ANIMATED RADIOSITY APPLICATION

BY

SUZANNE J. PATTERSON

BS, Pennsylvania State University, 2001

PROJECT

Submitted in partial fulfillment of the requirements for
the degree of Master of Science in Computer Science
in the Graduate School of
Binghamton University
State University of New York
2005

Submitted in partial fulfillment of the requirements for
the degree of Master of Science in Computer Science
in the Graduate School of
Binghamton University
State University of New York
2005

Richard Eckert_____August 22, 2005
Department of Computer Science, Binghamton University

Leslie Lander_____August 22, 2005
Department of Computer Science, Binghamton University

Abstract

Radiosity, developed in the early 1980s, has become a fundamental part of the Computer Graphics field. The Radiosity concept can act as a standalone rendering tool, but actually was developed as a smaller piece to the grander challenge of simulating light behavior in a scene. Because of its tradeoffs, Radiosity is often combined with other techniques in order to take advantage of the different benefits that each algorithm has to offer. This in turn has created a variety of hybrid versions as well as new twists on the Radiosity algorithm itself in an attempt to avoid the drawbacks, namely, its strain on system resources and also its lack of specular reflections.

This project revisits the early fundamental Radiosity algorithm. Understanding the baseline algorithm is no simple task. The implementation of the algorithm is even more challenging. Not only does the algorithm itself need to be coded, but an entire environment for which to apply the algorithm must be created. In this project a tutorial application has been designed and implemented that helps demystify some of the concepts behind basic Radiosity. Understanding the fundamental algorithm will provide the user with a foundation in this area and open the doors to understanding the complexities of more recent derived algorithms.

The major contribution of this project is to make the basic concept of Radiosity accessible to everyday programmers with some mathematical background by providing an animated application. This application will walk the user through all of the required steps from setting up an environment to the final rendered image. The steps will provide a combination of insight on mathematical technique and program structure that guide the user towards understanding and implementing the Gathering Full-Matrix Radiosity algorithm.

Table of Contents

Figures

1. Introduction

The Radiosity algorithm is a mathematical method used in modern computer graphics to render images of visual objects. Historically, it was a natural progression within the field due to its capabilities in the area of diffuse reflections. That is, an incident straight beam of light energy reflected in all directions off of a rough surface. Radiosity has become a significant development in this area, however, it is not without it's limitations. For example, in contrast to diffuse reflections, there are specular reflections, or a sharply defined beam resulting from reflection off of a smooth surface. Radiosity does not handle specular reflections. Over the years, combinations of various algorithms have been created in order to take advantage of many different illumination effects.

This project implements an interactive animated application to illustrate the principles behind the basic Gathering Full-Matrix Radiosity algorithm. The application demonstrates the mechanisms of this algorithm by walking the user through an extensive twelve-step tutorial. In addition, the application provides the user with the freedom to change input parameters through dialog boxes to see how the rendered scene is affected.

Visual C++ 6.0 and Macromedia Flash MX were used to develop the application. The interface is made up of menus, toolbar buttons, dialog boxes, and one or more view spaces. One scene is currently built-in. The scene contains a room, a box, and two light sources. These objects are immutable. If the scene is translated in any way, all objects are translated together. In addition, individual objects cannot be added or removed. Figure 1 shows the simple wireframe scene. The left side shows the scene before its surfaces are divided into a set of patches. The right side shows the scene's surfaces divided up into a default set of patches.

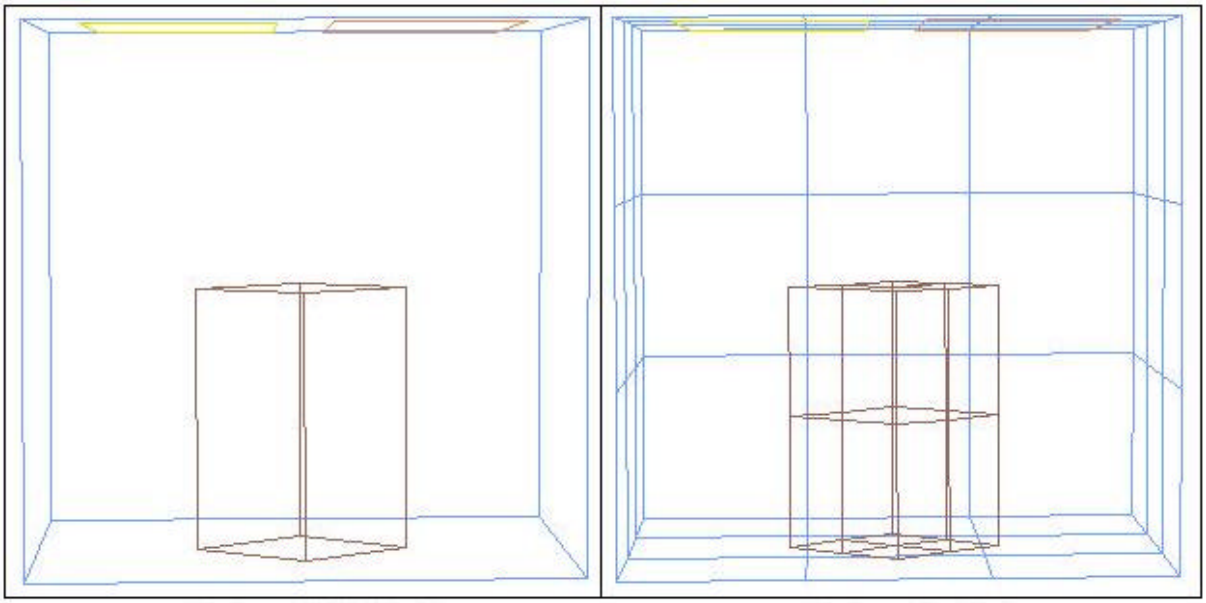

Figure 1: Wireframe Scene

Default parameters are set so the user may view the scene without any setup. Figure 2 shows the rendered scene. The left side shows the scene with only Radiosity and hidden surface removal applied. The manner in which the individual surfaces are divided is still apparent. The right side shows the scene with Gouraud shading. This shading technique smoothes the patches out and provides a more finished look to the scene.

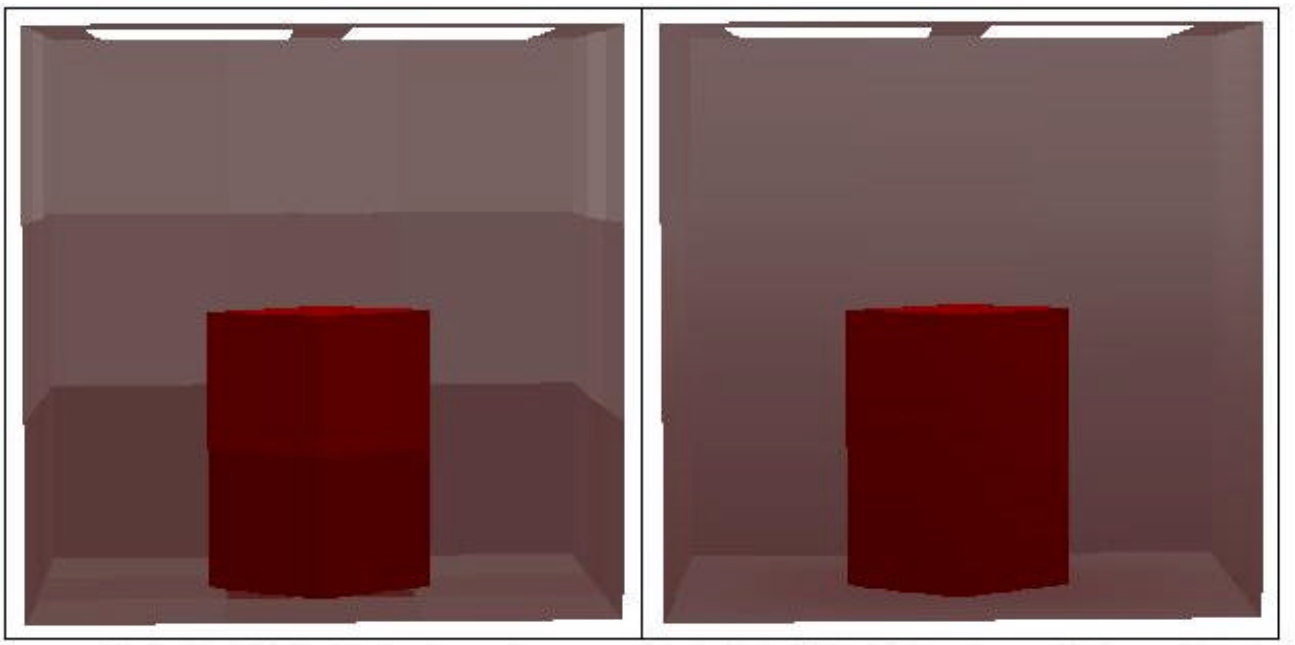

Figure 2: Rendered Scene

This application also contains an AVI movie player, which is used during the twelve-step tutorial. AVIs are incorporated into the tutorial to provide a clearer and more interesting presentation.

The Gathering Full-Matrix Radiosity algorithm has several advantages and disadvantages. The application distinctly points these out for this particular Radiosity algorithm. Understanding the underlying reasons for the pros and cons will prepare the user to grasp similar, more complex algorithms and create new ones.

Advantages:

?? Photorealism: The Radiosity algorithm models light within a scene based on the geometry that exists between surfaces. In turn the algorithm can produce some very realistic images.

?? Energy Transfer: The fundamental concept underlying Radiosity is the transfer of energy from surface to surface. This energy transfer results in a more accurate and physically correct solution that leads to more believable rendered images.

?? Soft Shadows: These are a more realistic representation of area light sources. Soft shadows provide a gradual transition from unlit to lit areas. They contain an umbra (no light visible) and a penumbra (some light visible).

?? Color Bleeding: The colors from diffuse reflective surfaces bleed into their surroundings.

?? View Independence: Once the Radiosity values have been calculated for the scene, the viewpoint can be changed without the Radiosity values having to be recomputed.

Disadvantages:

?? Expensive – Time & Storage: The form factor between every pair of surfaces must be computed and stored prior to calculating Radiosity values. As the number of surfaces increase, the time taken to compute Radiosities as well as data storage requirements also increase.

?? Curved surfaces must be approximated with polygon meshing: The scene used for this application does not contain curved surfaces. In general though, many small polygons (or surfaces) that are attached to each other are used to approximate curved surfaces.

?? No specular reflections (reflection of shiny surfaces): All surfaces within the scene are considered to be diffuse reflectors. Ideally, we would want the ability to create both diffuse and specular reflections.

2. Radiosity Overview

The Radiosity of a surface is the total light energy per unit area leaving a surface per unit time. Included is the light energy emitted and reflected. Equation (1) shows the basic Radiosity equation that can be used to render a scene.

$$B_i \; ? \; E_i \; ? \; ?_i \; ? \; B_j F_{ij} \qquad (1)$$

Each parameter in equation (1) serves an important purpose:

?? B_i = Radiosity (surface i)
?? E_i = Emission (surface i)
?? p_i = Reflectivity (surface i)
?? B_j = Radiosity (surface j)
?? F_{ij} = Form Factor (surface j relative to surface i)

B_i is the Radiosity of a surface and that is to be calculated. It is the total light energy per unit area leaving a surface per unit time. E_i is the emissivity of the surface. Emissivity is the amount of light energy emitted per unit area per unit time. Light sources for example, will have some non-zero emission value. Non-light sources will have an emission value of zero. p_i is the reflectivity of the surface. Reflectivity is the fraction of light energy that is reflected off of a surface. The reflectivity is defined as a number between 0 and 1. The higher the reflectivity of a surface, the larger the fraction of light that gets reflected off of that surface. B_j is the Radiosity of a surface elsewhere in the

scene. This is energy per unit area leaving a surface per unit time. F_{ij} is called a form factor and it represents a purely geometrical relationship between surfaces i and j. It is the fraction of light energy leaving patch i that arrives at patch j. It is independent of the viewpoint or any surface properties of the patches (surfaces).

Figure 3 shows the different energy components arriving at and leaving a surface. The surface might represent for example, the wall of a room or a smaller area of the wall.

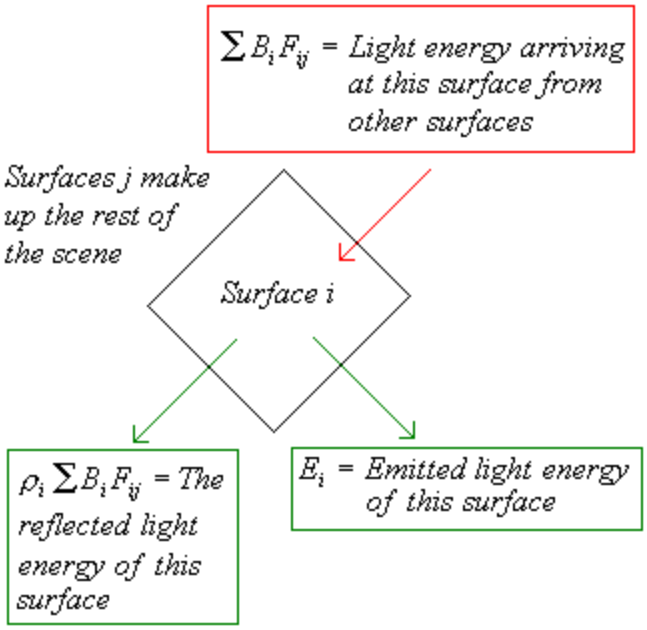

Figure 3: Surface Input/Output

The Radiosity equation itself is based on the Law of Conservation of Energy. The total energy leaving a patch equals the energy it emits plus the sum of any energy from other patches that it reflects. Figure 3 shows that there are two energy components leaving the patch. The emitted light energy that originates from this surface and the reflected light energy. This reflected light energy takes into account the sum of intensities of other surfaces multiplied by the form factor from this surface i to that surface j. This sum is then multiplied by the reflectivity of this patch i in order to determine what fraction of light energy that arrives at this patch, should leave this patch. There is one energy component arriving at the current patch i. This is the sum of light energy arriving from all other surfaces in the scene. This is the same quantity that is used to determine how much light should leave the patch, except that the full amount is expected to arrive at the patch and only a fraction is expected to leave.

An important part of the Radiosity calculation is the determination of form factors. The form factor between two surfaces is the fraction of energy emitted from one patch that is incident on the other patch. Figure 4 shows the geometric relationship between two patches of which form factors are based.

