A guide to the G3D renderer examples

This guide describes two renderers that are very similar to what's described in the "Rendering in Practice" chapter of the book. The first is a pathtracer; the second is a photon mapper that does final gathering. Both differ from the implementation in the book in modest ways that have to do with an update to G3D between the writing of the book and the final versions of the renderers. I'll assume that you've read the book already, or are doing so simultaneously with your reading of these notes.

The large-scale changes

There are four major changes in the code.

- The C++ standards committee decided that a notion of shared pointers should be adopted in to the language; that made many of G3D's datatypes outmoded. Any place where the code used to have something like Triangle::Ref, we now have shared pointers.
- The Surfel datatype changed.
- The BSDF datatype changed.
- What used to be called defaultCamera has been renamed debugCamera.

Large scale program structure

There are a few cooperating classes.

App: The application that loads up a world, renders it, displays the result, provides a GUI to let the user decide which world to load, how many rays to cast, etc. There are two options: by checking a checkbox, you can have the scene raytraced; by unchecking, you can have it pathtraced. In each case, the work is done by three procedures. For raytracing,

- The first is `rayTraceImage`, which sets up the image into which the scene is to be rendered, calls `trace`, which does the real work, and then "exposes" the result onto an image called m_result, which can be displayed for the user to see.
- The second is `trace`, which takes an xy-position on the film plane and from it computes some number (`m_primaryRaysPerPixel`) of rays, calls `rayTrace` on each of them, and averages the results.
- The third is `rayTrace` which takes a ray and a "bounce" argument, and does raytracing along that ray, using at most `m_maxBounce` levels of recursion; the `bounce` argument tells how many bounces have already taken place.

Path tracing follows a parallel pattern, using `pathTraceImage` and `ptrace` analogously. The only subtlety is in the line

```
GThread::runConcurrently2D(Point2int32(0, 0), Point2int32(width, height), this, &App::trace);
```

in which the fourth argument is invoked on all possible pairs of values specified by the first two arguments. The first two arguments range from (0,0) to (width-1, height-1); for each such xy-pair, we invoke `trace(x, y)`, using as many concurrent threads as are available.

The application also maintains a notion of the current camera frame (i.e., position and orientation parameters) and the previous one. The user can rotate the camera by dragging with the right mouse in the main window (or using the WASD keys, as in many video games); if the current and former positions differ by enough, a new view is rendered using raytracing (which is faster) so that it's easy to pose the camera the way you like it before invoking the pathtracer.

World: A description of the geometry and lights in a scene. Our implementation is very basic, and each of six scenes is hand-coded. The example code provided with G3D shows how to make a scene-loader that can read a nicely-formatted text file and produce a scene from it. If you're going to do a lot of work with either of these programs, adopting some of that code would be a good idea.

AreaLight: A representation of an area light source. A light source has a total power and a mesh representation. The power is radiated uniformly from the whole mesh, at each point radiating into the hemisphere described by the surface normal. In all our examples, the mesh is simply a rectangle, emitting from one side.

SurfaceSampler: A representation of geometry used by AreaLight. It provides a method for randomly sampling a point of the geometry uniformly with respect to surface area.

Aside from the G3D support classes, which are extensively used, those are the only classes used by the pathtracer.

The photon mapper uses a few additional classes:

LightList: a list of point and area lights

EPhoton: a photon that's been emitted from a light source

IPhoton: a photon that's arrived at a surface and needs to be stored in the photon map

PhotonMap: A place to store photons.

Commentary on the classes

What follows is the code for each class, with additional comments in magenta to explain what's going on.

The App Class (for the pathtracer project)

