```
/*
 * CS 148: Summer 2014. HW2.
 * David Ross (SCPD)
 */

/*
    pbrt source code is Copyright(c) 1998-2014
                        Matt Pharr, Greg Humphreys, and Wenzel Jakob.

    This file was modified by K. Breeden from the reference implementation for CS348B - Sp2014.
    contact: kbreeden@stanford.edu
 */

#include "stdafx.h"

// cameras/realistic.cpp*
#include "cameras/realistic.h"
#include <cmath>
#include "paramset.h"
#include "sampler.h"
#include "montecarlo.h"
#include "floatfile.h"
#include "imageio.h"

//#include <stdio.h>

bool not_drawn = true;
bool not_drawn_2 = true;
float total_lens_length = 0;
long long num_rays = 0;
long long num_hit = 0;

// TODO:
// Don't forget to include VDB here.
#include "renderers/vdb.h"

// RealisticCamera Utility Functions

/*  If an intersection is found between the given ray and spherical element,
 *      return true and set t, n.
 *
 *  Otherwise, return false.
 *
 *  radius: radius of curvature of the spherical lens element
 *  center: the (virtual) center of the spherical lens element is located at (0,0,center)
 *  ray:    the ray being traced through the element
 *  t:      the distance along [ray] at which an intersection occurs (if any)
 *  n:      the (normalized!) surface normal at the point of intersection (if any).
 *          Note: this should point outwards, ie, towards the incoming ray.
*/
static bool IntersectSphericalElement(float radius, float center,
                                      const Ray &ray, float *t, Normal *n) {


    /* Useful tidbits:
     * ray.o is of type Point and represents the origin of the ray
     * ray.d is of type Vector and is the direction of the ray
     *
     * the function "Quadratic(...)" located in core/pbrt.h
     * the function "Faceforward(...)" located in core/geometry.h
     * the function "Normalize(Vector)" also located in core/geometry.h
     */

    float a_quad = Dot(ray.d, ray.d);
    Point centr = Point(0, 0, center);
    Vector ro_c = ray.o - centr;
```

```
    float b_quad =  Dot((2 * ro_c), ray.d);
    float c_quad = Dot(ro_c, ro_c) - (radius * radius);
    float t0, t1;
    if(Quadratic(a_quad, b_quad, c_quad, &t0, &t1)){
        if(t0 < 0){
            *t = t1;
        } else {
            *t = t0;
        }
        Vector p0_pd = ((Vector)ray.o) + (t0 * ray.d);
        Vector nrml = Normalize(p0_pd - Vector(0, 0, center));
        Vector rel_nrml = p0_pd + nrml;
        *n = Normal(nrml.x,nrml.y,nrml.z);
        return true;
    } else {
        return false;
    }
}

/* returns true if the ray is transmitted.
 * returns false if there is total internal reflection
 *
 * n      = the surface normal at the refracting interface
 * wi     = the incident ray
 * eta_i  = index of refraction of inbound material
 * eta_t  = index of refraction of outbound material
 * wt     = the transmitted ray
 *
 * Note: core/geometry.h contains useful functions such as Normalize, Dot, ...
