg,48); //Fill Background with Page Color
  img1=loadImage("1_32.png"); img2=loadImage("2_32.png"); img3=loadImage("3_32.png");
                                                                          //Load Images
33  textFont(font);             //Display Text
  if (!view3D) {fill(0); img3.filter(GRAY);} else fill(160);
  text("Ray Tracing", 64, szImg + 28);
  if (lightPhotons || view3D) {fill(0); img1.filter(GRAY);} else fill(160);
  text("Photon Mapping", 368, szImg + 28);
  if (!lightPhotons || view3D) img2.filter(GRAY);

  stroke(0); fill(255);  //Draw Buttons with Icons
  rect(198,519,33,33); image(img1,199,520);
  rect(240,519,33,33); image(img2,241,520);
  rect(282,519,33,33); image(img3,283,520);
}

34 void render(){ //Render Several Lines of Pixels at Once Before Drawing
  int x,y,iterations = 0;
  float[] rgb = {0.0,0.0,0.0};

  while (iterations < (mouseDragging ? 1024 : max(pMax, 256) )){

35  //Render Pixels Out of Order With Increasing Resolution: 2x2, 4x4, 16x16... 512x512
    if (pCol >= pMax) {pRow++; pCol = 0;
      if (pRow >= pMax) {pIteration++; pRow = 0; pMax = int(pow(2,pIteration));}}
    boolean pNeedsDrawing = (pIteration == 1 || odd(pRow) || (!odd(pRow) && odd(pCol)));
    x = pCol * (szImg/pMax); y = pRow * (szImg/pMax);
    pCol++;

    if (pNeedsDrawing){
      iterations++;
      rgb = mul3c( computePixelColor(x,y), 255.0);              //All the Magic Happens in Here!
      stroke(rgb[0],rgb[1],rgb[2]); fill(rgb[0],rgb[1],rgb[2]);  //Stroke & Fill
      rect(x,y,(szImg/pMax)-1,(szImg/pMax)-1);}                  //Draw the Possibly Enlarged Pixel
  }
  if (pRow == szImg-1) {empty = false;}
}

36 void resetRender(){ //Reset Rendering Variables
  pRow=0; pCol=0; pIteration=1; pMax=2; frameRate(9999);
  empty=true; if (lightPhotons && !view3D) emitPhotons();}

37 void drawPhoton(float[] rgb, float[] p){          //Photon Visualization
  if (view3D && p[2] > 0.0){                        //Only Draw if In Front of Camera
    int x = (szImg/2) + (int)(szImg *  p[0]/p[2]); //Project 3D Points into Scene
    int y = (szImg/2) + (int)(szImg * -p[1]/p[2]); //Don't Draw Outside Image
    if (y <= szImg) {stroke(255.0*rgb[0],255.0*rgb[1],255.0*rgb[2]); point(x,y);}}
}

//-------------------------------------------------------------------------------------
38 //Mouse and Keyboard Interaction ----------------------------------------------------
//-------------------------------------------------------------------------------------
int prevMouseX = -9999, prevMouseY = -9999, sphereIndex = -1;
float s = 130.0; //Arbitary Constant Through Experimentation
boolean mouseDragging = false;
void mouseReleased() {prevMouseX = -9999; prevMouseY = -9999; mouseDragging = false;}
void keyPressed() {switchToMode(key,9999);}

void mousePressed(){
  sphereIndex = 2; //Click Spheres
  float[] mouse3 = {(mouseX - szImg/2)/s, -(mouseY - szImg/2)/s, 0.5*(spheres[0][2] + sphe-
res[1][2])};
  if (gatedSqDist3(mouse3,spheres[0],spheres[0][3])) sphereIndex = 0;
  else if (gatedSqDist3(mouse3,spheres[1],spheres[1][3])) sphereIndex = 1;
  if (mouseY > szImg) switchToMode('0', mouseX); //Click Buttons
}

void mouseDragged(){
  if (prevMouseX > -9999 && sphereIndex > -1){
    if (sphereIndex < nrObjects[0]){ //Drag Sphere
```

```
      spheres[sphereIndex][0] += (mouseX - prevMouseX)/s;
      spheres[sphereIndex][1] -= (mouseY - prevMouseY)/s;}
    else{ //Drag Light
      Light[0] += (mouseX - prevMouseX)/s; Light[0] = constrain(Light[0],-1.4,1.4);
      Light[1] -= (mouseY - prevMouseY)/s; Light[1] = constrain(Light[1],-0.4,1.2);}
    resetRender();}
 prevMouseX = mouseX; prevMouseY = mouseY; mouseDragging = true;
}

39 void switchToMode(char i, int x){ // Switch Between Raytracing, Photon Mapping Views
  if      (i=='1' || x<230) {view3D = false; lightPhotons = false; resetRender(); drawInterface();}
  else if (i=='2' || x<283) {view3D = false; lightPhotons = true;  resetRender(); drawInterface();}
  else if (i=='3' || x<513) {view3D = true; resetRender(); drawInterface();}
}
```

Render programma in Processing

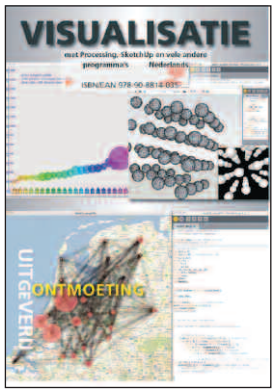
(open source)

Processing is een vereenvoudigde en gespecialiseerde vorm van Java met een eigen compiler en Preview venster. Gespecialiseerd op grafische toepassingen.

Wij gebruikten de inmiddels oude versie 1.5 op Mac OSX computer met 10.4.11. Uitgeverij Ontmoeting heeft daar een Nederlandstalige handleiding Visualisatie voor geschreven.
http://ontmoeting.nl/visualisatie-2.html

Inmiddels is het versie nummer verder opgeschoven naar 2 en wellicht zelfs hoger, maar wat belangrijker is, de bibliotheken die het fundament onder Processing zijn, hebben het onderspit moeten delven bij de laatste upgrade. Ook veel instructies zijn niet meer beschikbaar.
En dat is voor een Open Source programma een behoorlijk probleem. Met deze grote wijziging zijn ook de honderden interessante programma's van de site verwijderd.
Er zijn ook voordelen: OpenGL ondersteuning en diverse nog in aanbouw zijnde nieuwe commando's die het programma geschikt moeten maken om met z'n tijd mee te gaan.

Hieruit blijkt dat ook bij een Open Source programma zoals Processing, er op een gegeven moment drastische maatregelen worden genomen om beter aan te sluiten bij nieuwe ontwikkelingen. Net zoals bij Mac OSX en in mindere mate met MS Windows is het niet gezegd dat programma's van oudere versies nog werken in nieuwe operating systemen en omgekeerd.



Render programma beschrijving

Een korte impressie van de hoofdpunten uit het renderprogramma zijn:

1.
Scene Description, beschrijving van de standaard instellingen voor aantal objecten, omgevingslicht, uitgangspunt (oorsprong) definitie, startpunt in 3D van de lichtbron, plaatsing van de twee bollen met hun straal in 3D en ten slote de gebruikte oppervlakken die de kamermuren voorstellen.
2.
Photon Mapping, voor deze instelling van renderen de standaard gegevens. Het aantal Photons die worden berekend kan van 1000 wel iets worden verhoogd, maar veel verder gaat niet aangezien het programma bij 25. "photons[type][id] etc. vast loopt. Bij het aantal weerkaatsingen kan deze van 3, 2 naar 1 worden ingesteld, hoger dan 3 geeft dezelfde vastloper als hierboven.
Photon Integration Area (het kwadraat) geeft met float sqRadius = 0.7 de grootte aan, de werkelijke waarde is de wortel uit 0.7 = 0.83660026.
Op de volgende pagina drie schermafbeeldingen vanuit het programma met een wijziging van

deze sqRadius van 0.7 naar 4. Bij hogere getallen verschuiven de kleuren nog verder.

3.

Raytracing Globals, **int gType** geeft aan of het een bol of een oppervlak (wand, vloer of plafond) betreft. In dit programma is dat een simpele manier van werken. In professionele programma's is dat niet mogelijk en worden per straal een aantal keuzes afgewerkt. De instelling gDist = - 1.0 lijkt de minimale afstand voor rendering, maar is het niet, wijzigingen naar + 5000 leveren geen verschil in de Bounding box op van de rendering. De **gPoint** met z'n drie coördinaten is een belangrijk onderdeel van het hele renderingsprogramma, hier worden de x, y, en z-waarde van het treftpunt met het object opgeslagen.

4.

Ray-Geometry Intersections
Ook void raySphere geeft het trefpunt aan van bol en straal met r = straal richting en o de oorsprong van de straal.

5.

De gebruikte formule voor het bepalen van de juiste afstand van de straal tot het te treffen object is al eerder beschreven, het komt neer op de vierkantsvergelijking (abc formule) oplossen.

6.

Waarbij de Discriminant D bepaald of het een geschikte wortel is die gebruikt kan worden, of dat er geen oplossing is. Met
checkDistance(Dist,0,idx); wordt bekeken of het de dichtstbijzijnde trefpunt tot nu toe is.

7.

Indien er een trefpunt met een oppervlak van een van de wanden is. Er plaatsing van het vlak wordt berekend en als de r[axis] ongelijk 0 is dan loopt er een parallele straal met het gevolg geen trefpunt.

8.

Controle of het een bol of een wand oppervlak betreft.

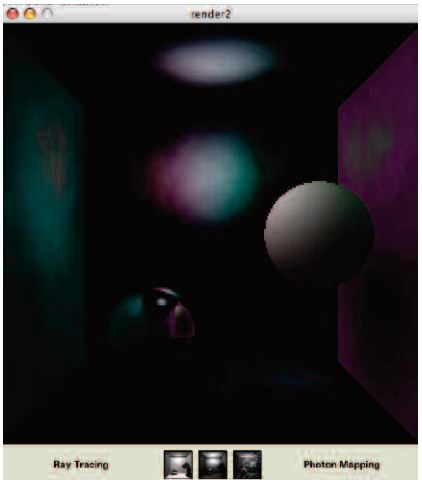
9.

Is het de dichtstbijzijnde trefpunt tot nu toe in voorwaartse richting? Als het antwoord ja is dan wordt het trefpunt opgeslagen in gType, gIndex en gDist

10.

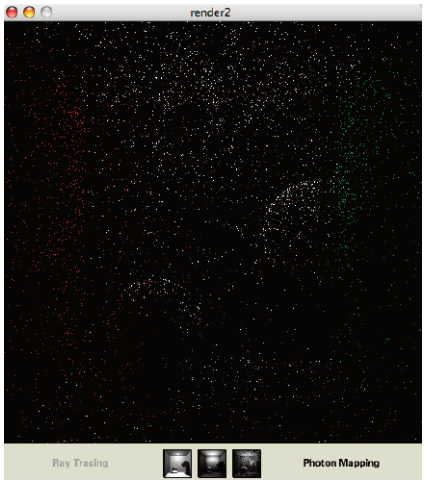
Lighting, de diverse instellingen voor de enige lichtbron op punt P in het programma.

sqRadius 4.



sqRadius 0.7

sqRadius 0.7 samples van de photon map zelf.



11.

Normalize3 is in de nieuwe versie vervangen door de instructie **norm ()**
Er is ook nog een instructie **normal ()** deze is voor het berekenen van de Normaal van een oppervlak (beiden van versie 2.0 Processing).
De laatste instructie return dot3(N,L);
voert de ook al eerder beschreven dot instructie uit. In vers. 2.0 van Processing is deze nog niet aanwezig bij de referenties maar wel bij de PVector Class, maar zal ongetwijfeld worden toegevoegd.
Hier wordt het dotproduct tussen de N (normaal)

en de L (lamp) weergegeven.
De L en N waarden zijn daarboven al bepaald.

12.

SphereNormal geeft de Normaal richting aan van het middelpunt van de bol naar het trefpunt. Bij de volgende float[] planeNormal wordt hetzelfde voor een oppervlak van een van de wanden gedaan.

13.

Raytracing
raytrace (float [] ray, float [] origin) maakt gebruik van de straal coördinaten in drie dimensies en de coördinaten van de oorsprong.

14.

Als start wordt **gIntersect** op uit gezet.

De maximale afstand op een hoog getal 999999.9 om alle voorwerpen in de scene mee te kunnen nemen. Voor dit kleine voorbeeld kan dat uiteraard nog kleiner worden gekozen.

Voor het aantal **nrTypes** (aantal objecten) en **nrObjects (2, 5)**. Twee bollen en 5 oppervlakken van de wanden.
Het rayObject wordt zo samengesteld waarbij t het aantal objecten en i het soort objecten weergeeft.

rayObject (t, i, ray, origin);

Op de volgende pagina de schermafdruk met de uitkomsten van “ray”.

15.

Ook de kleur van het pixel (x, y) is een van de kerntaken van een renderingsprogramma. In eerste instantie wordt RGB als 0 (zwart) gedefinieerd. De gekozen brandpuntafstand is 1.0
De pixels worden omgezet naar beeld coördinaten met ***float [] ray***, waarbij de grootte van het gedefinieerde venster wordt aangehouden. (szimg)
Met de laatste instructie ***raytrace (ray, gOrigin);*** wordt de trefpunten met de objecten in de Global State (wereld) bewaard.
De origin staat op 0.

PVector Class

* add() -- add vectors
* sub() -- subtract vectors
* mult() -- scale the vector with multiplication
* div() -- scale the vector with division
* mag() -- calculate the magnitude of a vector
* normalize() -- normalize the vector to unit length of 1
* limit() -- limit the magnitude of a vector
* heading2D() -- the heading of a vector expressed as an angle
* dist() -- the euclidean distance between two vectors (considered as points)
* angleBetween() -- find the angle between two vectors
* dot() -- the dot product of two vectors
* cross() -- the cross product of two vectors

dot ( )

Twee driedimensionale vectoren kunnen met het dotproduct worden berekend.

voorbeeld I

