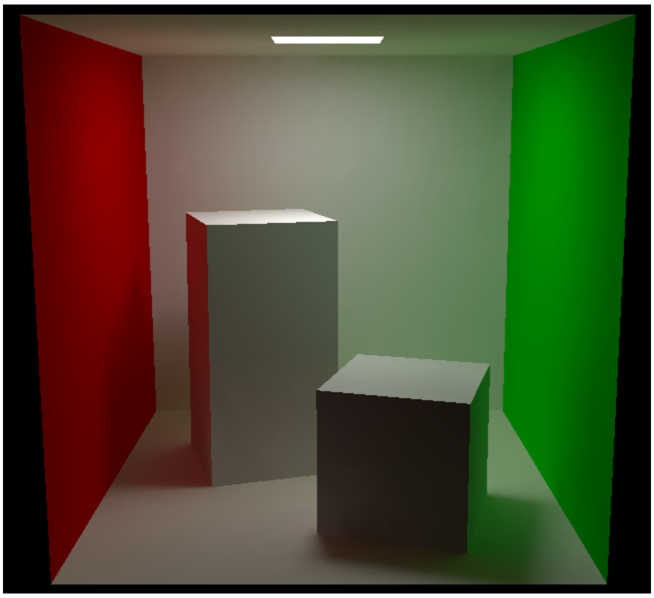
Technical University Munich
Department of Computer Science
Informatik 15
Computer Graphics & Visualization

Tobias Boldte *München, Mars 2004*

Graphics Programming and Applications (Assignment 4)



0 Topics

- Radiosity calculation
- Offscreen rendering
- Discontinuity meshing

1 Background

A *radiosity calculator* can be structured into many parts, each one performing a very special and separate task, like reading the scene description, calculating the form factors between the patches (scene polygons with different reflectance and light emittance), calculating the patch radiosities, using a discontinuity meshing (to increase the number of patches in regions with too large radiosity discontinuity), rendering the calculated scene (patches with radiosities) to visualize the result and many more. The key to designing a good radiosity calculator now is to identify the interacting entities of the program and to construct an object-oriented program skeleton that allows efficient implementation and flexible extension of functionality. The active parts of the code mentioned first, finally must be configured to the object implementations at the appropriate level in the object hierarchy. In this assignment, it's your task to improve a given skeleton for radiosity calculation.

You find the C++ source code of this skeleton on the GraPA Homepage. If you are not comfortable with its object design or the programming language, you are free to completely re-implement it, as long as the following requirements are met:

- The radiosity calculator is written in a pure object-oriented programming language, like C++, Java, Modula-2, Eiffel. Because object-orientation is a central point in this assignment, potentially mixed languages like C programs within C++ class wrappers, Lisp, ADA, Prolog, Pascal, Fortran and Basic are not allowed.
- Your implementation is able to calculate the radiosities for the given reference scenes from the data directory correctly and without complaints.

The base version of the skeleton for radiosity calculation compiles automatically by issuing the make command. According to the platform you are working on, the appropriate compiler is chosen. If you get an error message here, you have to take a look at the settings in the Makefile.

If all went well, you now have an executable radiosity.<arch>_<opt>, where <arch> denotes your system architecture and <opt> gives you a hint about the compiler options used. There are two other make targets for compiling a verbose debugging version (make dbg) or an optimized version (make opt). If you run the program without arguments, it will tell you how to provide it with the filename for the scene.

When the program is started, it opens an Inventor viewer that allows you to regard the read scene. Only the patches that emit light (these are the light sources of the scene) can be seen. The others are drawn black. If you change to the input mode and press <enter>, the first radiosity calculation is started. When it has finished, you can view the lit scene. When you press <w>, the calculated scene is saved to <filename>_RESULT.iv, so that you can view it later again with an Inventor viewer (ivview...).

