Aalto T-111.4310: Vuorovaikutteisen tietokonegrafiikan jatkokurssi

Spring 2013 / Prof. Jaakko Lehtinen / TA Miika Aittala `first.last@aalto.fi`

Programming Assignment 2: Radiosity

Due Sunday March 31 23:59 – Note Preceding Easter Holidays

Now that the intro is over, your mission is to apply your ray tracer for rendering diffuse global illumination (GI) using a simple radiosity method. Given enough samples, this will result in pretty cool looking indirect illumination (Fig. 1). Your code will produce nice effects such as "color bleeding", i.e., the effect where a brightly lit surface will transfer some of its color to other, nearby surfaces through reflection.

The requirements are simple enough:

1. Compute direct irradiance due to an area light source (4p).
2. Compute subsequent bounces of indirect lighting by evaluating hemispherical gathering integrals at each vertex (6p).

Successfully completing this assignment is not complicated — in particular, the amount of code required for full credit is not more than two or three screens — but it will require you to understand Monte Carlo integration, including importance sampling and PDFs, the relationship of irradiance and radiosity, and the solid-angle-to-area variable change that is used for direct lighting. *Note that you will have to integrate your own ray tracer from Assignment 1 in order to shoot rays.*

1 Introduction

Radiosity methods, in their most general meaning, store and display lighting information using basis functions defined on the scene geometry. This has two main benefits: first, it allows one to perform expensive precomputation off-line and visualize the results in real time — this is the main point of view in this assignment — and second, they allow one to compute illumination at a coarse resolution and interpolate the result, saving one from having to recompute everything for every pixel. As you will notice during the debugging of your solver, indirect lighting tends to be smooth, which means it is often nicely amenable to interpolation. It should be noted that basis function techniques and lightmapping are not restricted to diffuse global illumination. E.g. ambient occlusion is often precomputed

Figure 1: Direct and one-bounce indirect illumination in the Sponza Atrium, computed using approx. 1000 hemisphere rays per vertex for the indirect bounce.

and stored in textures. *Lightmaps*, as seen in lots of games, are the most popular radiosity technique. See here for an example[1].

Our setting is not entirely realistic in that we'll be using vertices and linear interpolation over triangles as the basis functions. This is in contrast to storing lighting in textures, as is most often done in games. However, *the computations required to produce the lighting stored in the vertices are almost precisely the same.* In other words: if you can do this, you can do light maps if you're given a properly UV parameterized scene. And people still do vertex interpolation as well; see, for instance, this paper.

[1]Editor's note: funny enough, at Remedy we did stuff like this in the late 90s and people still market it as new and fancy.

2 Starter code

The starter code provides an environment you are already familiar with: `M, O` load and save a mesh, `WASD` move the camera, and left mouse button + mouse movement rotate the camera. Use the mouse wheel to change movement speed. You can load and save the current configuration by pressing `ALT`+number and load the configuration by pressing a number key alone.

This time there is also an area light source involved, see `AreaLight.hpp/cpp`. It is not part of the scene geometry but has its own drawing code instead. A slider controls its size. There is no intensity slider, but it's easy to add yourself should you want to. Controlling its placement is crude but effective: fly the camera where you want the light to be and press `SPACE`.

The GUI also features buttons to load and save the lighting solutions.

2.1 Prerequisites

You will need to bring in your own ray tracer from Assignment 1. In addition, you'll need the `formBasis` function for generating a local coordinate system in which to shoot the indirect gathering rays.

2.2 Provided functionality

In this assignment you'll be computing irradiance at the vertices. The code drop provides ready-made mechanisms for displaying meshes with per-vertex colors. You can try setting the colors to random values upon mesh load (`App::loadMesh`) to see what happens, but you shouldn't need to poke around there yourself.

The `Radiosity` class contains almost all the code you will be working on. Specifically: you need to fill in `Radiosity::vertexTaskFunc()`, the function that computes direct or indirect irradiance on a single vertex and stores it in its internal per-vertex vectors. In addition to this, you should only need to touch the `AreaLight::sample()` function.

The starter code provides means of running a radiosity computation job asynchronously in the background while allowing you to fly around in the scene and watch how rendering proceeds. This is implemented through the Framework's `MulticoreLauncher` helper class, which is used to asynchronously call `vertexTaskFunc` for each vertex in succession. You'll notice that the function gets passed a `RadiosityContext` instance that contains all the relevant information for running the computation, including the scene, the light source, the ray tracer, which bounce is currently being computed, where to store the results, the parameters that determine how many rays to use, etc.