```
PVector v = new PVector (10, 20, 0);
float d = v.dot (60, 80, 0);
println (d);
```

print op het scherm “ 2200.0”

Berekening van het dot product van twee vectorern (in het voorbeeld dus 10, 20, 0 en 60, 80, 0.

voorbeeld II

```
PVector v1 = new PVector (10, 20, 0);
PVector v2 = new PVector (60, 80, 0);
float d = v1.dot(v2);
println(d);
```

print op het scherm “ 2200.0”

In een Euclidische ruimte zal het Dot product van twee eenheidsvectoren simpelweg de cosinus zijn van de hoek tussen de vectoren. Dit volgt uit de formule van het Dot product, de lengtes van de vectoren zijn beiden 1.

normalize ( )

Een driedimensionale vector kan worden ‘genormaliseerd’ tot een lengte 1. Een genormaliseerde vector wordt vaak voorzien van een dakje boven de naam van de vector.
Het is de waarde gedeeld door de lengte van de vector (optelling van kwadraten en daar de wortel uit).

```
PVector v = new PVector (10, 20, 2);
v.normalize();
println (v);
```

print op het scherm "[0.4454354, 0.8908708, 0.089087084] “

Println (ray)

met rechts de uitvoer per soort object [0, 1 of 2]

16.
Als er een trefpunt is dan wordt dat opgeslagen in ***gPoint*** met zowel de 3 coördinaten als de afstand.

Indien gType 0 is (0 = Sphere) EN gIndex 1 is (Plane / wand) dan zal het antwoord true zijn en de daarop volgende instructies worden uitgevoerd.

17.
Laat de straal tegen de wand reflecteren en volg de nieuwe gereflecteerde straal op zijn weg in de 3D scene. Het punt waarbij dat gebeurt is ***gPoint***.

18.
Indien gekozen is voor "Lighting via Photon Mapping" dan wordt de RGB waarde bij de trefpunten en de types en index vast gelegd.

19.
Indien de straal van het licht naar het object deze het eerst treft? Dan zal deze niet in de schaduw liggen en het licht wordt berekend.

20.
float[] reflect (float [] ray, float [] fromPoint)
Bereken bij de reflectie de Normaal van het oppervlak en retourneer normalize3.

21.
Photon Mapping
Voor deze keuze met de p, int type en int id gegevens de energy uitrekenen.

22.
Void emitPhotons()
randomSeed (0)