2 Description

As you can see, the radiosity discontinuity in some regions is very large because the scene is divided in too large patches. It is your task to invent a meshing class that enables you to subdivide the patches. You should start an initial subdivision before the radiosity calculation is started. It should be based on the area of a patch: all patches larger than a certain area have to be subdivided to smaller ones before first radiosity calculation. Hint: The best method for triangle subdivision is to create 4 new triangles out of one.

Furthermore your program should be able to do a discontinuity meshing based on the calculated radiosity values from the last calculation. This meshing strategy is needed to subdivide two neighboured patches if the discontinuity between their radiosity values is too large (Hint: neighboured patches should be patches with the same material that share two vertices. The angle between their normals should not be larger than 20 degrees.) Of course, you have to keep the neighbour informations. Otherwise the complexity for a new meshing would be n^2 for n patches.

Every time you restart the radiosity calculation after such a radiosity refinement you will get a better result. When you start increasing the number of patches in this way, you will see that the calculation starts taking too much time. To fasten up the radiosity calculation, your second task is to optimize the form factor calculation (complexity n^2 for n patches) in the class HemiCube. For that you have to use the graphic hardware for the projection of the other patches onto the HemiCube of one patch. (The class ZBufferCube may not be used any more in your final implementation). The rendering has to be done in an offscreen buffer as provided in GLOffscreen.h. Of course you can also write your own offscreen buffer class if you are not comfortable with this implementation.

In the base version the class Scene provides you only with the function showResult() that enables you to create a simple Inventor scene out of the radiosity results. This scene shows all patches with their radiosity as emissive color. For getting good results you have to supplement a function that creates you a smooth shaded version of this scene. Your task is to create this function and enable the user to toggle between the old and the smooth shaded version. Hint: as you can't create smooth shaded polygons with different emissive colors in their vertices, you have to use diffuse colors for the radiosities in common with another light model.

The `src` directory provides you with:

- **main.C**: This is the main program control. It creates an Inventor viewer with a simple key event handler. The viewer gets the currently shown scene from class Scene.h. If you uncomment the BATCH_ -define, you can get the calculation times of your scene calculations. The calculator now reads an Inventor scene, calculates the radiosities and writes back the calculated Inventor scene to disk. You can use the batch version to get the calculation time your program needs. For that you have to tell the program how many radiosity refinements (discontinuity meshing + new radiosity calculation) have to be done.
- **Scene.h**: This class controls all functions related with the scene shown by the Inventor viewer in main.C. It reads the scene from file `<input>.iv` and writes the calculated scene to the file `<input>_RESULT.iv`. It is also used as interface between the viewer as scene renderer and the functions for Radiosity calculation as it is able to convert the Inventor format into the format used in class Radiosity.h and vice versa.
- **Radiosity.h**: This class is used for radiosity calculation.
- **FormFactorCube.h**: This class calculates the delta formfactors for a HemiCube of a certain resolution. One delta formfactor is the formfactor value belonging to one pixel of the HemiCube.
- **ZBufferCube.h**: This class realizes a simple projection of patches onto the buffers (top and four sides) of the Hemicube of one patch. -This class has to be replaced by using OpenGL commands in subclass HemiCube.h. It may not be used any more in your final version !
- **HemiCube.h**: Used for formfactor calculation. Creates a two dimensional array with the formfactors for all patches.
- **Vertex.h**: This class represents a Vertex – unsurprisingly. It encapsulates the coordinate values (x,y,z) of a vertex. The class functions allows you to move them or rotate them around the coordinate axis. But you can also use this class for handling vectors. For example, if you want to move a vertex by a vector, you create two Vertex objects and add simply their coordinate values. (This is not a clear implementation, but it is easy to work with).
- **Color.h**: This class represents an RGBA color.
- **Patch.h**: In this class all patch informations needed for radiosity calculation are contained - a Vertex array for the vertices of the patch and its color informations: its emittance, its reflectance and its radiosity.
- **GLOffscreeen.h**: This class provides you with the creation of an offscreen buffer you can use for offscreen rendering. It is not used in the base version.