 *      also, Normals can be converted into vectors using Vector(Normal n)
 */
static bool Refract(const Normal &n, const Vector &wi, float eta_i, float eta_t, Vector *wt) {
    Vector nrml = Vector(n);
    Vector nrmlwi = Normalize(wi);
    float cos_ti = Dot(nrmlwi, nrml);
    float sin_ti = sqrtf(1.0 - (cos_ti * cos_ti));
    float sin_tt = sin_ti * (eta_i/eta_t);
    if(sin_tt > 1.0) return false;
    Vector r_in = (((eta_i/eta_t) * cos_ti) + sqrtf(1-((eta_i/eta_t)*(eta_i/eta_t)*sin_ti*sin_ti))) * nrml;
    Vector r_ip = (eta_i/eta_t) * nrmlwi;
    *wt = wi.Length() * (r_ip + r_in);
//  *wt = Normalize(r_ip + r_in);
    return true;
}

// RealisticCamera Method Definitions
RealisticCamera::RealisticCamera(const AnimatedTransform &cam2world,
                                 float sopen, float sclose,
                                 float apertureDiameter, float filmDist,
                                 float filmDiag, float focusDistance,
                                 bool sw, const char *lensFile, Film *f)
    : Camera(cam2world, sopen, sclose, f),
      apertureRadius(apertureDiameter / 2.f),
      focusedFilmDistance(filmDist), filmDiagonal(filmDiag),
      simpleWeighting(sw) {

    // Load element data from lens description file
    std::vector<float> lensData;
    if (ReadFloatFile(lensFile, &lensData) == false) {
        Error("Error reading lens specification file \"%s\".",
              lensFile);
        return;
    }

    // strides by 4; radius of curvature, thickness, index of refraction, aperture
    if ((lensData.size() % 4) != 0) {
```

```
        Error("Excess values in lens specification file \"%s\"; "
              "must be multiple-of-four values, read %d.",
              lensFile, (int)lensData.size());
        return;
    }

    // note, this halves the aperture to store aperture radius, not diameter
    for (int i = 0; i < (int)lensData.size(); i += 4) {
        LensElement le;
        le.curvatureRadius = lensData[i];
        le.thickness = lensData[i+1];
        le.eta = lensData[i+2];
        le.apertureRadius = lensData[i+3] / 2.f;
        elements.push_back(le);

        //elements.push_back((LensElement){lensData[i],lensData[i+1],
        //                                  lensData[i+2], lensData[i+3] / 2.f});
        if (lensData[i] == 0.f &&
            apertureDiameter > lensData[3]) {
            Warning("Specified aperture diameter %f is greater than maximum "
                    "possible %f.  Clamping it.", apertureDiameter, lensData[3]);
            apertureDiameter = lensData[3];
        }
    }

    // reverse to store the elements from nearest image space --> nearest object space
    std::reverse(elements.begin(), elements.end());

    // Compute film width and height from diagonal length
    float aspect = (float) film->yResolution / (float) film->xResolution;
    /*                x^2 + y^2 = filmDiag^2
     *                        y = aspect * x;
     * ==> x^2 + (aspect^2 x^2) = filmDiag^2
     *       x^2 (1 + aspect^2) = filmDiag^2
     *                        x = sqrt(filmDiag^2 / (1 + aspect^2);
     */

    filmWidth = std::sqrt(filmDiagonal * filmDiagonal /
                          (1.f + aspect * aspect));
    filmHeight = aspect * filmWidth;

    // Compute lens--film distance for the focus distance given in the input file
    float startFocusDistance = FocusDistance(focusedFilmDistance);

    Info("Initial film distance %f; the camera is focused at distance %f\n",
         focusedFilmDistance, startFocusDistance);

    // Find _filmDistanceLower_, _filmDistanceUpper_ that bound the focus distance
    float filmDistanceLower, filmDistanceUpper;
    filmDistanceLower = filmDistanceUpper = focusedFilmDistance;

    // increase filmDistLower until it's focusDist is <= the given focusDistance
    while (FocusDistance(filmDistanceLower) > focusDistance)
        filmDistanceLower *= 1.005f;

    // decrease filmDistUpper until it's focusDist is >= the given focusDistance
    while (FocusDistance(filmDistanceUpper) < focusDistance)
        filmDistanceUpper /= 1.005f;

    // Do binary search on film distances to focus
    for (int i = 0; i < 20; ++i) {
        float fmid = 0.5f * (filmDistanceLower + filmDistanceUpper);
        float midFocus = FocusDistance(fmid);
        if (midFocus < focusDistance)
            filmDistanceLower = fmid;
        else
```

```
            filmDistanceUpper = fmid;
    }

    // now, focusedFilmDistance will be as close as possible to putting objects at focusDistance in focus
    focusedFilmDistance = 0.5f * (filmDistanceLower + filmDistanceUpper);
    Info("Final film distance %f -> camera focus distance %f\n",
         focusedFilmDistance, FocusDistance(focusedFilmDistance));

    // Compute exit pupil bounds at sampled points on the film
    int nSamples = 64;
    for (int i = 0; i < nSamples; ++i) {
        float r = (float)i / (float)(nSamples-1) * filmDiagonal / 2.f;
        Bounds2f bounds = BoundExitPupil(Point2f(r, 0), focusedFilmDistance);
        exitPupilBounds.push_back(bounds);
    }
}


float RealisticCamera::FocusDistance(float filmDistance) const {
    // Find offset ray from film center through lens
    Bounds2f bounds = BoundExitPupil(Point2f(0, 0), filmDistance);
    float lu = 0.5f * bounds.pMax[0];
    Ray ray(Point(0,0,0), Vector(lu, 0, filmDistance), 0.f);

    if (!TraceThroughLenses(&ray, filmDistance)){
        Error("Focus ray at lens pos(%f,0) didn't make it through the lenses "
              "with film distance %f?!??\n", lu, filmDistance);
        return INFINITY;
    }

    // transform from mm to m
    ray.o.x *= .001f;
    ray.o.y *= .001f;
    ray.o.z *= .001f;

    // Compute distance _tFocus_ where ray intersects the principal (x) axis
    // org + t * dir = 0 (for x)
    //             t = -org/dir
    float tFocus = -ray.o.x / ray.d.x;
    if (tFocus < 0)
        tFocus = INFINITY;
    return tFocus;
}


Bounds2f RealisticCamera::BoundExitPupil(const Point2f &pFilm,
                                         float filmDistance) const {
    const int nSamples = 128;
    Bounds2f pupilBounds;
    float rearRadius = elements[0].apertureRadius;

    // Sample a grid of points on the rear lens to find exit pupil
    int numExitingRays = 0;
    for (int y = 0; y < nSamples; ++y) {
        for (int x = 0; x < nSamples; ++x) {
            // Find location of sample point on rear lens element
            Point pBack(Lerp((float)x / (nSamples-1), -rearRadius, rearRadius),
                        Lerp((float)y / (nSamples-1), -rearRadius, rearRadius),
                        filmDistance);

            // Skip ray if it's outside the rear element's radius
            if (pBack.x * pBack.x + pBack.y * pBack.y > rearRadius * rearRadius)
                continue;

            // Expand pupil bounds if ray makes it through the lens system
            Point Porg(pFilm.x, pFilm.y, 0.f);
```

```
            Ray ray(Porg, pBack - Porg, 0.f);
            if (TraceThroughLenses(&ray, filmDistance)) {
                pupilBounds = Union(pupilBounds, Point2f(pBack.x, pBack.y));
                ++numExitingRays;
            }
        }
    }

    // Return entire element bounds if no rays made it through the lens system
    if (numExitingRays == 0)
    {
        Info("Unable to find exit pupil at (%f,%f) on film.", pFilm.x, pFilm.y);
        return Bounds2f(Point2f(-rearRadius, -rearRadius),
                        Point2f( rearRadius,  rearRadius));
    }

    pupilBounds.Expand(2.f * rearRadius / (nSamples-1));
    return pupilBounds;
}


void RealisticCamera::RenderExitPupil(float sx, float sy,
                                      const char *filename) const {
    Point Porg(sx, sy, 0.f);

    const int nSamples = 512;
    float *image = new float[3*nSamples*nSamples];
    float *imagep = image;

    for (int y = 0; y < nSamples; ++y) {
        float fy = (float)y / (float)(nSamples-1);
        float ly = Lerp(fy, -elements[0].apertureRadius,
                             elements[0].apertureRadius);
        for (int x = 0; x < nSamples; ++x) {
            float fx = (float)x / (float)(nSamples-1);
            float lx = Lerp(fx, -elements[0].apertureRadius,
                                 elements[0].apertureRadius);

            Point Pback(lx, ly, focusedFilmDistance);
            Ray ray(Porg, Pback - Porg, 0.f);

            if (TraceThroughLenses(&ray, focusedFilmDistance)) {
                *imagep++ = 1.f;
                *imagep++ = 1.f;
                *imagep++ = 1.f;
            }
            else {
                *imagep++ = 0.f;
                *imagep++ = 0.f;
                *imagep++ = 0.f;
            }
        }
    }

    WriteImage(filename, image, NULL, nSamples, nSamples,
                 nSamples, nSamples, 0, 0);
    delete[] image;
}

/*
 * This method refracts a ray through each of the lens elements
 * returns true if ray makes it all the way through; false otherwise
 *
 * ray          : the ray whose path is being altered by the lens
 * filmDistance : distance to first lens element
 */
```

```
bool RealisticCamera::TraceThroughLenses(Ray *ray, float filmDistance) const {
    float pi = 3.141592;
    float x = 0.0;
    float y = 0.0;
    float z = 0.0;
    bool draw = false;
// Draws the intial scene.
    if(not_drawn){
        for (int i = 0; i < elements.size(); i++) {
            if(i%2){
                vdb_color(255/255,0/255,0/255);//red
            } else {
                vdb_color(0/255,0/255,255/255);//blue
            }
            for (float t = 0.0; t <= 2 * pi; t = t + pi / 1024){
                vdb_point(elements[elements.size() - 1 - i].apertureRadius * cos(t), elements[elements.size ↙
    () - 1 - i].apertureRadius * sin(t), z);
            }
            z = z - elements[elements.size() - 1 - i].thickness;
        }
        total_lens_length = z;
        vdb_color(255/255,0/255,0/255);//red
        vdb_line(0,0,0,100,0,0);
        vdb_color(0/255,255/255,0/255);//green
        vdb_line(0,0,0,0,100,0);
        vdb_color(255/255,255/255,255/255);//white
        vdb_line(0,0,0,0,0,-100);
        not_drawn = false;
    }
    float center = total_lens_length - elements[0].curvatureRadius;
    float ray_start = total_lens_length - filmDistance;
    float lens_length = total_lens_length;
    ray->o.z = ray->o.z + ray_start;
    for (int i = 0; i < elements.size(); i++) {
        float t = 0;
        Normal n;
// Special case of stop
        if(elements[i].eta == 0){
            float t_ap = (lens_length - ray->o.z) / ray->d.z;
            float x_ap = ray->o.x + (t_ap * ray->d.x);
            float y_ap = ray->o.y + (t_ap * ray->d.y);
            float r_ap = sqrtf((x_ap * x_ap) + (y_ap * y_ap));
            if(r_ap >= elements[i].apertureRadius){
                return false;
            } else {
                ray->o.x = x_ap;
                ray->o.y = y_ap;
                ray->o.z = lens_length;
                if(i < elements.size() - 1){
                    lens_length = lens_length + elements[i+1].thickness;
                    center = lens_length - elements[i+1].curvatureRadius;
                }
            }
        } else {
            if(IntersectSphericalElement(elements[i].curvatureRadius, center, *ray, &t, &n)){
                float t_ap = (lens_length - ray->o.z) / ray->d.z;
                float x_ap = ray->o.x + t_ap * ray->d.x;
                float y_ap = ray->o.y + t_ap * ray->d.y;
                float r_ap = sqrtf((x_ap * x_ap) + (y_ap * y_ap));
                if(r_ap >= elements[i].apertureRadius){
                    return false;
                } else {
                    if( rand() % 10000000 == 1 ){
                        vdb_color(255/255,0/255,0/255);//red
                        vdb_line(ray->o.x, ray->o.y, ray->o.z, ray->o.x + (t * ray->d.x), ray->o.y + (t *   ↙
    ray->d.y), ray->o.z + (t * ray->d.z));
```

```
                    }
                    Vector wt;
                    float next_eta = 0;
                    if((elements[i+1].eta == 0) || (i == elements.size() - 1)){
                        next_eta = 1;
                    } else {
                        next_eta = elements[i+1].eta;
                    }
                    if(Refract(n, ray->d, elements[i].eta, next_eta, &wt)) {
                        ray->o.x = ray->o.x + (t * ray->d.x);
                        ray->o.y = ray->o.y + (t * ray->d.y);
                        ray->o.z = ray->o.z + (t * ray->d.z);
                        ray->d.x = wt.x;
                        ray->d.y = wt.y;
                        ray->d.z = wt.z;
                        if(i < elements.size() - 1){
                            lens_length = lens_length + elements[i+1].thickness;
                            center = lens_length - elements[i+1].curvatureRadius;
                        }
                    } else {
                        return false;
                    }
                }
            } else {
                return false;
            }
        }
    }
    return true;
    /*
     * Useful tidbits:
     * - the lens element struct contains values such as thickness, curvature radius, and aperture radius
     * - if the lens element is a stop, the curvature radius will be 0.
     *      (in this case, the value t for the intersection is simply the ray/plane intersection point.)
     *- once you have computed the location of the intersection, don't forget to ensure that it happens
    within
     *      the aperture radius of that element!
     *
     *- A Point p along a Ray* ray can be accessed as: Point p = (*ray)(t);
     */
}


Point RealisticCamera::SampleExitPupil(const Point2f &pFilm,
        const Point2f &lensSample, float *pdf) const {
  // Find exit pupil bound for sample distance from film center
  float rFilm = std::sqrt(pFilm.x * pFilm.x + pFilm.y * pFilm.y);
  float r = rFilm / (filmDiagonal / 2.f);

  int pupilIndex = std::min((int) exitPupilBounds.size() - 1,
                            (int) std::floor(r * (exitPupilBounds.size() - 1)));

  Bounds2f pupilBounds = exitPupilBounds[pupilIndex];
  if (pupilIndex + 1 < (int) exitPupilBounds.size())
      pupilBounds = Union(pupilBounds, exitPupilBounds[pupilIndex+1]);

  // Generate sample point inside exit pupil bound
  Point2f pLens = pupilBounds.Lerp(lensSample);
  if (pLens.x * pLens.x + pLens.y * pLens.y >
      elements[0].apertureRadius * elements[0].apertureRadius) {
    *pdf = 0;
    return Point(0,0,0);
  }

  // Rotate sample point by angle of _pFilm_ with $+x$ axis
  float sin_t = pFilm.y / rFilm, cos_t = pFilm.x / rFilm;
```

```
  Point2f pLensRot(cos_t * pLens.x - sin_t * pLens.y,
                   sin_t * pLens.x + cos_t * pLens.y);
  *pdf = 1.f / pupilBounds.Area();
  return Point(pLensRot.x, pLensRot.y, focusedFilmDistance);
}


void RealisticCamera::TestExitPupilBounds() const {
  static RNG rng;

  float u = rng.RandomFloat();
  Point Porg(u * filmDiagonal / 2.f, 0., 0.);

  float r = Porg.x / (filmDiagonal / 2.f);
  int pupilIndex = std::min((int) exitPupilBounds.size() - 1,
                            (int) std::floor(r * (exitPupilBounds.size() - 1)));

  Bounds2f pupilBounds = exitPupilBounds[pupilIndex];

  if (pupilIndex + 1 < (int) exitPupilBounds.size())
      pupilBounds = Union(pupilBounds, exitPupilBounds[pupilIndex + 1]);

  // Now, randomly pick points on the aperture and see if any are outside
  // of pupil bounds...
  for (int i = 0; i < 1000; ++i) {
    Point2f pd;
    ConcentricSampleDisk(rng.RandomFloat(), rng.RandomFloat(), &pd.x, &pd.y);
    pd.x *= elements[0].apertureRadius;
    pd.y *= elements[0].apertureRadius;

    Ray testRay(Porg, Point(pd.x, pd.y, focusedFilmDistance) - Porg, 0.f);
    if (!TraceThroughLenses(&testRay, focusedFilmDistance))
        continue;

    if (!pupilBounds.Inside(pd)) {
      fprintf(stderr, "Aha! (%f,%f) went through, but outside bounds (%f,%f) - (%f,%f)\n",
              pd.x, pd.y, pupilBounds.pMin[0], pupilBounds.pMin[1],
              pupilBounds.pMax[0], pupilBounds.pMax[1]);
      RenderExitPupil((float) pupilIndex / exitPupilBounds.size() * filmDiagonal/2.f, 0.,
                      "low.exr");
      RenderExitPupil((float) (pupilIndex + 1) / exitPupilBounds.size() * filmDiagonal/2.f, 0.,
                      "high.exr");
      RenderExitPupil(Porg.x, 0., "mid.exr");
      exit(0);
    }
  }
  fprintf(stderr, ".");
}


float RealisticCamera::GenerateRay(const CameraSample &sample,
        Ray *ray) const {

    // Generate initial ray, _ray_, pointing at rearmost lens element
    Point2f s((sample.imageX - film->xResolution / 2.f) / film->xResolution,
              (sample.imageY - film->yResolution / 2.f) / film->yResolution);

    Point Porg(-s.x * filmWidth, s.y * filmHeight, 0.f);

    float pdf;
    Point Pback = SampleExitPupil(Point2f(Porg.x, Porg.y),
                                  Point2f(sample.lensU, sample.lensV),
                                  &pdf);
    if (pdf == 0.f)
      return 0.f;
```

```
    *ray = Ray(Porg, Pback - Porg, 0.f);
    float costheta = Normalize(ray->d).z;

    if (!TraceThroughLenses(ray, focusedFilmDistance))
        return 0.f;

    // Finish initialization of realistic camera ray
    // Transform _ray_ from millimeters to meters
    ray->o.x *= .001f;
    ray->o.y *= .001f;
    ray->o.z *= .001f;

    ray->time = Lerp(sample.time, shutterOpen, shutterClose);
    *ray = CameraToWorld(*ray);
    ray->d = Normalize(ray->d);
    ray->maxt = INFINITY;

    // Return weighting for lens system ray
    if (simpleWeighting)
        return (costheta * costheta) * (costheta * costheta);
    else
        return ((costheta * costheta) * (costheta * costheta)) /
            (focusedFilmDistance * focusedFilmDistance * pdf);
}


RealisticCamera *CreateRealisticCamera(const ParamSet &params,
        const AnimatedTransform &cam2world, Film *film) {

    float shutteropen = params.FindOneFloat("shutteropen", 0.f);
    float shutterclose = params.FindOneFloat("shutterclose", 1.f);

    if (shutterclose < shutteropen) {
        Warning("Shutter close time [%f] < shutter open [%f].  Swapping them.",
                shutterclose, shutteropen);
        std::swap(shutterclose, shutteropen);
    }

    // Realistic camera-specific parameters
    std::string lensFile = params.FindOneFilename("specfile", "");
    float filmDistance = params.FindOneFloat("filmdistance", 70.0);
    float apertureDiameter = params.FindOneFloat("aperture_diameter", 1.0);
    float filmDiagonal = params.FindOneFloat("filmdiag", 35.0);
    float focusDistance = params.FindOneFloat("focusdistance", 10.0);
    bool simpleWeighting = params.FindOneBool("simpleweighting", true);
    if (lensFile == "") {
        Error("No lens description file supplied!");
        return NULL;
    }

    return new RealisticCamera(cam2world, shutteropen, shutterclose,
                               apertureDiameter, filmDistance, filmDiagonal,
                               focusDistance, simpleWeighting,
                               lensFile.c_str(), film);
}
```