Pressing `ENTER` will start the computation job. The default implementation just replaces the color of each vertex with its color coded normal vector and sleeps a bit so you can see how the computation proceeds. The colors output by the processing function into its result buffer are copied over to the display mesh every half a second.

The default implementation is single-threaded, but this can be changed by merely switching the commented lines in `Radiosity::startRadiosityProcess()`. You will need to make sure your code, including the ray tracer, are thread safe.

Before diving in, take a moment to familiarize yourself with the software architecture.

3 Requirements

3.1 Direct Illumination (4p)

At the first bounce, you should compute, for each vertex, the incident irradiance cast by the area light source.

The irradiance incident onto the point x is defined as the cosine-weighted integral over the hemisphere. However, when there is only one source of illumination, it is *much* better to use the already-familiar change of variables between area and solid angle and integrate over the surface S of the light source instead:

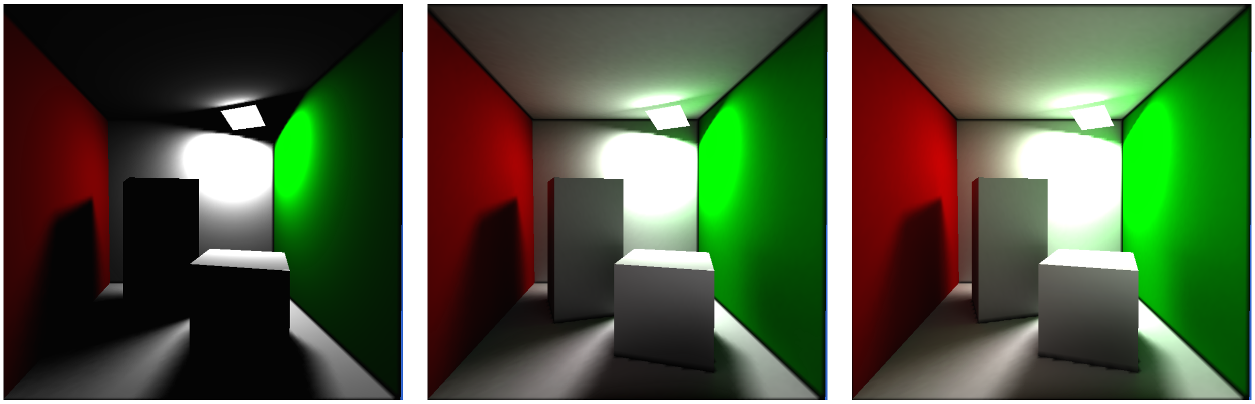
$$E(x) = \int_\Omega L_{\text{in}}(x \leftarrow \omega)\, \cos\theta \,\mathrm{d}\omega = \int_S E(y \to x) \frac{\cos\theta_l\, \cos\theta}{r^2}\, V(x,y)\,\mathrm{d}A_y. \qquad (1)$$

Here E is the radiance emitted by point y towards point x, which we assume to be constant over both x and y (i.e., the light emits equally into all directions from all points), θ is the angle between the incoming direction ω and the surface normal at x, and θ_l is the angle between the vector $\vec{yx}$ and the surface normal of the light. $V(x,y)$ is the visibility function, which you will evaluate using your ray tracer. Applying the Monte Carlo method to the rightmost integral results in

$$\int_S E(y \to x) \frac{\cos\theta_l\, \cos\theta}{r^2}\, V(x,y)\,\mathrm{d}A_y \approx \frac{1}{N} \sum_i \frac{E(y_i)\, \cos\theta_{l,i}\, \cos\theta_i\, V(x,y_i)}{r^2} \frac{1}{p(y_i)}, \qquad (2)$$

where the light source samples are drawn using from the PDF $p(y)$.

Turning the last formula into code is the first half of the assignment. This happens by filling in the first part of the commented-out section of `vertexTaskFunc`. Note how the result is stored into both the `m_vecResult` and `m_vecCurr` members of the context. The



Direct lighting only | Direct+1 indirect bounce | Direct+2 indirect bounces

Figure 2: TODO

result contains the colors used for display, and current holds the irradiance from the current bounce only.

For $p(y)$ we recommend the uniform distribution: unless you choose to do non-uniform emission, i.e., a textured light source, there is no need to do anything different. You need to fill in the `AreaLight::sample()` function that draws points on the light source and returns the points and value of the PDF. This really is not complicated: a purely uniform random solution will require one or two lines of code. You can use the built-in member `m_rand`, *but pay attention to that you will need to do something about it when you make your code thread safe in order to make multithreading work properly.*

When you've successfully implemented this part, you should, upon loading the Cornell box test scene, see a result that looks like Fig. 2, left.