The pathtracer provides not only code for pathtracing, which can be slow, but also for a raytracer to quickly show the scene that you'll be pathtracing when you press the "Render" button.

```
// App is derived from GApp, G3D's basic 3D application class, which provides
// the ability to print stuff to the screen, to display a customizable GUI,
// an easy-to-use camera manipulator, etc.

/**
  @file App.cpp PATH TRACING
 */
#include "App.h"
#include "World.h"

G3D_START_AT_MAIN(); //Makes windows programs start at the right
                     //place by defining WinMain

int main(int argc, char** argv) {
    GApp::Settings settings;
    settings.window.caption     = "JFH Path Trace";
    settings.window.width       = 900;
    settings.window.height      = 900;
    return App(settings).run(); //Initialize app with this window-size and caption,
                                //and run it.
}
```

```
// The member variables defined here are all things that control the rendering.
// For instance, m_pointLights, when false, causes the contribution of point
// lights to the scene to be ignored.
App::App(const GApp::Settings& settings) :
GApp(settings),
    m_maxBounces(3),        // Max recursive depth used in preview raytracer
    m_pointLights(false),   // Include point lights in computations?
    m_areaLights(true),     // Include area lights in computations?
    m_whichRenderer(false), // Raytrace instead of pathtracing when true
    m_whichWorld(2),        // Start by loading/rendering world number 2
    m_primaryRaysPerPixel(1),
    m_world(NULL),          // Reference to currently loaded world
    m_direct(true),         // Include direct light in computations?
    m_direct_s(true),       // Include specularly-reflected direct light?
    m_indirect(true),       // Include indirect light in computations?
    m_emit(true),           // Include emitted light in computations?
    m_scaleFactor(0.2f) // Image is displayed in a large window, but may be rendered
                        // smaller, for speed, and then scaled up. This says to
                        // initially render at 1/5 resolution in each axis, speeding
                        // results up by a factor of about 25, but losing a lot of
                        // detail.
{}
```

```
// Called when program is first invoked. Build the main window, the GUI, etc.
void App::onInit() {
    message("Loading..."); // Let the user know what's going on
    m_world = new World(m_whichWorld);

    showRenderingStats = false;

    // Build some visual components (the "developer Heads-Up Dispay") that
    // we mostly don't use, except the GUI;
    // Hide the components that we don't care about.
    createDeveloperHUD();
    developerWindow->setVisible(false);
    developerWindow->cameraControlWindow->setVisible(false);

    // Starting position
    debugCamera()->setFrame(CFrame::fromXYZYPRDegrees(0.0f, 0.3f, 0.0f, 0.0f, 0.0f, 0.0f));
    debugCamera()->filmSettings().setBloomStrength(0.0f); // turn off bloom filter


    // record previous pose as well, so we can tell when camera's moved
    m_prevCFrame = debugCamera()->frame();

    makeGUI(); // Build the panel from which we'll control the program
    onRender();// And pretend that the user pressed the "Render" button.
}
```

```
void App::makeGUI() {
    message("Building GUI");
// Build a GUI window. The small rectangle makes it grow only as needed.
    shared_ptr<GuiWindow> window = GuiWindow::create("Controls", debugWindow->theme(), Rect2D::xywh(0,0,0,0),
GuiTheme::TOOL_WINDOW_STYLE);
    GuiPane* pane = window->pane();
    pane->addLabel("Use WASD keys + right mouse to move");
// Add a button with the label "Render High Res." When the button is pressed, the
// onRender() method will be called.
    pane->addButton("Render High Res.", this, &App::onRender);

// Add a box with a slider next to it, that runs from 1 to 16 in steps of 3.
// When the slider's moved, or the user types a new number into the box,
// the value of m_maxBounces will be changed to the new number.
    pane->addNumberBox("Max bounces", &m_maxBounces, "", GuiTheme::LINEAR_SLIDER, 1, 16, 3);
    pane->addNumberBox("Prim. Rays/Pix", &m_primaryRaysPerPixel, "", GuiTheme::LOG_SLIDER, 1, 4096, 1);
    pane->addCheckBox("Use point lights?", &m_pointLights);
    pane->addCheckBox("Use area lights?", &m_areaLights);
    pane->addCheckBox("Direct light", &m_direct);
    pane->addCheckBox("Specular Direct light", &m_direct_s);
    pane->addCheckBox("Indirect light", &m_indirect);
    pane->addCheckBox("Emitted light", &m_emit);
    pane->addCheckBox("Raytrace instead of pathtrace", &m_whichRenderer);
    pane->addNumberBox("Which world?", &m_whichWorld, "", GuiTheme::LINEAR_SLIDER, 0, World::m_nWorlds-1, 1);
    pane->addNumberBox("Scale", &m_scaleFactor, "", GuiTheme::LINEAR_SLIDER, -0.0f, 1.10f, 0.1f);
    window->pack();// Resize the GUI to contain all this stuff without overlaps
    window->setVisible(true);
    addWidget(window); // Include the GUI into the GApp
    message("Done with GUI");
}
```

```
// Called whenever the window needs to be updated
void App::onGraphics(RenderDevice* rd, Array<shared_ptr<Surface>>& surface3D, Array<Surface2D::Ref>& surface2D) {
// If the previous and current camera positions differ, do a quick raytrace to
// show the new view
    if (! m_prevCFrame.fuzzyEq(debugCamera()->frame())) {
        rayTraceImage(0.2f, 1);
        m_prevCFrame = debugCamera()->frame();
    }
// rd is the “render device”; the following lines update
// the image shown on that device, by using the image we’ve produced (m_result)
// as a texture for the background.

    rd->clear();
    if (notNull(m_result)) {
        rd->push2D();
        rd->setTexture(0, m_result);
        Draw::rect2D(rd->viewport(), rd);
        rd->pop2D();
    }
    Surface2D::sortAndRender(rd, surface2D);
}

void App::onCleanup() {
}

// Most of the remaining code matches the book quite closely, so I won’t make as many comments
static const float RAY_BUMP_EPSILON = 0.5 * units::millimeters();
// A single random number generator used in all places that need it.
static Random rnd(0xF018A4D2, false);


// Display a message for the user to read
void App::message(const std::string& msg) const {
    renderDevice->clear();
    renderDevice->push2D();
        debugFont->draw2D(renderDevice, msg, renderDevice->viewport().center(), 12,
            Color3::white(), Color4::clear(), GFont::XALIGN_CENTER, GFont::YALIGN_CENTER);
    renderDevice->pop2D();

    // Force update so that we can see the message
    renderDevice->swapBuffers();
}
```

```
// This procedure is called whenever the user presses the "Render" button, and also during initialization
void App::onRender() {
    message("Rendering...");

    // set up a timer to see how long rendering takes
    Stopwatch timer;
    if (m_world) delete m_world; // start with a clean instance of the world

    m_world = new World(m_whichWorld);
    if (m_whichRenderer) { // If raytracing's selected, do it and time it.
        rayTraceImage(m_scaleFactor, m_primaryRaysPerPixel);
        timer.after("Ray Trace");
       }
    else { // Otherwise, invoke pathtracing and time it.
        pathTraceImage(m_scaleFactor, m_primaryRaysPerPixel);
        timer.after("Path Trace");
    }
    debugPrintf("%f s\n", timer.elapsedTime()); // show how long rendering took
    m_currentImage->save("result.png"); // And save the resulting image for later use
}

// Raytrace an image at some multiple (usually less than 1) of the size of the display window.
// Do so using numRays rays per pixel.
void App::rayTraceImage(float scale, int numRays) {
    int width  = int(window()->width()  * scale);
    int height = int(window()->height() * scale);

    if (isNull(m_currentImage) || (m_currentImage->width() != width) || (m_currentImage->height() != height)) {
        m_currentImage = Image3::createEmpty(width, height);
    }
    m_primaryRaysPerPixel = numRays;
   // invoke the trace(x, y) on every xy-pair from (0,0) to (width, height)
   // using as many threads as are available.
    GThread::runConcurrently2D(Point2int32(0, 0), Point2int32(width, height), this, &App::trace);

    // Post-process
    shared_ptr<Texture> src = Texture::fromImage("Source", m_currentImage);
    if (m_result) {
        m_result->resize(width, height);
    }
    m_film->exposeAndRender(renderDevice, m_debugCamera->filmSettings(), src, m_result);
}
```

```
// Trace rays from the eye through pixel (x, y)
void App::trace(int x, int y) {
    Color3 sum = Color3::black();

    if (m_primaryRaysPerPixel == 1) { // shoot ray through pixel center for single sample
        sum = rayTrace(debugCamera()->worldRay(x + 0.5f, y + 0.5f, m_currentImage->rect2DBounds()));
    } else {
        for (int i = 0; i < m_primaryRaysPerPixel; ++i) { // use jittered rays for multiple samples
            sum += rayTrace(debugCamera()->worldRay(x + rnd.uniform(), y + rnd.uniform(), m_currentImage->rect2DBounds()));
        }
    }
    m_currentImage->set(x, y, sum / m_primaryRaysPerPixel);
}

// Actually trace the ray into the scene; "bounce" is the number of bounces it's already taken.
Radiance3 App::rayTrace(const Ray& ray, int bounce) {
    Radiance3 radiance = Radiance3::zero();

    shared_ptr<Surfel> surfel;
    float dist = inf();
    if (m_world->intersect(ray, dist, surfel)) {
        // Shade this point (direct illumination)
        for (int L = 0; L < m_world->lightArray.size(); ++L) {
            const shared_ptr<Light>& light = m_world->lightArray[L];