```
randomSeed(0);
for (int i=0; i < 100; i++) {
  float r = random(0, 255);
  stroke(r);
  line(i, 0, i, 100);
}
```

Stel de seed waarde in om de random functie uit te voeren.
Standaard random() geeft elke keer dat programma werkt verschillende resultaten.
Met random (0) een constante, wordt telkens een zelfde pseudo aantal willekeurige getallen gegenereerd. Elk object wordt met nul Photons gestart.

23.
Teken 3 x de Photons om ze bruikbaar te maken. De startpositie voor de RGB kleur is wit. Ook de genormaliseerde ray wordt met random van richting gewijzigd.

24.
Photons mogen in principe alleen binnen de kamer blijven, andere lichtstralen uitrekenen heeft in wezen geen zin. Een behoorlijk ingewikkelde routine moet dat indammen.

25.
Raytrace, volg het pad van de photonen.

26.
Void storePhoton. Bij het verhogen van het aantal nrPhotons loopt het programma hier klem. Een reden zou kunnen zijn dat deze aantallen niet meer in de Array geplaatst kunnen worden om tijdelijk op te slaan.

27.
void shadowPhoton, de routine voor het berekenen van de schaduwpartijen.

28.
float [] filterColor, een kleurcorrectie filter ter correctie van de uiteindelijke rendering.

29.
Vector operations
De meeste vector bewerkingen worden hier uitgevoerd. Normaliseren, aftrekken van vectoren, optellen van vectoren, vermenigvuldigen met een Scalar,.

30.
User Interaction and Display

31.
void setup ()
Aan het einde van het programma komt in wezen de instelling zoals gebruikelijk bij Proces-

sing. De gebruikte bibliotheek JAVA2D, vroeger los in te voegen, tegenwoordig ingebouwd en het gebruikte font.

32.

void draw ()
Het teken (preview) gedeelte binnen Processing, alleen hier kunnen de routines naar het scherm worden gevoerd bij deze versie.
Indien de drie PNG afbeeldingen er niet bijzitten dan zal (view3D) false genereren en er wordt een zwart venster gemaakt zonder beeld.

33.

textFont (font)
De bijbehorende kleine teksten in het venster, waarbij de PNG iconen als drukknoppen worden gebruikt. Hier kan een keuze worden gemaakt tussen Ray Tracing standaard, Ray Tracing met Photon mapping, Photon map zelf.

34.

void render ()
Voor het tekenen eerst een aantal lijnen renderen. Iterations hebben een "for" structuur om herhaalde tekeningen uit te voeren.

35.

Render Pixels out of Order
renderen met telkens toenemende resoluties
2 x 2, 4 x 4, 16 x 16 en 512 x 512 ...
Volgens de auteur van het programma “All the Magic Happens in Here!
rgb = mul3c(computePixelColor (x, y), 255.0);
stroke and fill

36.

void resetRender ()
reset de render variabelen

37.

void drawPhoton

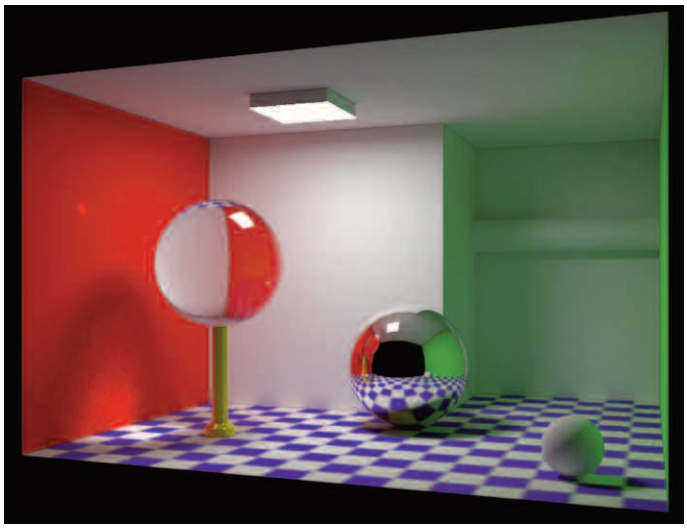
38.

Voi

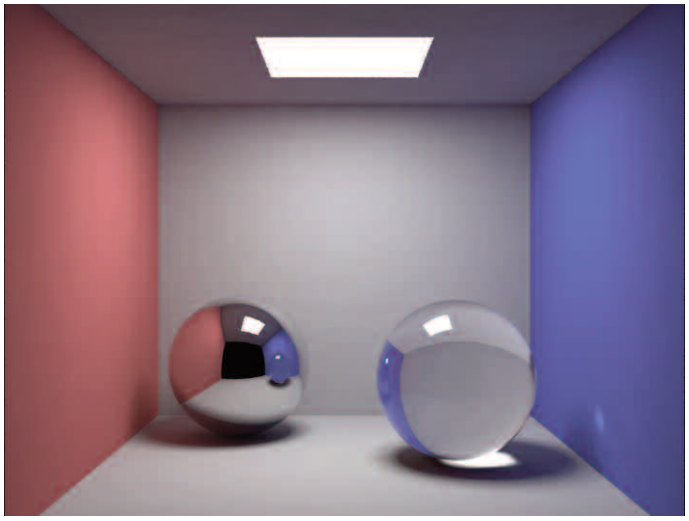
39.

Voi

Forum reactie op Ray Tracing en Photon Mapping programma
http://www.processing.org/discourse/beta/num_1201727156.html
Ray Tracing and Photon Mapping
30 januari 2008 10:05 van Grant
Hij krijgt lof toegezwaaid van andere forumleden



SunFlow, open source render programma



SunFlow, Cornell Box met caustics.

en bedankt ze. Hij zegt: “hou in het achterhoofd dat zowel de snelheid en de relatief korte code betekent dat het om een heel erg verkorte interpretatie van Photon mapping gaat. Er zijn geen caustics, geen sub-surface scattering en de scene is wel heel simpel.”

De auteur was zelf zeer verbaasd over de rendersnelheid en hij schrijft dat toe aan de kwaliteit van Processing. “Meestal doe ik dergelijke experimenten in CodeWarrior of XCode.
SunFlow heeft een veel uitgebreidere opzet en die is wel geschikt voor gebruik van allerlei polygonen.”

SunFlow

SunFlow is de Open Source Render Engine en is geheel in Java (vers. 1.5 of hoger) geschreven. De laatste aanvulling is al weer van april 2007.
SunFlow werkt met een cameralens met ondermeer de PinholeLens. Maar ook een Sferische-, Thin- en Fisheye lenzen zijn aanwezig.
http://sunflow.sourceforge.net/index.php?pg=gall
zie volgende pagina.

Java code intersection:

```
void intersection(sfvec3f rayStart,sfvec3f rayDirection,sfnode nearestNode,sfvec3f distance) {
        sfvec3f c = getSphereCentre().clone();
        c.sub(rayStart); // c=centre of sphere relative to start of ray
        double r = getSphereRadius(); // r=radius of sphere
        double distanceToSphere = Math::sqrt((c.x * c.x) + (c.y * c.y) + (c.z * c.z));
        if (r> distanceToSphere) { // the start point is already inside the sphere so the distance to it is zero
                nearestNode.setNode(this);
      distance.set(0.0);
                return;
        }
        double vx = rayDirection.x * distanceToSphere;
        double vy = rayDirection.y * distanceToSphere;
   double vz = rayDirection.z * distanceToSphere; // v = end point of ray when projected to sphere orbit
        double distanceFromEnd=Math::sqrt(((c.x - vx)*(c.x - vx)) + ((c.y - vy)*(c.y - vy)) + ((c.z - vz)*(c.z - vz)));
        if (distanceFromEnd > distanceToSphere) return; // the distance of the centre is greater than the dis-
tance to the beginning so its pointing away
        if (R < distanceFromEnd) return; // The distance of V from C is greater than R so its outside
        double d=distance.getValue();
        if ((d < 0.0) | (d > distanceToSphere - R)) {
        distance.set(distanceToSphere - R); // return distance to the nearest point
                nearestNode.setNode(this);
        }
}
```

C++ code intersection:

```
void sphereBean::intersection(sfvec3f* rayStart,sfvec3f* rayDirection,sfnode* nearestNode,sfvec3f* dis-
tance) {
        sfvec3f* c = getSphereCentre()->clone();
        c->sub(rayStart); // c=centre of sphere relative to start of ray
        double r = getSphereRadius(); // r=radius of sphere
        double distanceToSphere = Math::Sqrt((c->x * c->x) + (c->y * c->y) + (c->z * c->z));
        if (r> distanceToSphere) { // the start point is already inside the sphere so the distance to it is zero
                nearestNode->setNode(this);
      distance->set(0.0);
                return;
        }
        double vx = rayDirection->x * distanceToSphere;
        double vy = rayDirection->y * distanceToSphere;
   double vz = rayDirection->z * distanceToSphere; // v = end point of ray when projected to sphere orbit
        double distanceFromEnd=Math::Sqrt(((c->x - vx)*(c->x - vx)) + ((c->y - vy)*(c->y - vy)) + ((c->z -
vz)*(c->z - vz)));
        if (distanceFromEnd > distanceToSphere) return; // the distance of the centre is greater than the dis-
tance to the beginning so its pointing away
        if (R < distanceFromEnd) return; // The distance of V from C is greater than R so its outside
        double d=distance->getValue();
        if ((d < 0.0) | (d > distanceToSphere - R)) {
        distance->set(distanceToSphere - R); // return distance to the nearest point
                nearestNode->setNode(this);
        }
}
```

Deel uit programma "Photonsampler.cpp" van LuxRays
lux-d2eeecb44aec te downloaden van de site http://src.luxrender.net/

```
/***************************************************************************
 *   Copyright (C) 1998-2010 by authors (see AUTHORS.txt )                 *
 *                                                                         *
 *   This file is part of LuxRays.                                         *
 *                                                                         *
 *   LuxRays is free software; you can redistribute it and/or modify       *
 *   it under the terms of the GNU General Public License as published by  *
 *   the Free Software Foundation; either version 3 of the License, or     *
 *   (at your option) any later version.                                   *
 *                                                                         *
 *   LuxRays is distributed in the hope that it will be useful,            *
 *   but WITHOUT ANY WARRANTY; without even the implied warranty of        *
 *   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the         *
 *   GNU General Public License for more details.                          *
 *                                                                         *
 *   You should have received a copy of the GNU General Public License     *
 *   along with this program.  If not, see <http://www.gnu.org/licenses/>. *
 *                                                                         *
 *   LuxRays website: http://www.luxrender.net                             *
 ***************************************************************************/
```

init deel weggelaten, zie origineel

```
// Photon tracing
void PhotonSampler::AddFluxToHitPoint(const Sample *sample, const u_int lightGroup, HitPoint * const hp, const XYZColor flux)
{
	// TODO: it should be more something like:
	//XYZColor flux = XYZColor(sw, photonFlux * f) * XYZColor(hp->sample->swl, hp->eyeThroughput);
	hp->IncPhoton();

	sample->AddContribution(hp->imageX, hp->imageY,
		flux, hp->eyePass.alpha, hp->eyePass.distance,
		0, renderer->sppmi->bufferPhotonId, lightGroup);
};
//------------------------------------------------------------------------------
// Tracing photons for Photon Sampler
//------------------------------------------------------------------------------

void PhotonSampler::TracePhoton(
	Sample *sample,
	Distribution1D *lightCDF)
{
	Scene &scene = *renderer->scene;
	// I have to make a copy of SpectrumWavelengths because it can be modified
	// even if passed as a const argument !
	SpectrumWavelengths sw(sample->swl);

