Global Illumination Methods for Ray Tracing Spring 2011

Joe Cross
Undergraduate Student
Dept. of Mathematics
UMass Dartmouth, Dartmouth MA 02747
Email: jcross@umassd.edu

Abstract

This project's focus was on two problems. First, to gain a better understanding of how the many parts of a ray tracer work together, and how design decisions can help and hurt extensibility. Second, to understand the challenges faced in efficiently coding a Global Illumination method, and comparing the advantages and drawbacks of different solutions; ideally, to attempt to implement one of these solutions into a working skeleton ray-tracer. Although the first goal was achieved, much of the time allocated for the second goal was stolen by setbacks stemming from unfamiliarity with Linux and C++.

1 Global Illumination: What is it?

1.1 Defining the problem

Global Illumination refers to the group of algorithms used to make lighting more realistic in 3D scenes. Shadows are a good example of global illumination, because when rendering a shadow from one object, it can potentially affect the (global) set of all other objects in the scene. Rendering techniques that use global illumination methods are both more realistic than those using direct lighting techniques, and more computationally expensive. The GI algorithms compared in this project model diffuse inter-reflection, and, except for radiosity, also model specular reflection.

1.2 Techniques used to address the problem

Because ray tracing is computationally expensive, my go-to language of Python would be very suboptimal for this project. Interpreted languages are good for quick code projects and those involving broad, general use-cases. This project however, had very strict definitions and orders of magnitudes more calculations than any project I'd worked on before. Therefore, a compiled language was needed. Writing in C++ on a Linux distribution, an object-oriented paradigm was decided on for its ease of extension and organization. After deciding on radiosity as a GI method, I worked on converting a Python Finite Elements Method code into C++ with appropriate bindings for the project. At the suggestion of my advisor, Professor Adam Hausknecht, I began my work from a barebones raytracer written to accompany the book "Ray Tracing from the Ground Up" by Kevin Suffern.

2 Process

2.1 C++

My first task was to re-familiarize myself with coding in a typed language. I decided to take this time to also learn how to navigate my way around Ubuntu, a Linux distribution. After some help from Professor Hausknecht with library management and how to update and build packages on Linux, I began reviewing the C++ barebones raytracer using the Eclipse IDE. Compiled languages have more strict function formats than I was used to with Python. Overloading the "+" operator for the C++ Vector3D class and the vector python class were as follows:
C++:

```
Vector3D operator+(const Vector3D &lhs, const Vector3D &rhs){
    return Vector3D (lhs.x+rhs.x, lhs.y+rhs.y, lhs.z+rhs.z);}
```

Python:

```
class vector(object):
    ...
    def __plus__(self, other):
        return vector(self.x+other.x,self.y+other.y,self.z+other.z)
```

After getting used to the difference in coding syntax and the C++ workflow process of "code, compile, run" instead of the Python process of "code" -I used an interactive environment for most testing on my last project, essentially running code changes as I typed them- I was ready to dive into the actual ray tracer's code.

2.2 Ray-Tracing Basics

2.2.1 Extending Renderable Shapes

I began by inspecting the hit detection method of a class Sphere, with parent class GeometricObject. GeometricObject defines a method virtual bool hit(ray, t, sr) and each child class of GeometricObject provides the implementation for this method. This is the core of any renderable object's visibility- this function says "does the Ray object 'ray' with minimum interaction time 't' collide with this object?" This code is available in Appendix A. To find the formula for ray-sphere interaction, we first have the formula for a point p on a ray with origin o for t = 0:

$$p = o + td$$

and the vector equation for a sphere with center c and radius r:

$$(p - c) \cdot (p - c) - r^2 = 0$$

substituting our equation for a ray into our equation for a sphere:

$$(o + td - c) \cdot (o + td - c) - r^2 = 0$$

After expanding and grouping symbols, we have:

$$(d \cdot d)t^2 + [2(o - c) \cdot d]t + (o - c) \cdot (o - c) - r^2 = 0$$

Which is a quadratic equation and we can solve for t. If t's roots are both imaginary, the ray did not intersect the sphere. If there is one double root, the ray hits the sphere tangentially, and since the ray is perpendicular to the sphere's normal, will not be visible. When there are two real roots, the smaller of the two corresponds to the first intersection with the sphere, and is used. My first work was to try and add a second renderable shape, a torus, by solving the vector equation for the intersection of a torus and a ray.