While you are extending the base version, we like to remind you once more, that *the code has to be commented completely*. This includes the code you obtained with the base version ! If you do not comment the code completely, you will loose points (see the evaluation criteria at the end). The comment format must be usable for the `doxygen` source code documentation system. Try a `make doc` to see how manual pages and HTML documentation is generated automatically from these comments.

The final radiosity calculator must be able to

- create patches out of the patches from the read scene that are not larger than a given size (For the given reference scenes they should not have an area larger than 10000). size out of the read scene.
- calculate the radiosities for these patches correctly. The needed form factors have to be calculated by rendering in an offscreen buffer.
- perform a radiosity refinement. The user should be able to start a new refinement (based on discontinuity of calculated radiosities) by pressing `<enter>`.
- show the calculated scene in a smooth shaded version. The user should be able to toggle between the simple version (whole polygon is shown in one color) and the smooth shaded version.
- correctly light the reference scenes in the `data` directory and the scene myScene.iv you created on your own (see the evaluation criteria at the end). You needn't copy the result files you get to the working directory - we will generate them anyway with your calculator.

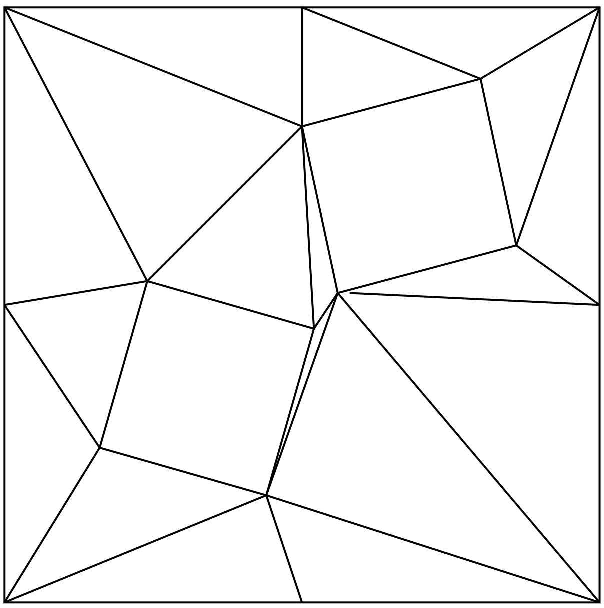
The calculation times for the reference scenes should be about the calculation times we got with our solution (see section 5).

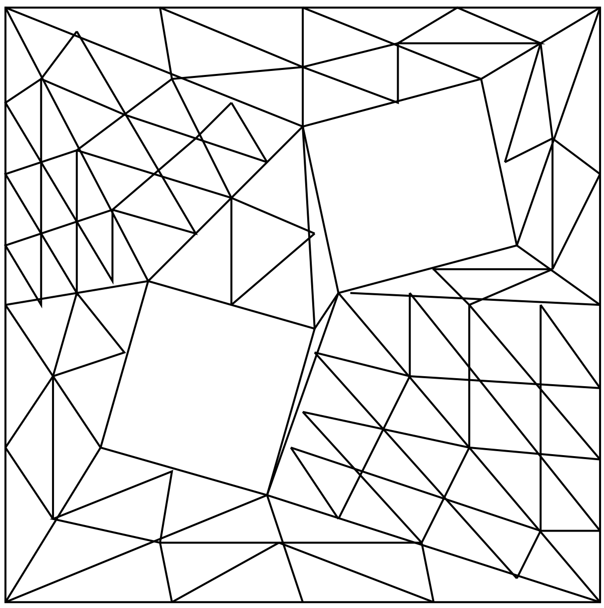
3 Requirements

This section will give you some hints how to realize the new functions for your radiosity calculator. It won't explain how radiosity works. So please also remember to consult your script of the *Computer Graphics* lecture !