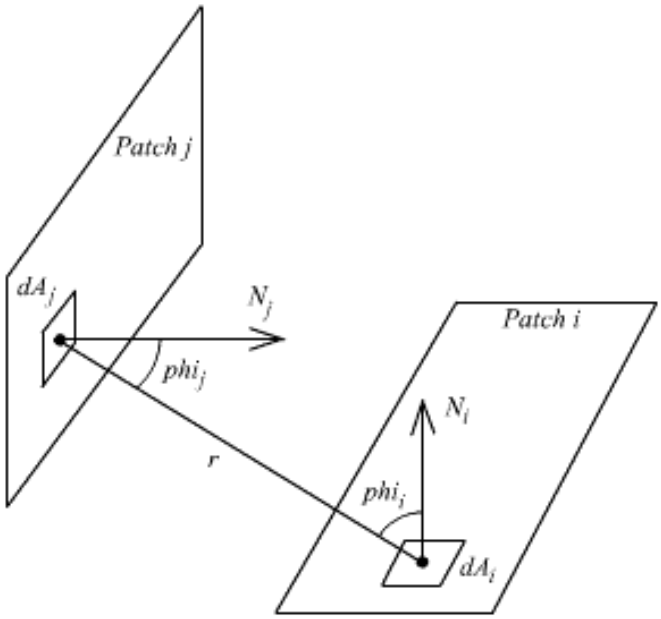

Figure 4: Patch Geometrical Relationship

The terms used in Figure 4 are:

- dA - Represents a differential area of each surface.
- R - A vector drawn from the center of the one differential area to the other.
- N - The normal vector drawn from the center of the differential area. It is perpendicular to the surface.
- phi - The angle between vector r and the normal vector.

Equation (2) describes this geometrical relationship between two patches.

$$F\ dA_i\ dA_j\ ?\ \frac{\cos ?_i\ \cos ?_j}{?\left|r\right|^2} \qquad (2)$$

The form factors between any two patches follow the Principle of Reciprocity. This principle says that the percentage of light energy emitted by one patch and received by the other patch is equal to the percentage of energy going in the other direction. We can see that this is true by looking at equation (2) for the form factor. Reversing the sending and receiving patches in that equation yields the same expression. The overall form factor can be found by performing a double integration on equation (2). Equation (3) shows this double integration.

$$F_{ij}\ ?\ \frac{1}{A_i}\ ?\ ?\ ?\ \frac{\cos ?_i\ \cos ?_j}{?\left|r\right|^2}\ dA_i\ dA_j \qquad (3)$$

Because equation (3) is rather complex, an approximation method was developed in order to simplify the Radiosity process. This is called the Nusselt analog. The Nusselt analog allows the simple and accurate calculation of the form factor between a surface and a point on a second surface. The method involves centering half of some symmetrical surface on a patch and dividing that surface into elements. According to the Nusselt analog, summing up "delta form factors" between an external patch j and each of the surface elements will give an excellent approximation to the exact form factor between the two patches that would result from evaluating the double integral. Figure 5 shows a hemisphere with unit radius built on the visible side of a patch. The other patch is then spherically projected onto this structure.

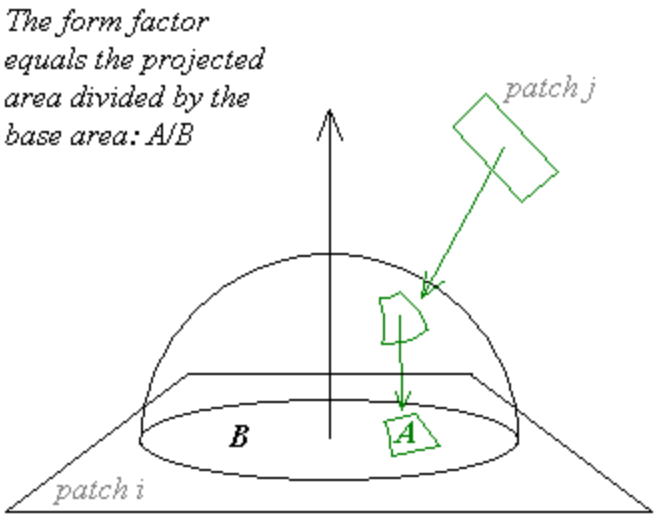

Figure 5: Nusselt Analog - Hemisphere

It is more convenient to use half a cube (called a hemicube) instead of a hemisphere. The hemicube is built over the center of the patch and each of its faces is divided into discrete rectangular elements called cells. The delta form factor for each cell is then easily computed. This application uses the hemicube to determine form factors between the patches in the scene. The hemicube is a valid method for determining the form factors because of the Nusselt analog. Figure 6 shows a hemicube built on the visible side of a patch with a second patch projected onto the structure.

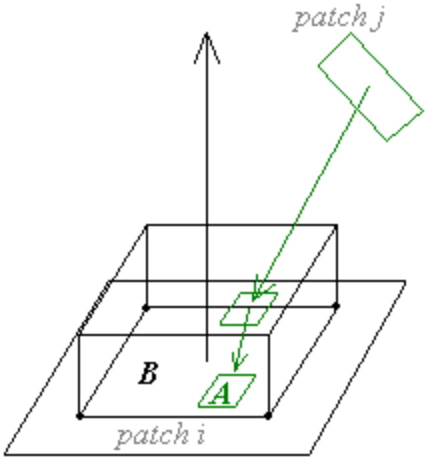

Figure 6: Nusselt Analog – Hemicube

This project focuses on the gathering version of Radiosity. Gathering means that light energy is gathered from all other surfaces in the scene. The gathered energy from each other surface in the scene may be computed by using the precalculated form factors in the basic radiosity equation. Thus the proportions of light energy received from other patches in the scene can simply be added together to calculate the total light energy that the current patch receives. The procedure for finding a form factor between a pair of patches is outlined in the following pseudocode:

```
for all patches i
        compute patch i center and build a hemicube over the patch
                for each face of hemicube
                        if top face
                                for each top face cell
                                        for all patches j
                                                if patch i = patch j
                                                        form factor = 0
                                                else
                                                        if ray from patch i center through current
                                                        cell intersects patch j
                                                                get ray length d
                                                                if(size < d)
                                                                        save patch j
                                        if a patch j has been saved
                                                form factor += delta form factor of current cell
                        else if side face
                                for each side face cell
                                        for all patches j
                                                if patch i = patch j
                                                        form factor = 0
                                                else
                                                        if ray from patch i center through current
```

```
                                    cell intersects patch j
                                        get ray length d
                                        if(size < d)
                                            save patch j
                    if a patch j has been saved
                        form factor += delta form factor of current cell
```

There will only be one form factor between a pair of patches. This psuedocode shows how the form factors between pairs of patches get incremented based on the hemicube approach. The form factors may only be found once the delta form factors for each hemicube have been determined.

A delta form factor must be calculated for each cell on a hemicube. The calculation is done differently depending on whether the cell is located on the top or one of the sides of the hemicube. Equation (4) shows the calculation for a hemicube top cell.

$$?ff\ ?\ \frac{dA}{?\ (a^2\ ?\ b^2\ ?\ 1)^2} \tag{4}$$

Figure 7 shows the a and b distances that are used in equation (4). dA represents the area of the cell on the top face of the hemicube.

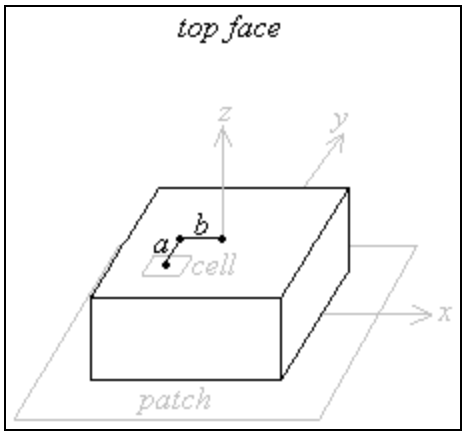

Figure 7: Delta Form Factor Parameters for a Hemicube Top Cell

Equation (5) shows the calculation required for the delta form factor on a hemicube side cell.

$$?ff\ ?\ \frac{b*dA}{?\ (a^2\ ?\ b^2\ ?\ 1)^2} \tag{5}$$

Figure 8 shows the a and b distances that are used in equation (5). dA represents the area of the cell on the side face of the hemicube.

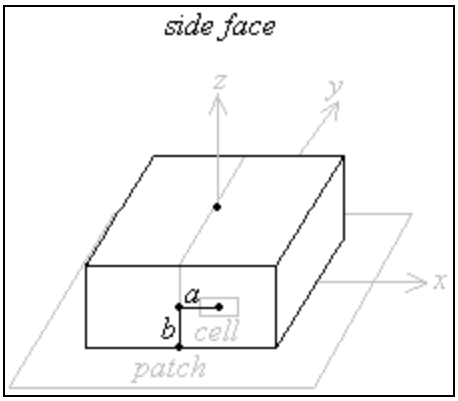

Figure 8: Delta Form Factor Parameters for a Hemicube Side Cell

Equations (4) and (5) can be used with the following psuedocode to calculate all delta form factors on every hemicube in the entire scene:

```
for all patches i
        for each hemicube face on patch I
                if top face
                        for each cell on top face
                                compute center of cell, dA, a, b
                                calculate and save delta form factor
                else if side face
                        for each cell on top face
                                compute center of cell, dA, a, b
                                calculate and save delta form factor
```

The result of calculating delta form factors and form factors is the ability to solve the Radiosity equation for each B_i in the scene. Equation (1) should first be rearranged to produce equation (6).

$$B_i \; ? \; ?_i \; ? \; B_j F_{ij} \; ? \; E_i \qquad (6)$$

Equation (6) can be broken out into a system of equations. Each i represents a patch, so there will be as many equations as patches in the scene. Equation (7) shows the system broken out into matrix form. All rho and E values are user defined. The form factors have been calculated. The only remaining unknown in the system is B_i. Note that the diagonal of the matrix is all 1's indicating a patch does not receive energy from itself.

$$\underbrace{\begin{bmatrix} 1 & -\rho_1 F_{12} & -\rho_1 F_{13} & \dots -\rho_1 F_{1N} \\ -\rho_2 F_{21} & 1 & -\rho_2 F_{23} & \dots -\rho_2 F_{2N} \\ \dots & \dots & \dots & \dots \\ -\rho_N F_{N1} & -\rho_N F_{N2} & -\rho_N F_{N3} & \dots 1 \end{bmatrix}}_{K} \begin{bmatrix} B_1 \\ B_2 \\ \dots \\ B_N \end{bmatrix} = \begin{bmatrix} E_1 \\ E_2 \\ \dots \\ E_N \end{bmatrix} \qquad (7)$$

The matrices in equation (7) represent one color channel out of red, green, and blue. A separate set of equations must be maintained for each of the three channels. Since both the reflectivity ? of a patch and the form factor F_{ij} between a patch i and another patch j must both be less than one, off-diagonal elements of the matrix are all less than one, making the matrix "diagonally dominant". Since that is the case, the Gauss-Seidel iterative method can be used to solve for the radiosities (Bi). This method continuously calculates new radiosities from old radiosities until a convergence is reached. A convergence is reached when all new Radiosity values do not change from the previous values or they only change within a small, predefined amount. Equation (8) shows the calculation for each new Radiosity. The sum is broken up into two parts. The new Radiosity value is used for surfaces j if it's been calculated for the current iteration. Otherwise the previously calculated Radiosity value is used.

$$Radiosity\_new[i] = E_i + \sum_{j=1}^{i} K[i][j] * Radiosity[j] + \sum_{j=i+1}^{TotalPatches} K[i][j] * Radiosity\_prev[j] \qquad (8)$$

Once red, green, and blue radiosities have been calculated for every patch in the scene, the values are proportionally mapped onto a 0 to 255 color scale on a true-color graphics display device.

In summary, the Radiosity algorithm requires several steps in order to view a final rendered scene. All of these steps are covered in detail within the application. The following list summarizes the steps:

- Define a scene: In order to use the Radiosity algorithm, there must first exist a defined environment.

- Divide all surfaces of the scene into distinct patches: A red, blue, and green Radiosity will be calculated for each individual patch.

- Build a hemicube on a patch: Divide each face of the hemicube into a finite number of cells. Calculate a delta form factor for each cell on the hemicube and store it. Once this is complete, repeat the process on every other patch in the scene.

- Calculate the form factors for every pair of patches in the scene: This can be approximated using delta form factors that were computed in the previous step.

- Calculate the radiosities for all patches: Once the form factors have been determined, all information that is required to solve for the radiosities has been obtained. Use the Gauss-Seidel iterative method to solve the system of equations.

?? Map all Radiosity values to a color scale: Use a 0 to 255 scale for a graphics display system that uses true color. Thus, the largest Radiosity value will get mapped to 255. The remaining values are scaled linearly. Once the values have been mapped, hidden surface removal can be performed and the rendered scene can be drawn.

?? Apply Gouraud shading to smooth out the look of the patches: This is technically not part of the Radiosity algorithm and along with the Z buffer hidden surface removal technique, is not view independent.

3. Application Interface

This section will discuss the features of the application. The features include the modes, menus, toolbar buttons, and other miscellaneous items.

3.1. Modes

The application runs in two different modes. These modes are Free and Animate. Once the scene is intialized by selecting Scene A from the Start menu, the user can toggle between the two modes by pressing the appropriate toolbar button.

In Free mode, the user can view the scene in wireframe mode with or without patches. They can view the final rendered scene with or without Gouraud shading. The scene can also be adjusted by translating, rotating, or scaling. In addition, the four viewing parameters theta, rho, phi, and distance can be adjusted. The user can do all of this as well as bring up dialog boxes to set various scene properties. Free mode essentially provides an experimental environment for the user to see how various scene properties affect the Radiosity algorithm. The animation controls are not available for use in this mode. If the user attempts to use an animation control, a message box will pop up indicating that the application should be toggled to Animate mode in order to use the button. Figure 9 shows a screenshot of the application in Free mode.

