ABSTRACT
Coloring monochrome images is not exactly a new problem. In the realm of film alone, a huge amount of effort has gone into producing realistic color versions from grayscale originals. Anyone who as tried manual colorization can attest to its difficult and time-consuming nature. The basic strategy behind most algorithms involves edge recognition and segmentation. It goes without saying that this can be a difficult problem when the only input is a monochrome image.

In this project, I explore the use of a rough 3D model and an advanced global illumination framework as a tool in this process. Specifically, I take as input a 3D model and a monochrome image and use an iterative process to generate lighting conditions and add details not contained within the input model. The results are good enough to merit attention as a new perspective on ways to attack this problem.

1. INTRODUCTION
The goal of this project was to investigate a new approach to coloring. Traditional methods only use the original monochrome image and rely on a human to specify colors at different points and guide the propagation process. This is time consuming and expensive. My essential objective was to investigate how a physical model of the scene could aid this process. In this paper I describe a tool built around a global illumination renderer that takes this new approach to coloring.

2. DEVELOPMENT PROCESS
One reason I chose this project was based on my experience implementing a ray tracer last semester. I immensely enjoyed the process and spent significant time afterward working to adapt it to more advanced techniques. As such, my initial plan was to use it for this project. The major advantage of this would have been comprehensive personal understanding of the system and easy low-level access into it. The fact is, however, that ray tracing is a well-developed field and the Radiance[1] tools represent an improvement in quality and speed far beyond the capabilities of my own. The major downsides to using them, however, included the burden of familiarization and having to do all programmatic control through file and pipe manipulation/chaining and invocations of the command line. Thus, a significant amount of initial effort was spent learning the tools and building wrapper classes to abstract out the file manipulation and command-line invocations. The subsequent sections will address the development following the construction of this basic foundation.

2.1 Iterative Simulation of Lighting
Early on, I focused on iteratively improving the model of the lighting in a scene. I studied the case of a single light source for which the position and geometry was known. This is a very reasonable scenario, since light position can easily be calculated based on shadows in a scene. The task then was to calculate the color and intensity of the source in order to match the rendered scene with the target monochromatic image. My approach involved setting the light to pure red, green and blue separately in order to gauge the response of each pixel to each component. I used this information to repeatedly guess a light color and improve it. After each guess, I rendered the scene and iterated through the pixels to see the degree to which they differed from the target. The weight of each pixel on a given component color of the light depended on its difference from the target intensity and the pixel’s relative response to light of that color. I explored a multitude of different algorithms under that general pretense and met reasonable but qualified success.

First of all, the process took a huge number of renderings to complete and thus was slow for scenes of significant complexity. More importantly, the usefulness of this process on its own merit breaks down when the model is only a rough approximation of the scene. Finally, a persistent issue was susceptibility to local minima. There would be a significant number of pixels that were too light as well as plenty that were too dark but the weighting calculations had reached a fixed point. I had some success by modifying the algorithm to first increase lighting until no pixel was too dark and then decrease it until none was too light. Simply put, this version constantly improved low and high bounds towards a fixed point. This had the downside, however, of increasing the number of renderings even further. The nail in the coffin was when imperfect models were used. The algorithm needed to have some sort of cutoff to keep it from trying to satisfy impossible pixels. I also experimented with randomization of the stepping size (using a Gaussian distribution) and randomly choosing one color to work with in each iteration. The noise added by these techniques proved mostly detrimental, however, and did not greatly help the issue of local minima.

2.2 Masking
The system should not rely on having perfect models. Consequently, there is much more to the issue than recreating precise lighting conditions. For example, consider a wooden object or brick. In both cases, there is significant color variation over the surface of an object that is difficult to add to a model. As another example, consider shadows cast by complex objects such as trees. There is no way that a 3D model could capture such things exactly as they appear in a picture.

My solution to this was masks. A mask specifies a per-pixel adjustment to the rendered image. In the case of dark wood grain, the mask darkens the areas where wood grain exists in the target pictures. My first attempts involved fully automated masking. This did not work well at all with the iterative lighting calculations. The basic problem was that there is no way to automatically recognize whether a given pixel has the wrong
intensity because of incorrect lighting or because a mask is needed. In the context of the wood example, there is no way for the program to know whether it is a light surface with dark discolorations or a dark one with light discolorations. The issue boiled down to the fact that masking should be limited and precise in its use, but the automatic system tended to apply it heavily-handedly to every intensity error in the rendering. Furthermore, fully automated masking can only apply intensity masks (so no color adjustments). This makes it unsuitable for many cases, with being brick an easy example.

2.3 Areas
At this point it had become clear that masking would play an integral role in the final system and that I had focused too strongly on lighting recreation. That being said, it was also clear that there needed to be a much stronger element of human interaction in the mask generation, particularly if color adjustments were desired. Furthermore, I had failed to take advantage of all the information available to me from the ray tracer and 3D model. Traditional coloring techniques rely on intensity edges to recognize color transitions. The ray tracer and 3D model provide many more opportunities for surface distinction. In order to match the terminology in my code, I will refer to these sections as areas.

The essential idea is facilitating the generation and application of masks to specific portions of an image. I implemented the generation of areas based on material name, surface name, target image intensity edges, distance from the viewpoint and edges in the difference between the target and current rendering. The overall process is a repeated iteration of: 1) Calculate lighting using successively higher-resolution renderings (in order to improve render times), 2) Select areas to which to apply a mask, 3) Setting specific pixels in the areas to either not be masked or else adjusted to a specific color. Users select areas by clicking on representations in the GUI. Modifier keys indicate whether the area clicked on should be added or subtracted from the current selection area. This is especially powerful when combined with the ability to switch between area modes. For example, a user can use target intensity areas to select the white mortar in a brick wall and then use a different mode to subtract off any bleeding onto adjacent surfaces.

As mentioned above, there are two ways to generate a mask in an area. The first way is to specify a point as having no mask. For example, on a wooden table with dark grain, one would click on the light area. Every other pixel in the area is then adjusted relative to this new zero. In principle, the lighting will then be able to adjust (whereas before it was stuck between making the light and dark parts look correct). This works remarkably well. The second way is to set a specific target color for the point. A color adjustment mask is then generated that adjusts that pixel to the new color. Other pixels in the area are adjusted according to how similar their diff (i.e. the difference between the rendered value and the target) is to the selected pixel. The reason that the diff is used instead of target intensity is to try to propagate the color even into shadowed regions. In principle this is the more powerful technique but I had difficulty finding an optimal formula for correlating the relative diff values and the amount of masking applied. An interesting adaptation might be to allow users to select multiple points, set a color for each, and then interpolate between the colors based on pixel diff values.

3. ARCHITECTURE OVERVIEW
My entire project is implemented in Python. This choice was based on facilitating rapid development and evolution. Furthermore, I anticipated that the time taken to do ray tracing would far outweigh the overhead costs of Python (versus C++ for example). The project consists of about 2000 lines of Python code (the equivalent in C++ would be several times larger at least). Image manipulation was done using PIL (Python Imaging Library). The GUI was done using Tkinter. Both of these are part of the standard Python library.

3.1 Ray Tracer Interface
The core rendering engine is a ray tracing framework developed by Greg Larson: "The Radiance Lighting Simulation and Rendering System," (hereby referred to as Radiance). Radiance is comprised of a set of command-line tools all geared toward production of state-of-the-art renderings using global illumination techniques. In order to make a rendering, a set of scene and lighting description files are passed to a tool called “oconv” to generate an optimized viewpoint-independent octree representation. This octree is then passed to one of several rendering tools along with a viewpoint and rendering settings to generate a final image.

In order to effectively use the system in a programmatic environment I created a python wrapper (raytracer.py) to handle invoking the ray tracer. A closely related module (rd.py) was defined to handle lower-level manipulation of the Radiance data files as well as BMP files. As it turns out, implementing these interfaces took an extraordinary amount of time, requiring parsing data in and around dozens of files and programs. The end result is that a Raytracer object can be instantiated in python with a list of scene files and viewpoint information. This object has a “genImage” method that takes a light color and file resolution and returns a bitmap image. In order to do this, a file describing the light source is created and added to the current octree, which is then used to render the scene.

The other essential use of the Radiance system was obtaining data about scene materials, surfaces and depth. The wrapper methods for this functionality can be found in rd.py as well.

3.2 Processing Images
Most of the basic image processing is defined in area.py and processImage.py. The former contains the classes Area and AreaGroup, which are designed to (reasonably) efficiently represent sets of pixels and allow unions and subtractions. Areas are listed as a map of y-values to a list of one or more ranges, each of which specifies a sequence of adjacent pixels in the area on that row. The latter file pertains mostly to generating areas and edge masks from images. It includes a filter function to apply arbitrary kernels to images (since PIL does not let you apply kernels to all kinds of images).

3.3 Main Engine
The center logic in my system is found in the class “Engine” defined in render.py. Given a Raytracer instance and a target image, it runs an iterative lighting process and develops masks. Ironically, this code represents less than 10% of the entire code base. It is designed to run a loop in its own thread until interrupted.
3.4 Main Application

The user interface logic is defined in renderApp.py and launchDialog.py. This RenderApp class is responsible for loading saved data and using the rest of the classes as aides in making areas, generating images, etc. This is where the GUI is generated and controlled.

![Image 1: The main interface to the application. Panes: rendering, visualization of diff, areas and visualization of mask (clockwise from top left)](image1)

4. EVALUATION

An example can be seen in the following figures. The coloring was done without looking at the actual colored scene and the entire process took no more than a few minutes. The basic model used as input (Figure 3) had no textures at all (notice the floor, the wood desk, the book, and the picture). It also did not have the window blinds. Finally, it had a simpler lighting model (notice how the everything only has a single shadow).

The final colored version (Figure 6) looks quite good but it is not perfect by any means. The tile floor turned out almost perfect. The wood grain is very good but not quite exact. There seems to be a slight greenish tint to parts of it, like the light washed it out. The text in the book looks identical to the original. The window shades have also been added with decent success, as has the plant in the picture on the wall. In addition, notice how there are now multiple shadows, just like the original. On the downside, there are fairly obvious artifacts, especially at the edges of shadows and in the reflection in the window. These are mostly the result of resolution scaling issues with masks. Also, the plant is lacking much of the detail in the original and the background does not match.

![Figure 2: Target monochrome image](image2)

![Figure 3: Rendering of the 3D model used](image3)

![Figure 4: Desk and book colored](image4)
As the example shows, the results are impressive on the one hand but at the same time do not come all that close to perfection. The biggest problems with the current implementation are scaling issues (for one thing mapping areas across resolutions without artifacts is not trivial). Also, the generation of color adjustment masks is not exactly robust: the code was derived rather empirically.

5. FUTURE WORK

I would very much have liked to apply the colorization techniques described in [2] on areas in images. This is especially the case given the difficulty I had with color propagation. The situation is slightly different, however, in that I want to assign a similar color to pixels of similar intensities even if they are not adjacent.

Another major improvement would be treating the result of masks as modifications to surfaces and updating the scene data sent to the ray tracer. I spent a fair amount of time trying to engineer a method to make this work but it just was not feasible. I did not want to impose undue requirements on the format of scenes and trying to update arbitrary data files (there are dozens of types of materials and object types) is quite a Herculean task.

6. CONCLUSION

On the whole, this project was a success despite ending in a different place than I initially anticipated. I originally hoped to have a more complete product that produced high enough quality output to fool the human eye. A huge part of the effort in this project was implementation issues. The PIL library was wonderful when it worked as expected but occasionally threw wrenches in my path. A very indicative example of the kind of issues I had to deal with was based on the fact that the Radiance tool for outputting BMP files sometimes reverses the line order and also sometimes seems to output a garbage BMP header. Some image programs (such as Preview on a Mac) could always open these but PIL would choke up and throw exceptions. As such, I had to write a line-reversal function that also had a bunch of special cases to handle the funny cases (and I had to stop testing for the BMP magic bytes since it sometimes wasn’t there). Another time, I spent nearly a week trying to figure out why my masks were having inconsistent results before discovering that one of the ~30 possible command line options to a Radiance tool had enabled an adaptive sampling mode, meaning that at the pixel level two renderings with slightly different lighting could look quite different. Things like these obviously are not a very big deal but it all adds up to a lot of time spent on implementation details and getting things to work. That being said, I am quite proud of the final program: it provides a new viewpoint on the problem of colorization. Furthermore, it hints at the potential gains to be had from combining it with other algorithms. Finally, and most importantly, this project has served me very well as a learning experience across an interesting range of topics, from advanced global illumination to image filtering and color theory.

7. Acknowledgements

I would like to particularly thank Prof. Julie Dorsey for her guidance and assistance over the past year, despite her own crazy schedule. I have learned an incredible amount and thoroughly enjoyed tackling the projects set before me.

8. REFERENCES

