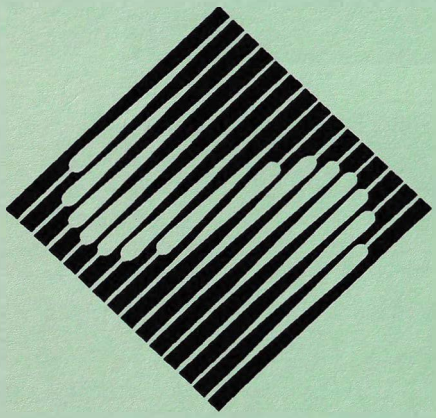
1993 SIGGRAPH

Educators' Slide Set



Rosalee Nerheim-Wolfe, editor

DePaul University

Drawing Lines and Circles

2: **Title slide.**
This section illustrates the basic principles of scan-converting lines and circles for raster displays. Scan-conversion is the process of determining which pixels should be illuminated in order to display a representation of a geometrical object which is as faithful as possible to the exact continuous geometry of the object.

3: **Lines on a display screen.**
This is a photograph of an actual raster display screen, which shows a teapot drawn as a number of lines. The photograph is taken a few inches away from the screen, and at this normal viewing distance, the lines appear to be nearly continuous.

4: **Lines as pixels.**
Now, we have zoomed in much closer towards the screen, and we can see from this photograph that the lines are not continuous at all! In fact, they are all made of discrete pixels. At this resolution, the lines look quite fragmented. (Note that no anti-aliasing techniques are being used in this example. If anti-aliasing were used, we would still see a line as a collection of pixels, but the intensities of the pixels would not be the same. This helps to 'smudge' out the jaggedness of the line.)

5: **Approximating a line with pixels.**
This slide shows the principle of representing a line using pixels. In this and subsequent slides we represent the array of pixels on the surface of a display as a grid of square pixels. (Almost all modern raster displays have square pixels). If a pixel is lit up, we draw it in yellow. The open grid squares are pixels in the background color of the display.

Note that we always refer to a pixel by the coordinates of its center: by 'the pixel at (x,y)' we mean 'the pixel whose center lies on (x,y)'.

On a raster display, every part of the image must be made of pixels. There is nothing else! Because the pixels are laid out in a fixed grid pattern, we can only approximate lines and curves on a raster display. The purpose of this set of slides is to describe an efficient method for choosing the best approximations. By 'best' we mean a choice of pixels which most closely matches the exact geometry we are tying to draw.

6: **The equation of a line.**
First, we forget about pixels, and review basic Cartesian geometry. The equation of a line may be written: $y = mx + b$. The slide shows that the slope m of the line is given by dy / dx, where dy is the "rise" in y and dx is the "run" in x for a particular segment of the line.

When drawing lines on a display, we're working in device coordinates, which means that the endpoints of a line segment are specified by integers. The quantities dx and dy are also integers.

7: **Brute force scan conversion.**
We can use the equation of the line directly to perform a scan-conversion. Starting at the pixel for the left-hand endpoint of the line, xl, we step along the pixels horizontally, until we reach the right-hand end of the line, xr. For each pixel we use the equation 'y = mx + b' to compute the corresponding y value. We then round this value to the nearest integer to select the nearest pixel. We might write this in C as follows:

```
x = xl;
while (x <= xr){
     ytrue = m*x + b;
     y = Round (ytrue);
     PlotPixel (x, y);              /* Set the pixel at (x,y) on */
     x = x + 1;
}
```

The slide shows that because the line equation is evaluated for every step in x, we need to perform a floating-point multiplication each time. This method therefore requires an enormous number of floating-point multiplications, and is therefore expensive.

8: **The DDA algorithm.**
We can improve on the brute force method by noting that if $dx = 1$, then the change in y will be m, and there is no need to evaluate 'y = mx + b' for each pixel. To get the next value of y, we add m to the last value of y. We need only compute m once, as the start of the scan-conversion. This method is called a Digital Differential Analyser, or DDA.