// Note that "geometric.normal" has changed to "geometricNormal"
// The next line uses the translation of the light's frame as
// the position of a point light, even if the actual light is an
// area light, because basic raytracing can't handle those.
            // Shadow rays
            if (m_world->lineOfSight(surfel->position + surfel->geometricNormal * 0.0001f, light->frame().translation)) {
                Vector3 w_i = light->frame().translation - surfel->position;
                const float distance2 = w_i.squaredLength();
                w_i /= sqrt(distance2);

                // Attenduated radiance
                const Irradiance3& E_i = light->bulbPower() / (4.0f * pif() * distance2);
// What used to be "evaluateBSDF" has changed to "finiteScatteringDensity",
// which is a better name, since the returned value is the non-impulse part of
// the BSDF
                radiance +=
```

```
                    surfel->finiteScatteringDensity(w_i, -ray.direction()) *
                    E_i *
                    max(0.0f, w_i.dot(surfel->shadingNormal));
// Note in line above that shading.normal has become shadingNormal.

                debugAssert(radiance.isFinite());
            }
        }

        // Indirect illumination
        // Ambient
        // radiance += surfel.material.lambertianReflect * world->ambient;
// The comment above here is code that would be included in a classic
// Whitted raytracer to handle "ambient" light.

        // Specular
        if (bounce < m_maxBounces) {
            // Perfect reflection and refraction
            Surfel::ImpulseArray impulseArray;
            surfel->getImpulses(PathDirection::EYE_TO_SOURCE, -ray.direction(), impulseArray);

            for (int i = 0; i < impulseArray.size(); ++i) {
                const Surfel::Impulse& impulse = impulseArray[i];
                Ray secondaryRay = Ray::fromOriginAndDirection(
                                        surfel->position, impulse.direction).bumpedRay(RAY_BUMP_EPSILON);
                debugAssert(secondaryRay.direction().isFinite());
                radiance += rayTrace(secondaryRay, bounce + 1) * impulse.magnitude;
                debugAssert(radiance.isFinite());
            }
        }
    } else {
        // Hit the sky
        radiance = m_world->ambient;
    }

    return radiance;
}
```

```
// The next three procedures are completely analogous to the three above.
void App::pathTraceImage(float scale, int numRays) {
    int width  = window()->width()  * scale;
    int height = window()->height() * scale;
    if (isNull(m_currentImage) || (m_currentImage->width() != width) || (m_currentImage->height() != height)) {
        m_currentImage = Image3::createEmpty(width, height);
    }
    m_primaryRaysPerPixel = numRays;
    // Don't do multithreading when you're trying to debug!
    const int maxThreads =
#ifdef G3D_DEBUG
  1;
#else
  System::numCores();
#endif
    GThread::runConcurrently2D(Point2int32(0, 0), Point2int32(width, height), this, &App::ptrace, maxThreads);

    // Post-process
    shared_ptr<Texture> src = Texture::fromImage("Source", m_currentImage);
    if (notNull(m_result)) {
        m_result->resize(width, height);
    }
    m_film->exposeAndRender(renderDevice, debugCamera()->filmSettings(), src, m_result);
    // show(m_result); // display result in small window without scaling up to full size
}

}

void App::ptrace(int x, int y) {
    Color3 sum = Color3::black();

    for (int i = 0; i < m_primaryRaysPerPixel; ++i) {
        sum += pathTrace(debugCamera()->worldRay(x + rnd.uniform(), y + rnd.uniform(),
                                                 m_currentImage->rect2DBounds()), true);
    }
    m_currentImage->set(x, y, sum / m_primaryRaysPerPixel);
}
```

```
// the pathTrace procedure serves a role parallel to the rayTrace proc, but its details are, of course, different
Radiance3 App::pathTrace(const Ray& ray, bool isEyeRay) {
    // trace a ray into the world. “isEyeRay” tells you whether this is the first part of a path or somewhere deeper.
    // For deeper parts, we don’t record the emitted light, because that will have already been accounted for during
    // the direct-light computations

    Radiance3 radiance;

    shared_ptr<Surfel> surfel;
    float dist = inf();
    if (m_world->intersect(ray, dist, surfel)) {
        // this point could be an emitter...
        if (isEyeRay && m_emit) {
            radiance += surfel->emittedRadiance(-ray.direction());
        }

        // Shade this point
        if ((! isEyeRay) || m_direct) {
            radiance += estimateDirectLightFromPointLights(surfel, ray);
            radiance += estimateDirectLightFromAreaLights(surfel, ray);
        }

        if ((! isEyeRay) || m_indirect) {
            radiance += estimateIndirectLight(surfel, ray, isEyeRay);
        }
    }

    return radiance;
}
```

```
// The remaining portions of the code are essentially identical to the ones in the book,
// with only small modifications analogous to those in the rayTrace procedure.
Radiance3 App::estimateDirectLightFromPointLights(const shared_ptr<Surfel>& surfel, const Ray& ray){
    Radiance3 radiance(0.0f);

    if (m_pointLights) {
        for (int L = 0; L < m_world->lightArray.size(); ++L) {
            const shared_ptr<Light>& light = m_world->lightArray[L];
            // Shadow rays
            if (m_world->lineOfSight(surfel->position + surfel->geometricNormal * 0.0001f, light->frame().translation)) {
                Vector3 w_i = light->frame().translation - surfel->position;
                const float distance2 = w_i.squaredLength();
                w_i /= sqrt(distance2);

                // Attenuated radiance
                const Irradiance3& E_i = light->bulbPower() / (4.0f * pif() * distance2);

                radiance +=
                    (surfel->finiteScatteringDensity(w_i, -ray.direction()) *
                    E_i *
                    max(0.0f, w_i.dot(surfel->shadingNormal)));
                debugAssert(radiance.isFinite());
            }
        }
    }
    return radiance;
}
```

```
Radiance3 App::estimateDirectLightFromAreaLights(const shared_ptr<Surfel>& surfel, const Ray& ray) {
    Radiance3 radiance(0.0f);
    // Shade this point (direct illumination from AreaLights)
    if (m_areaLights) {
        for (int L = 0; L < m_world->lightArray2.size(); ++L) {
            shared_ptr<AreaLight> light = m_world->lightArray2[L];
            shared_ptr<Surfel> lightsurfel = light->samplePoint(rnd);
            const Point3& Q = lightsurfel->position;

            if (m_world->lineOfSight(surfel->position + surfel->geometricNormal * 0.0001f,
                                     Q + 0.0001f * lightsurfel->geometricNormal)) {
                Vector3 w_i = Q - surfel->position;
                const float distance2 = w_i.squaredLength();
                w_i /= sqrt(distance2);

                radiance +=
                    surfel->finiteScatteringDensity(w_i, -ray.direction()) *
                    (light->power() / pif()) *
                    max(0.0f, w_i.dot(surfel->shadingNormal)) *
                    max(0.0f, -w_i.dot(lightsurfel->geometricNormal) / distance2);
                debugAssert(radiance.isFinite());
            }
        }
        if (m_direct_s) {
            Surfel::ImpulseArray impulseArray;
            surfel->getImpulses(PathDirection::EYE_TO_SOURCE, -ray.direction(), impulseArray);
            for (int i = 0; i < impulseArray.size(); ++i) {
                const Surfel::Impulse& impulse = impulseArray[i];
                const Ray& secondaryRay = Ray::fromOriginAndDirection(surfel->position,
                                                                      impulse.direction).bumpedRay(RAY_BUMP_EPSILON);
                shared_ptr<Surfel> surfel2;
                float dist = inf();
                if (m_world->intersect(secondaryRay, dist, surfel2)) {
                    if (m_emit) {
                        radiance += surfel2->emittedRadiance(-secondaryRay.direction()) * impulse.magnitude;
                    }
                }
            }
        }
    }
    return radiance;
}
```

```
Radiance3 App::estimateIndirectLight(const shared_ptr<Surfel>& surfel, const Ray& ray, bool isEyeRay){
    Radiance3 radiance(0.0f);
    // Use recursion to estimate light running back along ray from surfel that arrives from
    // INDIRECT sources, by making a single-sample estimate of the arriving light.

    Vector3 w_o = -ray.direction();
    Vector3 w_i;
    Color3 coeff;

    if (!(isEyeRay) || m_indirect) {
        // Scatter has a new argument saying which direction the path is taking...
        surfel->scatter(PathDirection::EYE_TO_SOURCE, w_o, false, rnd, coeff, w_i);
        if (! w_i.isNaN()) {
            // see Veach, page 167.
            const float refractiveScale = square(surfel->etaPos / surfel->etaNeg);

            // the final "false" makes sure that we do not include direct light.
            radiance += refractiveScale * coeff *
                        pathTrace(Ray(surfel->position, w_i).bumpedRay(RAY_BUMP_EPSILON *
                                                                       sign(surfel->geometricNormal.dot(w_i)),
                                  surfel->geometricNormal), false);
            // there's no  "w_i.dot(surfel.geometric.normal)" because it's included in "scatter" for the lambert case, and
            // should NOT be there at all for the reflection or transmission case...
        }
    }
    return radiance;
}
```