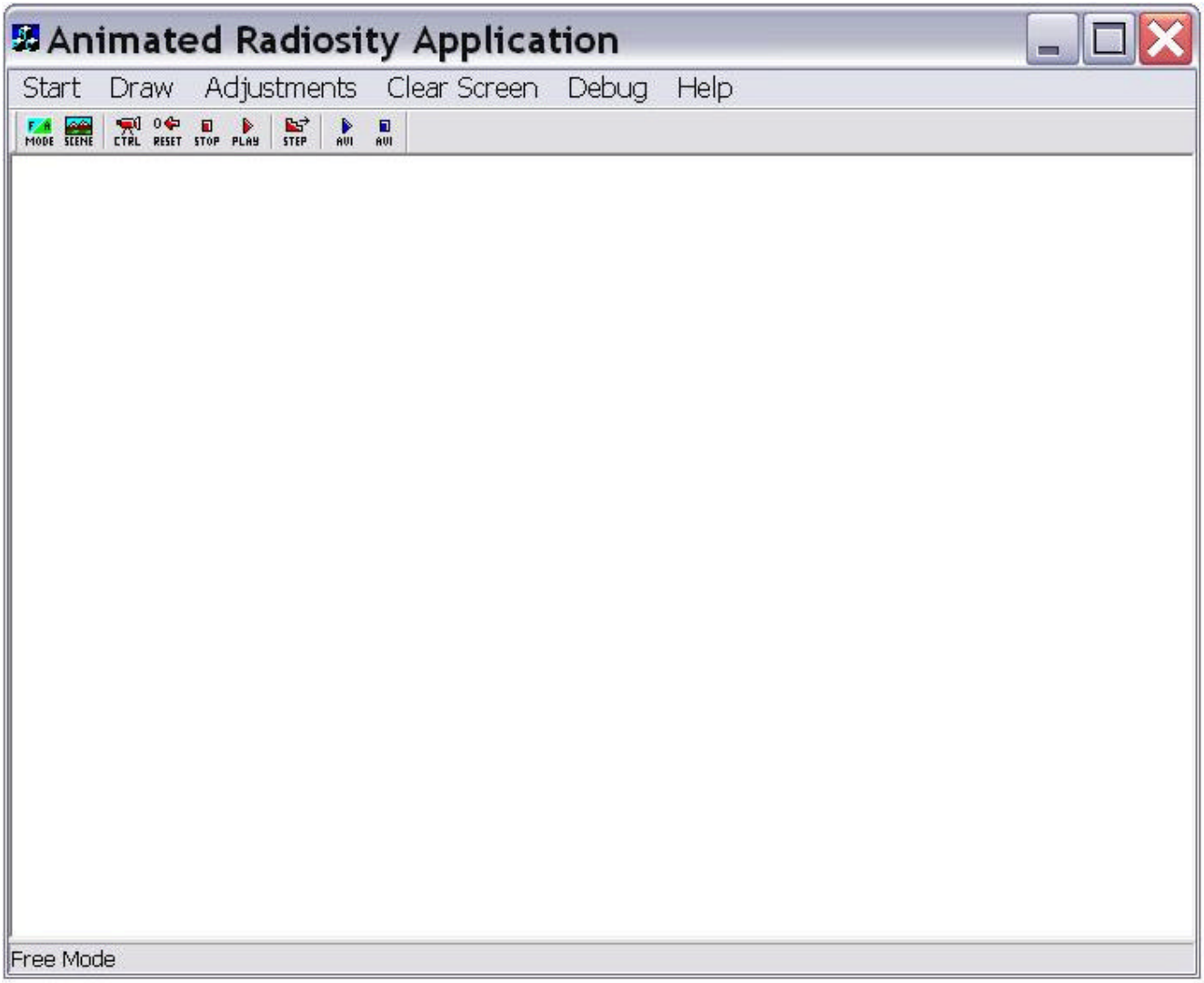

Figure 9: Free Mode

Animate mode provides the tutorial on the Radiosity algorithm. The user is no longer free to adjust the scene in any way. However, all of the animation controls can be used. Animate mode has divider bars that appear on the screen. This distinguishes its appearance from Free mode. The divider bars provide some modularity to the screen in order for the animation to make efficient use of the screen space. Figure 10 shows a screenshot of the application in Animate mode.

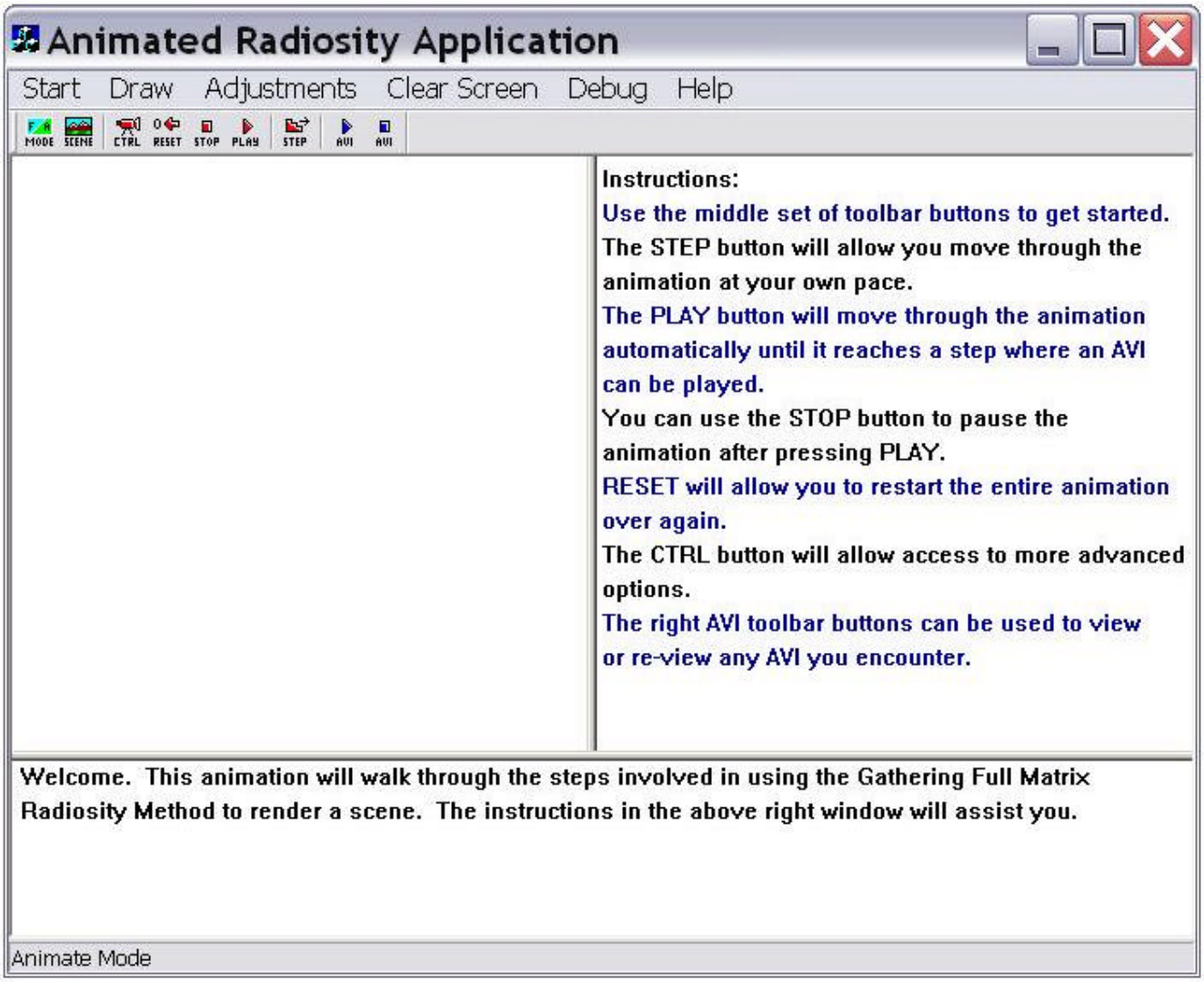

Figure 10: Animate Mode

3.2. Menus

There are six menus located at the top of the application. These are Start, Draw, Adjustments, Clear Screen, Debug, and Help.

3.2.1. Start

The Start menu is used to initialize or reinitialize a scene. Figure 11 shows the contents of the Start menu.

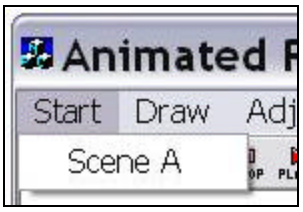

Figure 11: Start Menu

There is currently only one scene available for selection. Scene A must be selected prior to using the Draw menu and also prior to switching to Animate mode.

3.2.2. Draw

Any selection from the Draw menu can be used in Free mode only. Figure 12 shows the selections available.

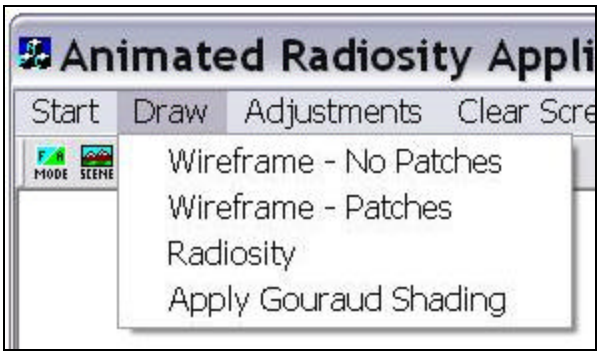

Figure 12: Draw Menu

The scene can be drawn to the screen in a wireframe mode either with or without patches shown. The Radiosity menu item will show the rendered scene without Gouraud shading to smooth out the patches. Apply Gouraud Shading will show the final rendered scene and can only be selected after Radiosity has been selected. The Draw menu is useful for viewing the scene after it has been adjusted and also after various scene properties are set using the Edit Scene Properties toolbar button.

3.2.3. Adjustments

There are various ways to adjust the scene. The scene can be translated, scaled, and rotated. The viewing parameters can also be increased or decreased. Once a selection is made, left clicking anywhere in the viewing space of the window will continue to adjust the scene in that manner.

Figure 13 shows the options available for translating the scene. The scene can be translated in the positive or negative X, Y, or Z direction.

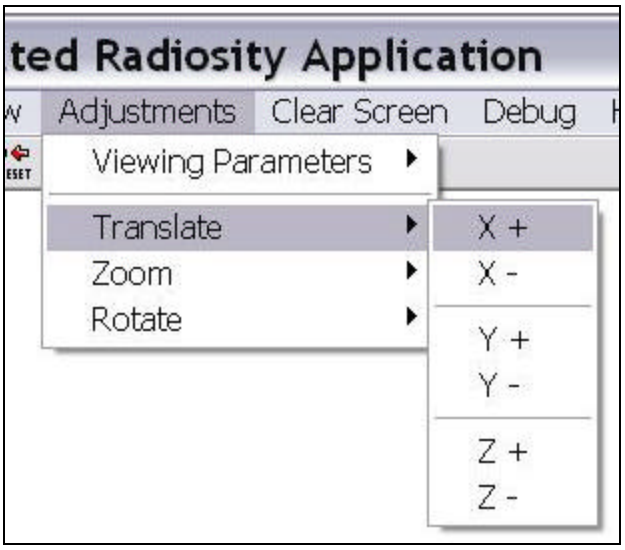

Figure 13: Translate Menu Options

Each time the scene is adjusted, the World Coordinates are recalculated. Because of this, the new World Coordinates must be converted to Screen Coordinates through the viewing pipeline. Figure 14 shows how translating the scene along the different axis' will adjust it to a new position. The arrow along the axis' indicate a positive direction.

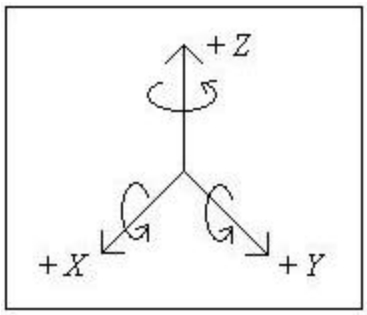

Figure 14: Axis For Scene Adjustments

Figure 15 shows the options available for rotating the scene. The scene can be rotated in the positive or negative X, Y, or Z direction. Figure 14 shows how rotating the scene about a different axis will adjust it to a new position. The arrows around each axis indicate a positive direction. When looking down each axis towards the origin, counter-clockwise is the positive direction for rotation.

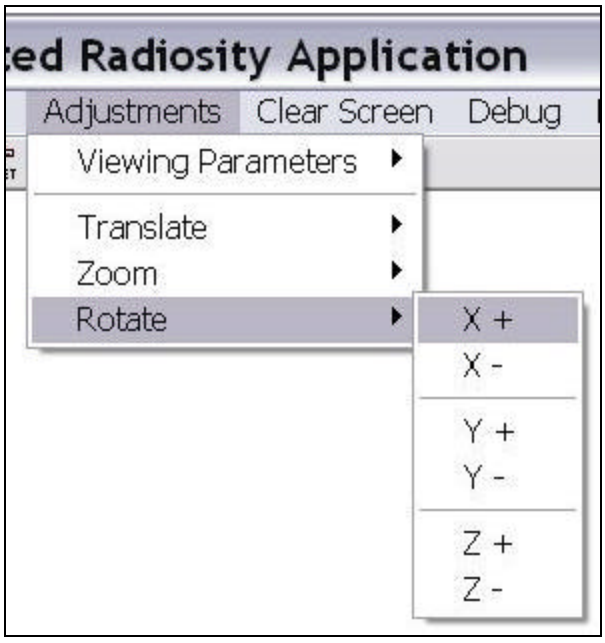

Figure 15: Rotate Menu Options

Figure 16 shows the options available for zooming. The scene can be zoomed in or out. These menu items will scale the scene up or down around a specific point.

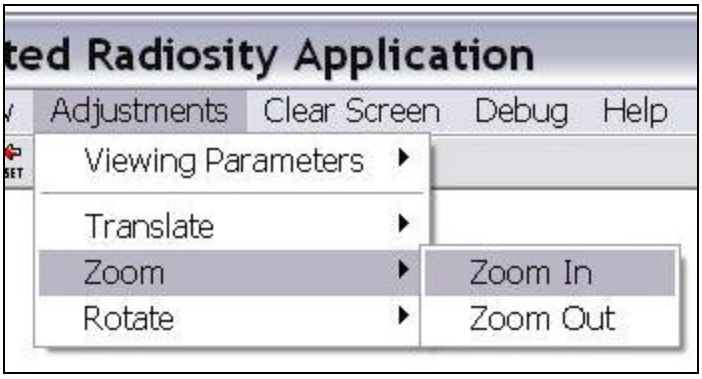

Figure 16: Zoom Menu Options

Finally, the Viewing Parameters for the 4-Parameter Viewing Pipeline can be adjusted up or down. Figure 17 shows the menu options for this. In this viewing pipeline "Rho" (not to be confused with radiosity reflectivities) is the distance between the observer and the origin of the world coordinates system, "Theta" the observer's azimuthal angle, "Phi" the observer's polar angle, and "Distance" the distance between the observer and the projection plane.

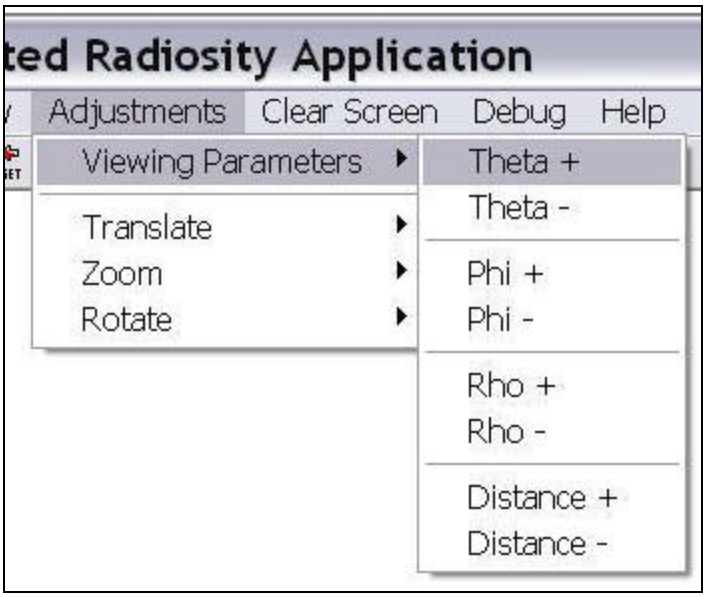

Figure 17: Viewing Parameter Menu Options

Figure 18 shows the 4-parameter viewing system. Scene vertices are initially defined as world coordinates. Those vertices are then converted to viewing coordinates. Viewing coordinates are located behind a specific viewpoint. Next the 3D vertices are projected onto a 2D plane and converted to vertices in a viewport, or screen coordinates that the user can see.