	// Trace a photon path and store contribution
	float u[2];
	// Choose light to shoot photon from
	float lightPdf;
	u_int lightNum = lightCDF->SampleDiscrete(GetOneD(*sample, 0, 0), &lightPdf);
	const Light *light = scene.lights[lightNum];

	// Generate _photonRay_ from light source and initialize _alpha_
	BSDF *bsdf;
	float pdf;
	SWCSpectrum alpha;
	GetTwoD(*sample, 0, 0, u);
	if (!light->SampleL(scene, *sample, u[0], u[1],
		GetOneD(*sample, 1, 0), &bsdf, &pdf, &alpha))
		return;

	Ray photonRay;
	photonRay.o = bsdf->dgShading.p;
	float pdf2;
	SWCSpectrum alpha2;
	GetTwoD(*sample, 1, 0, u);
```

```
		if (!bsdf->SampleF(sw, Vector(bsdf->dgShading.nn), &photonRay.d,
			u[0], u[1], GetOneD(*sample, 2, 0), &alpha2,
			&pdf2))
			return;
		alpha *= alpha2;
		alpha /= lightPdf;

		// The weight of the photon of the pass should be one, see ContribSample.
		alpha /= renderer->sppmi->photonPerPass / renderer->scene->camera->film->GetSamplePerPass();

		const bool directLightSampling = renderer->sppmi->directLightSampling;
		// store the state of the path:
		//	- if directLightPath is true, the photon is still on a direct light
		//	path and should not be accounted for if directLightSampling is true
		//	- else, the photon has survived an indirect bounce, so it must be
		//	accounted in the density estimation.
		bool directLightPath = true;

		if (!alpha.Black()) {
			// Follow photon path through scene and record intersections
			Intersection photonIsect;
			const Volume *volume = bsdf->GetVolume(photonRay.d);
			BSDF *photonBSDF;
			u_int nIntersections = 0;
			u_int diffuseVertices = 0;
			while (scene.Intersect(*sample, volume, false,
				photonRay, 1.f, &photonIsect, &photonBSDF,
				NULL, NULL, &alpha)) {
				++nIntersections;

				// Handle photon/surface intersection
				Vector wi = -photonRay.d;

				// Deposit Flux (only if we have hit a diffuse or glossy surface)
				// Note: the hitpoint BSDF allready handle this test, but it optimise a bit and avoid same bias
				if(!directLightPath || !directLightSampling)
					if (photonBSDF->NumComponents(BxDFType(BSDF_REFLECTION | BSDF_TRANSMISSION |
BSDF_GLOSSY | BSDF_DIFFUSE)) > 0)
					{
						PhotonData photon;
						photon.p = photonIsect.dg.p;
						photon.wi = wi;
						photon.alpha = alpha;
						photon.lightGroup = light->group;

						renderer->hitPoints->AddFlux(*sample, photon);
					}

				if (nIntersections > renderer->sppmi->maxPhotonPathDepth)
					break;

				// Sample new photon ray direction
				Vector wo;
				float pdfo;
				BxDFType flags;
				// Get random numbers for sampling outgoing photon direction
				float *data = GetLazyValues(*sample, 0, nIntersections);

				// Compute new photon weight and possibly terminate with RR
				SWCSpectrum fr;
				if (!photonBSDF->SampleF(sw, wi, &wo, data[0],
					data[1], data[2], &fr, &pdfo, BSDF_ALL,
					&flags))
					break;

				diffuseVertices += (flags & BSDF_DIFFUSE) ? 1 : 0;
				if (diffuseVertices > 0) {
					// Russian Roulette
					const float continueProb = min(1.f, fr.Filter(sw));
					if (data[3] > continueProb)
						break;
```

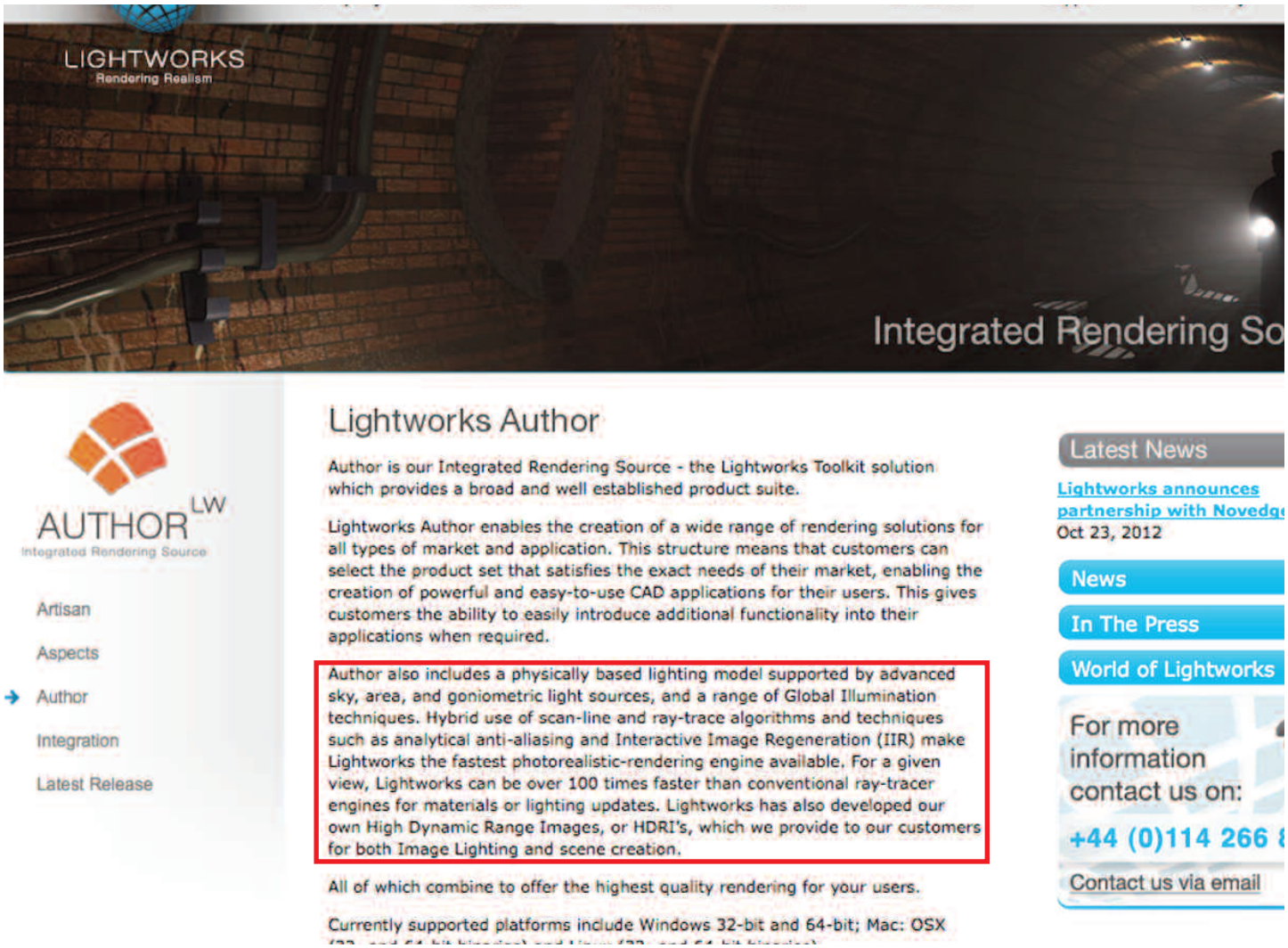
```
                                alpha /= continueProb;
                        }

                        alpha *= fr;
                        photonRay = Ray(photonIsect.dg.p, wo);
                        volume = photonBSDF->GetVolume(photonRay.d);

                        // Check if the scattering is not a passthrough event
                        if (flags != (BSDF_TRANSMISSION | BSDF_SPECULAR) ||
                                !(photonBSDF->Pdf(sw, wo, wi, BxDFType(BSDF_TRANSMISSION | BSDF_SPECULAR)) > 0.f)) {
                                // this is not a passthrough event, so now the photon path is indirect light
                                directLightPath = false;
                        }
                }
        }
        sample->arena.FreeAll();
}

//------------------------------------------------------------------------
// Photon Sampler
//------------------------------------------------------------------------

void PhotonSampler::ContribSample(Sample *sample)
{
        // cheat the sample count of the photon buffer
        // normally the photon buffer should be normalized by the number of photon
        // (hence the automatic +1 of AddSample which needs to be removed by a -1.f)
        // instead we normalize it by the number of pass, so the number of
        // contribution is 1.0 / photonPerPass
        //
        // WARNING: this is link to AMCMC weighting
        // (SPPMRenderer::ScaleUpdaterSPPM) and alpha in TracePhoton.
        sample->contribBuffer->AddSampleCount(-1.0 + 1.0 / renderer->sppmi->photonPerPass * renderer->scene->camera->film-
>GetSamplePerPass());
        dynamic_cast<Sampler*>(this)->AddSample(*sample);
}


void PhotonSampler::TracePhotons(
                        Sample *sample,
                        Distribution1D *lightCDF,
                        scheduling::Range *range)
{
        range->begin();
        while(range->next() != range->end())
        {
                GetNextSample(sample);

                TracePhoton(sample, lightCDF);

                ContribSample(sample);
        }
}

//------------------------------------------------------------------------
// Halton Photon Sampler
//------------------------------------------------------------------------