9: **The DDA algorithm for lines with -1 < m < 1.**
This slide shows the modified form of the algorithm. We start at (xl,yl), and at each step add 1 to x, and m to y. As before, we round the true y-value to determine the nearest pixel:

```
x = xl;
ytrue = yl;
while (x <= xr){
     ytrue = ytrue + m;
     y = Round (ytrue);
     PlotPixel (x, y);
     x = x + 1;
}
```

10: **Gaps occur when m > 1.**
The figure on the left of this slide shows the results of the DDA method when the slope of the line is greater than 1. Notice the gaps in the line, which arise because a step of 1 in x corresponds to a step in y of more than one pixel. The solution in this case, shown in the figure on the right, is to reverse the roles of x and y, using a unit step in y, with a corresponding step of 1/m for x. In this slide, the slope of the line is 2. To avoid gaps, we determine each pixel location by adding 1 to y and 1/2 to x.

11: **Bresenham's algorithm.**
The DDA algorithm runs rather slowly because it requires real arithmetic (floating-point operations). We now describe an improvement on this algorithm called Bresenham's algorithm, which uses only integer arithmetic, and runs significantly faster.

(Historical note: this algorithm was first formulated in 1965, for the control of a pen-plotter. Ref: Bresenham, J.E. 'Algorithm for computer control of a digital plotter', IBM Systems Journal, January 1965, pp. 25-30.)

12: **Choosing between two pixels.**
This slide shows the fundamental principle of scan-conversion. We have a line with a slope between 0 and 1, and the pixel most recently chosen is filled in yellow. There are two choices for the next pixel, each marked "?". For the most accurate representation of the line, we must choose the pixel which is closest to the true path of the line.

13: **Finding the closer pixel.**
To determine which pixel is closest to the line, we need to measure the distance of its center from the path of the line. For this example, we see that the lower pixel (shown as white in the slide) is the closer pixel. Because we are using an integer to approximate the actual y-coordinate of the line, we've created an error. The line's y-value lies above the integer approximation, so the error is a positive quantity.

14: **Another example.**
This shows the scan conversion of a different line. Here, we choose the top pixel (shown in white). Again, in using an integer value to approximate the line's y-coordinate, we have created an error. This time we "overshot" the true y-coordinate, so the error is a negative quantity.

15: **Introducing an error term.**
We have seen that using an integer to approximate a real (floating-point) quantity causes an error. If we record the error as it accumulates, we can use this information to choose the pixels which best approximate the line.

16: **Using the error term.**
Here we see how the cumulative error influences pixel choice. The figure depicts a line, and the pixels that best approximate that line. In the bottom figure is a graph of the cumulative error.

In this example, the line's slope is 1/3. When we move one pixel to the right, the error increases by 1/3 (or the amount of the slope.) As long as the error is less than 1/2, the lower pixel is closer to the line. When the error is greater than 1/2, we choose the upper pixel. Notice that when the pixel's y-coordinate increases by one, the pixel jumps to a position above the true line, and the error decreases by one.

If we use this approach, we can use integers to store the x and y coordinates of the pixel location, and we ask the question

is the error > 1/2 ?

to decide whether to adjust the y-coordinate of the next pixel.

17: **The error term is fractional.**
Unfortunately, the error term is a fractional quantity. We are striving for efficiency and wish to eliminate floating-point operations. To do this, we must reformulate the error term to convert it to an integer quantity.

18: **Rewriting the error term.**
This slide shows the original error, and a scaled version of it, which is an integer. We need to multiply the original error by some quantity to create an integer. To find that quantity, we analyze the operations performed on the error:

1) add m to the error
2) compare the error to 1/2
3) subtract 1 from the error.

The first two of these operations involve fractions. If we multiply the error by $2dx$, to give a new 'scaled error' term, then the operations become:

1) add $2dy$ to the scaled error
2) compare the scaled error to dx
3) subtract $2dx$ from the scaled error.

19: **Using the integer scaled error term.**
Here we see that the integer scaled error term correctly chooses the correct pixels. Here we are comparing the integer scaled error term to dx to determine pixel position. Is there a yet faster comparison that we could use? What comparison is faster than comparing an integer to some arbitrary value? Could we use that comparison here?

We now have the basis for a line scan-conversion algorithm that uses only integer arithmetic. The following is C code that implements Bresenham's algorithm for lines whose slopes are between 0 and 1. All of the variables are integers. Notice that all of quantities that are added to or subtracted from 'ie' (the integer scaled error term) have been scaled by $2 * dx$. The code may easily be modified to work for all slopes.