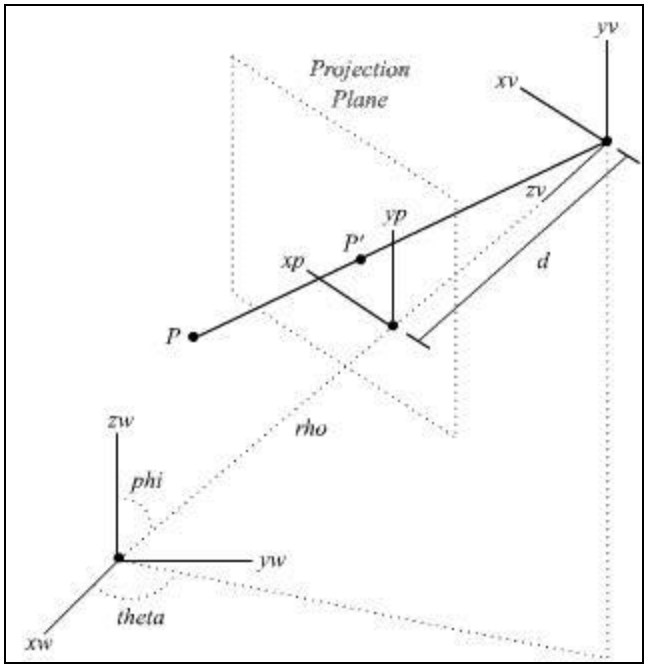

Figure 18: Four Parameter Viewing Pipeline

3.2.4. Clear Screen

The Clear Screen menu item can be chosen to clear the entire screen. If in Free mode, the screen will be completely blanked out. In Animate mode, the screen will be blanked out except for the divider bars. The Clear Screen menu item shown in Figure 17 does not have any submenus.

3.2.5. Debug

This menu can be used for debugging purposes while code is being developed. Selecting Data Dump from this menu dumps data into the output file out.txt within the workspace folder. This menu option may be useful for future application improvements. Figure 19 shows the contents of the Debug menu.

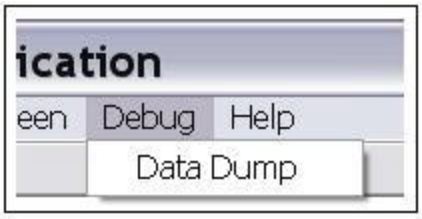

Figure 19: Debug Menu

3.2.6. Help

Figure 20 shows the contents of the Help menu.

Figure 20: Help Menu

The Getting Started menu will display a dialog box containing some basic instructions for using the application. The dialog box is non-modal so the user may keep it open while continuing to use the application interface. Selecting About RadiosityF from the help menu will display the version information.

3.3. Toolbar Buttons

Figure 21 shows the entire toolbar. Dividers have been strategically placed in order to provide a more intuitive interface. Both controls for the AVI player are located together. The regular animation buttons have been placed together and colored red. The step button contains dividers around it to set it off among the rest of the animation controls since it is the most frequently used button.

Figure 21: Toolbar Buttons

3.3.1. Mode

There must be an interface item to allow the user to toggle between Free and Animate modes in order to access the functionality of both. The Mode toolbar button accomplishes this. When this button is pressed within the application, the OnScreenMode handler is called to complete the transition. Figure 22 shows the Mode button.

Figure 22: Mode Button

3.3.2. Edit Scene Properties

This toolbar button lets the user access various dialog panels in order to set some scene properties. These properties include setting:

- Colors for the wireframe scene
- Reflectivities and emissivities for rendered scene
- The number of patches in the scene
- The number of cells in the hemicubes

When this button is pressed within the application, the OnSceneProperties handler is called. If the application is in Animate mode, a message box will appear asking the user to switch to Free mode first. Figure 23 shows the Scene button.

Figure 23: Scene Button

3.3.3. Edit Animation Controls

Clicking this toolbar button will open a dialog box giving the user two options to set. The first option will ask the user how many seconds he or she would like the animation to wait in between frames. When the user plays the animation, the application will wait that particular number of seconds after showing each frame. The other option asks which section number the user would like the animation to start at. The tutorial is subdivided into section numbers, each of which explains and/or demonstrates one set of features of the radiosity algorithm. Since the animation is lengthy, this provides an intuitive way for the user to jump to the material that he or she would like to see without watching the animation from the beginning. If the application is in Free Mode, a message box will appear asking the user to switch to Animate mode first. This button calls the OnAnimationControls handler. Figure 24 shows the Ctrl button.

Figure 24: Ctrl Button

3.3.4. Reset Animation

At any time while the user is either stepping through or playing the animation, this button can be pressed to reinitialize the animation. Once this button is pressed, the user can select play or step to start the animation from the beginning. Pressing this button will call the OnAnimateRestart handler. If the application is in Free mode, a message box will appear asking the user to switch to Animate mode first. Otherwise, any Windows timers are killed and the frame number of the animation is set back to 0. Figure 25 shows the Reset button.

Figure 25: Reset Button

3.3.5. Stop

The Stop button will call the OnPauseAnimation handler. If the application is in Free mode, a message box will appear asking the user to switch to Animate mode first. Otherwise, this button kills the Windows timer. If the animation is playing and the timer is killed, the application will cease stepping through the animation automatically. It will hold at the current frame number until the user selects something. The user may choose to press the play button. A new timer will be started and the animation will continue automatically stepping through frames. The user may also choose to step through the rest of the animation manually. This button simply provides the user with more control over the animation. Figure 26 shows the Stop button.

Figure 26: Stop Button

3.3.6. Play

Pressing this button will call the OnAnimateFwd handler. If the application is in Free mode, a message box will appear asking the user to switch to Animate mode first. Otherwise, this button will allow the user to play the animation through, starting with the current frame number. This button sets a Windows timer and calls the OnAnimateStepFwd for each next frame in the animation. The OnAnimateStepFwd actually dictates what gets drawn to the screen for each frame. OnAnimateFwd simply starts the timer to automate the calling of each next frame. The timer value is either the default value or the value that the user set in the Edit Animation Controls dialog. Figure 27 shows the Play button.

Figure 27: Play Button

3.3.7. Step

Pressing this button will call the OnAnimateStepFwd handler. If the application is in Free mode, a message box will appear asking the user to switch to Animate mode first. Otherwise, this button will allow the user to step through to the next frame of the animation. No Windows timers are set. This button essentially allows the user to walk through the animation at his or her own pace. Figure 28 shows the Step button.

Figure 28: Step Button

3.3.8. AVI Play

The AVI Play button can be used to play or replay any AVI animation that is encountered within the application. This button calls the OnAviPlay handler. This button will not resume a paused AVI animation, but will restart the AVI from the first frame. Figure 29 shows the AVI Play button.

Figure 29: AVI Play Button

3.3.9. AVI Stop

The AVI Stop button can be used to pause any AVI animation that is encountered within the application. This button calls the OnAviStop handler. Figure 30 shows the AVI Stop button.

Figure 30: AVI Stop Button

3.4. Miscellaneous

The following features are intentionally included:

?? The window is specifically sized to 700 x 550 and is centered on the screen when the application starts. The user cannot resize the window. This is to maintain better control over how the data is presented within the application.

?? The toolbar buttons contain tooltips. If the user mouses over the toolbar buttons, a box will pop up giving the user the name of the button. This makes the purpose of each button more clear.

?? A status bar is located at the bottom of the application. The status bar indicates which mode the application is in (Free or Animate). If in Free mode, the status bar will also appropriately indicate what is drawn to the screen. If in Animate mode, the status bar will also indicate what section the animation is in.

4. Code structure

This section describes the program state variables, dialog classes, enumerated lists, structs, and functions that were created for the application. Each plays an important role in the overall functionality of the application. The implemented animation describes some of these if they pertain directly to the Radiosity algorithm.

The code is mostly procedural with as few classes as possible. This is so that the animation can be clearer on how the Radiosity algorithm works without the algorithm being performed using complex classes.

4.1. Program State Variables (Booleans)

Several Boolean variables are declared in the RadiosityFView.h file. These variables keep track of specific events that have occurred in order to provide some program control.

initComplete:
Set to False when application is started. This variable is set to True when the OnInitA handler is called indicating that the scene has been initialized, or the data structures have been filled. In Free mode, nothing from the Draw menu may be selected until Scene A has been initialized. The user may not toggle to Animate mode until Scene A has been initialized. Scene A can be initialized by selecting Scene A from the Start menu.

True: Scene initialized
False: Scene not initialized

RadiosityCalculated:
Set to False when application is started. This variable is set to True at the end of the OnRender function indicating that the Radiosity values have been calculated. The OnShading function will not perform Gouraud shading on the scene until the Radiosities have been calculated.

True: Radiosities calculated
False: Radiosities not calculated

justStarted:
Set to True when application is started. This variable is set to True in the OnInitA handler. When the user selects Scene A from the Start menu, the message box that pops up will indicate if the scene has been initialized for the first time or if the scene has been reinitialized.

True: Initialization message box indicates scene is being initialized for the first time.
False: Initialization message box indicates scene is being reinitialized.

callFormFactors:
Set to True when application is started. Set to False during the Radiosity Demo section of the animation after the form factors have been calculated once. This provides some efficiency as the animation steps through the patches being calculated. The form factors only need to be calculated once. In the OnRender handler, the form factors will only be calculated if callFormFactors is set to True.

True: OnRender calculates form factors before calculating Radiosities.
False: OnRender does not calculate form factors before calculating Radiosities.

wireframe_key:
When the application is started, this variable is set to False. In Free mode, the handlers OnNoPatchesAllFaces and OnPatchesAllFaces may be freely accessed by the user to draw the wireframe scene to the screen. This is done when the user selects a wireframe option from the Draw menu. In Animate mode, the wireframe options from the Draw menu are unavailable. The handlers OnNoPatchesAllFaces and OnPatchesAllFaces are inaccessible unless the Boolean variable wireframe_key is set to True. The animation steps call these handlers to display the scene. In order to access the handlers, wireframe_key is set to TRUE and then set back to False once the call has been made.

True: Access to OnNoPatchesAllFaces and OnPatchesAllFaces granted in Animate mode.
False: Access not granted in Animate mode.

render_key:
When the application is started, this variable is set to False. In Free mode, the handler OnRender may be freely accessed by the user to draw the rendered scene to the screen. This is done when the user selects the Radiosity option from the Draw menu. In Animate mode, the Radiosity option from the Draw menu is unavailable. The handler OnRender is inaccessible unless the Boolean variable render_key is set to True. The animation steps call this handler to display the scene. In order to access the handler, render_key is set to True and then set back to False once the call has been made.

True: Access to OnRender granted in Animate mode.
False: Access not granted in Animate mode.

enableAVICtrls:
When the application is started, this variable is set to False. It remains False until Animate mode is selected and a step is reached where the application plays an AVI. enableAVICtrls is set to True in order to access the OnAVIPlay and OnAVIStop handlers. It is set back to False as soon as the AVI is closed. This is to indicate to the user when the AVI Play and Stop buttons cannot be used.

True: Access to OnAVIPlay and OnAVIStop granted.
False: Access not granted.

Drawing State Variables:
In order for the application to refresh the screen properly, the OnDraw handler is used for anything drawn to the screen except for an AVI animation. When the application is minimized, dragged across the screen, or put in the background, it will resume showing the correct contents of the screen. Boolean variables are used to control what is visible. The following Boolean variables are set to True in OnDraw to show their corresponding contents and set to False when the contents should be hidden.

drawFrameBuffer: Controls the display of whatever is located within the 2D frame buffer. The frame buffer is used to hold pixel colors for a rendered scene.

drawAnimationScreen: The animation screen refers to the vertical and horizontal bars that appear when in animate mode. This variable controls when the bars should appear.

drawBitmapL1, drawBitmapL2, drawBitmapR1, drawBitmapR2: This application has the ability to display four bitmaps at one time. The assumption is that two bitmaps could be displayed in each of the left and right windows in animation mode. DrawBitmapL1 and drawBitmapL2 are set to True in order to display bitmaps in the left window. Likewise, drawBitmapR1 and drawBitmapR2 are set to True in order to display bitmaps in the right window.

drawTextRight[17]: This array of 17 Booleans is used to display text in the right window of the application. Each array entry corresponds to one line of text.

drawTextRight[5]: This array of 5 Booleans is used to display text in the lower window of the application. Each array entry corresponds to one line of text.

The following variables control which objects in the wireframe scene are displayed:

- drawRoomPatches
- drawRoomNoPatches
- drawBoxPatches
- drawBoxNoPatches
- drawLight1Patches
- drawLight1NoPatches
- drawLight2Patches
- drawLight2NoPatches

The application always displays all objects in the wireframe scene as opposed to a few.

Figure 31 summarizes the available drawing controls and shows which type of content can be displayed in which windows.