//------------------------------------------------------------------------
// Adaptive Markov Chain Sampler
//-------------------------------------------------------------------------

void AMCMCPhotonSampler::TracePhotons(
                        Sample *sample,
                        Distribution1D *lightCDF,
                        scheduling::Range *range)
{
        // Sample uniform
        do
        {
                GetNextSample(sample, true);
                TracePhoton(sample, lightCDF);
```

```
	} while(!pathCandidate->isVisible());

	swap(); // Current = Candidate

	range->begin();
	while(range->next() != range->end())
	{
		// Sample Uniform
		GetNextSample(sample, true);
		TracePhoton(sample, lightCDF);

		if(pathCandidate->isVisible())
		{
			swap();
			osAtomicInc(&renderer->uniformCount);
		}
		else
		{
			++mutated;

			// Sample mutated
			GetNextSample(sample, false);
			TracePhoton(sample, lightCDF);

			if(pathCandidate->isVisible())
			{
				++accepted;
				swap();
			}

			const float R = accepted / (float)mutated;
			mutationSize += (R - 0.234f) / mutated;

		}
		pathCurrent->Splat(sample, this);
		ContribSample(sample);
	}

	LOG(LUX_DEBUG, LUX_NOERROR) << "AMCMC mutationSize " << mutationSize << " accepted " << accepted << " mutated " << mutated << " uniform " << renderer->uniformCount;
}

// -----------------------------------
// AMCMCPhotonSampler sampler data
// -----------------------------------

void AMCMCPhotonSampler::AMCMCPhotonSamplerData::Mutate(const RandomGenerator * const rng, AMCMCPhotonSamplerData &source, const float mutationSize) const {
	for (size_t i = 0; i < n; ++i)
		values[0][i] = MutateSingle(rng, source.values[0][i], mutationSize);
}

float AMCMCPhotonSampler::AMCMCPhotonSamplerData::MutateSingle(const RandomGenerator * const rng, const float u, const float mutationSize) {
	// Delta U = SGN(2 E0 - 1) E1 ^ (1 / mutationSize + 1)

	const float du = powf(rng->floatValue(), 1.f / mutationSize + 1.f);

	if (rng->floatValue() < 0.5f) {
		float u1 = u + du;
		return (u1 < 1.f) ? u1 : u1 - 1.f;
	} else {
		float u1 = u - du;
		return (u1 < 0.f) ? u1 + 1.f : u1;
	}
}
```

- -

*YouTube film en op de internet pagina: "**it's physically correct**".*

*Als **'physically correct'** op wetenschap zou berusten, dan is het ook mogelijk om het te bewijzen of althans (wiskundig) aannemelijk te maken. Dat wordt echter nooit gedaan. Het blijft bij de 'one-liner'.*



Open Source Renderers

Sommige programma's en websites zullen al niet meer op te roepen zijn, andere programma's zijn blijven steken in de tijd. De ontwerper/programmeur van dergelijke programma's hebben andere interesses gekregen of zijn te druk in hun huidige werk. Er ontbreekt dan tijd om er mee door te gaan. Wellicht dat u uit bijgaande lijst en links één of meerdere interessante programma's aantreft met download van zowel het programma als de broncode plus ondersteunende informatie. Het is een goed startpunt om render programma's te bestuderen.

* Angelina
* Aqsis
* Art of Illusion
* Blender
* Inyo
* jrMan
* lucille
* Manta
* MegaPOV
* MiniLight
* Pane
* PARC
* PBRT
* perceptuum
* Pixie
* POV-Ray
* Radiance
* Ray of Light
* RenderPark
* R.I.S.E.
* toxic
* XFRT
* YafRay
* YACORT
* WinOSi

De **vet weergegeven links** zijn begin 2013 nog gecontroleerd en interessant om te bezoeken. De aanvullingen daarvan zijn redelijk up-to-date, en aan het programma wordt nog steeds gewerkt. De **rood** weergegeven programma's zijn vrijwel stopgezet. Maar bevatten soms toch nog interessante informatie.

http://lucille.atso-net.jp/svn/
Angelina

http://www.aqsis.org/
Aqsis
1.4.2 released eind 2012

http://www.artofillusion.org/
Art of Illusion
Haal ook de uitgebreide documentatie op voor extra informatie.

http://www.blender3d.org/
Blender
actief programma

http://inyo.sourceforge.net/
Inyo
"is more or less dead", overgestapt op SunFlow.

http://www.jrman.org/
jrMan
Laatste nieuws van februari 2007
jrMan is een open source versie van het REYES rendering algoritme die door Pixar's PhotoRealistic Renderman wordt gebruikt. Onderstaand een afbeelding uit jrMan.

http://lucille.sourceforge.net/
lucille
"lucille goes next stage. After 7 years of the development, lucille become commercial product to provide you better lucille experience: highly optimized rendering, richer functionality and professional support.
For more information, please visit the product page (its avaiable only in Japanese at this time) http://www.fixstars.com/ja/lucille/
Old open source version of lucille (but no

maintainance, no support. Just for your reference) is still avaiable at this sourceforge site."

De commerciële Windows uitvoering van deze Global Illumination renderer:

http://www.fixstars.com/en/lucille/

"Physically accurate. Unnaturally fast, Customizable, Cross-platform and RenderMan compatible." Cross-platform betekent op dit moment alleen x86 Windows 7 64-bit en IBM Linus 64-bit. Alhoewel er wel een MacOSX 10.7 building uitvoering blijkt te zijn. Met een inzending voor Siggraph 2012. Geschreven in M^3 (M-cubed) een software platform dat op meerdere CPU (GPU ?) core omgevingen incl. clusters werkt. Particle rendering is 50x zo snel geworden met de nieuwste Violin 6616 Flash Memory Array (SATA HDD). LUCILLE nieuwe stijl is gevestigd in Tokyo Japan.

http://code.sci.utah.edu/Manta/index.php/Main_Page
Manta
http://mantawiki.sci.utah.edu/manta/index.php/Main_Page
Welcome to the Manta-wiki. Manta is a highly portable interactive ray tracing environment designed to be used on both workstations, clusters, and super computers and is distributed under the MIT license. Manta was initially created at the Scientific Computing and Imaging (SCI) Institute at the University of Utah and is now used by a larger community including academic research groups and companies.

http://megapov.inetart.net/
MegaPOV
http://megapov.inetart.net/samples.html
Gallerie van renderingen uit 20-8-2005. Windows en een aantal oudere operating systeem versies voor Mac PPC.

http://www.hxa7241.org/minilight/minilight.html
MiniLight
Mac, Linux en Windows in drie maanden geschreven door auteur Harrison Ainsworth.

MiniLight is a minimal global illumination renderer. It is primarily an exercise in simplicity. But that makes it a good base and benchmark (in some sense) for development and experimentation. And it just might be the neatest renderer around (on average, about

650 lines). There are translations into several programming languages.

Er is bij de ontwikkeling gebruik gemaakt van Global Illumination Test Scenes:
http://www.cs.utah.edu/~bes/papers/scenes/

MiniLight vertelt op de website over Unbiased en correct.

MiniLight Unbiased?
Monte-carlo path-tracing is described as 'unbiased'. **The term is now sometimes used in the manner of a marketing tag**. Its real meaning is too abstract for that purpose. It denotes a lack of consistent error—the only error is in the effects of the randomness. Such niceties are appropriate for mathematics but for engineering we must live by practicalities. All implementations are biased and incorrect in various ways. The task is to arrange them acceptably.

MiniLight Ultimately
The value of this method is its simplicity, while having full generality. It allows the implementation to be almost proven correct by inspection. (**Correct meaning 'following the rules for a particular standard approximation'**.) The images produced can be authoritative references for other renderers.

The core idea has also proven remarkably durable and adaptable since Whitted's paper of 1980. One could reasonably call it the key engineering concept for rendering, and it looks to remain so.

The features are:

* Monte-carlo path-tracing transport
* Emitter sampling
* Progressive refinement
* RGB light
* Diffuse materials
* Triangle modelling primitives
* Octree spatial index
* Pin-hole 'lens'
* Ward linear tone-mapping

Download de programmeer codes onder de nieuwe BSD licentie regels:
http://www.hxa.name/minilight/#algorithm
In totaal 22 bestanden die met een simpele tekstverwerker kunnen worden opgeroepen.

3D rendering in Firefox browser

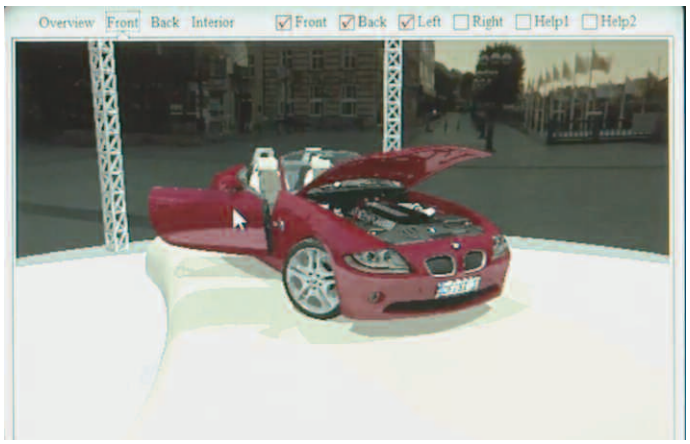
http://news.slashdot.org/story/10/03/04/1351211/3D-Graphics-For-Firefox-Webkit
"A group of researchers plans to release a version of the **Firefox browser** that includes the built-in ability to view 3D graphics. They've integrated real-time ray tracing technology, called RT Fact, into Firefox and Webkit. Images are described using XML3D, and the browser can natively render the 3D scene." The browser will be released within a few weeks, the researchers say, and they are checking with the Mozilla Foundation about whether they can call it Firefox.

Pseudo code **MiniLight** in 38:1 schaal.
17 code regels, één voor 38 uiteindelijke coderegels in het programma.