The AreaLight Class (for the pathtracer project)

An area light is a polyhedral mesh that radiates uniformly in all outward directions from all points of the mesh. It's constructed via a factory method (`fromSampler`) that takes a sampler (which we'll describe in a moment) and a power for the area light and uses these to initialize the members `baseSampler` and `power`. The base sampler is just a wrapper around surface meshes: you give it a mesh, and it provides a method that lets you pick a point uniformly at random on the mesh. We'll use this in constructing area lights in our sample worlds.

```
#include "AreaLight.h"

AreaLight::AreaLight(void) { // protected constructor – not meant to be used
}

shared_ptr<AreaLight> AreaLight::fromSampler(const shared_ptr<SurfaceSampler>& baseSampler, const Power3& power) {
    shared_ptr<AreaLight> s(new AreaLight());
    s->m_baseSampler = baseSampler;
    s->m_power = power;

    return s;
}


// The radiance from a Lambertian emitter of area A is power/(4 pi A).
Radiance3 AreaLight::radiance() {
    return m_power / (m_baseSampler->surfaceArea() * pif());
}


Radiance3 AreaLight::power() {
    return m_power;
}


shared_ptr<Surfel> AreaLight::samplePoint(Random& r) {
    return m_baseSampler->samplePoint(r);
}
```

The SurfaceSampler Class (for the pathtracer project)

The surface sampler is just a wrapper around a mesh (represented by a vertex array and triangle array. When an instance is constructed, it computes the total area *A* of the triangle mesh by summing up the areas of individual triangles, and recording the running total in a summed area table. Then to sample a point, we pick a number *x* between 0 and *A*, and do binary-search in the summed area table to find the triangle whose addition made the area greater than *x*. We then pick a point at random from this triangle. The probability of picking any triangle to sample is therefore proportional to the triangle's area.

To pick a point at random in the triangle, we pick random barycentric coordinates uniformly at random, and use these to select a point of the triangle.

The header looks like this:

```
class SurfaceSampler : public ReferenceCountedObject {
protected:
    double              m_area;
    Array<float>        m_triangleSummedAreaTable;

    CPUVertexArray      m_vertexArray;
    Array<Tri>          m_triArray;

    GMutex              m_mutex;

    shared_ptr<Surface> m_base;

    void areaPrecompute();
    SurfaceSampler(const shared_ptr<Surface>& base);

public:

    static shared_ptr<SurfaceSampler> fromSurface(const shared_ptr<Surface>& baseSurface);
    virtual ~SurfaceSampler() {}

    static Point3 barycentricSample(Random& r);
    Point3 triangleSample(const Point3& baryCoords, int triangleIndex);
    shared_ptr<Surfel> samplePoint(Random& r);
    double surfaceArea();
};
```

The C++ code is fairly straightforward

```
#include "SurfaceSampler.h"
#include "G3D/Random.h"

SurfaceSampler::SurfaceSampler(const shared_ptr<Surface>& base) : m_base(base) {
    areaPrecompute(); // find the total area, and build a summed area table
}


shared_ptr<SurfaceSampler> SurfaceSampler::fromSurface(const shared_ptr<Surface>& baseSurface) {
    return shared_ptr<SurfaceSampler>(new SurfaceSampler(baseSurface));
}


// pick a point in the unit square; if x < y, reflect into the other half-square. That gives points
// uniformly distributed in the x >= y triangle. The barycentric coords of this point wrt the vertices
// are then x-y, y, 1-x.
Point3 SurfaceSampler::barycentricSample(Random& r){
    float x = r.uniform();
    float y = r.uniform();
    if (x < y) {
        x = 1.0f - x;
        y = 1.0f - y;
    }
    return Point3(x - y, y, 1.0f - x);
}


double SurfaceSampler::surfaceArea() {
    return m_area;
}

// pick a point uniformly from the triIndex triangle.
Point3 SurfaceSampler::triangleSample(const Point3& baryCoords, int triIndex) {
    const Tri& t = m_triArray[triIndex];
    const Point3& z = baryCoords[0] * t.vertex(m_vertexArray, 0).position + baryCoords[1] * t.vertex(m_vertexArray,
1).position + baryCoords[2] * t.vertex(m_vertexArray, 2).position;
    return z;
}
```

```
void SurfaceSampler::areaPrecompute() {
    double areaSoFar = 0.0;

    Surface::getTris(Array<shared_ptr<Surface>>(m_base), m_vertexArray, m_triArray);
    for (int i = 0; i < m_triArray.size(); ++i) {
        const Tri& t = m_triArray[i];
        double area = t.area();
        areaSoFar += area;
        m_triangleSummedAreaTable.append(areaSoFar);
    }
    m_area = areaSoFar; // total of all triangles
}


shared_ptr<Surfel> SurfaceSampler::samplePoint(Random& r) {
    // find a random point on triangle i, where
    // a random number 0 <= x < totalArea
    // satisfies SAT(i-1) < x <= SAT(i)

    double x = r.uniform(0.0f, m_area);
    int k = m_triangleSummedAreaTable.size();
    // invariant: SAT(mini-1) <= x < SAT(maxi), so mini < maxi
    int mini = -1; // we (conceptually) enlarge SAT so that SAT(-1) = 0.
    int maxi = k-1;
    int split = mini + (1 + maxi - mini) / 2; // obsn: mini < split <= maxi.
    while ((maxi - mini) > 1) {
        if (x < m_triangleSummedAreaTable[split]) {
            maxi = split;
        } else {
            mini = split;
        }
        split = mini + 1 + (maxi - mini) / 2;
    }

    Tri::Intersector intersection;
    intersection.cpuVertexArray = &m_vertexArray;
    intersection.tri = &m_triArray[maxi];

    const Vector3& B = barycentricSample(r);
    intersection.u = B.x;
    intersection.v = B.y;
```

```
    intersection.backside = false;

    alwaysAssertM(maxi >= 0 && maxi < m_triArray.size(), format("Index out of bounds: maxi = %d, m_triArray.size() = %d",
maxi, m_triArray.size()));
    const shared_ptr<Material>& m = m_triArray[maxi].material();
    return m->sample(intersection);
}
```

The World Class (for the pathtracer project)