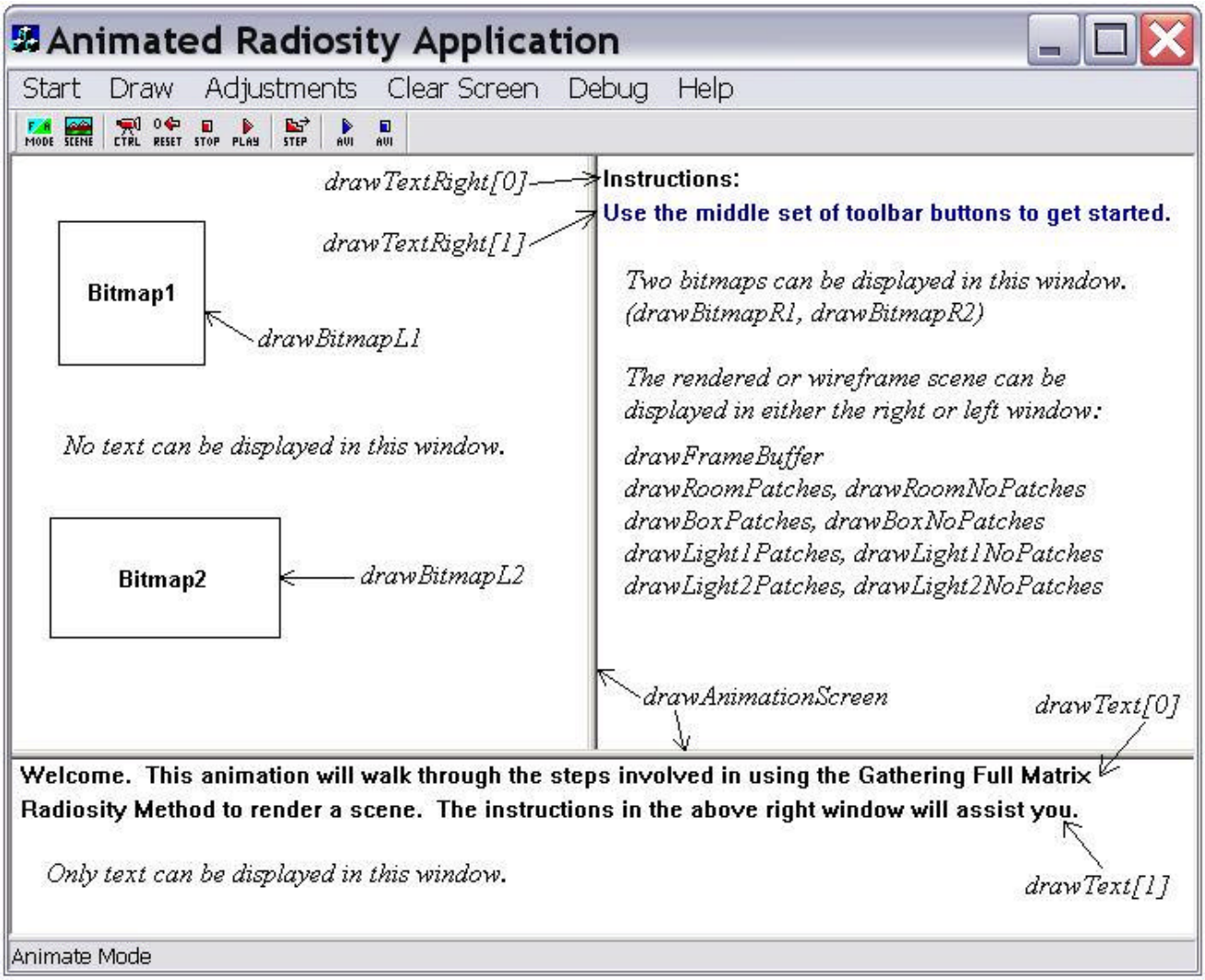

Figure 31: Drawing State Controls

4.2 Dialog Class Descriptions

Although the majority of the application is procedural without the use of complex object-oriented programming, some classes are still needed in order to make use of MFC dialog boxes. The dialog boxes can be accessed by clicking on toolbar buttons within the application. Each dialog box has an associated class. The dialog options provide the user with more application versatility, though the application will use default values if the dialog box options are not chosen.

4.2.1. CAnimationControls

Editing Animation Controls provides a way for the user to set a timer and select a section from which to begin. Setting a timer will specify the number of seconds the animation should wait in between frames before proceeding. This only applies if the

user chooses to play the animation, not step through it. The animation contains many steps, but is divided into several sections. If the user does not wish to start from the beginning of the animation, a section may be selected in this dialog box.

AnimationControls.cpp converts the user input strings into usable data and ensures that the entries are within the valid range. RadiosityFView.cpp creates the CAnimationControls object. The dialog object is set as modal, meaning all other application functionality freezes until the user makes a selection within the dialog box. If the user set a timer value, then the Windows timer variable is set. If the user set a section number, the step_ctr variable is set. This dialog box may only be used in Animate mode. Figure 32 shows the Animation Controls dialog box.

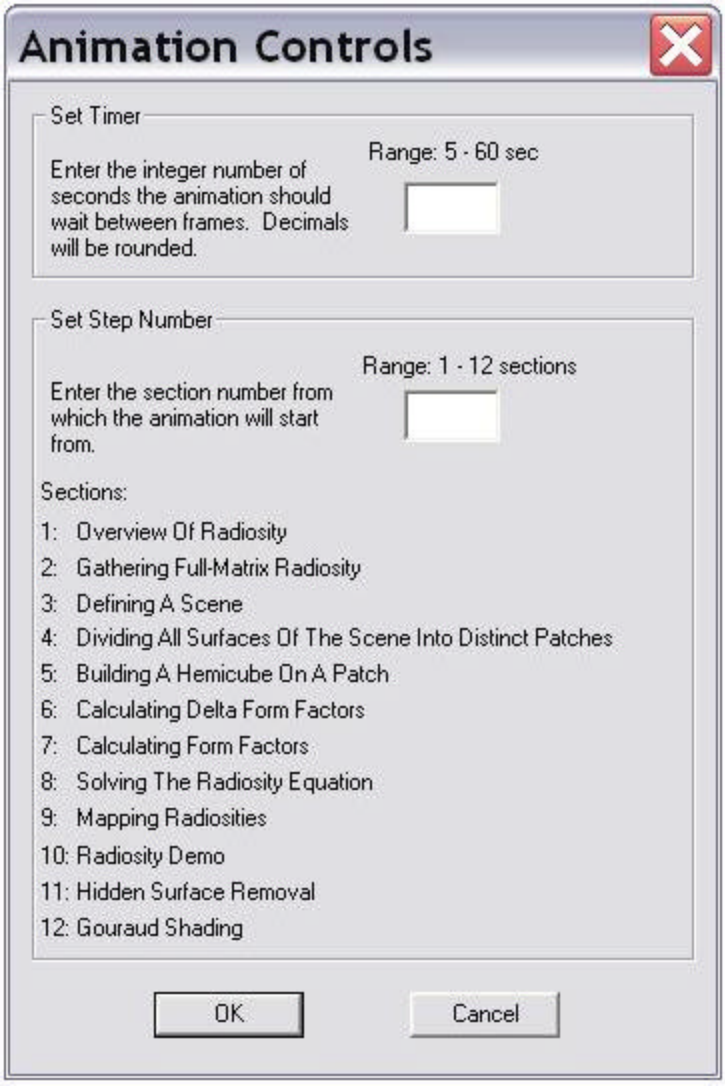

Figure 32: Animation Controls Dialog Box

4.2.2. CSceneAOptions

Editing Scene A Options provides a way for the user to access four other dialog boxes. These are Wireframe Colors, Render Properties, Patches, and Hemicubes. CSceneAOptions creates a dialog object for each and sets them to modal so all other functionality freezes until a selection is made. This dialog box may only be accessed in Free mode. Figure 33 shows the Scene A Options dialog box.

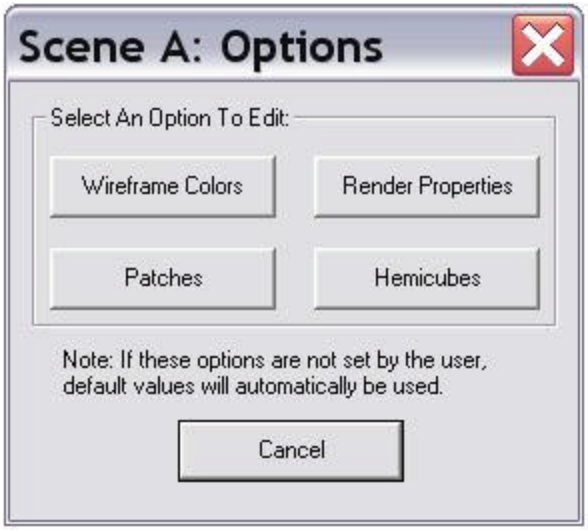

Figure 33: Scene A Options Dialog Box

4.2.3. CHemiOps

Editing Hemicube Options provides a way for users to choose how many cells they would like on each hemicube face. The top face and the side faces of the hemicube are set separately. All hemicubes that are created for every patch will then be divided into the specified number of cells.

HemiOps.cpp converts the user input strings into usable data and ensures that the entries are within the valid range. The range is quite strict since the storage requirements and time taken to make calculations escalates significantly as the number of hemicube cells increases. Figure 34 shows the Hemicube Options dialog box.

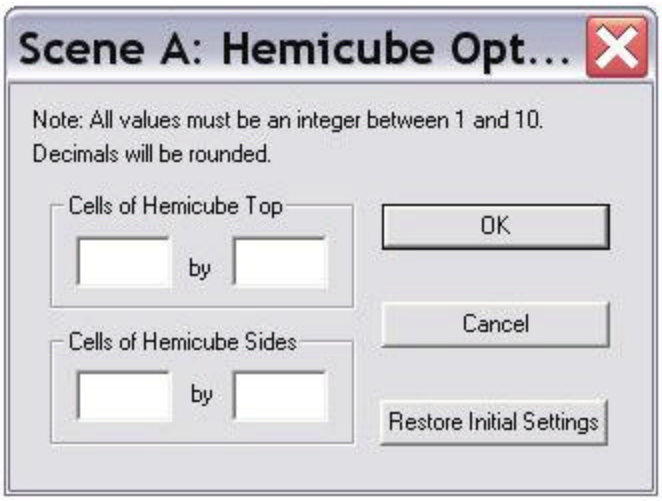

Figure 34: Hemicube Options Dialog Box

4.2.4. CPatchOpsA

Editing Patch Options provides a way for users to choose how many patches they would like to be created on various parts of Scene A. The dialog box setup is intuitive once the user sees at least a wireframe version of the scene. The entries are labeled appropriately from the users' perspective. For example, the left wall of the room is the wall on the users' left side.

PatchOpsA.cpp converts the user input strings into usable data and ensures that the entries are within the valid range. The range is quite strict since the storage requirements and time taken to make calculations escalates significantly as the number of patches increases. Figure 35 shows the Patch Options dialog box.

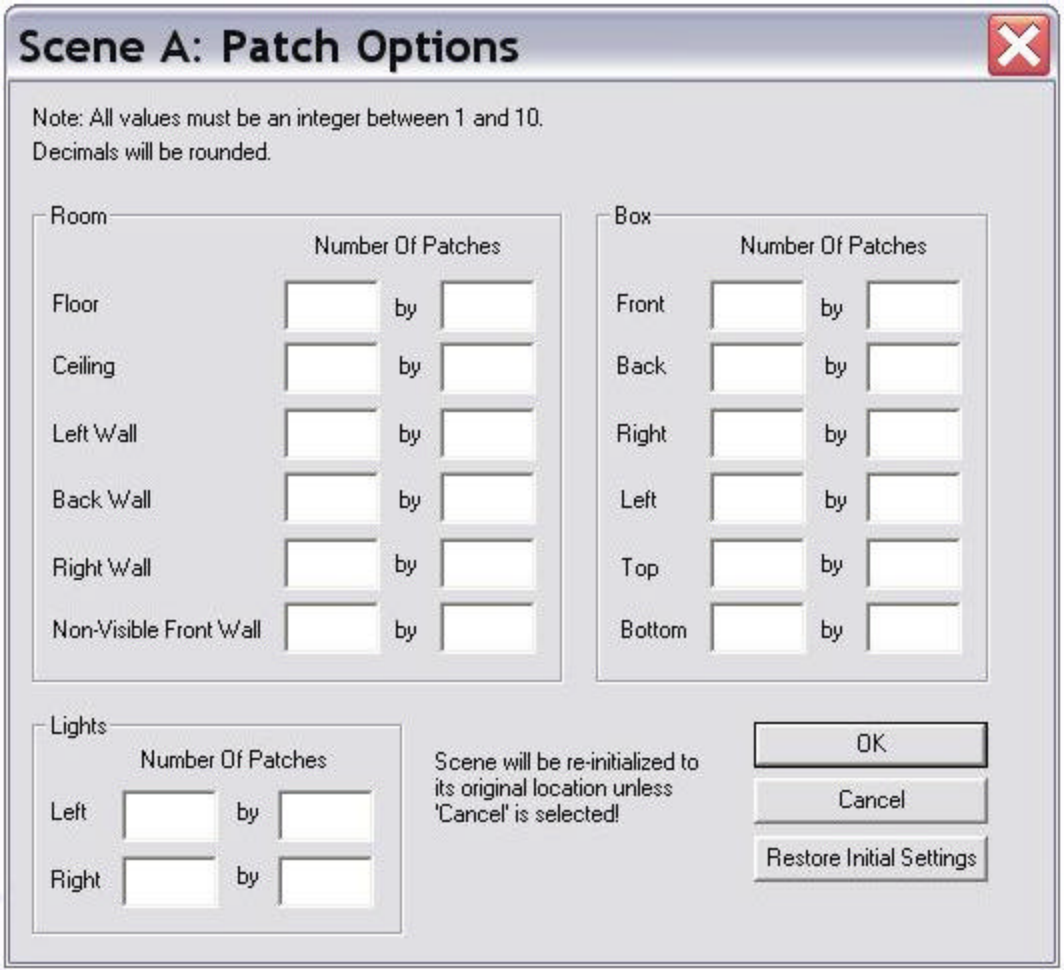

Figure 35: Patch Options Dialog Box

4.2.5. CRenPropsA

Editing Render Options provides a way for the user to set reflection and emission values for each part of Scene A. Every object in the scene has red, green and blue reflectivities, but only light sources have emission values.

RenPropsA.cpp converts the user input strings into usable data and ensures that the entries are within the valid range. Reflectivities must be a value between 0 and 1 in order for the Gauss-Seidel iteration to be guaranteed to converge. The limits on the emission values were set somewhat arbitrarily. Figure 36 shows the Render Options dialog box.

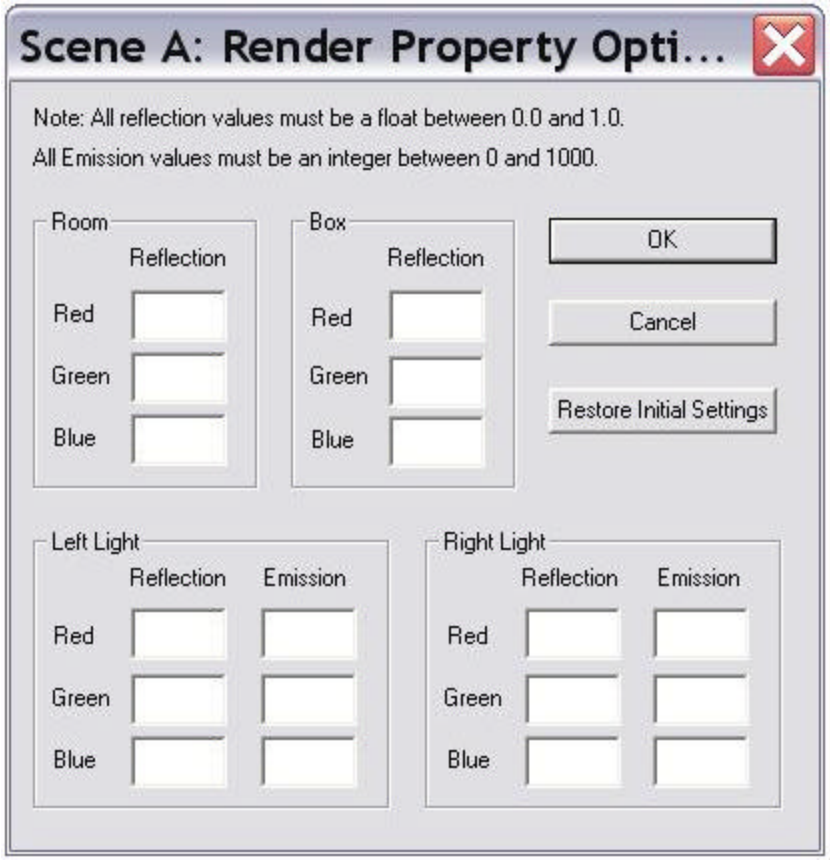

Figure 36: Render Property Options Dialog Box

4.2.6. CWFColorsA

Editing Wireframe Color Options provides a way for the user to set colors for Scene A. Setting these colors has no impact on the final rendered scene, only the wireframe model.