```
- render, by progressive refinement
  - render a frame, to the image
    - sample each image pixel
      - make sample ray direction
      - get radiance returning in sample direction
        - intersect ray with scene
        - if intersection hit
          - get local emission (only for first-hit)
          - calculate emitter sample
          - calculate recursive reflection
          - sum local emission, emitter sample,
            and recursive reflection
        - else no hit
          - use default/background scene emission
      - add radiance to image
  - save image
    - divide pixel values by frame count
    - apply tonemapping factor
```

Camera.c	30 januari 2011, 14:27	8 KB	Platte-tekstdocument
Camera.h	30 januari 2011, 14:27	4 KB	Platte-tekstdocument
Exceptions.c	30 januari 2011, 14:27	4 KB	Platte-tekstdocument
Exceptions.h	30 januari 2011, 14:27	4 KB	Platte-tekstdocument
Image.c	30 januari 2011, 14:27	8 KB	Platte-tekstdocument
Image.h	30 januari 2011, 14:27	4 KB	Platte-tekstdocument
MiniLight.c	30 januari 2011, 14:27	12 KB	Platte-tekstdocument
Primitives.h	30 januari 2011, 14:27	4 KB	Platte-tekstdocument
Rand...Mwc.c	30 januari 2011, 14:27	4 KB	Platte-tekstdocument
Rand...Mwc.h	30 januari 2011, 14:27	4 KB	Platte-tekstdocument
RayTracer.c	30 januari 2011, 14:27	8 KB	Platte-tekstdocument
RayTracer.h	30 januari 2011, 14:27	4 KB	Platte-tekstdocument
Scene.c	30 januari 2011, 14:27	8 KB	Platte-tekstdocument
Scene.h	30 januari 2011, 14:27	4 KB	Platte-tekstdocument
Spati...ndex.c	30 januari 2011, 14:27	12 KB	Platte-tekstdocument
Spati...dex.h	30 januari 2011, 14:27	4 KB	Platte-tekstdocument
Surfa...oint.c	30 januari 2011, 14:27	8 KB	Platte-tekstdocument
Surfa...oint.h	30 januari 2011, 14:27	4 KB	Platte-tekstdocument
Triangle.c	30 januari 2011, 14:27	8 KB	Platte-tekstdocument
Triangle.h	30 januari 2011, 14:27	4 KB	Platte-tekstdocument
Vector3f.c	30 januari 2011, 14:27	8 KB	Platte-tekstdocument
Vector3f.h	30 januari 2011, 14:27	4 KB	Platte-tekstdocument

MiniLight programma files, 22 in totaal.



FireFox
http://youtu.be/H66FetbJ5cI
CEBIT: XML3D brings 3D graphics to the Web as an extension to HTML

Flying Frog Ray Tracer

http://www.ffconsultancy.com/dotnet/fsharp/raytracer/code/1/raytracer.fs

http://www.ffconsultancy.com/dotnet/fsharp/raytracer/index.html
Flying Frog Consultancy
Putting the fun in functional since 2005

RAY TRACER
Ray tracing is a simple way to create images of 3D scenes. The method casts rays from the camera into the 3D scene and determines which object the ray intersects first. This approach makes some problems easy to solve:

* Shadows can be simulated by casting a ray from the intersection point towards the light to see if the intersection point has line-of-sight to the light.
* Reflections can be simulated by casting a second ray off the surface in the reflected direction.

BRONCODE Flying Frog in F# geschreven


```
/// Ray tracer in F#
/// (C) Flying Frog Consultancy Ltd., 2007
/// http://www.ffconsultancy.com

open Math
open Math.Notation
open System.Drawing
open System.Windows.Forms
open System.Threading

let pi = 4. * atan 1.
let delta = sqrt epsilon_float
let sqr x : float = x * x
let clamp l u x = if x < l then l else if x > u then u
else x
let vec x y z = vector [x; y; z; 1.]
let zero = vec 0. 0. 0.
let ( .+ ) s r = Vector.create 3 s + r
let dot a b = a.[0] * b.[0] + a.[1] * b.[1] + a.[2] *
b.[2]
let length2 r = sqr r.[0] + sqr r.[1] + sqr r.[2]
let length r = sqrt(length2 r)
let unitise r = 1. / length r $* r
let init = Matrix.init 4 4
let translate (r : vector) = init (fun i j -> if i = j
then 1. else if j = 3 then r.[i] else 0.)
let identity = Matrix.identity 4
```

Intersection Flying Frog

The core of this ray tracer is a recursive intersection function that hierarchically decomposes the sphereflake into its constituent parts:

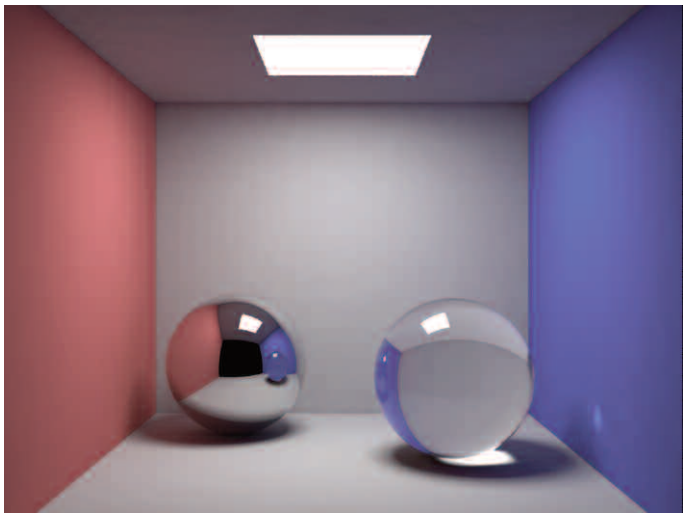
```
let rec intersect_spheres ray (lambda, _, _ as hit) =
function
 | Sphere (sphere, material) ->
    let lambda' = ray_sphere ray sphere in
    if lambda' >= lambda then hit else
    let normal = unitise (ray.origin + (lambda' $*
ray.direction) - sphere.center) in
    lambda', normal, material
 | Group (bound, scenes) ->
    let lambda' = ray_sphere ray bound in
    if lambda' >= lambda then hit else
List.fold_left (intersect_spheres ray) hit scenes
```

```
let rot_x t =
  let c = cos t and s = sin t in
  matrix [[ 1.; 0.; 0.; 0.];
          [ 0.;  c;  s; 0.];
          [ 0.; -s;  c; 0.];
          [ 0.; 0.; 0.; 1.]]
let rot_y t =
  let c = cos t and s = sin t in
  matrix [[  c; 0.;  s; 0.];
          [ 0.; 1.; 0.; 0.];
          [ -s; 0.;  c; 0.];
          [ 0.; 0.; 0.; 1.]]
let rot_z t =
  let c = cos t and s = sin t in
  matrix [[  c;  s; 0.; 0.];
          [ -s;  c; 0.; 0.];
          [ 0.; 0.; 1.; 0.];
          [ 0.; 0.; 0.; 1.]]

// Ray tracing primitives
type material = { color: vector; shininess: float }
type sphere = { center: vector; radius: float }
type obj = Sphere of sphere * material | Group of
sphere * obj list
type ray = { origin: vector; direction: vector }

// Direction of the light
let light = unitise (vec -1. -3. 2.)

// Levels of spheres
let level = 6

// Ratio of one level's radius to the next
let s = 1. /. 3.
```

```
// Sky material
let sky_material = { color = vector [0.; 0.1; 0.2];
shininess = 0. }

// Floor material as a function of coordinate
let floor_material x y =
  let z = Complex.mkRect(x, y) in
  let arg = Complex.phase z and s = Complex.mag-
nitude z in
  if int_of_float(s + arg / pi) % 2 = 0 then
    { color = vector [0.8; 1.; 0.7]; shininess = 0.15 }
  else
    { color = vector [0.; 0.; 0.]; shininess = 0. }

// Sphere material as a function of level
let sphere_material n =
  let t = pi / 180. * 15. * float n in
  { color = vector[sin t; sin(t + 2. * pi / 3.); sin(t + 4.
* pi / 3.)]; shininess = 0.5 }

// Ray-sphere intersection
let ray_sphere ray sphere =
  let v = sphere.center - ray.origin in
  let b = dot v ray.direction in
  let disc = sqr b - length2 v + sqr sphere.radius in
  if disc < 0. then infinity else
    let disc = sqrt disc in
    let t2 = b + disc in
    if t2 < 0. then infinity else
      let t1 = b - disc in
      if t1 > 0. then t1 else t2

// Intersect a ray with the tree of spheres
let rec intersect_spheres ray (lambda, _, _ as hit) =
function
  | Sphere (sphere, material) ->
      let lambda' = ray_sphere ray sphere in
      if lambda' >= lambda then hit else
        let normal = unitise (ray.origin + (lambda' $*
ray.direction) - sphere.center) in
        lambda', normal, material
  | Group (bound, scenes) ->
      let lambda' = ray_sphere ray bound in
      if lambda' >= lambda then hit else
List.fold_left (intersect_spheres ray) hit scenes