The code, as written, is still not quite as fast as it could be. What modifications would make the algorithm even faster?

```
void Bresenham (int xl, int yl, int xr, int yr)
                /* xl, yl:    coordinates of the left endpoint  */
                /* xr, yr:    coordinates of the right endpoint */
      {
          int x,y;          /* coordinates of pixel being drawn  */
          int dy, dx;       /* rise, run                         */
          int ie;           /* integer scaled error term         */

          x = xl;           /* start at left endpoint            */
          y = yl;
          ie = 2 * dy - dx; /* initialize the error term         */

          /* pixel-drawing loop */
          while (x <= xr){
             PlotPixel (x,y);              /* draw the pixel    */
             if (ie > 0) {
                y = y + 1;
                ie = ie - 2 * dx;       /* replaces e = e - 1  */
             }
             x = x + 1;
             ie = ie + 2 * dy;          /* replaces e = e + m  */
          }
      }
```

20: **Scan-converting circles.**
We now turn to an extension of this method for drawing circles. In terms of processing, circles are more expensive than lines to scan-convert. However, a circle is a symmetrical shape, and we can use this property to speed up the scan-conversion.

21: **The eight-fold symmetry of the circle.**
This slide illustrates the eight-fold symmetry of a circle. To draw a complete circle we only need to calculate one eighth of the circle (known as an octant), and then reflect these points several times to obtain the whole circle.

We assume the center of the circle to be at the origin of the coordinate system. The initial octant is shown in yellow, on the top right of the circle. Now, we can reflect these points about the line $x = y$, to get the next octant clockwise, drawn in green. We now have a quadrant of the circle, and we can reflect all these points about the x-axis (the line y=0), to give the next quadrant (drawn cyan). Now we have a half circle, and if we reflect this about the y-axis (the line $x = 0$), we obtain the whole circle.

22: **Computing the initial octant.**
To compute the initial octant we use a scan-conversion procedure derived from line scan-conversion. The slide shows the grid of pixels, together with the true path of the circle (green).

23: **Choosing the next pixel.**
When drawing the initial octant, we begin by choosing the pixel at the apex of the circle. To find the next pixel, we consider the pixels to the right and diagonally down from the current pixel. The slide depicts the last pixel chosen by the algorithm, and the two candidates for the next pixel. The algorithm will choose the pixel which is closer to the true path of the circle.

If the pixel just chosen has its center at (x,y), what are the coordinates of the pixel to its right? What are the coordinates of the pixel located diagonally down?

24: **Using a reference point P.**
Instead of using a brute-force calculation to determine which candidate pixel is closer to the circle, we make use of a reference point P. If P is inside the circle, then the candidate to the right of the last pixel is closer to the true circle.

Horizontally, P is halfway between the last pixel, and the candidate to the right. Vertically, P is halfway between the two candidate pixels. What are the coordinates of P?

25: **P is outside the circle.**
When the reference point P lies outside the circle, the lower of the two candidate pixels is the one closer to the circle.

26: **Determining if a point lies inside a circle.**
How do we actually determine whether a point lies inside or outside a circle? This slide shows a method using the equation of the circle. Again, we assume the circle is centered on the origin, and write its equation as:

$$x^2 + y^2 = r^2$$

Consider the function:

$$F(x,y) = x^2 + y^2 - r^2$$

For all points (x,y) which lie on the circle, $F(x,y) = 0$.

The diagram on the left side of the slide illustrates a point lying inside the circle. In this case, $F(x,y) < 0$. On the right, the point lies outside the circle, and $F(x,y) > 0$.

The value of this function at the reference point P serves the same purpose as the value of the error term in the line-drawing algorithm. It influences the choice of the next pixel. For efficiency reasons, we do not want to compute this function very often. We can compute the explicit form of this function once, when we draw the first pixel, and then use addition to update its value for subsequent pixels.

27:**Credits.**
Toby Howard created the storyboard and developed the supporting text for this section of the slide set. Jenny Morlan, Assistant Professor of Art at DePaul University, generously gave of her time and talents in the development of this section of the slide set.