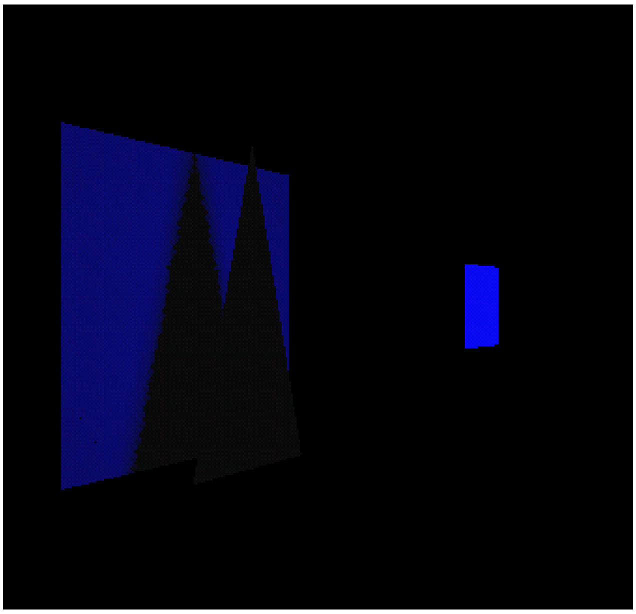
How to do it

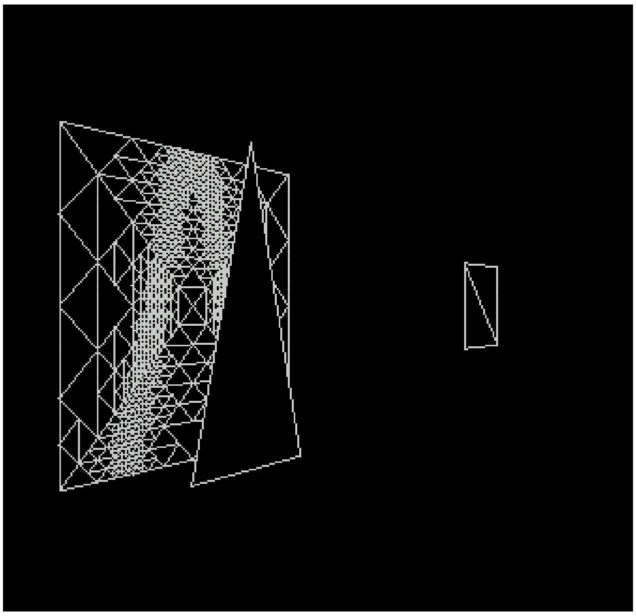
Your first task is, as described abouve, to invent an initial patch meshing that allows you to subdivide the patches of the read scene. After this meshing the area of all patches in the list (or any other data structure that is efficient) have to be smaller than a given size. The images below show the floor of the scene `REF_8.iv` before and after such a meshing:





In your second task you should invent a discontinuity meshing that depends on the radiosity discontinuities of the previous calculation. It is needed to increase the number of patches in regions with shading lines as in the scenes `REF_1.iv` and `REF_2.iv` as you can see in the image results:

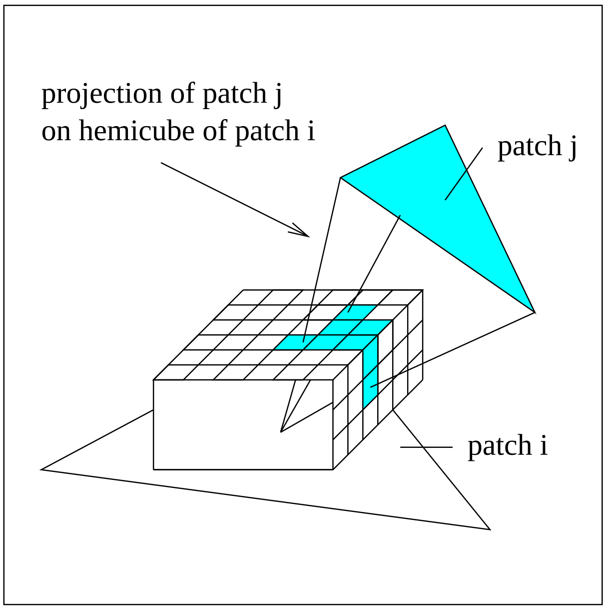




Both meshing strategies should be realized with the following restrictions: The subdivision level of two neighboured patches may at least differ in one level: It should not be allowed that a patch has more than two neighbours on one side. The neighbouring informations you have to calculate before the first meshing have to be kept and supplemented in every subdivision, because a new calculation of these informations (needed for the discontinuity meshing) would take too much time after a few refinements.