// Intersect a ray with the floor
let intersect_floor ray (lambda, _, _ as hit) =
  let lambda' = -ray.origin.[1] / ray.direction.[1] in
  if ray.direction.[1] > 0. || lambda' >= lambda
then hit else
    let r = ray.origin + (lambda' $* ray.direction) in
    lambda', vec 0. 1. 0., floor_material r.[0] r.[2]

// Find the first intersection of the given ray with
scene and floor
let intersect ray scene =
  intersect_floor ray (intersect_spheres ray (in-
finity, zero, sky_material) scene)

(* Trace a single ray *)
let rec ray_trace weight light ray scene =
  let lambda, normal, material = intersect ray
scene in
  if lambda = infinity then vector [0.; sqrt(ray.di-
rection.[1]); 1.] else
    let o = ray.origin + (lambda $* ray.direction) +
(delta $* normal) in
    (* Recursively examine specular reflections *)
    let color =
      if weight < 0.1 then Vector.create 3 0.5 else
        let d = ray.direction in
        let ray = { origin = o; direction = ray.direction
- (2. * dot ray.direction normal $* normal) } in
        material.shininess $* ray_trace (weight * ma-
terial.shininess) light ray scene in
    (* Calculate the final color, taking account of
shadows, specular and
       diffuse reflection. *)
    let s = match intersect { origin = o; direction =
zero - light } scene with
        slambda, _, _ when slambda = infinity -> max
0. (-dot normal light)
      | _ -> 0. in
    (s .+ color) .* material.color

// Find the bounding sphere of a given scene
let rec bound b = function
  | Sphere(s, _) -> { b with radius = max b.radius
(length(b.center - s.center) + s.radius) }
  | Group(_, scenes) -> List.fold_left bound b sce-
nes

// Build the scene
let scene =
    let rec aux level r m =
      let sphere = { center = m * zero; radius = r } in
      let material = sphere_material level in
      let obj = Sphere (sphere, material) in
      if level = 1 then obj else begin
        let make_top i =
          m * rot_y(pi / 6. + 2. * pi / 3. * float i) *
rot_z(pi / 4.) * translate(vec 0. ((1. + s) * r) 0.) in
        let make_bottom i =
          m * rot_y(pi / 3. * float i) * rot_z(110. * pi /
180.) * translate(vec 0. ((1. + s) * r) 0.) in

        let objects =
          List.map make_top [0 .. 2] @ List.map
make_bottom [0 .. 5] |>
            List.map (aux (level - 1) (s * r)) in

        Group (List.fold_left bound sphere objects,
obj :: objects)
      end in
    aux level 1. (translate (vec 0. 1. 0.))

// Render a pixel
```

```
let pixel w h x y =
  let ray =
    let o = vec 0. 2. -6. in
    let x, y = float x - 0.5 * float w, float y - 0.5 *
float h in
    let d = unitise(vec x y (float (max w h))) in
    let d = rot_x -0.1 * d in
    { origin = o ; direction = d } in
  let c = ray_trace 1. light ray scene in
  let scale x = clamp 0 255 (int_of_float(0.5 + 255.
* x)) in
  Color.FromArgb(scale c.[0], scale c.[1], scale c.[2])

// As threads complete rasters they are pushed
onto this stack
let rasters = ref (ref [])

// Render a raster and then pop it on the raster
stack
let raster r w h y =
  // If this image becomes obsolete then we raise
and catch an exception to return immediately
  try
    let data = Array.init w (fun x -> if !rasters != r
then raise Exit; pixel w h x y) in
    Idioms.lock !rasters (fun () -> r := (h - 1 - y,
data) :: !r)
  with Exit -> ()

// Spawn a thread for each raster
let render (form : #Form) =
  for y = 0 to form.Height - 1 do
    assert(ThreadPool.QueueUserWorkItem(fun _ -
> raster !rasters form.Width form.Height y;
form.Invalidate()));
  done

type Form1 = class
  inherit Form

  val mutable bitmap : Bitmap

  new() as form = { bitmap = null } then
    // Start rendering rasters
    render form;

    // Stop Windows from drawing the background
for this form
    form.SetStyle(Enum.combine[ControlStyles.All-
PaintingInWmPaint; ControlStyles.Opaque], true);

    form.Text <- "Ray tracer";
    form.bitmap <- new Bitmap(form.Width,
form.Height, Imaging.PixelFormat.Format24bp-
pRgb);
    form.Show()

  override form.OnPaint e =
    // Copy any rasters that have been rendered
into this form's bitmap
    // (the bitmap is only accessible from inside this
rendering thread)
    Idioms.lock !rasters (fun () ->
      let draw(y, data) = Array.iteri (fun x c -> try
form.bitmap.SetPixel(x, y, c) with _ -> ()) data in
      List.iter draw (! !rasters);
      !rasters := []);

    // Draw the bitmap on the form
    e.Graphics.DrawImage(form.bitmap,
form.ClientRectangle, new Rectangle(0, 0,
form.Width, form.Height), GraphicsUnit.Pixel)

  override form.OnResize e =
    // Reset the raster and replace the bitmap
    rasters := ref [];
    form.bitmap <- new Bitmap(form.Width,
form.Height, Imaging.PixelFormat.Format24bp-
pRgb);
    render form;
    form.Invalidate()

  override form.OnKeyDown e = if e.KeyCode =
Keys.Escape then form.Close()
end

do let form = new Form1() in while form.Created
do Thread.Sleep(1); Application.DoEvents() done
```

SunFlow caustic met Cornell Box.



http://www.csit.fsu.edu/%7Ebeason/pane/
Pane
Florida State University

http://www.flarg.com/parc/
PARC

http://www.pbrt.org/

PBRT

Physically Based Rendering, from theory to implementation
Laatste nieuws van 12 juli 2010.
Auteurs:
Matt Pharr (Intel architect) en Greg Humphreys (lid van de OptiX groep van NVIDIA)
Programma is aangevuld met een boek (2de editie prijs $ 61,-) dat nog steeds op universiteiten wordt gebruikt.

Enige renderproramma's die verwant zijn aan PBRT:

* MiniLight
* Perceptuum 2
* P3 ToneMapper
* P3 WhiteBalancer
* C++ Library
* Octree
* P3 Architecture

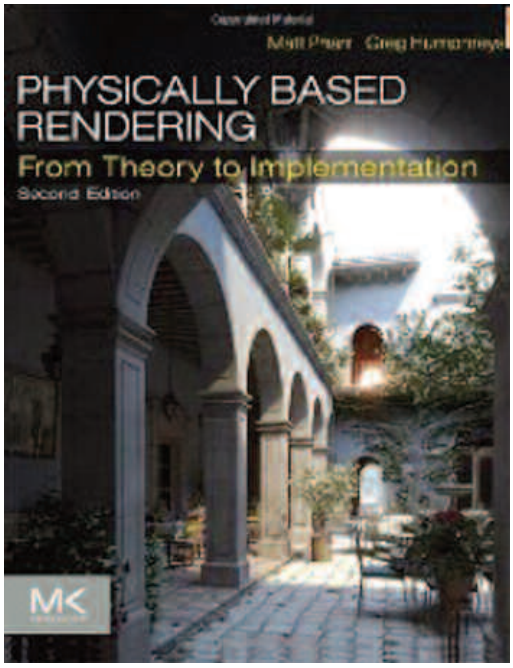
http://www.hxa7241.org/perceptuum/perceptuum.html

Perceptuum 2

Perceptuum is an Windows open-source standalone general purpose Physically-based renderer
simulating Global illumination with monte-carlo ray tracing. 30.000 code regels in C++.
Broncode beschikbaar.
Enige kernpunten:

* monte-carlo ray tracing basis
* illumination by direct lighting and directly visualised photon mapping
* generalised BRDF light interaction

Pane renderer



Physically Based Rendering, 2nd Edition describes both the mathematical theory behind a modern photorealistic rendering system as well as its practical implementation. A method - known as 'literate programming'- combines human-readable documentation and source code into a single reference that is specifically designed to aid comprehension. The result is a stunning achievement in graphics education. Through the ideas and software in this book, you will learn to design and employ a full-featured rendering system for creating stunning imagery.

* New sections on subsurface scattering, Metropolis light transport, precomputed light transport, multispectral rendering, and much more.

* Includes a companion site complete with source code for the rendering system described in the book, with support for Windows, OS X, and Linux. Please visit, www.pbrt.org.

* Code and text are tightly woven together through a unique indexing feature that lists each function, variable, and method on the page that they are first described.

* high dynamic range CIE XYZ light representation
* motion blur, depth of field, and glossy scattering capabilities
* quasi-random stochastic generators
* progressive refinement rendering
* persistable and deterministic sampling state
* triangle mesh models
* model hierarchical instancing
* image texturing with material and bump mapping

http://www.cs.utexas.edu/%7Eokan/Pixie/pixie.htm
Pixie
Department of Computer Science

http://www.povray.org/
POV-Ray
The Persistence of Vision Raytracer is a high-quality, totally free tool for creating stunning three-dimensional graphics. It is available in official versions for Windows, Mac OS/Mac OS X and i86 Linux. The source code is available for those wanting to do their own ports.

In 2013 zou er een dubbele nauwkeurigheid uitvoering verschijnen. Samen met het Parallela platform - Supercomputer for everyone.

Supercomputer voor iedereen

http://www.kickstarter.com/projects/adapteva/parallella-a-supercomputer-for-everyone

Deze bij Kickstarter aanwezige fabrikant wil via de website een investerings kapitaal van 750.000 dollar ophalen om zijn droom uit te zien komen.
De droom is om een supercomputer uit te brengen die voor iedereen is aan te schaffen.

1ste druk van Physically Based Rendering
Preface

Just as other information should be available to those who want to learn and understand program source code is the only means for programmers to learn the art from their predecessors.
Programming cannot grow and learn unless the next generation of programmers has access to the knowledge and information gathered by other programmers before them.
- Erik Naggum.

Een van de vele programma onderdelen van R.I.S.E. is deze RayIntersection.h