In terms of the theory covered in class, you are now computing $\mathcal{P}\mathcal{T}E$, the discrete (projected) approximation of the direct irradiance: when you interpolate the results from the vertices into the triangles' interior and draw the mesh (not forgetting to divide by π and multiply by albedo, but worry not, this is already done for you in the drawing code), you've actually created a finite representation of the actual "infinite-dimensional" direct irradiance. Clearly, tessellating the mesh with smaller triangles will yield a better match to the actual irradiance. This corresponds to using a "finer" $\mathcal{P}$, but the transport operator $\mathcal{T}$ stays the same. Also note that there are two kinds of approximation going on: $\mathcal{P}\mathcal{T}E$ would be the projection of the *perfectly sampled* (noise-free) direct irradiance $\mathcal{T}E$, but we are approximating $\mathcal{T}E$ pointwise using Monte Carlo integration.

You can control the number of rays used for direct lighting using a slider in the GUI. Obviously, lower numbers will give you noisier results.

3.2 Indirect Bounces (6p)

Once the direct irradiance is computed and stored at the vertices, we turn to computing the next bounce of lighting.

In terms of operators and Neumann series, this is denoted $\mathcal{PTPT}E$: the projected direct irradiance $\mathcal{PT}E$ is transported once more through $\mathcal{T}$ and then projected again. Subsequent bounces merely repeat the same process. In practice this is simple: you shoot rays from each vertex into the hemisphere, see where they hit, interpolate the irradiance from the previous bounce from the vertices onto the hit point, turn it into outgoing radiosity, and use this as the incoming radiance. In mathspeak:

$$E_{n+1}(x) = \int_\Omega L_{\text{in},n}(x \leftarrow \omega)\, \cos\theta \, \mathrm{d}\omega = \int_\Omega L_{\text{out},n}(r(x,\omega))\, \cos\theta \, \mathrm{d}\omega \tag{3}$$

i.e., the irradiance for bounce $n+1$ is computed by integrating the outgoing radiance from bounce n over the hemisphere at each vertex x. Remember that $r(x,\omega)$ is the ray casting function that returns the point hit by the ray from x towards ω.

Like before, you now apply Monte Carlo integration and approximate the above by

$$E_{n+1}(x) \approx \frac{1}{N} \sum_i \frac{L_{\text{out}}(r(x,\omega))\, \cos\theta_i}{p(\omega_i)}. \tag{4}$$

It's not strictly required, but we strongly recommend you use the cosine PDF $p(\omega) = \cos\theta/\pi$ which you already know how to sample. Remember to be careful and see you have all the π and other factors accounted for!

Let's denote $r(x,\omega)$ by y below. The (not-too-difficult) trick is that in order to compute the outgoing radiosity $L_{\text{out},n}(y)$ for the point hit by the ray $x \to \omega$, we need to interpolate the results from the previous bounce. Fortunately, your ray tracer contains functionality for returning the barycentric coordinates for the hit point!

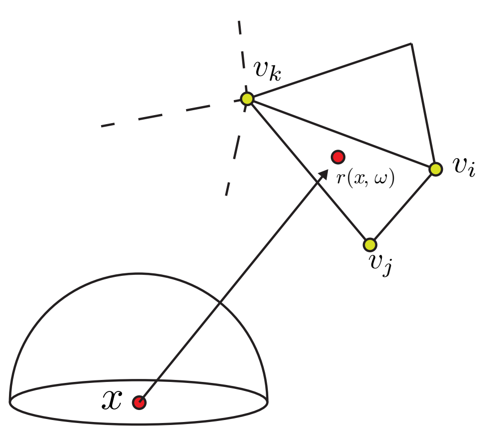


Figure 3: Schematic for interpolation of irradiance from the previous bounce.

Fig. 3 illustrates. Your ray from x has hit the red point that falls within the triangle that consists of vertices v_i, v_j, and v_k. Your ray tracer gives you the barycentrics for this point, i.e., you know α, β such that $y = \alpha v_i + \beta v_j + (1-\alpha-\beta) v_k$. Because you also know the irradiance at these vertices — you just computed it in the previous pass! — it's easy to interpolate it from the vertices using the same weights, i.e., $E_n(y) = \alpha E_{n,i} + \beta E_{n,i} + (1-\alpha-\beta) E_{n,k}$, where $E_{n,i}$ is the irradiance after n bounces at vertex number i.