In the meshing class you should also get the information, which vertices are shared by which patches. For that you could also change the data structure of the patches and replace the vertex informations of one patch by pointers (iterators) on vertices in a separate vertex list (in which every vertex that is shared by several patches is only listed once). You need these informations to create the smooth shaded version of the scene. There you can take the mid radiosity color of all neighboured patches as vertex color in the shown inventor scene.

Your last task is to implement a new, more eficcient form factor calculation by rendering in an offscreen buffer. For that you have to use the strategy of calculating the form factors by using a hemicube. In this strategy a hemicube is placed on the top of a patch. All the other patches of the scene are projected on the four sides and the top of the hemicube. Its center lies in the middle of the patch:



You get the form factor between patch i and patch j by summing up the delta form factors that correspond to the pixels of the projected patch on the hemicube. These delta form factors are precalculated for a hemicube of given resolution in class FormFactorCube.
You can realize the projection on the hemicube by rendering the scene from the view from one patch (eyepoint in the center of the patch and viewpoints in the directions of the top center and centers of the four sides of the cube) five times in an offscreen buffer.

To get the information which patch is seen in a certain pixel you can store the patch numbers (number of patch in list) as RGB colors: for example the upper byte in the red component and the lower byte in the green component.

In order to verify the correctness of your implementation, you

- find input files named `REF*.iv` in the `data` directory along with the resulting image computed with the reference implementation. They can help you to identify the most common errors. Your implementation must be able to render all `REF`-files correctly ! The file `REF_1.iv` may help you to find out whether the projection on the top side of your Hemicube in form factor calculation works well. The file `REF_2.iv` shows you whether the projection on the sides of your Hemicube is implemented correctly. Both refence files can be used to prouve the correctness of your discontinuity meshing.
- should investigate the calculation time of your implementation with the `time` tool and compare it with the timings of the `REF`-files (see section 5).

How not to do it

Please do *not consider* the following to be a good idea:

- *Hacks* to “speed up” the existing code. We know that it’s not as efficient as it could be in some places, but for the sake of clarity it was done this way ! Detecting optimization potential there is the task of your friendly compiler.
- *External code* fragments, programmed in a non-object-oriented language for the sake of “super speed” like C-hacks, Fortran library calls, Assembler blobs. The purpose of teaching the application of object-oriented programming techniques forbids it.

- *Unknown code behavior*: Apart from trivial demands like dynamic memory allocation, you should take care of the efficiency of your code in the sense, that you analyze the behavior for different scene complexity. The scene complexity (number of patches for next radiosity calculation) augments automatically with each refinement. To give you a sense of run-time requirements, compare the timings in section 5 for each scene in the `data` directory.

4 Scene file format and restrictions

This section describes the format of the scene files. It is based on the Inventor file format but does not support all Inventor nodes. You may extend the format, but the old format has still to be supported.

patch description The patches have to be described with the following Inventor fields: The Vertices of the patches are stored in an array of Coordinate3.point. The vertices of the patches are set in an array of IndexedFaceSet.coordIndex. Their emittance and reflectance have to be described as Materials: the emittance of light sources as emissive color and the reflectance of the object patches as diffuse color. To get clear light sources, the diffuse color of a light source (patch with color emittance) should be set to 0.

restrictions The scene should be desribed with a continuous meshing: the vertices of neighboured patches should be shared.
We have to prevent that patches can be seen from their back side in form factor calculation. That means that from the front facing side of every patch there can only be seen front facing sides of other patches. All back facing patches have to be behind front facing ones from that view.