WFColorsA.cpp converts the user input strings into usable data and ensures that the entries are within the valid range. The 0 to 255 RGB scale dictates the range of values allowed. Figure 37 shows the Wireframe Color Options dialog box.

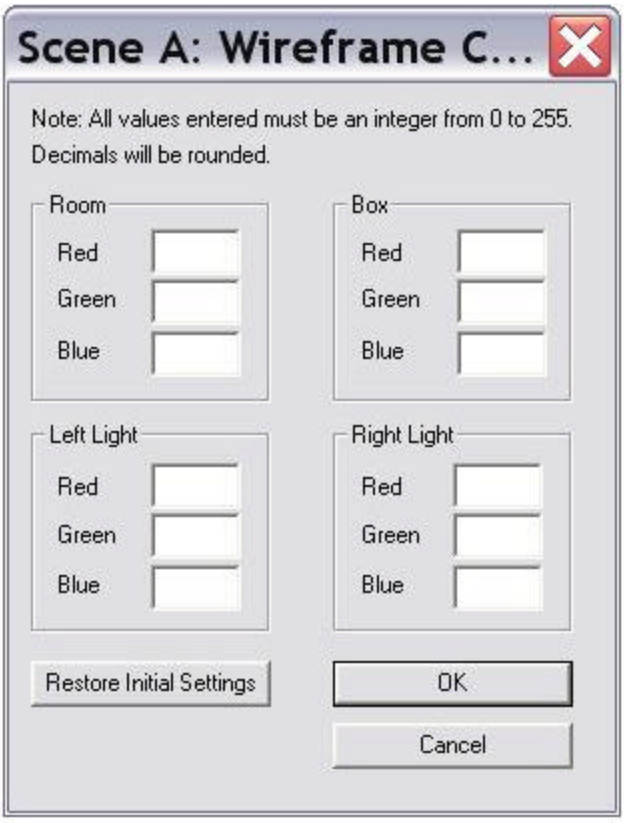

Figure 37: Wireframe Color Options Dialog Box

4.2.7. CHelp

The CHelp class creates a modeless dialog box containing general instructions for the application. The dialog box does not contain any inputs or outputs except for an 'OK' button.

The OnInstructions handler in RadiosityFView.cpp creates a CHelp object when the Help/Getting Started menu item is selected. Figure 38 shows the Help dialog box.

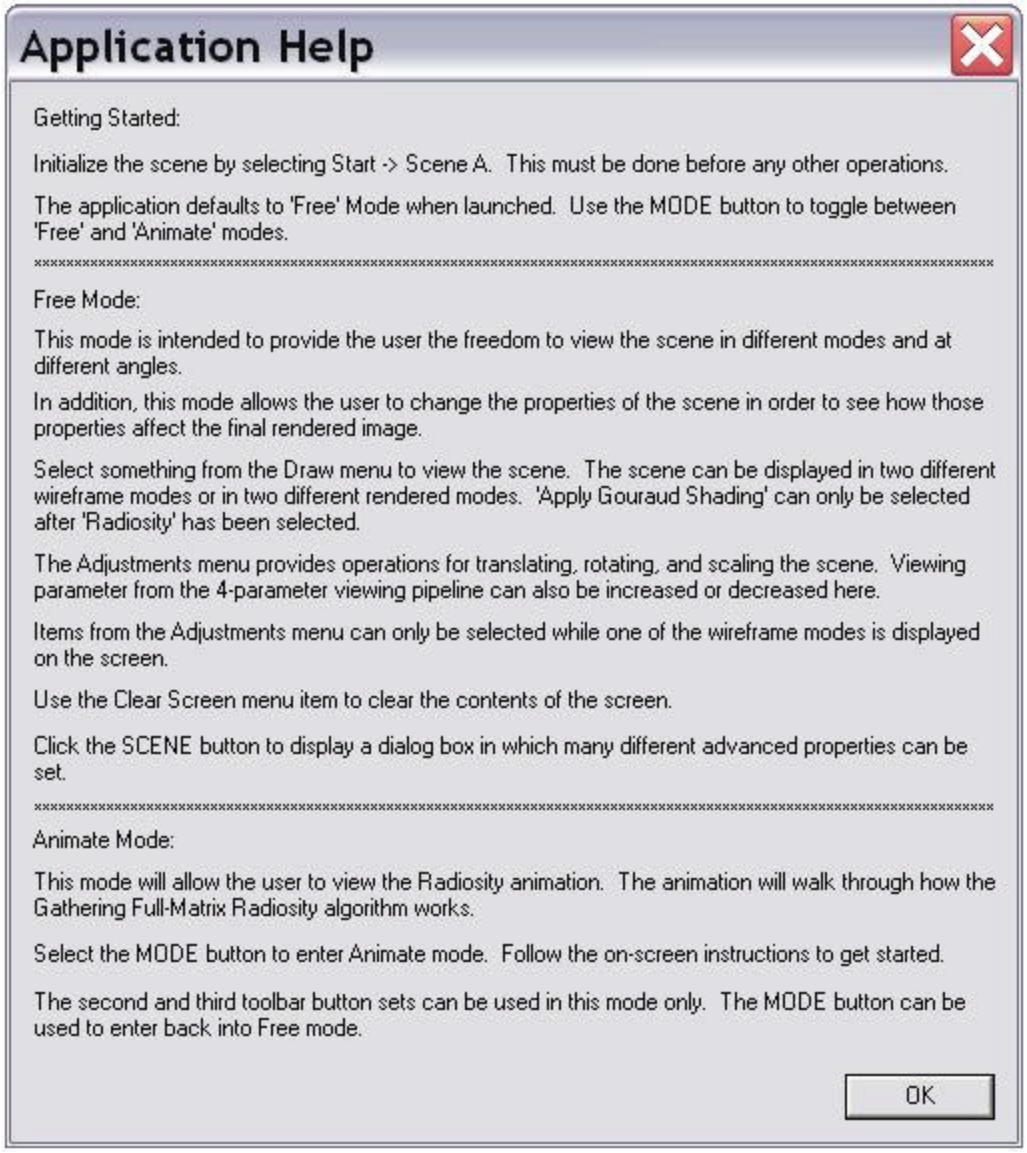

Figure 38: Help Dialog Box

4.3. Data Structures

The methods of storage in this application were designed to provide an easy way to access data. Enumerated lists provide an intuitive way to organize lists of items. Structs group similar data together. All enumerated lists and structs are defined in the RadiosityFView.h file.

4.3.1. Enumerated Lists

View Mode:
The view mode keeps track of what is currently drawn to the screen. By doing this, the application will know what work needs to be done if another mode is selected. There are five possible modes:

```
typedef enum {
        CLEAR,
        WIREFRAME_NOPATCHES_ALLFACES,
        WIREFRAME_PATCHES_ALLFACES,
        RENDERED,
        SHADED
} viewModeType;
```

This variable keeps track of the current view mode at all times while the application is running. It is initialized to CLEAR when the application is started:

```
int viewMode;
```

Screen Mode:
There are two possible screen modes, Free and Animate. Only certain operations can be performed in each mode. By keeping track of the mode, the application maintains easy control over which selections that the user makes can be performed. If the user makes a selection that is not allowed in the current mode, a message box will appear telling the user to toggle to the other mode. By having multiple modes, the application provides more versatility to the user:

```
typedef enum {
        FREE,
        ANIMATE
} screenModeType;
```

This variable keeps track of the current screen mode at all times while the application is running. It is initialized to FREE when the application is started:

```
int screenMode;
```

Translation Type:
The application offers the user several options for adjusting the scene in Free mode. The current translation (adjustment) type is maintained so the user may select the adjustment once and then left click in the window to continue performing that operation. When the current translation type is maintained, the LButtonDown message handler will know what operation to perform on the scene:

```
typedef enum {
        NONE,
        TRANSXPLUS,
        TRANSXMINUS,
        TRANSYPLUS,
        TRANSYMINUS,
        TRANSZPLUS,
        TRANSZMINUS,
        ZOOMPLUS,
        ZOOMMINUS,
        ROTXPLUS,
        ROTXMINUS,
        ROTYPLUS,
        ROTYMINUS,
        ROTZPLUS,
        ROTZMINUS,
        THETAPLUS,
        THETAMINUS,
        PHIPLUS,
        PHIMINUS,
        RHOPLUS,
        RHOMINUS,
        DISTANCEPLUS,
        DISTANCEMINUS

} currentTranslationType;
```

This variable keeps track of the current translation type at all times while the application is running:

```
int currentTranslation;
```

Axis Type:

The axis type is used for rotate adjustment operations. When the application calls the rotate function, it must specify which axis to rotate about. The rotate function is designed to accept one of the following three axis types:

```
typedef enum {
        XAXIS,
        YAXIS,
        ZAXIS
} axisType;
```

Since this is not an enumerated list to keep track of the program state, there is no variable that must maintain a value at all times while the program is running.

Scene Type:
Originally the application was designed with the idea that multiple scenes might later be added. At this time, there is only one possible scene. The scene type keeps track of which scene is currently selected. When no scene is selected, very few operations can be performed. The user can select a scene from the Start menu. If Scene A is selected, then the current scene will be set to SCENEA:

```
typedef enum {
        NOSCENE,
        SCENEA
} sceneType;
```

This variable keeps track of the current scene type at all times while the application is running. It is initialized to NOSCENE when the application is started:

```
int currentSceneType;
```

Object ID:
The patch list keeps track of what object each patch belongs to. This is important because each object in the scene has different properties (i.e. reflection and emission values) that are used for Radiosity calculations. In addition, the non-visible front wall of the room maintains its own object ID so that it is easier for the application to not draw it to the screen, yet include it in all other calculations.

```
typedef enum {
        ROOM1,
        NVROOM1,
        BOX1,
        LIGHT1,
        LIGHT2
} objectType;
```

Since this is not an enumerated list to keep track of the program state, there is no variable that must maintain a value at all times while the program is running.

4.3.2. Structs

3D Coordinate:
This struct allows a 3D floating point coordinate to be declared. This is useful whenever working with World or View coordinates:

```
typedef struct {
        float x;
        float y;
        float z;
```

```
} float_coord_3d_type;
```

Here is an example of how this struct can be used:

```
// Declaration
float_coord_3d_type myCoordinate;

// Set the x-coordinate to 10
myCoordinate.x = 10.1f;
```

Number of Patches:
This struct allows the application to declare the number of patches wide and high required for a polygon or the number cells wide and high for a hemicube face:

```
typedef struct {
        int w;
        int h;
} int_num_patches_type;
```

Here is an example of how this struct can be used:

```
// Declaration
int_num_patches_type myPatches;

// Set the number of patches to 5x4
myPatches.w = 5;
myPatches.h = 4;
```

Edge Definition:
The interpolation_utility function uses this struct for Hidden Z Buffer and Gouraud shading operations.

```
typedef struct {
        int a;
        int b;
        int ia_red;
        int ia_green;
        int ia_blue;
        int ib_red;
        int ib_green;
        int ib_blue;
} int_edge_define_type;
```

The following local declaration is made within the interpolation_utility function:

```
int_edge_define_type edge_list[4];
```

Here is an example of how an item may be accessed:

```
// Sets the endpoint b of edge 2 to vertex 3 of the current polygon
edge_list[2].b = 3;
```

An array of 4 is declared, so that each item in the struct will be maintained for a different edge of a 4-vertex polygon. a and b contain the integer vertex number of the endpoints of the edge. ia and ib contain the intensity values for each edge endpoint. There is a red, green, and blue value. These are necessary in order to perform Gouraud shading.

Color Definition:

This struct allows for storage of a color. It is intended to be used with the 0 to 255 RGB scale. For example, the frame buffer uses this struct to store colors in a 2D array.

```
typedef struct {
        int r;
        int g;
        int b;
} int_color_type;
```

Here is an example of how this struct can be used:

```
// Declaration
int_color_type myColor;

// Sets myColor to bright red
myColor.r = 255;
myColor.g = 0;
myColor.b = 0;
```

3D Vector:

This struct allows a 3D vector to be declared. Though this is similar to the 3d coordinate struct, there is an important distinction between a 3d coordinate and vector that is observed within the application. A struct for each is provided to avoid confusion within the code.

```
typedef struct {
        float i;
        float j;
        float k;
} float_vector_3d_type;
```

Here is an example of how this struct can be used:

```
// Declaration
float_vector_3d_type myVector;
```

```
// The vector 3.2i + 4.2j + 5.2k is defined:
myVector.i = 3.2f;
myVector.j= 4.2f;
myVector.k = 5.2f;
```

Patch List:

The patch list contains an entry for every patch in the entire scene. This struct is set up to keep track of the important properties of each patch:

```
typedef struct {
        float_coord_3d_type patch_coords[4];
        int obj_id;
        float area;
        float rad_red;
        float rad_green;
        float rad_blue;
        int_color_type vertex_intensity[4];
} patch_list_type;
```

patch_coords are a list of the four corner vertices of the patch.
The obj_id tells what object the patch belongs to (i.e. room, box, etc).
area stores the area of the patch
rad provides storage for the Radiosities that are calculated for red, green, and blue channels. The Radiosity mapped 0 to 255 values are also stored here.

The following declaration is made in the RadiosityFView.h file:

```
patch_list_type *patch_list;
```

The patch list array is dynamically allocated since the user has the ability to change the number of patches through a dialog box option.

Hemicube and Delta Form Factor Data:

This struct is used to store hemicube vertex data and delta form factor data The following struct items are intended to be stored for every cell of every face of every hemicube:

```
typedef struct {
        float *dff;             // Delta form factor
        float *d;               // Closest patch so far seen by a cell.
        int *ff;                // Holds index to most recent FF for a particular cell
        float_coord_3d_type *cell_vertices; // All Hemicube cell vertices
} hemicube_type;
```

dff stores the delta form factor for the current hemicube cell.

d stores the distance of the closest patch projecting to that cell.
ff stores the patch index of the closest patch projecting to that cell. The patch index is the index into the patch_list array.
cell_vertices stores the four corner vertices of the hemicube cell.

The following declaration is made in the RadiosityFView.h file:

hemicube_type (*hemicube_attributes)[5];

This is declared as a 2D array: number of patches x 5 hemicube faces. The bottom face of the hemicube is disregarded since no other patches in the scene will project to that face. Note that the number of patches part of the array is dynamically allocated since the user can change the number of patches within the application.

4.4. Function Descriptions

Each function defined within the application plays an important role. This section describes each function in the RadiosityFView.cpp file in detail.