The remaining task is to turn the irradiance into outgoing radiosity, which, as you know, is same thing as radiance for a diffuse surface: $L_{\text{out},n}(y) = \rho(y)E_n(y)/\pi$. If the triangle does not have a texture, we use a constant ρ from the material structure (`Material::diffuse`).

If the surface is textured, you have to sample the texture to find $\rho(y)$ as it now changes over the triangles. The barycentrics again come to rescue. Again, using the above recipe, you first interpolate the texture UV coordinates from the vertices to the hit point. (These are stored in the `t` member of the vertex structure.) Once you've interpolated the UVs, you multiply them by the width W and height H of the texture map, truncate the results to integers, wrap the coordinates to lie in the ranges $[0, W-1]$ and $[0, H-1]$, and read the color off the image using `Image::getVec4f()`. Note you only need the first three components.

The scenes provided in the starter package do not demonstrate the effect of textures on the color of bounced light very well. We will provide more scenes to better see this in a little while.

You will note the commented bit of the starter code again stores the results into `m_vecCurr` to be subsequently fed into the next bounce; however, the result is also added to `m_vecResult` that keeps track of the accumulated irradiance for all bounces.

You can separately control the number of rays used for indirect lighting in the GUI. Again, obviously, lower numbers will yield noisier results.

4 Extra Credit

There are tons of exciting ways to extend your solution! All points are subject to change depending on the quality of your implementation, but don't worry — we'll try to be generous.

4.1 Easy: Multithreading (1p)

If your code runs on all available cores, you get one extra point.

4.2 Better Sampling

If you choose to implement better than uniform light sampling, we recommend you insert GUI switches to control which method to use. These don't give a whole lot of points because they were already listed as extras in Assignment 1, but they *will* make your renderer produce much nicer results in the same amount of time.

4.2.1 Easy: Stratified Sampling (2p)

One extra point for stratified sampling of the area light source. Note that you will need to quantize the number of direct light samples so that you get a nice tiling of the light source area.

Another extra point for stratifying the indirect lighting computation as well (note that this will require you to implement the Shirley-Chiu mapping; stratifying the disk directly is hard).

4.2.2 Medium: Quasi Monte Carlo Sampling (2p)

Implement QMC light sampling using a suitable sequence (Halton, Sobol', etc.). Again, one point for light sampling, and another for the hemisphere in the indirect bounces.

4.3 Medium: Area Sampling (4p)

You'll notice that sampling lighting exclusively at the vertices results in hidden vertices "bleeding" darkness into visible areas. This is visible also in Fig. 2 in the corners[2]. This is not unexpected. A way to combat this is to let go of sampling the irradiance at the vertex only, but instead compute the vertex value by taking weighted averages of the incident irradiance over the support of the piecewise linear basis function defined by each vertex (cf. Fig. 4). In other words, you compute the irradiance of the vertex i by the double integral

$$E_i = \frac{\int_{\mathrm{supp}(B_i)} \int_\Omega B_i(x)\, L_{\mathrm{in}}(x \leftarrow \omega)\, \cos\theta \,\mathrm{d}A_x \,\mathrm{d}\omega}{\int_{\mathrm{supp}(B_i)} B_i(x)\,\mathrm{d}A_x}, \quad (5)$$

where $\mathrm{supp}(B_i)$ is the support of the piecewise linear basis function $B_i(x)$ of the ith vertex (i.e., just the set of all connected triangles). This formula looks hairy, but isn't: the inner integral over the solid angle is the same you're already computing; we've merely added the spatial weighting and averaging.

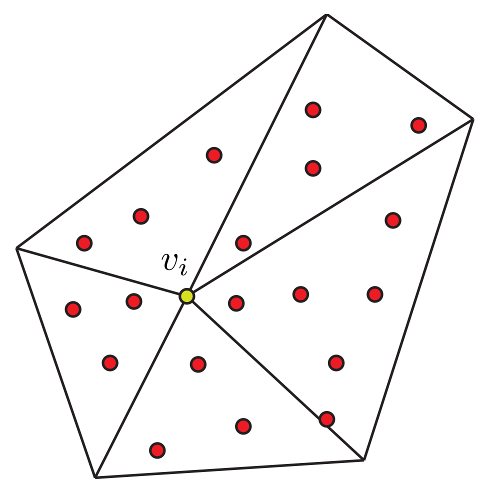


Figure 4: Distributing spatial samples (red) over the support of the basis function $B_i(x)$ associated with vertex v_i.