A "world" is a representation of a scene on which a raytracer or pathtracer can act. Our World class is very simple: you pass a small integer *k* to the constructor, and it produces the *k*th predefined world. The world has several responsibilities, but the main one is to answer ray-intersection queries (i.e., find the first scene-point hit by a ray). Worlds get constructed by either adding certain geometries (Spheres, squares) or point lights, or square area-lights. The header file is this:

```
[...]
class World {
private:

    Array<Tri>                      m_triArray; // the geometric data in the scene
    Array<shared_ptr<Surface> >     m_surfaceArray;
    TriTree                         m_triTree;

    enum Mode {TRACE, INSERT}       m_mode; // TRACE mode is for ray/path tracing; INSERT mode is used during
                                            // construction of the world

private:

    void loadWorld0();
    void loadWorld1();
    void loadWorld2();
    void loadWorld3();
    void loadWorld4();
    void loadWorld5();
    void loadWorld6();
    void loadWorld7();

public:

    Array<shared_ptr<Light> >       lightArray; // the arrays of point- and area-lights
    Array<shared_ptr<AreaLight> >   lightArray2;
    Color3                  ambient;       // an ambient light color for use by the raytracer if wanted.
    static const int        m_nWorlds = 8; // there are only 8 choices

    World(int whichWorld);

    /** Returns true if there is an unoccluded line of sight from v0
        to v1.  This is sometimes called the visibilty function in the
        literature.*/
```

```
    bool lineOfSight(const Vector3& v0, const Vector3& v1) const;

    void begin();
    void end();

    void insert(const shared_ptr<ArticulatedModel>& model, const CFrame& frame = CFrame());
    void insert(const shared_ptr<Surface>& m);

// tools for adding geometry to a scene that's being constructed
    void addPointLight(const Point3& position, const Color3& powerInWatts);
    void addSquareLight(float edgeLength, const Point3& center, const Vector3& axisTangent, const Vector3& normal, const
UniversalMaterial::Specification& material, const Color3& powerInWatts);
    UniversalMaterial::Specification makeMaterial(const Color3& lambertColor, float lambertFraction, const Color3&
specularColor, float specularFraction, float shininess);
    void addSphere(const Point3& center, float radius, const UniversalMaterial::Specification& material);
    void addSquare(float edgeLength, const Point3& center, const Vector3& horizontalTangent, const Vector3& normal, const
UniversalMaterial::Specification& material);
    void addTransparentSphere(const Point3& center, float radius, float interiorEta, float exteriorEta,
                              const Color3& transmissiveColor, float transmissiveFraction,
                              const Color3& lambertianColor, float lambertianFraction,
                              const Color3& specularColor, float specularFraction, float shininess);

    /**\brief Trace the ray into the scene and return the first
       surface hit.

       \param ray In world space

       \param distance On input, the maximum distance to trace to.  On
       output, the distance to the closest surface.

       \param surfel Will be initialized by the routine.

       \return True if anything was hit
     */
    bool intersect(const Ray& ray, float& distance, class shared_ptr<Surfel>& surfel) const;

    const Array<shared_ptr<Surface> >& surfaces(); // get all the surfaces in the world
};
#endif
```

The World C++ code

This is a long and highly repetitive piece of code, so I've elided a great deal, writing "[...]" to indicate elisions.

```
World::World(int whichWorld) : m_mode(TRACE) {
    if (whichWorld >= m_nWorlds) {
        debugPrintf("Can't load world %d\n", whichWorld);
        exit(-1);
    }

    switch (whichWorld) {
        case 0:
            loadWorld0();
            break;
         [...]
        case 7:
            loadWorld7();
            break;
        default:
            debugPrintf("Can't load world %d\n", whichWorld);
            exit(-1);
            break;
    }
}


void World::begin() { // Begin the construction of a world by flushing everything
    debugAssert(m_mode == TRACE);
    m_surfaceArray.clear();
    m_triArray.clear();
    m_mode = INSERT;
}

// insert a G3D articulated model (i.e., the kind most often used) into the scene,
// using the given frame to position it.
void World::insert(const shared_ptr<ArticulatedModel>& model, const CFrame& frame) {
    Array<shared_ptr<Surface>> posed;
    model->pose(posed, frame);
    for (int i = 0; i < posed.size(); ++i) {
        insert(posed[i]);
```

```
    }
}


void World::insert(const shared_ptr<Surface>& m) {
    debugAssert(m_mode == INSERT);
    m_surfaceArray.append(m);
}


void World::end() { // finish the construction of a world, and set the m_mode to TRACE so that it can be used
    m_triTree.setContents(m_surfaceArray);
    debugAssert(m_mode == INSERT);
    m_mode = TRACE;
}


// Check whether v1 is visible from v0.
bool World::lineOfSight(const Point3& v0, const Point3& v1) const {
    debugAssert(m_mode == TRACE);

    const Vector3& d = v1 - v0;
    float len = d.length();
    const Ray& ray = Ray::fromOriginAndDirection(v0, d / len);
    float distance = len;
    Tri::Intersector intersector;

    return ! m_triTree.intersectRay(ray, intersector, distance);
}

// Use G3D's triTree intersection routines to determine where a given ray first hits the world.
bool World::intersect(const Ray& ray, float& distance, shared_ptr<Surfel>& surfel) const {
    debugAssert(m_mode == TRACE);

    Tri::Intersector intersector;
    if (m_triTree.intersectRay(ray, intersector, distance)) {
        surfel.reset(new UniversalSurfel(intersector));
        return true;
    } else {
        return false;
    }
}
```

```
const Array<shared_ptr<Surface>>& World::surfaces() {
    return m_surfaceArray;
}

// Load up world 0; contains a ground plane, a back wall, and a red sphere, a point light (for
// the raytracer to use) and an area light (for the pathtracer).
void World::loadWorld0() {
    Array<shared_ptr<Surface>> posed;

    begin();
    {
        // Point light

        addPointLight(Vector3(0, 2.9f, -2), Color3::white() * 80); // in Watts

        float squareLightEdgeLength = 3.0f;
        Power3 squareLightPower = Power3(Color3::white() * 60);

        // adds a light centered at the given point, with the two vectors as the edge-directions for the square.
       addSquareLight(squareLightEdgeLength, Point3(0.0f, 3.0f, -1.0f), Vector3(1,0,0), Vector3(0, -1, 0),
            // light itself is made from non-reflective material
            makeMaterial(Color3::black(), 1.0f, Color3::black(), 0.0f, 1.0f),  squareLightPower);

        {
            // A sphere, slightly to right, shiny and red.
            addSphere(Point3(1.00f, 1.0f, -3.0f), 1.0f,
                makeMaterial(Color3::fromARGB(0xff0101), 0.8f, Color3::white(), 0.2f, 1.0f));

            // And a  ground plane...
            // once again determined by a point and two direction vectors.
            addSquare(4.0, Point3(0.0f, -0.2f, -2.0f),  Vector3(1.0f, 0.0f, 0.0f),  Vector3(0.0f, 1.0f, 0.0f),
                makeMaterial(Color3::fromARGB(0xffffff), 0.9f, Color3::white(), 0.0f, 1.0));

            // And a  back plane...
            addSquare(4.0, Point3(0.0f, 2.0f, -4.00f),  Vector3(1.0f, 0.0f, 0.0f),  Vector3(0.0f, 0.0f, 1.0f),
                makeMaterial(Color3::fromARGB(0xbbbbff), 0.9f, Color3::white(), 0.0f, 1.0));
        }
    }
    end();
}
```

```
// Normal model-loading in G3D lets you position and rotate a model, but not scale it. Fortunately,
// ArticulatedModel provides a way to scale a model as you read it from a file.
static shared_ptr<ArticulatedModel> loadScaledModel(const std::string& filename, float scale) {
    ArticulatedModel::Specification s;
    s.filename = System::findDataFile(filename);
    s.scale = scale;
    return ArticulatedModel::create(s);
}


void World::loadWorld1() {

    Array<shared_ptr<Surface>> posed;
    UniversalMaterial::Specification areaLightMaterial;
    UniversalMaterial::Specification rightMaterial;
    shared_ptr<ArticulatedModel> squareLight = loadScaledModel("squarex8.ifs", 10.0f);

    begin();
    {
        addPointLight(Vector3(0, 10, -3), Color3::white() * 1200); // 1200 watts

        float squareLightEdgeLength = 10.0f;
        Power3 squareLightPower = Power3(Color3::white() * 600);
        addSquareLight(squareLightEdgeLength, Point3(0.0f, 10.0f, 5.0f), Vector3(1,0,0), Vector3(0, -1, 0),
            makeMaterial(Color3::black(), 1.0f, Color3::black(), 0.0f, 1.0f),  squareLightPower);

        {
            // A sphere, slightly to right, shiny and red.
            addSphere(Point3(1.00f, 1.0f, -3.0f), 1.0f,
                makeMaterial(Color3::fromARGB(0xff0101), 0.8f, Color3::white(), 0.2f, 1.0f));
        }
        // LEFT sphere, glass version
        {
            // small green glass sphere on left
            addTransparentSphere(
                Point3(-0.95f, 0.7f, -3.0f), 0.7f, // position, radius
                1.55f, 1.00f,                      // interior and exterior indices of refraction
                Color3::fromARGB(0xccffcc), 1.0f,  // Transmissive color and fraction
                Color3::black(), 0.0f,             // Lambertian color and fraction
                Color3::white(), 0.0f,             // Specular color and fraction
                1.0f);                             // shininess
```