4.4.1. Setup

PreCreateWindow:
Return: Bool (Part of MFC setup)
Parameters: None used.
Calls functions: None
Creates Object: None
Description: Sets color defaults, etc

4.4.2. Scene Definition

OnInitA:
Return: None
Parameters: None
Calls functions: OnClearScreen, GetSceneA, wcs_to_scs
Creates Object: None
Description: Message handler to initialize Scene A. Initializes buffers.

GetSceneA:
Return: None
Parameters: None
Calls functions: object_patch_setup
Creates Object: None
Description: Local helper function to fill data structures with additional Scene A information.

patch_list_setup:

Return: None
Parameters:
Out - patch_list_type patch_list_update[ ]: Contains the updated patch list
Calls functions: None
Creates Object: None
Description: Local helper function to update information in the patch list. The patch list needs to be updated whenever the scene is adjusted.

4.4.3. Coordinate System Conversion

wcs_to_vcs:
Return: None
Parameters:
In - int num_vertices: Number of vertices to be converted
In - float_coord_3d_type wcs_in[ ]: Listing of 3D World Coordinate vertices
Out - float_coord_3d_type vcs_out[ ]: Output listing of 3D Viewing Coordinate vertices
In - float theta_deg_in: Azimuthal angle in Degrees
In - float phi_deg_in: Polar angle in Degrees
In - float rho_in: Distance from World Coordinate System origin

Calls functions: get_view_matrix
Creates Object: None
Description: Local helper function to convert vertices from the 3D World Coordinate System to the 3D Viewing Coordinate System.

This function is part of the 4-parameter viewing pipeline.

vcs_to_scs:
Return: None
Parameters:
In - int num_vertices: Number of vertices to be converted
In - float_coord_3d_type vcs_in[ ]: Listing of 3D Viewing Coordinate vertices
Out - Point scs_xy_out[ ]: Output listing of 2D Screen Coordinate vertices
In - float d_in: Distance from Viewing Coordinate System origin
In - float window_viewport_setup[ ]: 2D Viewing Transformation matrix
Calls functions: None
Creates Object: None
Description: Local helper function to convert vertices from the 3D Viewing Coordinate System to the 2D Screen Coordinate System.

wcs_to_scs:
Return: None
Parameters:
In - int num_vertices: Number of vertices to be converted
In - float_coord_3d_type wcs_in[ ]: Listing of 3D World Coordinate vertices

Out - Point scs_xy_out[ ]: Output listing of 2D Screen Coordinate vertices
In - float theta_deg_in: Azimuthal angle in Degrees
In - float phi_deg_in: Polar angle in Degrees
In - float rho_in: Distance from World Coordinate System origin
In - float d_in: Distance from Viewing Coordinate System origin
In - float window_viewport_setup[ ]: 2D Viewing Transformation matrix

Calls functions: get_view_matrix
Creates Object: None
Description: Local helper function to convert vertices from the 3D World Coordinate System to the 2D Screen Coordinate System.

get_view_matrix:
Return: None
Parameters:
Out - float out[ ]: Output 4x4 viewing transformation matrix
In - float theta_deg_in: Azimuthal angle in Degrees
In - float phi_deg_in: Polar angle in Degrees
In - float rho_in: Distance from World Coordinate System origin

Calls functions: None
Creates Object: None
Description: Local helper function to set up a 3D viewing transformation matrix for translation operations.

4.4.4. Scene Adjustments

OnTransXPlus:
Return: None
Parameters: None
Calls functions:
OnDraw
translate
wcs_to_scs
patch_list_setup

Creates Object: None
Description: Menu Message Handler to translate the scene in +X direction. Can only be selected in Free Mode. Can only be selected in a Wireframe Mode.

OnTransXMinus:
Return: None
Parameters: None
Calls functions:
OnDraw
translate

wcs_to_scs
patch_list_setup

Creates Object: None
Description: Menu Message Handler to translate the scene in -X direction. Can only be selected in Free Mode. Can only be selected in a Wireframe Mode.

OnTransYPlus:
Return: None
Parameters: None
Calls functions:
OnDraw
translate
wcs_to_scs
patch_list_setup

Creates Object: None
Description: Menu Message Handler to translate the scene in +Y direction. Can only be selected in Free Mode. Can only be selected in a Wireframe Mode.

OnTransYMinus:
Return: None
Parameters: None
Calls functions:
OnDraw
translate
wcs_to_scs
patch_list_setup

Creates Object: None
Description: Menu Message Handler to translate the scene in -Y direction. Can only be selected in Free Mode. Can only be selected in a Wireframe Mode.

OnTransZPlus:
Return: None
Parameters: None
Calls functions:
OnDraw
translate
wcs_to_scs
patch_list_setup

Creates Object: None
Description: Menu Message Handler to translate the scene in +Z direction. Can only be selected in Free Mode. Can only be selected in a Wireframe Mode.

OnTransZMinus:
Return: None
Parameters: None
Calls functions:
    OnDraw
    translate
    wcs_to_scs
    patch_list_setup

Creates Object: None
Description: Menu Message Handler to translate the scene in -Z direction. Can only be selected in Free Mode. Can only be selected in a Wireframe Mode.

OnZoomPlus:
Return: None
Parameters: None
Calls functions:
    OnDraw
    scale
    wcs_to_scs
    patch_list_setup

Creates Object: None
Description: Menu Message Handler to zoom out on the scene. Can only be selected in Free Mode. Can only be selected in a Wireframe Mode.

OnZoomMinus:
Return: None
Parameters: None
Calls functions:
    OnDraw
    scale
    wcs_to_scs
    patch_list_setup

Creates Object: None
Description: Menu Message Handler to zoom in on the scene. Can only be selected in Free Mode. Can only be selected in a Wireframe Mode.

OnRotXPlus:
Return: None
Parameters: None
Calls functions:
    OnDraw
    rotate
    wcs_to_scs

patch_list_setup

Creates Object: None
Description: Menu Message Handler to rotate the scene in the +X direction. Can only be selected in Free Mode. Can only be selected in a Wireframe Mode.

OnRotXMinus:
Return: None
Parameters: None
Calls functions:
OnDraw
rotate
wcs_to_scs
patch_list_setup

Creates Object: None
Description: Menu Message Handler to rotate the scene in the -X direction. Can only be selected in Free Mode. Can only be selected in a Wireframe Mode.

OnRotYPlus:
Return: None
Parameters: None
Calls functions:
OnDraw
rotate
wcs_to_scs
patch_list_setup

Creates Object: None
Description: Menu Message Handler to rotate the scene in the +Y direction. Can only be selected in Free Mode. Can only be selected in a Wireframe Mode.

OnRotYMinus:
Return: None
Parameters: None
Calls functions:
OnDraw
rotate
wcs_to_scs
patch_list_setup

Creates Object: None
Description: Menu Message Handler to rotate the scene in the -Y direction. Can only be selected in Free Mode. Can only be selected in a Wireframe Mode.

OnRotZPlus:

Return: None
Parameters: None
Calls functions:
    OnDraw
    rotate
    wcs_to_scs
    patch_list_setup

Creates Object: None
Description: Menu Message Handler to rotate the scene in the +Z direction. Can only be selected in Free Mode. Can only be selected in a Wireframe Mode.

OnRotZMinus:
Return: None
Parameters: None
Calls functions:
    OnDraw
    rotate
    wcs_to_scs
    patch_list_setup

Creates Object: None
Description: Menu Message Handler to rotate the scene in the -Z direction. Can only be selected in Free Mode. Can only be selected in a Wireframe Mode.

OnThetaPlus:
Return: None
Parameters: None
Calls functions:
    OnDraw
    wcs_to_scs
    patch_list_setup

Creates Object: None
Description: Menu Message Handler to adjust the Theta viewing parameter in the (+) direction. Can only be selected in Free Mode. Can only be selected in a Wireframe Mode.

OnThetaMinus:
Return: None
Parameters: None
Calls functions:
    OnDraw
    wcs_to_scs
    patch_list_setup

Creates Object: None
Description: Menu Message Handler to adjust the Theta viewing parameter in the (-) direction. Can only be selected in Free Mode. Can only be selected in a Wireframe Mode.

OnPhiPlus:
Return: None
Parameters: None
Calls functions:
- OnDraw
- wcs_to_scs
- patch_list_setup

Creates Object: None
Description: Menu Message Handler to adjust the Phi viewing parameter in the (+) direction. Can only be selected in Free Mode. Can only be selected in a Wireframe Mode.

OnPhiMinus:
Return: None
Parameters: None
Calls functions:
- OnDraw
- wcs_to_scs
- patch_list_setup

Creates Object: None
Description: Menu Message Handler to adjust the Phi viewing parameter in the (-) direction. Can only be selected in Free Mode. Can only be selected in a Wireframe Mode.

OnRhoPlus:
Return: None
Parameters: None
Calls functions:
- OnDraw
- wcs_to_scs
- patch_list_setup

Creates Object: None
Description: Menu Message Handler to adjust the Rho viewing parameter in the (+) direction. Can only be selected in Free Mode. Can only be selected in a Wireframe Mode.

OnRhoMinus:
Return: None

Parameters: None
Calls functions:
    OnDraw
    wcs_to_scs
    patch_list_setup

Creates Object: None
Description: Menu Message Handler to adjust the Rho viewing parameter in the (-) direction. Can only be selected in Free Mode. Can only be selected in a Wireframe Mode.

OnDistancePlus:
Return: None
Parameters: None
Calls functions:
    OnDraw
    wcs_to_scs
    patch_list_setup

Creates Object: None
Description: Menu Message Handler to adjust the Distance viewing parameter in the (+) direction. Can only be selected in Free Mode. Can only be selected in a Wireframe Mode.

OnDistanceMinus:
Return: None
Parameters: None
Calls functions:
    OnDraw
    wcs_to_scs
    patch_list_setup

Creates Object: None
Description: Menu Message Handler to adjust the Distance viewing parameter in the (-) direction. Can only be selected in Free Mode. Can only be selected in a Wireframe Mode.

OnLButtonDown:
Return: None
Parameters: None used.
Calls functions:
    OnTransXPlus
    OnTransXMinus
    OnTransYPlus
    OnTransYMinus
    OnTransZPlus

OnTransZMinus
OnZoomPlus
OnZoomMinus
OnRotXPlus
OnRotXMinus
OnRotYPlus
OnRotYMinus
OnRotZPlus
OnRotZMinus
OnThetaPlus
OnThetaMinus
OnPhiPlus
OnPhiMinus
OnRhoPlus
OnRhoMinus
OnDistancePlus
OnDistanceMinus

Creates Object: None
Description: Menu Message Handler to adjust for left mouse button clicks. If a scene adjustment is active, left clicking anywhere within the window will continue to adjust the scene (i.e. if the user selects the Translate X+ menu item and then clicks in the window area, the scene will translate in the X+ direction again).

translate:
Return: None
Parameters:
In - int num_vertices: Number of vertices to be translated
In - float_coord_3d_type p_in[ ]: Listing of 3D input vertices
Out - float_coord_3d_type p_out[ ]: Listing of 3D output vertices
In - int tx: x-distance that vertices will be translated
In - int ty: y-distance that vertices will be translated
In - int tz: z-distance that vertices will be translated

Calls functions:
get_translate_matrix
get_transformed_vertices

Creates Object: None
Description: Local helper function to translate 3D vertices a specified amount.

scale:
Return: None
Parameters:
In - int num_vertices: Number of vertices to be scaled
In - float_coord_3d_type p_in[ ]: Listing of 3D input vertices to be scaled

Out - float_coord_3d_type p_out[ ]: Listing of 3D scaled output vertices
In - float sx: x-direction scale factor
In - float sy: y-direction scale factor
In - float sz: z-direction scale factor
In - float x_in: x-coordinate to scale about
In - float y_in: y-coordinate to scale about
In - float z_in: z-coordinate to scale about

Calls functions:
get_translate_matrix
get_composite_matrix
get_transformed_vertices

Creates Object: None
Description: Local helper function to scale 3D vertices a specified amount about a specified point.

rotate:
Return: None
Parameters:
In - int rot_axis: Which axis to rotate about (x, y, or z)
In - int num_vertices: Number of vertices to be rotated
In - float_coord_3d_type p_in[ ]: Listing of 3D input vertices to be rotated
Out - float_coord_3d_type p_out[ ]: Listing of 3D rotated output vertices
In - float theta_deg: Angle of rotation in Degrees
In - float x_in: x-coordinate to rotate about
In - float y_in: y-coordinate to rotate about
In - float z_in: z-coordinate to rotate about

Calls functions:
get_translate_matrix
get_composite_matrix
get_transformed_vertices

Creates Object: None
Description: Local helper function to rotate 3D vertices about a specified axis and point.

get_transformed_vertices:
Return: None
Parameters:
In - int num_vertices: Number of vertices to be computed
In - float_coord_3d_type p_in[ ]: Listing of 3D input vertices
Out - float_coord_3d_type p_out[ ]: Listing of 3D output vertices
In - float c[ ]: Composite matrix

Calls functions: None

Creates Object: None
Description: Local helper function for translating, rotating, and scaling vertices. Multiplies vertices by a composite matrix.

get_translate_matrix:
Return: None
Parameters:
- Out - float a_out[ ]: Translation matrix
- In - float dx: x-distance to be translated
- In - float dy: y-distance to be translated
- In - float dz: z-distance to be translated

Calls functions: None
Creates Object: None
Description: Local helper function to set up a translation matrix. Used by other functions when translating, scaling, or rotating vertices, or whenever vertices need to be translated.

get_composite_matrix:
Return: None
Parameters:
- Out - float c[ ]: Output 4x4 matrix
- In - float a[ ]: First 4x4 matrix to be multiplied
- In - float b[ ]: Second 4x4 matrix to be multiplied

Calls functions: None
Creates Object: None
Description: Local helper function to multiply two 4x4 matrices.

4.4.5. Draw Utilities

OnNoPatchesAllFaces:
Return: None
Parameters: None
Calls functions: OnDraw
Creates Object: None
Description: Menu message handler to draw the currently selected wireframe scene without patches to the screen. Scene A must first be initialized by selecting Start/Scene A.

OnPatchesAllFaces:
Return: None
Parameters: None
Calls functions: OnDraw
Creates Object: None
Description: Menu message handler to draw the currently selected wireframe scene

with patches to the screen. Scene A must first be initialized by selecting Start/Scene A.

OnRender:
Return: None
Parameters: None
Calls functions:
- get_hemicube
- wcs_to_scs
- object_patch_setup
- compute_patch_center
- dot_product_3d
- line_plane_intersection_test
- line_segment_size
- interpolation_utility
- OnDraw

Creates Object: None
Description: Menu message handler to draw the currently selected rendered scene to the screen. The rendered scene includes Radiosity intensities for patches with hidden surface removal applied. Scene A must first be initialized by selecting Start/Scene A.