The figure drawn in slide 3 uses the data points of Newell's teapot.

Many thanks to Steve Cunningham and the Winter '92 "Environmental Graphics" class at DePaul University for viewing these slides and making many helpful suggestions.

Aliasing and Antialiasing

28: **Title Slide.**
This section of the slide set will demonstrate how aliasing affects the rendering of images, and how antialiasing methods can soften or reduce the effects of aliasing.

29: **Aliasing.**
When a program renders a picture on a display device, it must compute a color for each pixel on the display. Computing the color for a pixel involves taking samples from the model or scene being drawn. A display screen has a finite number of regularly-spaced pixels, which means that a rendering program will take a finite number of samples from the model. Aliasing occurs when the finite number of samples do not contain enough information to draw an accurate picture of the model.

The next three slides demonstrate sampling.

30: **Original scene.**
The bowl of fruit was modeled using constructive solid geometry (CSG). One scanline in the fruitbowl is highlighted. The graph shows the luminosity (brightness) function of the highlighted scan line.

31: **Sampling the scene.**
The rectangular grid superimposed over the fruit bowl shows the size of the pixels. The dot in the middle of each square shows the position of a sample. The color at the sample point will be the color of the pixel in the rendered image.

The graph shows the corresponding sampled luminosity function of the highlighted scanline. Notice that a lot of information has been lost.

32: **Rendered Image.**
The rendered image differs greatly from the original scene, as does the luminosity signal. Notice that the green leaf has moved to the right in the rendered image.

33: **Effects caused by aliasing.**
The errors caused by aliasing are called artefacts. Common aliasing artefacts include jagged profiles, disappearing or improperly rendered fine detail, and disintegrating textures.

34: **Jagged profiles.**
The picture on the left shows the sampling grid superimposed on the original scene. The picture on the right is the rendered image. A jagged profile is quite evident in the rendered image. Also known as "jaggies", jagged silhouettes are probably the most familiar effect caused by aliasing. Jaggies are especially noticeable where there is a high contrast between the interior and the exterior of the silhouette.

35: **Improperly rendered detail.**
The original scene on the left shows a group of small polygons. In the rendered scene, one of the two red rectangles disappears entirely, and the other doubles in width. Two of the orange triangles disappear. Although the two yellow triangles are identical in size, one is larger than the other in the rendered image.

36: **Disintegrating textures.**
This is a checkered texture on a plane. The checkers should become smaller as the distance from the viewer increases. However, the checkers become larger or irregularly shaped when their distance from the viewer becomes too great. Simply increasing the resolution will not remove this artefact. Increasing the resolution will only move the artefact closer to the horizon.

37: **Antialiasing.**
Antialiasing methods were developed to combat the effects of aliasing. The two major categories of antialiasing techniques are prefiltering and postfiltering.

38: **Prefiltering.**
Prefiltering methods treat a pixel as an area, and compute pixel color based on the overlap of the scene's objects with a pixel's area.

39: **Basis for Prefiltering Algorithms.**
The original scene is a filled orange circle on a red background. All of the pixels inside the circle are 100 percent orange. All the pixels on the boundary of the circle have some area that is red and some area that is orange.

Forty percent of the highlighted pixel is orange and 60 percent of its area is red. The computed color for the highlighted pixel is 40 percent orange and 60 percent red.

40: **Prefiltering Demonstration.**
Both phrases were rendered at a resolution of 512x512. Even at this resolution, the jaggies are apparent in the phrase that hasn't been antialiased.

41: **Closeup.**
Without antialiasing, the jaggies are harshly evident.

42: **Closeup of Prefiltered Image.**
Along the character's border, the colors are a mixture of the foreground and background colors.

43: **Postfiltering.**
Postfiltering, also known as supersampling, is the more popular approach to antialiasing. For each displayed pixel, a postfiltering method takes several samples from the scene and computes an average of the samples to determine the pixel's color.

The two steps in the postfiltering process are:

1. Sample the scene at *n* times the display resolution. For example, suppose the display resolution is 512x512. Sampling at three times the width and three times the height of the display resolution would yield 1536x1536 samples.

2. The color of each pixel in the rendered image will be an average of several samples. For example, if sampling were performed at three times the width and three times the height of the display resolution, then a pixel's color would be an average of nine samples. A filter provides the weights used to compute the average.

44: **Sampling in the Postfiltering method.**
In both figures, the display resolution is four pixels wide by three pixels high. The superimposed grid depicts the size of a pixel. Both figures show supersampling at three times the height and three times the width of the display resolution.

In the right figure, the samples are regularly spaced. In the left figure, the positions of samples are displaced by a random amount. The random amount is small relative to the size of the pixel. This method of perturbing the sample positions is known as "jittering." Jittering adds noise to the rendered image. The advantage of jittering is that the human eye tolerates noise more easily than it tolerates aliasing artefacts, and as a result, humans perceive a higher quality in the rendered image.

45: **Filters.**
Filters combine samples to compute a pixel's color. The weighted filter shown on the slide combines nine samples taken from inside a pixel's boundary. Each sample is multiplied by its corresponding weight and the products are summed to produce a weighted average, which is used as the pixel color. In this filter, the center sample has the most influence.

The other type of filter is an unweighted filter. In an unweighted filter, each sample has equal influence in determining the pixel's color. In other words, an unweighted filter computes an unweighted average.

46: **Using a filter to compute a pixel's color.**
On the left is a group of samples that will be combined to produce pixel colors. Some of the samples are orange, and some are red. The superimposed numbers are the weights from the filter shown in the last slide. The highlighted group of samples on the left are combined using the filter to produce the color of the highlighted pixel on the right.

47: **Student work.**
Students incorporated several postfiltering techniques in a raytracer. All of the following raytraced images were rendered at a resolution of 512x342.

48: **No antialiasing.**

49: **3x3 supersampling, 3x3 unweighted filter.**
The students combined 1536x1026 regularly-spaced samples using an unweighted filter. Each of the nine samples taken for a pixel had equal influence on the pixel's computed color.

50: **3x3 supersampling, 5x5 weighted filter.**
The samples are regularly spaced. Here's the weighted filter used in this rendering:

1/81	2/81	3/81	2/81	1/81
2/81	4/81	6/81	4/81	2/81
3/81	6/81	9/81	6/81	3/81
2/81	4/81	6/81	4/81	2/81
1/81	2/81	3/81	2/81	1/81

This filter allows samples taken from neighboring pixels to influence the color of the current pixel. The result is softer profiles and edges.

51: **3x3 supersampling, jittered samples, 3x3 weighted filter.**
Each sample is perturbed by a random amount in x and y. The random amount is constrained to be less than one sixth of the pixel's width.

52: **Credits:**
Tony Loza, Cynthia Gryniewicz and David Abramoske wrote the raytracing package that created slides 48-51. The raytracer is written in 'C' and runs on several Unix platforms. David incorporated the postfiltering methods, and Cynthia created the model.

Special thanks to Jenny Morlan for her invaluable assistance in designing the slides.

Thanks also to Ed Allemand, Steve Cunningham, Henry Harr, Steve Jost, Warren Krueger, Glenn Lancaster, Charlotte Williams and the 1992 Winter Quarter "Environmental Graphics" class for viewing the slides and making many helpful suggestions.

Examining Radiosity

53: **Title Slide.**
This section describes an approach to generating computer graphics based on the concept of energy transfer between surfaces. This approach is commonly known as 'radiosity'. We first describe the basic algorithm, and then cover extensions to it.

For this method of image generation, we make some basic assumptions. We treat the scene being rendered as a closed environment containing a number of surfaces. A surface may be a source of illumination (a light), or an object which reflects light. To create an image of the scene we consider the exchange of light energy between all the objects in the closed environment.

54: **Direct and Indirect Light.**
Every surface in an environment is illuminated by a combination of 'direct light' and 'reflected light'.

The direct light is light energy which comes directly from a light source or light sources, attenuated only by some interposed media, such as smoke, fog or dust. The reflected light is light energy which, after being emitted from a light source, is reflected from one or more surfaces in the environment, and may then arrive at other surfaces.

When light energy is reflected from a surface it is attenuated by the 'reflectivity' of the surface, as some of the light energy may be absorbed by the surface, and some may pass through the surface. The reflectivity of a surface is often defined as its color.