```
////////////////////////////////////////////////////////////////////

//
//  RayIntersection.h - A class that describes an intersection of
//        a ray with some object, it incorporates the geometric aspects
//
//  Author: Aravind Krishnaswamy
//  Date of Birth: October 31, 2001
//  Tabs: 4
//  Comments:
//
//  License Information: Please see the attached LICENSE.TXT file
//

////////////////////////////////////////////////////////////////////


#ifndef RAY_INTERSECTION_
#define RAY_INTERSECTION_
#include "RayIntersectionGeometric.h"

namespace RISE
{
        class IMaterial;
        class IRayIntersectionModifier;
        class IObject;
        class IShader;
        class IRadianceMap;
        class RayIntersection
        {
        public:
                RayIntersectionGeometric geometric;             // geometric elements of ray intersection
```

Voor rond de honderd dollar en zo aan te sluiten op een TV. Daarmee parallel computing voor iedereen aantrekkelijk te maken. In de PDF Werkstations meer daar over.

http://radsite.lbl.gov/radiance/

Radiance

Synthetic Imaging System (2009)
Open Source
Radiance User Interface for Windows
RADIANCE is a highly accurate ray-tracing software system for UNIX computers that is licensed at no cost in source form. Radiance was developed with primary support from the U.S. Department Of Energy and additional support from the Swiss Federal Government. Copyright is held by the Regents of the University of California.

http://claustres.lautre.net/rol/
Ray of Light
2004
Download source code met een invulformulier.

http://www.cs.kuleuven.ac.be/cwis/research/graphics/RENDERPARK/
RenderPark (vers. 3.3 augustus 2001)
Computer Graphics Research Group of Katholieke Universiteit Leuven
RenderPark is a test-bed system for physically based photo-realistic image synthesis. It's a free software package providing a solid implementation of a wide variety of state-of-the-art ray-tracing and radiosity algorithms.

```
const IMaterial*pMaterial;  // the material at the intersection

const IShader*pShader; // the shader at the intersection

const IRayIntersectionModifier*pModifier; // something that modifies the intersection

// this should be a list of somesort... eventually
   const IObject*pObject; // the object that was hit

  const IRadianceMap*pRadianceMap; // the radiance map at the intersection
                    RayIntersection( const Ray& ray,
const RasterizerState& rast ) :
                      geometric( ray, rast ),
                      pMaterial( 0 ),
                      pShader( 0 ),
                      pModifier( 0 ),
                      pObject( 0 ),
                      pRadianceMap( 0 )
                     {}

  RayIntersection( const RayIntersectionGeometric& rig ) :
 geometric( rig ),
pMaterial( 0 ),
pShader( 0 ),
pModifier( 0 ),
pObject( 0 ),
pRadianceMap( 0 )
{}
RayIntersection( const RayIntersection& r ) :
geometric( r.geometric ),
pMaterial( r.pMaterial ),
pShader( r.pShader ),
pModifier( r.pModifier ),
pObject( r.pObject ),
pRadianceMap( r.pRadianceMap )

 {}
   };
 }

#include "../Interfaces/IMaterial.h"
#include
"../Interfaces/IRayIntersectionModifier.h"
#include "../Interfaces/IObject.h"
#include "../Interfaces/IShader.h"
#include "../Interfaces/IRadianceMap.h"

#endif
```

Sourcecode.pdf (uit 1999) in bibliotheek pdf beschikbaar, uitleg over het programma, broncode is echter niet in deze PDF aanwezig.

http://rise.sourceforge.net/
R.I.S.E.
R.I.S.E. is no longer under active development, when the Sourceforge team decides to purge this project, we will not object. In the mean time, if you are looking for a nifty open source renderer I highly recommend checking out Sunflow. Broncode R.I.S.E. nog steeds beschikbaar.
Een deel van het programma staat op de voorgaande twee pagina's aan de rechterkant.

http://www.toxicengine.org/
toxic

http://xfrt.sourceforge.net/
XFRT

http://www.yafray.org/

YafRay

YafaRay is a free open-source raytracing engine. Raytracing is a rendering technique for generating realistic images by tracing the path of light through a 3D scene.
Versie 0.1.2 beta voor plug-in voor Blender 2.6. Met interessante documentatie beschikbaar. De bijgevoegde renderingen zien er goed uit, we zullen nog meer horen van deze open source render. Enkele belangrijke features van dit programma:

Global Illumination
Supported GI algorithms are Pathtracing, Photon Mapping + Final Gather and Bidirectional Pathtracing.

Background Illumination
The illumination can be obtained from simple backgrounds, HDR images and complete SunSky solutions.

Caustics effects.
YafaRay uses photon mapping techniques to produce fast yet accurate caustics effects. Caustics can be also calculated with Pathtracing paths.

Materials.
Four shader types are implemented in YafaRay, which are ShinyDiffuse, Glossy, CoatedGlossy and Glass. They are flexible enough and with many parameters available for each of them to produce a wide range of materials, transparent surfaces and reflections.

Textures.
YafaRay supports several mapping types, blending of different textures and modulation of material parameters with textures. YafaRay support several procedural textures as well. A procedural texture is a computer generated image produced by an algorithm and intended to create a realistic representation of natural elements.

Cameras
YafaRay implements four camera types to reproduce different optic effects. YafaRay also supports raytraced depth of field (DOF) to focus a part of the scene.

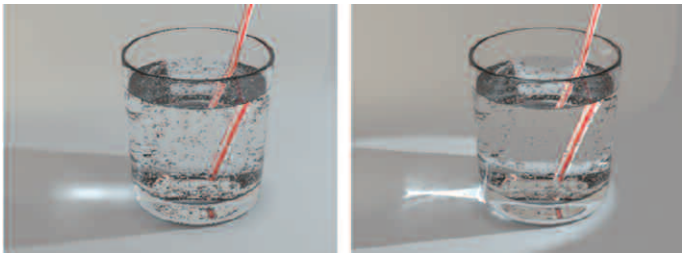
Volumetrics.
YafaRay's volumetric features provide a simulation of light interacting with particles suspended in a region of space. YafaRay uses a **realistic physics-based mode**l to render volumetrics, and provides a basis for creating not just believable beams of light, but also smoke, clouds, fog, and other volumetric effects.

http://www.yafaray.org/node/462

YafRay heeft een ideeën pagina in navolging van het Google Summer of Code programma. Zo kan er een uitwisseling van informatie plaats vinden die continu beschikbaar blijft. Eén van de eerste onderwerpen die aan de orde komt is Photon mappng projection maps.

YafaRay vervolg

YafaRay maakt geen gebruik van de methode waarbij lichtbronnen weten waar ze photonen naar toe moeten schieten.
Dat brengt een probleem mee in hoog geconcentreerde Photon Maps zoals bij caustics en volumetric. Voornamelijk als een arealight lichtbron wordt gekozen. Om de volgorde van de stroom photonen zo te richten op belangrijke gebieden levert Henrik Wann Jensen de oplossing: het maken van projectie maps die typisch als bitmaps zijn opgebouwd die om een kader scheren van de lichtbron. Waarbij elk bit bepaald of er geometrie aanwezig is die belangrijk is in voor die richting.



Vergelijking tussen Caustics maps die met een Arealight is gemaakt (rechts) en een spotlight (links).

http://bat710.univ-lyon1.fr/%7Ebsegovia/yacort/index.html
YACORT

http://www.winosi.onlinehome.de/

WinOSi

auteur Dipl. Ing. FH Michael Granz, Duitsland

WinOSi is an Open Source (vers. 0.46), freeware rendering tool for creating real photorealistic images by a new algorithm, which I have called 'Iterative Two Pass Optical Simulation Raytracing'. This algorithm combines the advantages of conventional raytracing and radiosity methods together with stunning optical effects like caustics, color dispersion and global illumination, with an accuracy not found in most rendering applications.
Typical rendering time for a simple scene in low resolution is 1 - 2 days (depending on CPU power, here about 1 Ghz) before the noise becomes invisible, leaving a perfect illuminated smooth and shiny image.
Aangezien de auteur 1 GHz computer gebruikt of zelfs nog lager mag worden opgemaakt dat het gaat om een programma dat jaren geleden al werd ontwikkeld. Multithreading wordt nog niet ondersteund. In het 'Merge Project" is het de bedoeling dat meerdere computers met elkaar kunnen worden gekoppeld.

Latest News:
- New release 0.46, VModel, RaVi-Demo, gallery updated & more!

Broncode beschikbaar. Evenals Windows applicaties.