OnShading:
Return: None
Parameters: None
Calls functions:
- interpolation_utility
- OnDraw

Creates Object: None
Description: Menu message handler to apply Gouraud shading and hidden surface removal to a rendered image. The frame buffer is updated with the new intensities in this function.

OnDraw:
Return: None
Parameters:
- In – CDC* pDC: Pointer to a device context

Calls functions: Calls no user created functions
Creates Object: None
Description: Handles all drawing operations to the screen except for AVI animations.

OnClearScreen:
Return: None
Parameters: None
Calls functions: None

Creates Object: None
Description: Menu message handler to clear the entire screen. In Free mode, the entire screen is cleared. In Animate mode, the entire screen is cleared except for the divider bars.

clear_text_area:
Return: None
Parameters: None
Calls functions: None
Creates Object: None
Description: Local helper function to clear the text area of the screen. Intended to be used for animation.

clear_right_area:
Return: None
Parameters: None
Calls functions: None
Creates Object: None
Description: Local helper function to clear the right area of the screen. Intended to be used for animation.

clear_left_area:
Return: None
Parameters: None
Calls functions: None
Creates Object: None
Description: Local helper function to clear the left area of the screen. Intended to be used for animation.

4.4.6. 3D Utilities

get_hemicube:
Return: None
Parameters:
- In - float_coord_3d_type patch_in[ ]: Vertices of patch in World Coordinates
- In - float_coord_3d_type hemicube_out[ ]: Hemicube vertices in World Coordinates
- In - int hemicube_faces_out[6][4]: y-offset to display bitmap in Screen Coordinates

Calls functions:
- compute_patch_center
- cross_product_3d

Creates Object: None
Description: Local helper function to compute the 3D vertices of a hemicube on a patch. The patch_in array must have the patch vertices defined from 0 to 3 in a clockwise direction when viewing the patch from the visible side. Otherwise the hemicube will be

built on the wrong side of the patch.

interpolation_utility:
Return: None
Parameters:
In - float_coord_3d_type wcs_in[ ]: Patch World Coordinates
In - int red: Red intensity if Gouraud shading not selected
In - int green: Green intensity if Gouraud shading not selected
In - int blue: Blue intensity if Gouraud shading not selected
In - BOOL shade: True selects Gouraud shading + hidden surface removal
False selects only hidden surface removal
In - int patch_number: Patch number in scene patch list

Calls functions:
wcs_to_vcs
wcs_to_scs

Creates Object: None
Description: Local helper function for hidden surface removal and Gouraud shading. Performs double interpolation on an input patch. Calculates Z Buffer values for a specific patch and can calculate Gouraud shading intensity values upon request. The Z Buffer must be reinitialized prior to calling this function for the first patch in a scene. The shading option cannot be selected unless the current scene mode is RENDERED. This function is tailored to the patch_list_type struct and will expect a 4-vertex patch input.

line_plane_intersection_test:
Return: BOOL intersect: True if line intersects plane, otherwise False.
Parameters:
In - float_coord_3d_type line_vertex1_in: Center of patch and center-bottom of hemicube. Line begins here (World Coordinates).
In - float_coord_3d_type line_vertex2_in: Center of cell on a hemicube face. provides a direction to project the ray (World Coordinates).
In - float_coord_3d_type plane_in[4]: Four vertices of patch to which the intersection test is being performed.
Out - float_coord_3d_type intersection_pt:If the line intersects the plane, the intersection point is returned, otherwise a code is returned.

Calls functions:
cross_product_3d
dot_product_3d

Creates Object: None
Description: Local helper function to determine whether a ray originating from a hemicube intersects a patch within the 3D scene. The vertices of the plane_in array must define the patch in either a clockwise or counter-clockwise manner. The

application uses a counter-clockwise manner. The code that is returned if the line does not intersect the plane indicates which test failed in this function:

-999999, -999999, -999999: The line is parallel to the plane.

-677777, -677777, -677777: The intersection point calculated does not lie within the bounded plane.

-888888, -888888, -888888: The line is finite in one direction. This code means that the calculated intersection point lies on the line's infinite path, but not on a defined part.

line_segment_size:

Return: float line_size

Parameters:

In - float_coord_3d_type line_vertex1_in: Start vertex in World Coordinates

In - float_coord_3d_type line_vertex2_in: End vertex in World Coordinates

Calls functions: None

Creates Object: None

Description: Local helper function that computes the length of a 3D line segment.

compute_patch_center:

Return: None

Parameters:

In - float_coord_3d_type wcs_in[ ]: Vertices of patch in World Coordinates

Out - float_coord_3d_type center: Center of patch in World Coordinates

Calls functions:

cross_product_3d

dot_product_3d

Creates Object: None

Description: Local helper function to computer the center of a 4-vertex polygon in 3D space. The vertices for the polygon must be defined in a circular manner (either clockwise or counter-clockwise).

cross_product_3d:

Return: None

Parameters:

In - float_coord_3d_type vector1_in: First input vector in World Coordinates

In - float_coord_3d_type vector2_in: Second input vector in World Coordinates

Out - float_coord_3d_type vector_out: Result of cross product in World Coordinates

Calls functions: None

Creates Object: None

Description: Local helper function to compute the cross product of two 3D vectors.

vector_out = vector1_in X vector2_in

dot_product_3d:
Return: None
Parameters:
In - float_coord_3d_type vector1_in: First input vector in World Coordinates
In - float_coord_3d_type vector2_in: Second input vector in World Coordinates
Out - float scalar_out: Result of dot product

Calls functions: None
Creates Object: None
Description: Local helper function to compute the dot product of two 3D vectors.
scalar_out = vector1_in . vector2_in

object_patch_setup:
Return: None
Parameters:
In - float_coord_3d_type wcs_vertices_no_patches: List of vertices without patches computed in World Coordinates
In – int poly_mesh_no_patches[ ][4]: Indices mapped to vertices for the 4-vertex polygon to divide up.
In – int_num_patches_type num_patches: Number of patches requested (i.e. 10 wide, 5 high).
Out – float_coord_3d_type wcs_vertices_patches[ ]: List of vertices with patches computed (World Coordinates).
Out – int poly_mesh_patches[ ][4]: Indices mapped to vertices for each patch in the input polygon.
In – int f: Index to the polygon number of a scene object.

Calls functions: None
Creates Object: None
Description: Local helper function to divide a 4-vertex 3D input polygon into a specified number of smaller 4-vertex polygons. This function is used to divide a scene up into patches and to divide a hemicube into cells.

4.4.7. Scene Options

OnScreenMode:
Return: None
Parameters: None
Calls functions:
OnDraw
OnInitA
OnClearScreen

Creates Object: None
Description: Toolbar message handler to toggle between Free and Animate modes.

OnSceneProperties:
Return: None
Parameters: None
Calls functions: updatePatches
Creates Object: CSceneAOptions aDlg
Description: Toolbar message handler for changing Scene A properties.

updatePatches:
Return: None
Parameters: None
Calls functions: object_patch_setup
Creates Object: None
Description: Local helper function to recalculate new number of patches. It is a stripped down version of the GetSceneA function. It only updates the patches data arrays (not the 'no patches' data arrays).

4.4.8. Animation

OnAnimateRestart:
Return: None
Parameters: None
Calls functions: OnClearScreen
Creates Object: None
Description: Toolbar message handler to reinitialize the animation. Can only be used in Animate mode.

OnAnimateStepFwd:
Return: None
Parameters: None
Calls functions:
- OnClearScreen
- clear_text_area
- clear_text_area
- clear_right_area
- clear_left_area
- OnDraw
- OnNoPatchesAllFaces
- OnPatchesAllFaces
- OnInitA
- OnRender
- interpolation_utility

Creates Object: None
Description: Toolbar message handler to step through the animation. Can only be used in Animate mode.

OnAnimateFwd:
Return: None
Parameters: None
Calls functions: OnAnimateStepFwd
Creates Object: None
Description: Toolbar message handler to play the animation all the way through. Can only be used in Animate mode.

OnPauseAnimation:
Return: None
Parameters: None
Calls functions: None
Creates Object: None
Description: Toolbar message handler to pause a running animation. Can only be used in Animate mode.

OnTimer:
Return: None
Parameters:
In - UINT nIDEvent: Event ID

Calls functions: OnAnimateStepFwd
Creates Object: None
Description: Windows timer used to time the pause between frames in the animation. The timer is used when the user plays the animation. It is not used for stepping through the animation.

OnAnimationControls:
Return: None
Parameters: None
Calls functions: OnAnimateRestart
Creates Object: CAnimationControls acDlg
Description: Toolbar message handler for setting animation controls. Can only be used in Animate mode.

OnAVIPlay:
Return: None
Parameters: None
Calls functions: None
Creates Object: None
Description: Toolbar Message Handler for playing an AVI within the Radiosity animation.

OnAVIStop:
Return: None
Parameters: None

Calls functions: None
Creates Object: None
Description: Toolbar Message Handler for stopping an AVI within the Radiosity animation.

4.4.9. Debug, Help & Cleanup

OnDatadump:
Return: None
Parameters: None
Calls functions: None
Creates Object: None
Description: Menu message handler to dump data to an output text file. The text file out.txt will be located within the workspace folder.

OnInstructions:
Return: None
Parameters: None
Calls functions: None
Creates Object: Pointer to CHelp object
Description: Message Handler for displaying application instructions.

OnDestroy:
Return: None
Parameters: None
Calls functions: None
Creates Object: None
Description: Windows message handler to clean up dynamically allocated space when the application window is closed.

4.5. Coding Challenges

Some of the coding challenges encountered during the development of the Radiosity part of the application motivated the design decisions for the animation part. These included:

- Organized storage of large amounts of data required for implementing Radiosity.
- Defining helper functions to reduce the amount of code and provide modularity to the application.
- The text that is drawn to the screen in the animation does not appear in a consistent manner across different operating systems. For example, in Windows 2000, six lines of text fit in the lower text area of the screen in Animate mode. In Windows XP, only five lines fit. The animation can be fully viewed on both systems.

5. Future Work

This section describes some of the known coding bugs that were discovered as well as improvements that could be made to the application.

5.1. Known Code Bugs

The following code bugs have been identified, but were not addressed:

- When in Free mode, if the scene is adjusted in certain ways and then rendered, there are unexpected artifacts (white spaces) in the scene. This is most likely due to a floating point problem.

- If a new scene were to be created, a hemicube that is too large may cause a problem in the line_plane_intersection_test function. If a patch elsewhere in the scene intersected the hemicube, it may not be detected as an intersection between the ray casted from the hemicube to that patch. Any intersection point found inside the hemicube is not considered to be an intersection and will not be used for a form factor calculation.

- Getting Started help menu does not go to the background if the application window is selected. This is inconvenient for users with lower screen resolution.

5.2. Application Improvements

The following improvements could be made in the future in order to enhance the application:

- The code could likely be updated to be more efficient. Depending on the specific PC the application is being run on, the scene options for number of patches and number of hemicube cells could possibly use up all system resources if set too high.

- Add the ability to change the color of individual room walls and box sides. This would allow more colors to be seen in one rendered image.

- Add an additional scene for the user to experiment with in Free mode.

- Implement a Step Back button for the animation so the user does not have to start at the beginning of the section in order to see a previous step.

- At this time, the Radiosity renderer only handles 4-vertex polygons. The code could be expanded to allow 3-vertex polygons. This will allow more flexibility in designing a new scene.

- There are other versions of Radiosity. This application could implement some of them and provide advanced tutorials for the user. Some of these might include

Progressive Radiosity, Shooting Radiosity, or having the hemicubes centered at random angles on the patch.

?? Setting a new number of patches in the scene through the Scene Options dialog box re-initializes the scene to its original location. After adjusting the scene to a new location, it would be useful to be able to set the number of patches without the scene moving.

?? Have a dialog box for the user to set specific adjustment parameters. For example, if the user want to rotate the scene about the x-axis in a positive direction, they must adjust the scene using the predefined increment instead of being able to set their own incremental value.

?? Implement an AVI Resume button. When the AVI Stop button is pressed, the only alternative is to replay the AVI from the beginning.

?? Prepare the application for use on the web. This would make the project more accessible to others.

6. References

[1] Anton, Howard. Elementary Linear Algebra. Ed. 7. New York: John Wiley & Sons Inc, 1994.

[2] Ashdown, Ian. Radiosity, A Programmer's Perspective. New York: John Wiley & Sons, Inc, 1994.

[3] Chen, Shenchang Eric. VI.I Implementing Progressive Radiosity With Use Provided Polygon Display Routines. Academic Press Inc, 1991.

[4] Eckert, Richard R. CS-460/560 Notes. 2004. 9 April 2005 <http:// www.cs.binghamton.edu/~reckert/460/460notes.htm>.

[5] Goldman, Ron. Radiosity. 2004. 9 April 2005 <http://www.owlnet.rice.edu/ ~comp360/lectures/Radiosity.pdf>.

[6] Greenberg, Donald P., et al. Radiosity. SIGGRAPH Course Notes 21, 1990.

[7] Hanly, Jeri R., Elliot B. Koffman, and Joan C. Horvath. Program Design For Engineers. Massachusetts: Addison-Wesley Publishing Company Inc, 1995.

[8] Hearn, Donald, and M. Pauline Baker. Computer Graphics, C Version. Ed. 2. New Jersey: Prentice Hall Inc, 1997.

[9] Horton, Ivor. Ivor Horton's Beginning Visual C++ 6. Birmingham, U.K.: Wrox Press Ltd, 1998.

[10] Main, Michael, and Walter Savitch. Data Structures And Other Objects Using C++. Massachusetts: Addison Wesley Longman, 1997.

[11] Marchener, Steve. CS465 Notes, Simple Ray-Triangle Intersection. 2003. Cornell University. 29 April 2005 < www.cs.cornell.edu/Courses/ cs465/2003fa/homeworks/raytri.pdf>.

[12] Mathews, John H. Module For Jacobi And Gauss-Seidel Iteration. 2003. 11 April 2005 <http://math.fullerton.edu/mathews/n2003/GaussSeidelMod.html>

[13] Microsoft Visual C++. 2005. Function X Inc. 9 April 2005 <http://functionx.com/ visualc/>.

[14] Owen, Scott. Overview Of Radiosity, SIGGRAPH 1993 Education Slide Set. 1998. 9 April 2005 <http://www.siggraph.org/education/materials/HyperGraph/ Radiosity/Radiosity.htm>.

[15] Schafer, Stephan. Efficient Object-Based Hierarchical Radiosity Methods. 2000. 6 June 2005 <www.eg.org/EG/DL/dissonline/doc/schaefer1999.pdf>

[16] Shklyar, Dmitry. 3D Rendering History, Part 2, To Photorealism And Beyond. 2003. CGNetworks. 10 April 2005 <http://www.cgnetworks.com/ story_custom.php?story_id=1724&page=1>.

[17] Stewart, James. Calculus. Ed. 3. California: Brooks/Cole Publishing Company, 1995.

[18] Weisstein, Eric, et al. Mathworld. 2005. Wolfram Research. 9 April 2005 <http://mathworld.wolfram.com/>.