55: **Examples of Rendering Methods.**
Three images are displayed on this slide, illustrating several facets of computer image generation.

The image in the upper right corner was rendered with a scanline rendering algorithm, where the ambient component of light is approximated with a constant value. This results in similar shading for all parts of the image, even in areas where shadows or less illumination would be expected. No shadows are calculated.

The image in the middle and the image in the lower left corner were both rendered with a ray tracing global illumination algorithm. The image in the middle exhibits some of the characteristics of a typical ray tracing algorithm: mirror-like reflections and no ambient light component. Notice the hard-edged shadows cast by the light source. The image in the lower left corner retains the mirror-like reflection typical of a ray tracing algorithm, and adds an accurate ambient component of light, by allowing many reflections of light energy through the environment, as well as soft shadows.

56: **Diffuse Interreflection.**
If a surface is defined to be a 'diffuse reflector' of light energy, any light energy which strikes the surface will be reflected from it in all directions. The amount of light reflected depends on how reflective the surface is, and on the angle between the surface normal and the direction of the incoming light. This relationship is expressed as Lambert's law.

Light which is reflected from a surface is attenuated by the reflectivity of the surface, which is closely associated with the 'color' of the surface. The reflected light energy often is colored, to some small extent, by the color of the surface from which it was reflected.

This reflection of light energy in an environment produces a phenomenon known as 'color bleeding', where a brightly colored surface's color will 'bleed' onto adjacent surfaces. The image in this slide illustrates this phenomenon, as both the red and blue walls 'bleed' their color onto the white walls, ceiling and floor.

57: **Introduction to Radiosity.**
The 'radiosity' method of computer image generation has its basis in the field of thermal heat transfer. Heat transfer theory describes radiation as the transfer of energy from a surface when that surface has been thermally excited. This encompasses both surfaces which are basic emitters of energy, as with light sources, and surfaces which receive energy from other surfaces and thus have energy to transfer.

This 'thermal radiation' theory can be used to describe the transfer of many kinds of energy between surfaces, including light energy.

As in thermal heat transfer, the basic radiosity method for computer image generation assumes that surfaces are diffuse emitters and reflectors of energy, emitting and reflecting energy uniformly over their entire area. It also assumes that an equilibrium solution can be reached, such that all of the energy in an environment is accounted for, through absorption and reflection.

The basic radiosity method is viewpoint independent. The distribution of energy in the scene is the same regardless of the viewpoint of the image.

58: **The Radiosity Equation.**
The 'radiosity equation' describes the amount of energy which can be emitted from a surface, as the sum of the energy inherent in the surface (a light source, for example) and the energy which strikes the surface, being emitted from some other surface. The energy which leaves a surface (surface j) and strikes another surface (surface i) is attenuated by two factors:

1. the 'form factor' between surfaces i and j, which accounts for the physical relationship between the two surfaces,

2. the reflectivity of surface i, which will absorb a certain percentage of light energy which strikes the surface.

59: **The Form Factor.**
The 'form factor' describes the fraction of energy which leaves one surface and arrives at a second surface. It takes into account the distance between the surfaces, computed as the distance between the centers of the surfaces, and their orientation in space relative to each other, computed as the angle between each surface's normal vector and a vector drawn from the center of one surface to the center of the other surface. A form factor is a dimensionless quantity.

The form factor, as initially shown, describes the form factor between two differential areas. This is a point-to-point form factor. To use this form factor with surfaces which have a positive area, the equation must be integrated over one or both surface areas. However, the form factor between a point on one surface and another surface with positive area can be used if the assumption is made that the single point is representative of all of the points on the surface.

60: **The Nusselt Analog.**
Differentiation of the basic form factor equation is difficult even for simple surfaces. Nusselt developed a geometric analog which allows the simple and accurate calculation of the form factor between a surface and a point on a second surface.

The 'Nusselt analog' involves placing a hemispherical projection body, with unit radius, at a point on a surface. The second surface is spherically projected onto the projection body, then cylindrically projected onto the base of the hemisphere. The form factor is, then, the area projected on the base of the hemisphere divided by the area of the base of the hemisphere.

61: **The Hemlcube.**
The 'hemicube' form factor calculation method involves placing the center of a cube at a point on a surface, and using the upper half of the cube (the 'hemicube' which is visible above the surface) as a projection body as defined by the 'Nusselt analog'.