2.2.2 Anti-Aliasing

Because the screen we render the 3D scene onto has a finite number of pixels, no amount of sampling can perfectly render the actual objects into the 2D view. A good example of this problem is an infinite checkerboard, extending into the horizon. Seen by the eye, we would eventually see the board fade to gray as the interaction of photon wavelengths very close to each other blur together. That is to say, after a certain distance, we cannot resolve the exact color of a point in our field of view. With ray tracing however, we can do this for distances as large as our data structures allow us (ints, long ints, long long ints, etc). Thus, when sampling the checkerboard with rays, we will have pixels near the horizon that are absolutely white or black, corresponding to what the color of the square was that the ray hit. This looks unrealistic, because the pattern breaks down as we move towards the horizon, as the frequency of the black-white pattern increases above the pixel resolution. Anti-aliasing attempts to correct this by over-sampling each pixel, and instead of using a single absolute color, returns an average of each ray within that pixel. Thus, edges smooth out around spheres and other objects, reducing the 'jaggies' on the edges of objects. In the two images below, the left is a close-up of the edge of a sphere rendered with one ray per pixel. The right image is the same section of the same sphere, rendered with 25 rays per pixel. Note that while neither method produces a perfectly smooth edge, as we are trying to express a continuous surface with a discrete number of pixels, the right image produces an edge that looks smoother from a distance, because the jagged edges of the left are averaged out. This method of anti-aliasing averages the color from 25 rays, each color found by detecting a hit or miss as normal, and then averaging those values.

Figure 1: Close-up of a sphere rendered with one ray per pixel

Figure 2: Close-up of a sphere rendered with 25 rays per pixel

2.3 Comparing GI methods

When selecting a Global Illumination method, there are many options. Most of these methods make sacrifices to accuracy or image refinement however, for the sake of computation time. Due to recent advances in technology however, these methods have been mostly discarded, such as cone tracing. The two primary options I looked into were Radiosity and Distributed Ray Tracing. Distributed Ray Tracing is not simply ray tracing using distributed computing- that method is called parallel ray tracing. DRT samples multiple points over an interval and averages them together. Much like the method of normal anti-aliasing described in section 2.2.2, this requires many more rays, but also allows for softer features of lighting and shadows. Unlike simple anti-aliasing however, DRT uses multiple rays per pixel for hit detection, and off of each of those rays, uses multiple rays for transmission and reflection rays. By sweeping an average of the rays over a small angle about the "true" reflection direction, for example, adds soft features of diffuse light. This method allows for a relatively straightforward implementation- for each ray used, be it hit, lighting, reflection or transmission, simply use nested loops of averaged rays. Radiosity, summed up nicely by Wikipedia, is "an application of the finite element method [FEM] to solving the rendering equation for scenes with purely diffuse

surfaces." Radiosity has a number of drawbacks, but some appreciable benefits that led to my decision to focus on implementing it. Radiosity's weaknesses lie in its restrictions: the soft lighting features and rich shadows are best created when dealing with diffuse surfaces, and the method is not ideal for highly reflective and transparent surfaces. Because finite element methods have notable troubles around sharp discontinuities, scenes where hard shadows and sudden, sharp changes between light and shadow are hard to create with radiosity. On the other hand, radiosity can be solved for a scene once, and is camera independent. This means that once the lighting calculations have been made once for the scene, the rays traced from camera to scene have no influence on lighting, and the camera can be moved freely without needing to recalculate the light values. For any applications considering fixed scenes where users wish to view objects from multiple angles, radiosity allows for a drastic reduction in the number of rays required per render (instead of a hit detection ray and multiple lighting rays per hit detection, as well as multiple reflection + transmission rays per lighting ray, only hit rays need be calculated). Because I already had a basic understanding of FEM from my numerical analysis classes and my work with Dan Higgs during the summer of 2010, I decided to try to implement radiosity into the barebones ray tracer I was using.