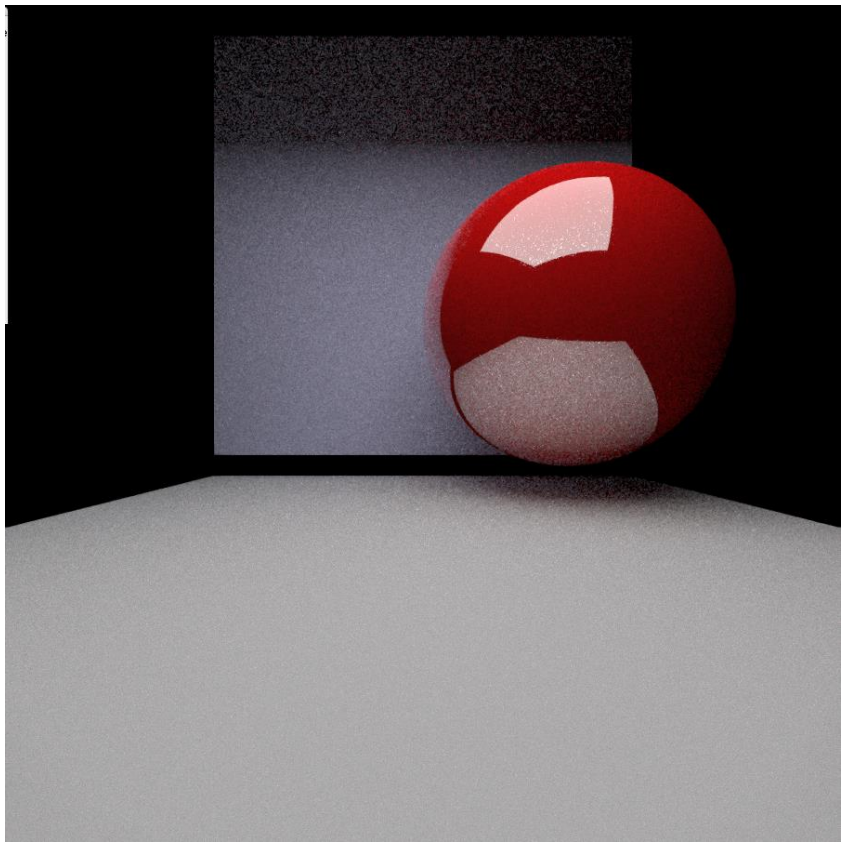
```
        }

        // And a  ground plane...
        addSquare(4.0, Point3(0.0f, -0.2f, -2.0f),  Vector3(1.0f, 0.0f, 0.0f),  Vector3(0.0f, 1.0f, 0.0f),
            makeMaterial(Color3::fromARGB(0xffffff), 0.9f, Color3::white(), 0.0f, 1.0));

        // And a  back plane...
        addSquare(4.0, Point3(0.0f, 2.0f, -4.00f),  Vector3(1.0f, 0.0f, 0.0f),  Vector3(0.0f, 0.0f, 1.0f),
            makeMaterial(Color3::fromARGB(0xbbbbff), 0.9f, Color3::white(), 0.0f, 1.0));
    }
    end();

}

//=============================================================================================================

void World::loadWorld2() {
[... other loadWorldX methods omitted ...]

//=============================================================

void World::addPointLight(const Vector3& position, const Color3& powerInWatts) {
    lightArray.append(Light::point("light", position, powerInWatts));
}


void World::addSquareLight(float edgeLength, const Point3& center, const Vector3& axisTangent, const Vector3& normal, const
UniversalMaterial::Specification& material, const Color3& powerInWatts){

    // radiance is power over pi * area
    Radiance3 squareLightRadiance = powerInWatts/ (pif() * edgeLength*edgeLength);

    UniversalMaterial::Specification areaLightMaterial = material;
    areaLightMaterial.setEmissive(squareLightRadiance);

    shared_ptr<ArticulatedModel> square = loadScaledModel("squarex8.ifs", edgeLength);
    // The square has only two triangles in a single mesh array; set the material for that to be
    // the one we've built
    square->rootArray()[0]->meshArray()[0]->material = UniversalMaterial::create(areaLightMaterial);

    // First, insert the square into the scene.
    Vector3 uNormal = normal / normal.length();
```

```
    Vector3 firstTangent(axisTangent / axisTangent.length());
    Vector3 secondTangent(uNormal.cross(firstTangent));

    Matrix3 rotmat(
        firstTangent.x, secondTangent.x, uNormal.x,
        firstTangent.y, secondTangent.y, uNormal.y,
        firstTangent.z, secondTangent.z, uNormal.z);
    // This is a matrix that rotates the x-axis to firstTangent; the y-axis to secondTangent; the z-axis to Normal
    // Because the square is defined in the xy-plane, this makes the plane of the posed square be the one
    // determined by the two tangent vectors
    CoordinateFrame cFrame(rotmat, center);
    insert(square, cFrame);

    // And now add it to the light list as well
    Array<shared_ptr<Surface>> posed;
    square->pose(posed, cFrame);
    for (int i = 0; i < posed.size(); ++i) {
        lightArray2.append(AreaLight::fromSampler(SurfaceSampler::fromSurface(posed[i]), powerInWatts));
    }
}

// shininess ranges from 0.0 (dull) to 1.0 (mirror)
UniversalMaterial::Specification World::makeMaterial(const Color3& lambertColor, float lambertFraction, const Color3&
specularColor, float specularFraction, float shininess) {
    UniversalMaterial::Specification material;
    material.setGlossy(specularColor * specularFraction);
    material.setShininess(shininess);
    material.setLambertian(lambertColor * lambertFraction);
    return material;
}


void World::addSphere(const Point3& center, float radius, const UniversalMaterial::Specification& material){
    shared_ptr<ArticulatedModel> sphere = loadScaledModel("sphere.ifs", radius);
    sphere->rootArray()[0]->meshArray()[0]->material = UniversalMaterial::create(material);
    insert(sphere, CFrame::fromXYZYPRDegrees(center.x, center.y, center.z, 0));
}


void World::addTransparentSphere(const Point3& center, float radius, float interiorEta, float exteriorEta,
    const Color3& transmissiveColor, float transmissiveFraction,
    const Color3& lambertianColor, float lambertianFraction,
```