Again, like in Assignment 1, it is better to think of this as one 4D integral rather than two nested 2D integrals. This means you still shoot a number of indirect rays into the hemi-

[2] Strictly speaking the corner vertices are not "hidden", but it's too much to ask of the ray tracer to support cases where there is a polygon going right through the origin of the ray!

sphere, but now the starting point of each ray varies, and you perform additional weighting computations. It is best to implement this by looping over all triangles and choosing a number of uniformly distributed samples them. Naturally, you use each sample for updating the results for all basis functions whose support it overlaps. Note that the PDF $p(x)$ changes with the area of the triangle. You will need to be careful that everything adds up properly over all the connected triangles.

Using QMC for drawing the 4D samples is highly recommended.

4.4 Medium: Adaptive Tessellation (4p)

Start from a coarsely tessellated mesh, and subdivide triangles if you detect significant variation in the irradiances of the vertices during computations. This is not very hard as such, but will likely require a re-engineering of some of the starter code.

The scenes package contains an untessellated `sponza.obj` you can start from.

4.5 Medium: Shooting Iteration (6p)

The starter and solution code slavishly follow the Neumann series: you compute bounce by bounce, and gather the lighting to each vertex in succession. However, it is also possible to solve for the illumination by iterating in another fashion: repeatedly finding the basis function B_i with most “unshot” energy, “shooting” it to all the receivers, and setting the unshot energy of B_i to zero. The iteration converges when there is only little unshot energy left in the scene.

The shooting operation can be implemented by treating the triangles that belong to the shooting basis function as area light sources with piecewise linear emission[3].

See Michael Cohen et al. for details, and don’t be scared of terminology like *form factors*. They are merely the coefficients of the discretized radiosity matrix we saw in class. All that ever happens is the same as here: we have some radiosities/irradiances for the vertices, and we see how that irradiance propagates to other vertices/patches.

Implementing shooting is highly recommended: the result will visually converge much faster than the kind of gathering iteration we are doing in the basic form. If you choose to attempt shooting, implement a GUI switch for choosing which method to use. Also verify that your solution gives the same result as gathering if you let both run a long time.

[3]The solver used in Max Payne 2 used shooting iteration. The implementation was slightly different — I used a “hemicube” placed at the shooting polygon and rasterized the receivers into it — but the main idea is the same.

4.6 Hard: Lightmapping (8p)

Find a way to parameterize the scene using unique UV coordinates and compute the lighting at the texels instead of vertices. You will need to update the realtime renderer rather significantly to support lightmaps and rendering using them. You can look at Microsoft's D3DX utility library (version 9, not later!), or your favorite software package.

4.7 Very Hard: Meshless Hierarchical Radiosity (max. 15p)

Implement a radiosity solver based on the meshless hierarchical basis introduced in our 2008 SIGGRAPH paper and the earlier technical report. This means letting go of the triangles and textures altogether and computing irradiance at points scattered over the surfaces. The solution can be displayed either directly by interpolating from the scattered points to all pixels using the GPU with the aid of a "deferred shader", or by resampling the obtained solution back to vertices. The references contain all the necessary details.

A single level meshless radiosity solution (without hierarchy) gives 5 points. Implementing a hierarchy and an adaptive renderer that computes the solution finely only where you detect variation gives another 5. Implementing a GPU renderer that directly visualizes the meshless solution gives another 5 for a grand total of 15 points. (You can get 10 points by skipping the hierarchy.)

5 Submission Instructions

You are to write a `README.txt` that answers the following questions:

- Your name and student number at the beginning of the document.
- Did you collaborate with anyone in the class? If so, let us know who you talked to and what sort of help you gave or received.
- Were there any references (books, papers, websites, etc.) that you found particularly helpful for completing your assignment? Please provide a list.
- Are there any known problems with your code? If so, please provide a list and, if possible, describe what you think the cause is and how you might fix them if you had more time or motivation. This is very important, as we're much more likely to assign partial credit if you help us understand what's going on.
- Did you do any of the extra credit? If so, let us know how to use the additional features. If there was a substantial amount of work involved, describe what how you did it.

- Got any comments about this assignment that you'd like to share?

As with the previous assignment, you should create a single `.zip` archive containing:

- Your whole Visual Studio project, *including the source code!*
- The `README.txt` file.
- Artifacts: You should provide a screen shot of eat least the Cornell and Sponza scenes rendered using the same settings as the state files 1 and 2 provided with the code distribution. Press `Alt+PrtScr` to capture a PNG into the current directory.

  In addition, impress us with nicely lit scenes with interesting camera angles, using either the provided geometries or something else! You can find lots of `.obj`s online.

Submit it online in Optima by **March 31 23:59**.