3 Progress

3.1 Ray-Tracing Basics

Unfortunately, halfway through working on adding a second shape to the ray tracing engine, I switched tasks. While giving my second elevator talk, I was asked to work on a problem that was currently being investigated, instead of one that had already been solved. While other students were working on projects that had focuses on existing work and this seemed a bit conflicting, I shifted my focus to Global Illumination methods. What time I did spend working on this portion of the project was very valuable, as it simultaneously reinforced my understanding of OOP design patterns and extending objects, as well as C++ syntax, forcing me to be sure I was using proper and efficent data types and structures. It also had me solving a fourth-degree polynomial, and looking into efficent methods of solving such an equation. Had I continued with this portion of the project, my next focus would have been on Bounding Boxes and Axis-Aligned Bounding Boxes, two methods of coarse hit-detection which speed up hit calculations for complex shapes, such as torii.

3.2 GI method

I decided to try and implement a radiosity GI method, and did not complete this task. I considered starting the Finite Element Method code for this from scratch, but given the remaining time on this project, decided to see if modifying and translating some existing 1D FEM code in Python would work. I was able to convert my base code into a 2D Alternating Direction Implicit (ADI) method, but was not able to translate the code to C++ in time. My code relied on a few numpy package methods, such as a method for solving tri-banded matrices. Further, memory allocation in C++ was not something I was familiar with, and combined with some dynamic resizing issues I would have to address when converting from Python to C++, I was not able to complete this task. However, my 2D ADI method does work in Python, which I consider to be a success. The main method is below; the full code can be found in Appendix B.

```
def ctr_diff(U, x0, xf, y0, yf, t0, tf, dt):
    """U must be square, dx = dy,
        dx = (xf-x0)/U.shape[0],
        dy = (yf-y0)/U.shape[1]"""

    Nx, Ny = U.shape
    dx = (xf-x0)/float(Nx)
    dy = (yf-y0)/float(Ny)
    X = np.linspace(x0,xf,Nx)
    Y = np.linspace(y0,yf,Ny)

    s = dt / (2* dx**2)
```

```
    diags = (-s, 1+2*s, -s)
    A = TriDiag(Nx, Ny, diags)
    c = (-diags[0],2-diags[1],-diags[2])

    b = np.zeros((Nx,),float)

    UNew = np.copy(U)

    for step in xrange(steps):
        #Stage 1 ADI, loop over y-direction
        for L in xrange(1,Ny-1):
            for J in xrange(1, Nx-1):
                b[J] = c[0] * U[J,L-1] + c[1] * U[J,L] + c[2] * U[J,L+1]
            UNew[:,L] = SolveD(A, b)

        #Stage 2 ADI, loop over x-direction
        for J in xrange(1, Nx-1):
            for L in xrange(1, Ny-1):
                b[L] = c[0] * UNew[J-1,L] + c[1] * UNew[J,L] + c[2] * UNew[J+1,L]
            U[J,:] = SolveD(A, b)
    #Fix bounds to 0
        U[ 0 , : ] = 0.0
        U[-1 , : ] = 0.0
        U[ : , 0 ] = 0.0
        U[ : ,-1 ] = 0.0
```

4 What I learned

4.1 C++

I had forgotten all the ”joys” of a compiled language- rebuilding, semicolon errors requiring a rebuild, strict typing, etc. In Python, I was used to making a mistake with one of these things, and being able to quickly check and correct it. While I don’t dislike C++, I realized that I should spend a lot more time with it, as writing computationally intense routines in an interpreted language is a bit masochistic. This did teach me to create my own basic type-checking decorators, so that instead of putting a ”if (type(val) != int)” line at the beginning of every method, I could instead put ”@tcheck(Ray,float,numeric)” before a method and it would ensure that when called, the first variable passed to the function is a Ray object, the second is a float object (or castable to a float) and that the third is any acceptable numeric object (int, float, etc). This way, my code can fail safely- if I have a function that frees memory by deleting large list structures, such as:

```
def delList(aList):
        del(aList)
        return True
```

It is possible that by mistake I might pass in a file, or critical experiment values instead of a list. Using the following, however:

```
@TCHECK(list)
def delList(aList):
        del(aList)
        return True
```

If the variable ”aList” is not of type ”list” the method will not run. While it is still possible to delete critical lists, it is no longer possible to delete different variables (unless they are functionally equivalent to lists, and are interpreted as lists everywhere else). The strict c++ type checking I encountered throughout this project encouraged me to create such a safety, so that if I don't want to use it, I can simply remove the decorator. For functions handling data however, I can quickly add a safety against sloppy coding. better type enforcing and checking in my python code taught me to create auto-caching decorators in python so that I could use an interpreted language with a JIT compiler

4.2 Ray Tracers

Ray tracers are really complex. This is the first project I've worked on with so many classes and attributes that I had to get a second whiteboard to keep track of everything. After printing out the code and numbering classes and methods, I created a diagram for just GeometricObjects and Worlds and mapped out all of the interactions between the two. Working on such a complex piece of software made me realize that most of the difficulty was, when working on one class, not using variables that other classes expected, and being very careful with data encapsulation. This was also the first project I worked on where it was very difficult to sit down and just spend 10-20 minutes working on a small problem. Because the classes were so inter-connected, working on a problem required at least 10 or 20 minutes to get comfortable with how the structures worked together. It felt like I had to spend much more time pre-loading a picture of how it all worked together before I could start to inspect one aspect of a class. This also made interruptions very frustrating. Often, it was as though this model I was delicately balancing and the code I was working on would collapse when bothered, and I would have to spend another 5 minutes trying to reset the model I was looking at. That said, I really enjoyed working on a project of this size, as it gave me a much greater appreciation for writing code as a team. Had I managed to implement a GI method such as radiosity, I would have needed to consider certain caching schemes, data compression structures, and transparancy calculations. Without other people to bounce ideas off of and help implement code, I don't think those would have been possible. dozens of parts, requires very careful organization to keep track of how the pieces fit together. best to set apart large blocks of time to work on the problem- hard to just ”pick up and go”

5 CSUMS in review

5.1 Best Feature

Elevator talks were by far the best addition to this semester of CSUMS. Forcing us to look at our projects from a much higher level allowed us a better idea of where the low-level details fit into the grand scheme of things. By spacing them between our two larger presentations, it kept us from falling into a ”put this equation into code, now put that equation into code...” routine, and kept us asking questions such as ”How does this formula mesh with the rest of the project? Do I have enough time to explain it in my elevator talk, or should I just mention it by name, and explain it in detail during my large presentation?” I think these were very valuable, and I found the talks much more understandable this semester than last. Organization improved greatly, and students' transitions between components of a project were fluid, tying the sections together instead of making abrupt changes in topic.

5.2 Worst Feature

Still not being a morning person, 9:30AM was a bit early for me, but this isn't a fault of the CSUMS program itself. The biggest thing I missed from the summer CSUMS program was the open space to collaborate with students and days with no talks. While the course does place a lot of emphasis on teaching good presentation, it would be nice to have a day or two every month to have structured collaboration times. By this I mean to say that collaboration outside of class time was good, but the atmosphere created by having professors in the room and other students struggling with projects feels very conducive to collaboration. These days could also have mini-talks that students aren't required to listen to. So a student may give a brief overview of ”Python's metaclasses and why dynamic inheritance is important for your work” but there's no reason to have the physics students working in FORTRAN listening in. By allowing the students to choose to work on their projects or listen to the talk, the audience is more refined to the topic and because

there are likely less students interested, the presenter can have a more active discussion with those listening, instead of holding questions until the end.

5.3 Things to Keep

The structure created by having two formal presentations and two quick elevator talks was great. The shifts in presentation tone, pacing, and style that the short-long-short-long presentation structure created made it easier to get interested in projects we'd already heard about, especially for those going on their second semester. The day where everyone checked in with Professor Gottlieb and Kim was very good, and I would highly recommend adding a second day of this, somewhere around halfway through the course. I felt more focused and more confident in my organization by talking through my plan with professors that had obviously had to manage such deadlines before.

5.4 Things to Change

As mentioned in the "Worst Feature" section, consider scheduling days in the middle of a block of presentation days with no scheduled "required attention" presentations, so that students can collaborate and take a break from intense concentration. Allow presentations or mini-workshops on scientific packages, coding tricks and the like, but allow students to choose not to listen and instead work on their projects. A presentation showing the relative ease with which one can switch from Matlab to Python would be great for everyone to hear (in my biased opinion) but I realize that for those working in FORTRAN, the talk would have little application. add more "free work" or "mini-workshop" days where attention isn't necessary, and students can break off into groups and collaborate

6 Acknowledgements

I would like to thank my advisor, Professor Adam Hausknecht, for encouraging and supporting me throughout this project, and the National Science Foundation for their generous funding for this project. I would also like to thank Professors Saeja Kim and Sigal Gottlieb for their endless patience, encouragement, and reassurance throughout every CSUMS class, offering students a chance to work on something that would have otherwise been another "maybe someday" hope.

7 Appendix A: GeometricShape Sphere's hit function (C++)

```
bool
Sphere::hit(const Ray& ray, double& tmin, ShadeRec& sr) const {
        double          t;
        Vector3D        temp    = ray.o - center;
        double          a               = ray.d * ray.d;
        double          b               = 2.0 * temp * ray.d;
        double          c               = temp * temp - radius * radius;
        double          disc    = b * b - 4.0 * a * c;

        if (disc < 0.0)
                return(false);
        else {
                double e = sqrt(disc);
                double denom = 2.0 * a;
                t = (-b - e) / denom;    // smaller root

                if (t > kEpsilon) {
                        tmin = t;
```

```
                        sr.normal        = (temp + t * ray.d) / radius;
                        sr.local_hit_point = ray.o + t * ray.d;
                        return (true);}

                t = (-b + e) / denom;    // larger root

                if (t > kEpsilon) {
                        tmin = t;
                        sr.normal   = (temp + t * ray.d) / radius;
                        sr.local_hit_point = ray.o + t * ray.d;
                        return (true);}
        }

        return (false);
}
```

8 Appendix B: 2D ADI (Python)

```
import scipy.linalg
import numpy as np

import JUtil
outW = JUtil.outW
outP = JUtil.outP
outD = JUtil.outD
outE = JUtil.outE

import os
import random
import matplotlib as mpl
import matplotlib.colors
mpl.use("Agg", False)
import matplotlib.pylab as plt

Normalize = matplotlib.colors.Normalize
solve_banded = scipy.linalg.solve_banded

def SolveD(A,D):
    ud = np.insert(np.diag(A,1), 0, 0)
    d = np.diag(A)
    ld = np.insert(np.diag(A,-1), len(d)-1, 0)

    ab = np.matrix([ud, d, ld,])

    return solve_banded((1,1), ab, D)

def TriDiag(x, y, (d1, d2, d3), wrap = False):
    A = np.zeros((x, y))
    for row in xrange(y):
        if not wrap:
            if row > 0:
```

```
                A[row-1,row] = d1
            A[row,row] = d2
            if row+1 < x:
                A[row+1,row] = d3
        else:
            A[row-1,row] = d1
            A[row,row] = d2
            A[row+1,row] = d3
    return A

def ctr_diff(U, x0, xf, y0, yf, t0, tf, dt, fLoc, fName, logger, out = 0,
             colFMT = 'xytv', delim = '\t'):
    """U must be square, dx = dy,
        dx = (xf-x0)/U.shape[0],
        dy = (yf-y0)/U.shape[1]"""

    Nx, Ny = U.shape
    dx = (xf-x0)/float(Nx)
    dy = (yf-y0)/float(Ny)
    X = np.linspace(x0,xf,Nx)
    Y = np.linspace(y0,yf,Ny)

    s = dt / (2* dx**2)
    diags = (-s, 1+2*s, -s)
    A = TriDiag(Nx, Ny, diags)
    #c = (s, 1-2*s, s)
    c = (-diags[0],2-diags[1],-diags[2])

    b = np.zeros((Nx,),float)

    UNew = np.copy(U)

    fileNames = []
    JUtil.ensdir(fLoc)

    def saveData(i, A, (maxcnt, cnt, fileindex)):
        de = delim
        fmt = colFMT

        fmt_ =  "{"+fmt[0]+":0=+12.6f}" +"{d}" +"{"+fmt[1]+":0=+12.6f}" +"{d}"
        fmt_ += "{"+fmt[2]+":0=+12.6f}" +"{d}" +"{"+fmt[3]+":0=+12.6f}" +"\n"
        _fmt =  "{"+fmt[0]+"}" +"{d}" +"{"+fmt[1]+"}" +"{d}"
        _fmt += "{"+fmt[2]+"}" +"{d}" +"{"+fmt[3]+"}" +"\n"

        newLines = A.shape[0]*A.shape[1]
        if (cnt + newLines) >= maxcnt:
            fileindex += 1
            cnt = 0
            filename = fLoc+fName+"_data_{0}.txt".format(fileindex)
            File = open(filename,'w')
            File.close()
```

```
        logger.log("Generated:␣" + filename, 1, 0, outD(out))
        fileNames.append(filename)

    cnt += newLines
    filename = fLoc+fName+"_data_{0}.txt".format(fileindex)
    if cnt != 0 and fileindex == 0:
        logger.log("Generated:␣" + filename, 1, 0, outD(out))
        fileNames.append(filename)
    #if i==0:
        #File = open(filename, 'w')
        #File.write(_fmt.format(x='x',y='y',t='t',v='v',d=de))
        #File.close()
    File = open(filename, 'a')
    t = t0 + dt*i
    for xi, x in enumerate(X):
        for yi, y in enumerate(Y):
            File.write(fmt_.format(x=x,y=y,t=t,v=A[xi,yi],d=de))
    File.close()
    logger.log("Index␣{0}␣complete".format(i), 1, 0, 0)
    return (maxcnt, cnt, fileindex)

steps = int((tf-t0)/dt)
logger.log("Generating␣{0}␣frames.".format(steps+1),
           1, 0, outP(out))


sts = saveData(0, U, (1000000, 0, 0))
for step in xrange(steps):
    #Stage 1 ADI, loop over y-direction
    if outE(out):
        print "\t␣Stage␣1"
    for L in xrange(1,Ny-1):
        for J in xrange(1, Nx-1):
            b[J] = c[0] * U[J,L-1] + c[1] * U[J,L] + c[2] * U[J,L+1]
        UNew[:,L] = SolveD(A, b)

    #Stage 2 ADI, loop over x-direction
    if outE(out):
        print "\t␣Stage␣2"
    for J in xrange(1, Nx-1):
        for L in xrange(1, Ny-1):
            b[L] = c[0] * UNew[J-1,L] + c[1] * UNew[J,L] + c[2] * UNew[J+1,L]
        U[J,:] = SolveD(A, b)
#Fix bounds to 0
    if outE(out):
        print "\t␣Fixing␣Bounds␣at␣0.0"
    U[ 0 , : ] = 0.0
    U[-1 , : ] = 0.0
    U[ : , 0 ] = 0.0
    U[ : ,-1 ] = 0.0
```

```
        sts = saveData(step+1, U, sts)

    #return fileNames

crank_nicolson = ctr_diff

def toFrames(x0, xf, y0, yf, fLoc,
             fLocSub, fName, dpi,
             logger, out = 0):

    logger.log("Started␣saving␣frames.", 1, 0, outP(out))

    logger.start()
    items = os.listdir(fLoc)
    valid_files = [i for i in items if i[-3:] == 'npy']
    n = len(fName)
    selected_frames = [v for v in valid_files if v[:n] == fName]
    n = len(selected_frames)
    bottom = 0
    top = 0
    JUtil.ensdir(fLoc)
    JUtil.ensdir(fLoc+fLocSub)
    for i, frame in enumerate(selected_frames):
        A = np.load(fLoc+frame)
        local_min = min(A.flat)
        local_max = max(A.flat)
        if i == 0:
            bottom = local_min
            top = local_max
        else:
            bottom = min(bottom, local_min)
            top = max(top, local_max)
    norm = Normalize(bottom, top)
    logger.stop()
    logger.logMessageWithElapsed(
        "Normalized␣frames␣values.", ret = True, out = outP(out))

    plt.clf()
    fig = plt.gcf()
    ax = fig.gca()
    A = np.load(fLoc+selected_frames[0])
    Nx,Ny = A.shape
    x = np.linspace(x0, xf, Nx)
    y = np.linspace(y0, yf, Ny)
    for i,frame in enumerate(selected_frames):
        A = np.load(fLoc+frame)
        ScalarMap = ax.pcolorfast(x, y, A, norm = norm)
        if i == 0: fig.colorbar(ScalarMap)
        ax.axis('image')
        fig.savefig(fLoc+fLocSub+frame[:-4]+".png", dpi = dpi)
        logger.log("Saved␣{0}".format(frame[:-4]+".png"), 1, 0, outD(out))
```

```
        ax.cla()

def rndSpot(U,pct,mag):
    x, y = U.shape
    Nx = (x * pct) / 100
    Ny = (y * pct) / 100
    rx = random.randint(0,x-Nx)
    ry = random.randint(0,y-Ny)
    U[rx:rx+Nx,ry:ry+Ny] = mag

def addSpots(U,pct,mag,n):
    for i in xrange(n):
        rndSpot(U,pct,mag)

def addGoodSpots(U,pct,p,n):
    addSpots(U,pct,2**(p-1),n)
```