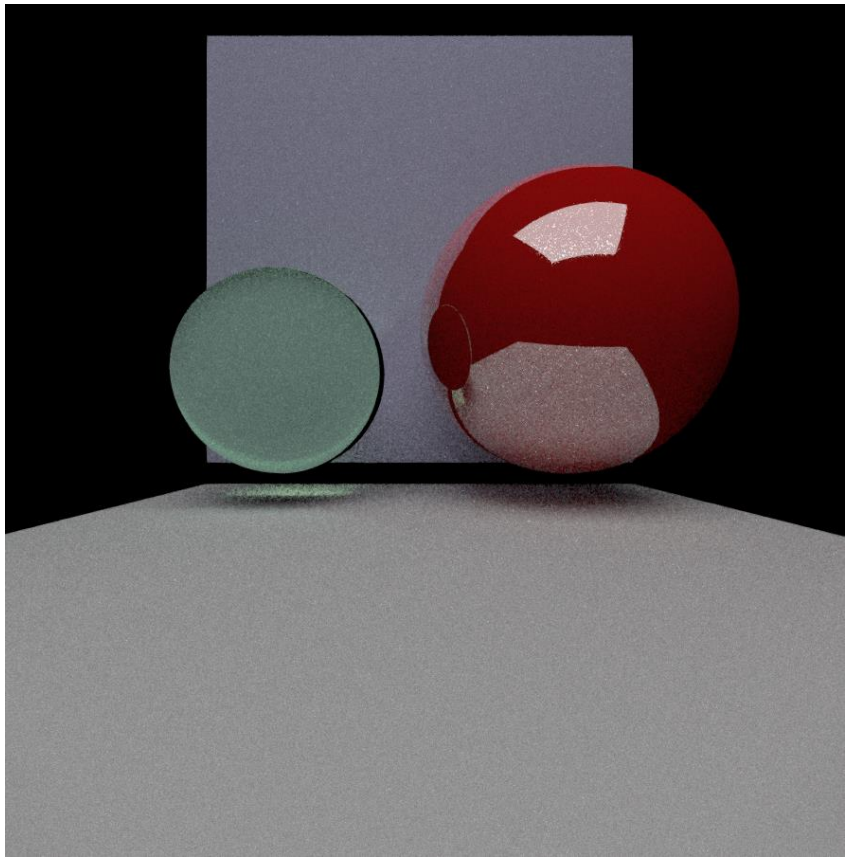
```
    const Color3& specularColor, float specularFraction, float shininess){

    shared_ptr<ArticulatedModel> sphere = loadScaledModel("sphere.ifs", radius);
    // Use the outside of the object as the interface into glass from air

    UniversalMaterial::Specification glassAir;
    glassAir.setGlossy(specularColor * specularFraction);
    glassAir.setShininess(shininess);
    glassAir.setLambertian(lambertianColor * lambertianFraction);
    glassAir.setEta(interiorEta, exteriorEta);
    glassAir.setTransmissive(transmissiveColor * transmissiveFraction);

    sphere->rootArray()[0]->meshArray()[0]->material = UniversalMaterial::create(glassAir);
    sphere->rootArray()[0]->meshArray()[0]->twoSided = true;

    insert(sphere, CFrame::fromXYZYPRDegrees(center.x, center.y, center.z, 0));
}


void World::addSquare(float edgeLength, const Point3& center, const Vector3& axisTangent, const Vector3& normal, const
UniversalMaterial::Specification& material){
    shared_ptr<ArticulatedModel> square = loadScaledModel("squarex8.ifs", edgeLength);
    square->rootArray()[0]->meshArray()[0]->material = UniversalMaterial::create(material);

    Vector3 uNormal = normal / normal.length();
    Vector3 firstTangent(axisTangent / axisTangent.length());
    Vector3 secondTangent(uNormal.cross(firstTangent));

    Matrix3 rotmat(
        firstTangent.x, secondTangent.x, uNormal.x,
        firstTangent.y, secondTangent.y, uNormal.y,
        firstTangent.z, secondTangent.z, uNormal.z);

    CoordinateFrame cFrame(rotmat, center);
    insert(square, cFrame);
}
```

As a guide to those who wonder what the worlds should look like when rendered, here are a few screen grabs; they're rendered with 15 primary rays per pixel:

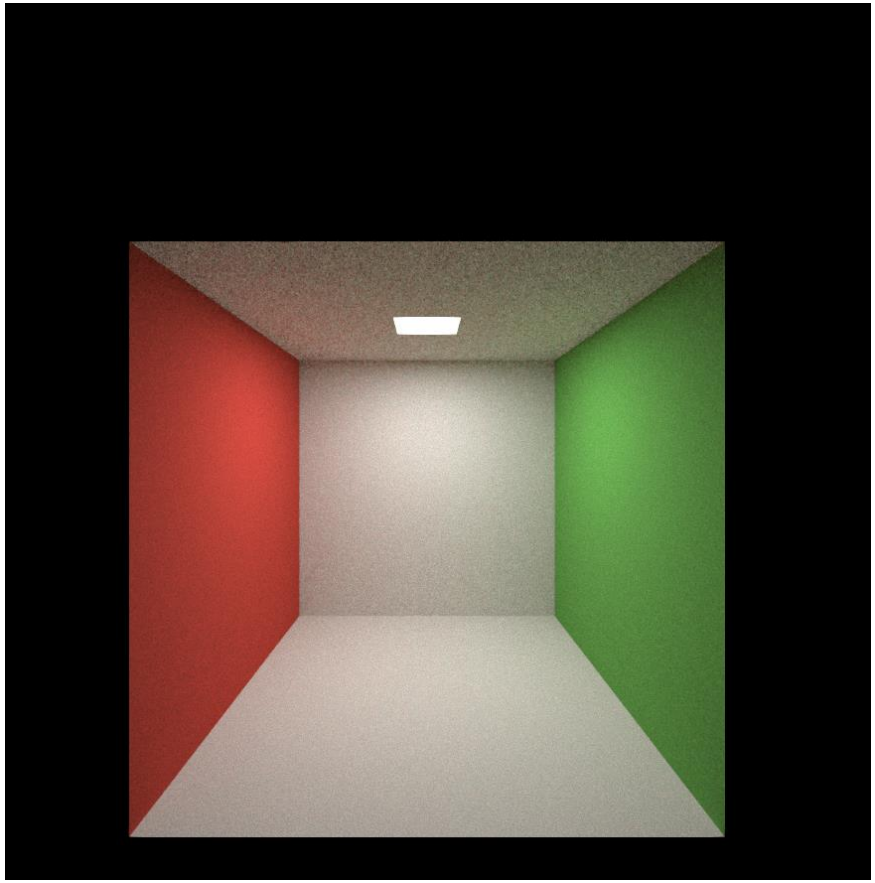
World 0:



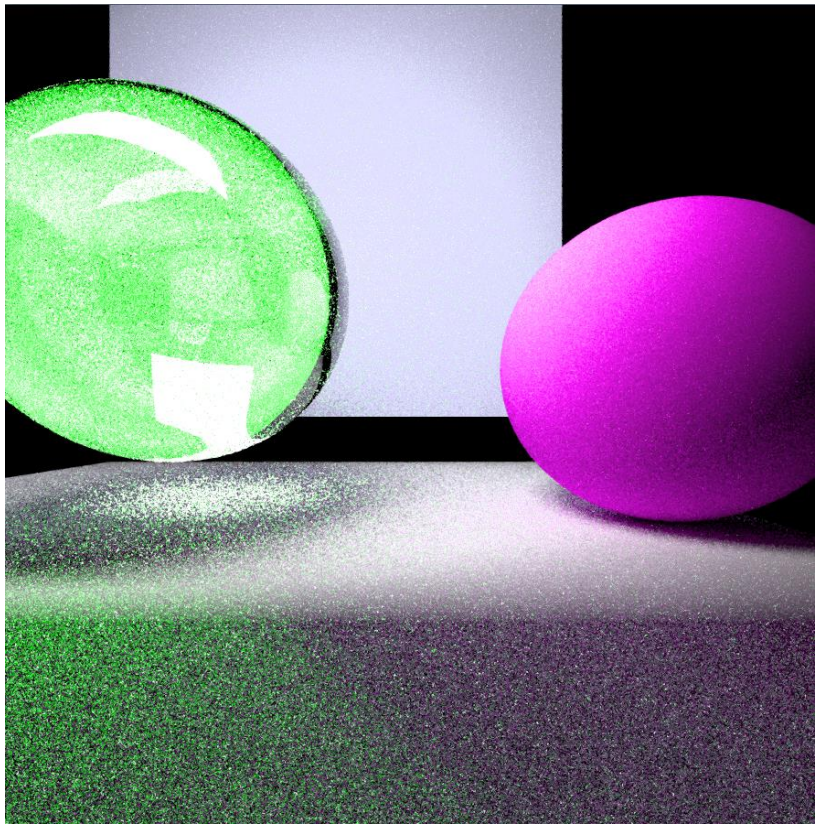
World 1:



World 2:

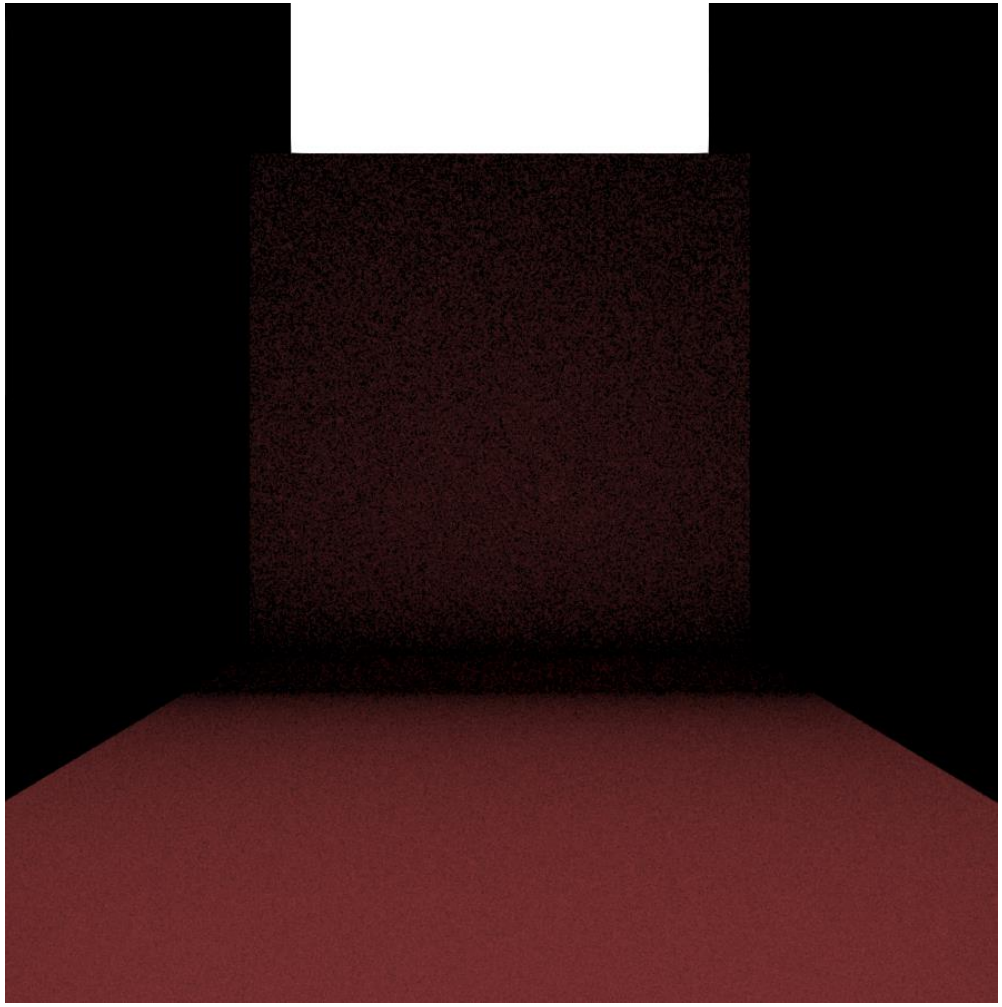


World 3:

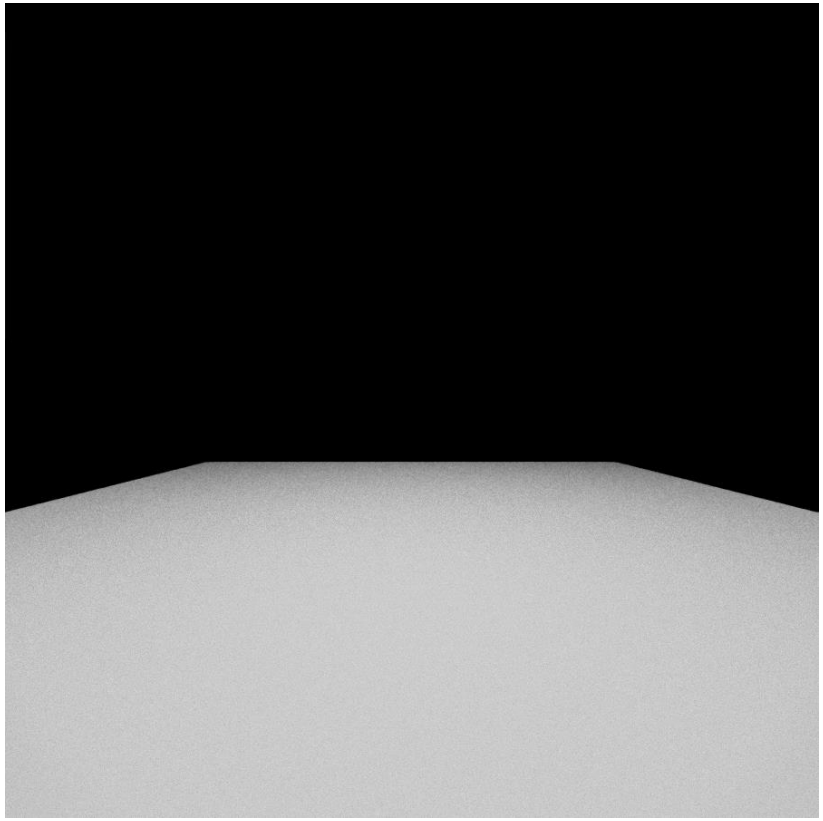


World 4:

World 5:



World 6 (which is just a light above a ground-plane – not a very interesting scene):



World 7: