Berkeley/Princeton Radiosity Walkthrough System
MIT LCS Installation and Customization

Brian Gladstein
6.199
May 1996
Prof. Seth Teller

INTRODUCTION

The Berkeley/Princeton Radiosity Walkthrough (BPRW) System is a three-dimensional computer graphics rendering system that was designed by Seth Teller, Celeste Fowler, Thomas Funkhouser, and Pat Hanrahan. The system's main goal was to create a rendering environment that would allow real-time traversal of very large radiositized three-dimensional models, meaning models that would normally require an amount of memory far too great to be stored conventionally. Such examples usually involve lighting simulations inside structures ranging from small offices to multi-story buildings.

The BPRW system was created at Berkeley and Princeton, and now it has been brought to MIT. It was my job to get the entire system running here on our computer system. My tasks involved:

1) Setting up the source tree.
2) Eliminating useless source directories.
3) Getting the source to compile and link correctly.
4) Creating a master makefile to build all sources and utilities.
5) Getting working copies of several models.
6) Creating a method to build all the models.
7) Creating a WWW page to document the project.
8) Converting Unigraphics model data files to Inventor format.

Many of these tasks involved the traversal and manipulation of large amounts of files. To assist with this, I took advantage of the text processing capabilities of the Perl programming language, providing a mechanism to wade through the several hundreds of files involved in the BPRW system.

There were also several problems and obstacles I ran into while completing this project. While some were trivial, others required a large amount of work to overcome. It was this problem-solving that really took up the bulk of my time. After all, the main goal of the project was to circumvent or solve the problems that had appeared in porting the system.

OVERVIEW

Before I go into the details of my project, it is important to give an overview of the BPRW system as a whole. This system was a breakthrough when it was developed, showing how it was possible to store highly detailed geometric information of an entire building in a database and access that information in real time. This enabled the user to wander through the building at will without needing an

unreasonable amount of memory in the machine. While it may not seem impossible to store all this information, the key feature about the BPRW system was that the information was radiositized. Radiosity is an complex lighting model process, explained below.

Radiosity

Once a geometric model is generated in a computer's memory, it needs to be drawn to the screen. In order to do this, the computer needs to know how bright each object should be. This means it needs some kind of lighting model to base relative brightnesses upon. There are several ways of creating this lighting model, the simplest being to introduce a standard, ambient light which shines equally on all surfaces. This creates a very cartoon-like image where light sources within the scene have no effect on the brightness of individual objects.

Another method, called interpolative shading, involves calculating the normal vector to a surface at certain points along each polygon in the scene. By comparing these normals with the vector to the light sources and the vector to the viewpoint, a fairly accurate lighting model can be created. This process is known as interpolative because generally the shading of an object in between two points where the normal is calculated is interpolated. That is, the pixels in between calculated points are shaded in a linear or ordered manner, smoothly from one to the next.

A third method, better than the previous two, is radiosity. This is the lighting model used by the BPRW system. Radiosity involves calculating the energy emitted from every point on every surface in the geometric model and that energy's effect on every other surface in the model. This method is obviously orders of magnitude more complex than the other methods mentioned, but its results are the most realistic.

Radiosity is a huge calculation to execute, especially given all the interactions between every pair of objects in the object database. Because of this, solutions to radiosity equations generally take a long time to compute. The BPRW system needed to provide a way to generate a radiosity solution for an entire building, while allowing the user to traverse the building in real-time.

Until the BPRW system was developed, no one had been able to produce a radiosity renderer for a model as complex as a building that could calculate its images in a reasonable time period, even on a very powerful computer. The BPRW system accomplishes this by exploiting two aspects of these large models that significantly reduce the working size of the data the renderer needs to process at any given time. Partitioning the model is a process which divides the model into smaller groups based on what is visible at every point in the model. Ordering is a method which determines the best order to process the information within and between each partition. Without these two techniques, the large-scale rendering that the system accomplishes would probably not be possible.

When the model gets partitioned, each point in the model is examined, and relationships between points are considered. The goal is to weed out any calculations which are unnecessary, and thus reduce the number of elements that the solver needs to take into account. As a result, when the system is calculating the radiosity information for a particular office, it will only process information for that office, parts of the surrounding hallway, and whatever else might be visible within a specific number of degrees of removal. This greatly reduces the work that the renderer needs to carry out. The system is also ordered to maximize the efficiency of the solver. Ordering mostly deals with more system-dependent parameters, such as disk reading/writing and memory management. Since these items

can be very time-expensive in a process on this scale, ordering can play an enormous part in the renderer's efficiency. By choosing a proper order for the calculations, running time is dramatically reduced. Finally, the radiosity information for each unit is stored in the database, so that when the user walks through the visual representation of the model, that information is accessible. The user can thus see the radiositized model in a real-time environment, going wherever is desired simply by moving the mouse.

PROCESSES

The BPRW system is a large collection of source code, libraries, and executables. The current system is further enhanced by an enormous tree of geometric data. The source code is mostly written in C and C++, with a little bit of Motif, and compiled under Unix for X-Windows with the standard cc and CC compilers. The geometric data is written in Unigraphics format, which is a language developed for the storage of such data.

All the geometric data is stored in a large, central tree. To form a model that can be used as input to be renderer, a new Unigraphics file is created in a new directory. Here, this new file is filled with instances of objects from the central tree. Each object is created with coordinates that describe where in the model the object is placed. The new file is also equipped with information about the lighting in the model, usually a link to another Unigraphics file created exclusively for this model. This file then gets run through a flattener (ugflatten) which replaces each instance of each object with its more basic form of vertices and edges. Finally, the file gets run through a program called axialsplit which further organizes the data base on its visibility.

Once the model data is collated into a main Unigraphics file, it is ready to be processed through the BPRW executables, which will ultimately create the visual, radiositized walk-through model. There are six main executables in the system, and several other less important ones. The main executables are called wkcreate, wksplit, wkadd, wkvis, wkrad, and wkmotif. They are all called within four scripts which have been developed to manage the process of creating the radiosity model.

The wkcreate executable is called first. It creates and initializes a new database file. After this, the wksplit program creates cells throughout the database, based on the input Unigraphics file. Wkadd is then used to actually add all the walls, objects, and lights into the database, one by one. The wkvis program computes visibility information throughout the system, which is eventually used in the partitioning and ordering techniques described above. Finally the wkrad program computes the radiosity solution for the database. The resultant file contains all the radiosity information for the viewable model. This file can be viewed with the wkmotif executable. The user can fly through the model by simply clicking and moving the mouse.

TASKS

When I received the code for the BPRW system, there was a fair amount of work I needed to do to place it in a state where it could be compiled. Once it was in that state, I needed to create a master makefile, making it easy for the code to be compiled from the execution of one simple command. Finally, I needed to augment the master makefile with the capability to generate the model data in addition to the executable files.

I decided to take care of a skeleton makefile as a first step. Once I had this, I could attempt to

compile the code and follow the trail of errors it created. I created a makefile whose targets were the top-level libraries from which the family of executables was built. This makefile was little more than a central location which would go through its targets and activate the makefile of each library's base directory. This approach required me to keep track of each library's individual dependencies myself, but at this point, all I needed was a simple method for compiling code on a large scale.

After my first pass at a large-scale compilation, I was actually surprised its the success. Although there were quite a number of errors, many of them were of a similar background. On the whole, there were two main reasons why the compilation was not completely successful, both of them seemingly differences in the compiler's features. The first was a problem with how the compiler read C++-style comments. Apparently, the original compiler understood this style of comment (//) when compiling regular C files. This kind of comment is normally unsupported by the ANSI standard. When I tried to compile the C code, I found it to be littered with C++ comments, all causing errors with the compiler we were using.

At first, I thought that I would need to actually replace all the unsupported comments throughout the entire tree to get the code in shape. I began to sketch out a Perl script which would go through each directory and replace the comments of every file with the correct format. It wasn't long, however, before I learned of a much simpler way of doing this. There is a flag in the compiler which can be set on the command line, enabling the compiler to understand the newer comment style. By setting this flag in each makefile, the compiler will automatically understand the double-slash comments. This still required me to edit each makefile, but it would turn out to be far less painless than my earlier approach. I developed a new Perl script to complete this task. This script traversed the tree, checking the makefile in each directory to see if the proper variable name contained the new flag, and upon finding an instance without the flag, would add it in the proper place.

The second error I encountered was one having to do with derived class type casts. It seems as though the original compiler was considerably more forgiving than the compiler the use here at MIT. The compiler found an abundance of errors of this sort, and each one required me to edit the code and insert an explicit type cast where it was called for. Luckily I didn't run into an unmanageable amount of these errors, and even more fortunately, many of them were caused by identical accesses of a specific variable, thereby all correctable via the same fix.

There were only a few more errors to rectify past those listed above, and it didn't take me much longer to bring all the code in the libraries into a working state. Once I had these in order, I created a similar master makefile for the executables' side of the tree. This code seemed to compile extremely well, and the only significant errors I encountered had to do with which libraries were linked into the final program. This actually took a fair amount of time, trying to search through the library tree for a lone variable name given as a linker error. Some of the programs weren't even complete, and it took a bit of asking around to find which were the important executables, which were the non-essentials, and which were non-compilable, developmental entities. Eventually, though, this was all completed, giving me a full family of fully-functional executables.

As mentioned above, there are six main executables in this family, but there are about fifteen others built in the BPRW compilation process. It was at this point that I began to trim some of the useless code out of the system. Each library in the source code tree serves to provide its own area of functionality in the final group of programs. Many of these libraries were in the tree as a result of the ongoing process of development and enhancements involved in a project the size of the BPRW project.

It must be expected that any project of that magnitude will continue beyond what its authors imagined, and even when there is a final project, there is an abundance of other modules that are either out of date or not fully implemented.

In order to determine which of the files were important and which were superfluous, it became important for me to sort through and study each library's makefile. Inside each of these files is a list of the other libraries which must be linked in with current one in order to fully compile the code. By constructing a list of all the libraries and checking this against those which exist, I was able to eliminate a significant amount of dead code. This was all verifiable with the code in working order; any necessary files which I accidentally deleted would give me a link-time error, and I could immediately replace them. Fortunately, most of my guesses were accurate, and the number of missing libraries was small.

With all these programs ready to run, the next obvious step was to begin the formation of the model data I was required to assemble. I was fortunately not starting from scratch with these models, there were several existing models I could both use and learn from.

I need to form four separate sets of geometric models. The first was to be an empty office, a simple test of the system. The office contained nothing more than four walls, a floor, a ceiling, and two fluorescent light strips, each represented by a series of light sources. The second model was to be a more complex office, consisting of desks, chairs, bookshelves, a door handle, and various other objects such as trash cans and books. All of these were enclosed in the original empty office. The third model was one step better: the entire fifth floor of the already existing model of Soda Hall at Berkeley. Finally, the fourth model was to be the entire Soda Hall building from the third floor to the roof, a total of about six full stories.

All of the data for each of these for models existed well before I got to it. Once again, it was my job to make sure it was all set up correctly. Not everything was perfect, but things were mostly alright once I overcame a small number of obstacles. The bulk of the work came from implementing the proper build scripts into each model's base directory.

The problems I ran into trying to build the models were, for the most part, rather trivial. It was quite early on that I encountered the first problem, one of environment variables. When the BPRW system compiles the Unigraphics code, it uses several environment variables to operate tell it where to look for references to other included files, much like a path variable. This environment variable, called UGMODELPATH, needs to be set in order for the system to locate the appropriate files. Another environment variable, BASE, gets set to the base of the entire source/library/model tree. This variable is more of a convenience for scripting and makefiles, but it is integrated enough into the compilation system that it is important.

The task of creating the build process for these model databases was a straight-forward one, but it was not without its problems as well. I began my work off the existing procedure that was present in some of the model directories I found throughout the system. This procedure consisted of a master file, similar in purpose to a makefile, called makeit. Makeit was nothing more than an envelope for four other procedures called prepit, addit, visit, and radit, which were scripts designed to execute the programs created in the source/library tree in the proper order as described above.

Prepit is the first script called, and it initializes a new database using the ugflatten, axialsplit, and wkcreate procedures. The next script, addit, goes through the model data and adds all the objects into the

database with wkadd. Visit is a script which calculates the visibility information for the database using wkvis, and finally radit uses the wkrad program to calculate the radiosity information. I also created a cleanit script to act as a special 'make clean' for the model directories. This enabled all the information to be rebuilt entirely from scratch in an easy manner. To get these scripts fully functional, I needed to make sure all the proper files were in each individual model's directory, and that each reference to a file in the script was correct. There were a fair number of problems with this, but since the scripts are relatively small, it wasn't at all difficult to find everything. For a long time, I refrained from using the radit script because calculating the radiosity information was a time-consuming process, and the wkmotif viewer works fine without this calculation, providing a constant-color, ambient-lit model. Because of this, it took some time for me to discover a small bug in the radit script which caused the radiosity information to be calculated for only one iteration. Usually, one desires at least three iterations of the radiosity algorithm. Before I could see if the scripts, and therefore the build process, did in fact work, I just needed to go through and make sure all the correct files were being included in the compilation. It turned out that there were some problems with the lighting files, as well as wrongly included data in the larger models. This turned out to be a trivial problem, and it was not long before the models were in working order. It took a little tweaking to get them perfect, but once they were, everything worked together quite well.

I had finally gotten to a point here where I could begin with nothing but source code and model data, and end up with a full set of libraries, executables, and model databases, and the executables provided the expected manipulation and viewing of those model databases. The one thing I did not have was a coherent way of managing all this code and the process from a central location within the tree. My next task was the creation of a makefile to control the building and cleansing procedures of all the libraries, executables, and models. Since I had some smaller makefiles already completed, it was not too difficult to begin this job.

I decided to place the final master makefile in the base directory, where it could access everything easily, and still be somewhat central in locale. My goal in creating this makefile was to design a smart, simple, makefile that could be used for all purposes. Additionally, it needed to be smart about its targets. That is, if one library depends on another, and the other library is rebuild, the first library should also be rebuild.

I found a lot of the dependencies for each library and program already existed in that component's directory, and thus it became merely a matter of including a rule for each library that would call make on the library. This was a welcome feature in the existing code; it certainly made my job significantly easier. I did continue to lay out a dependency map for the relationships of different parts of the code. For example, if you rebuild the lib directory, you will need to rebuild the bin library and the model databases. Besides these targets, I also included some cleansing targets, such as a make clean to clean out intermediate files, and a make superclean to remove end product files. This makefile ended up taking a fair amount of time to complete, but mostly because it took so long to compile all the libraries that errors would not be readable for quite some time after the code was modified. There was unfortunately no easy way I could see around this, so I just went with it, and eventually all the problems were worked out.

The setup I have described above constituted the major portion of my project, however, there were also some other additional tasks I undertook. The most significant among these was the creation of a Perl script to convert the entire Soda Hall model directory from its original Unigraphics format into the more modern and more standard Inventor format. I also had to design a web site documenting the

procedure for installing the BPRW files on a new computer system.

In order to convert the Unigraphics files to Inventor, I needed to utilize two existing tools. The first is called Ug2Inv, which is a small C program designed at MIT to convert Unigraphics files to Inventor. This program takes care of all the syntactical conversion of the most basic level Unigraphics file to a simple Inventor format. However, the Soda Hall files are not in the most basic level of Unigraphics, and this is where the second tool I used came in. This tool was the ugflatten executable described above for the BPRW system. This is the Unigraphics equivalent of a C++ to C converter. It takes the Unigraphics code and flattens out all the instances and declarations of objects, and reduces all the data into a basic set of about four main keywords (for vertices, edges, colors, and textures), plus any additional primitive keywords. These keywords are the set of tokens understood by Ug2Inv, and so the combination of the two will successfully turn any Unigraphics file into an Inventor one. My Perl script took care of the coordination of these two tools, the recursion and duplication of the directory structure, and the error checking that let me know where Ug2Inv and ugflatten failed. There were some augmentations that needed to be made to these programs, but those enhancements were out of my hands.

The purpose of the web page I developed is mostly to provide instructions of the installation procedures for other people who want to install the BPRW tree on their systems. The work I completed from throughout the course of this project will reside there for future use. Additionally, most of the information in this report is located there to help out wherever it can.

CURRENT STATE

At this point I would like to describe the directory structure and the state of the environment of the BPRW system. This information is useful to those who want to install the system on another computer system. It will give a basic overview of how the structure is organized, and what each section does.

The root of the entire BPRW tree is a directory stored by the environment variable $CVS. CVS (Concurrent Version System) is the source code control system that was used for this project. It is an augmentation of the RCS (Revision Control System) source code control system, enhanced to allow for an entire hierarchical system to be archived, as opposed to just the one directory supported by RCS. In this root directory, there exist four subdirectories and a master makefile. Execution of the makefile results in a build of the entire directory tree. All the helper applications and tools are built, along with models if so specified. The four subdirectories include the following:

1) ff: This module is the lowest level of code in the BPRW system.
2) gur: This module provides the real foundation for the walkthrough system, and is dependent on the ff module.
3) megagur: This module enhances the gur module, and completes the basic structure of the BPRW system.
4) walthru: This is where most of the source, libraries, models, and executables live. This directory contains only one item, a subdirectory version11.1, referenced by the $BASE environment variable. From now on, I will refer to this version 11.1 directory as $BASE.

The $BASE tree is comprised of the following six subdirectories:

1) bin: This directory holds the executables for the BPRW system.
2) util: This directory contains the high level source code for the programs in $BASE/bin.

3) lib: This directory holds the lower level libraries that are used to compile the programs in $BASE/bin.
4) src: This directory holds the lower level source code that creates the libraries in $BASE/lib.
5) models: This directory is the root of the tree which contains all the geometric descriptions of the items in Soda Hall. It is a repository for the models of all the individual items, and thus serves as the building blocks for actual models to be rendered. This directory must be included in the $UGMODELPATH environment variable.
6) largemodels: This directory contains each of the actual model collections I revised, i.e., the office and building collections which are rendered through the makeit scripts. This directory is referenced by the $MODELBASE environment variable.

And so, the environment variables which must be defined are as follows:

1) $CVS - The root of the BPRW tree.
2) $CVSROOT - Your personal CVS repository. If you do not use CVS, it is unimportant, however, if you are using the CVS system, you need to tell CVS where to look for the archived files. Note that this variable is only used by CVS itself.
3) $BASE - The base directory of most of the code, equal to $CVS/walkthru/version11.1.
4) $UGMODELPATH - The path of the Unigraphics model primitives. I used a path which also included the current directory. The variable looked like this: .:$BASE/models
5) $MODELBASE - The path of the final Unigraphics models, which are to be rendered, equal to $BASE/largemodels.

Finally, the master makefile, located in $BASE, contains many targets. I will list here the important ones:

1) make default - Build the executables (populates $BASE/bin)
2) make all - Builds all executables and radiosity models. This target is not the default because of the huge amount of time it takes to build the large radiosity models.
3) make lib - Builds the libraries (populates $BASE/lib)
4) make allmodels - Renders the models with the existing executables (does not rebuild these executables). Does not run radit, and so there is no radiosity information in the final databases.
5) make allmodelsrad: Same as allmodels, but does include the radiosity information.
6) make clean - Deletes all object files and intermediate files from the library and executable trees. This leaves all final targets intact, including .lib files and the final executables.
7) make superclean - Deletes all object files, all intermediate files, and all output files. Leaves only the source code. The program must be rebuilt to be used.
8) make cleanmodels - Deletes all output files in the models. The model databases must be rebuild to be used.
9) make ultraclean - This is simply a combination of superclean and cleanmodels, deleting all output in the entire system.

CONCLUSION

In conclusion, this project provided a good introduction to the BPRW system. Although I was not able to learn as much about it as I would have liked to, I did learn a lot about its structure and its purpose. The results of this project were very visible, which is always a nice feature when taking on such a task. It was nice to have a final, working system at the end of the project, as well as a working

knowledge of that system and its components.

This document was dictated with Kurzweil VOICE 2.0 for Windows 95.