5 Reference timing averages

As the calculation time depends on the scene complexity in the radiosity calculation, it depends of course on the used parameters. So please use the parameters given in class Radiosity: a hemicube resolution of 150 (150*150 pixels on top and 150*75 pixels on the sides), the maximal size of 4000 for all patches after initial subdivision, and a threshold of 0.03 for the discontinuity meshing. When you use these parameters, the number of patches in the first calculation (after initial patch meshing) and in the refinement calculations (after discontinuity meshing) should be about the same.

You find the `SOLUTION_` files in the `data` directory. They show the calculated scenes without and with smooth shading. The calculation time for the smooth shaded version and the simple version is the same.

The following table lists the average calculation times. The SOLUTION_ files have been calculated with the batch version of our solution implementation. It has been compiled with compiler optimization turned on. On an SGI workstation, the command `hinv` tells you about the hardware installed. On Linux systems take a look in `/proc/cpuinfo`. The command `glxinfo` tells you which graphic card you are working on.

reference scene	REF_1	REF_2	REF_3	REF_4	REF_5	REF_6	REF_7	REF_8
number of refinements	5	3	3	2	0	0	0	4
number of patches								
for first calculation	22	76	368	204	1904	2048	1532	829
in first refinement	70	262	638	681				2068
in second refinement	247	898	1244	2049				3658
in third refinement	718	2311	2882					5254
in fourth refinement	1915							6985
in fifth refinement	4942							
wallclock seconds on								
Intel P4 2GHz Linux with a GeForce4 Ti 4800 SE	64	21	30	19	12	15	8	305
Intel P4 2GHz Linux with a GeForce FX 5950 Ultra	45	16	23	16	10	11	7	228
Intel P4 2GHz Linux with a GeForce FX 5900 Ultra	45	16	23	15	9	10	6	249

6 Evaluation criteria

You can get a maximum of 20 points for this assignment if the following criteria are satisfied:

4 points	Both the base code and the code written by you is fully commented in clear English language. The comments explain in detail how the algorithm has been implemented. The comments are formatted to provide input for the `doxygen` source code documentation system.
2 points	The calculator is able to do an initial patch subdivision. For that it creates a list (or vector...) of patches about a certain size out of the read list of patches.
3 points	Your program provides a radiosity calculation based on a discontinuity meshing. The meshing strategy is implemented efficient (neighbouring informations are only calculated once before starting the initial subdivision and supplemented in each subdivision.
5 points	The form factor calculation is realized by an efficient rendering in an offscreen buffer.
2 points	Your program is able to show the calculated scene with smooth shaded patches. The radiosity discontinuities are no longer visible. The information which patches share the same vertices is only calculated once before starting the initial subdivision and supplemented in each subdivision.
2 points	There is a file `myScene.iv` in the `data` directory that shows a different scene from the scenes of all GraPA participants. It has to be designed with the constraints described before (or your program has to be adapted in a way that it's still able to read scenes of described old format). The scene has at least a ceiling, a floor, three walls, two objects and two light sources with different light colors. The radiosities of the scene can be calculated correctly with your radiosity calculator.
2 points	The calculation times for the `REF*.iv` files with your extended program is not significantly longer than the reference averages.

A Literature

- Cohen, Wallace, *Radiosity and realistic image synthesis*, Academic Press, 1993
- Sillion, Puech, *Radiosity & global illumination*, Morgan Kaufmann, 1994
- Watt, Alan, *3D-Computergraphik*, Addison Wesley, 2002

X Credits

This assignment is part of the practical course “Graphics Programming and Applications (GraPA)” developed at the Computer Graphics Group, University of Erlangen-Nürnberg and Computer Graphics & Visualization Group, Technical University of Munich ©1999-2004.

This assignment has been created by Tobias Boldte, `boldte@in.tum.de`, student of Technical University Munich.