Each face of the hemicube is subdivided into a set of small, usually square ('discrete') areas, each of which has a pre-computed form factor value. When a surface is projected onto the hemicube, the sum of the form factor values of the discrete areas of the hemicube faces which are covered by the projection of the surface is the form factor between the point on the first surface (about which the cube is placed) and the second surface (the one which was projected).

The speed and accuracy of this method of form factor calculation can be affected by changing the size and number of discrete areas on the faces of the hemicube.

62: **The Hemlcube In Action.**
This illustration demonstrates the calculation of form factors between a particular surface on the wall of a room and several surfaces of objects in the room.

A standard radiosity image generation algorithm will compute the form factors from a point on a surface to all other surfaces, by projecting all other surfaces onto the hemicube and storing, at each discrete area, the identifying index of the surface that is closest to the point. When all surfaces have been projected onto the hemicube, the discrete areas contain the indices of the surfaces which are visible to the point. From there the form factors between the point and the surfaces are calculated.

For greater accuracy, a large surface is broken into a set of small surfaces before any form factor calculation is performed.

63: **The 'Full Matrix' Radiosity Algorithm.**
Two classes of radiosity algorithms have been developed which will calculate the energy equilibrium solution in an environment.

The 'full matrix' radiosity solution calculates the form factors between each pair of surfaces in the environment, then forms a series of simultaneous linear equations, as shown in the upper figure. This matrix equation is solved for the 'B' values, which can be used as the final intensity (or color) value of each surface.

This method produces a complete solution, at the substantial cost of calculating form factors between each pair of surfaces and determining the solution of the matrix equation. Each of these steps can be quite expensive if the number of surfaces is large. Complex environments have upwards of ten thousand surfaces, and environments with one million surfaces are not uncommon. This leads to substantial costs in computation time and storage.

64: **The 'Progressive' Radiosity Algorithm.**
The 'progressive' radiosity solution is an incremental method, yielding intermediate results at much lower computation and storage costs. Each iteration of the algorithm requires the calculation of form factors between a point on a single surface and all other surfaces, rather than all n^2 form factors (where n is the number of surfaces in the environment). After the form factor calculation, radiosity values for the surfaces of the environment are updated.

This method will eventually produce the same complete solution as the 'full matrix' method, though, unlike the 'full matrix' method,it will also produce intermediate results, each more accurate than the last. It can be halted when the desired approximation is reached. It also exacts no large (n^2) storage cost.

65: **Progressive Radiosity Examples.**
This slide illustrates the iterative nature of the progressive method. The composite image shows that as the number of iterations increase, the accuracy of the intensity solution (and, hence, the resulting image) also increases. Of particular interest is the contribution of the color of the walls of the room to the overall color of the room in the rightmost section

of the composite image. This image was created by calculating three separate images, each halted at a different stage of rendering, and composited together afterwards.

66: **Progressive Radiosity Variants.**
Several variations on the basic progressive radiosity algorithm have been developed, in an effort to find the optimal method for producing the most pleasing results with minimum cost. Each variant calculates the form factors from a point on one surface to all other surfaces.

The 'gathering' variant collects light energy from all other surfaces in the environment, attenuated by the calculated form factors, and updates the 'base' surface. The 'base' surface is arbitrarily chosen. As in the 'gathering' variant, the 'base' surface is arbitrary.

The 'shooting' variant distributes light energy from the 'base' surface to all other surfaces in the environment, attenuated by the calculated form factors.

The "shooting and sorting" variant first calculates the surface with the greatest amount of unshot light energy, then uses this surface as the 'base' surface in the "shooting" variant.

In addition to these, an initial 'ambient' term can be approximated for the environment and adjusted at each iteration, and is gradually replaced by the true ambient contribution to the rendered image. The 'shooting and sorting' method is the most desirable, as it finds the surface with the greatest potential contribution to the intensity solution and updates all other surfaces in the environment with its energy.

67: **Comparison of Progressive Variants.**
This slide shows four variations on the basic progressive radiosity algorithm, each halted after one hundred iterations. The upper left image is the 'gathering' variant, the upper right image the 'shooting' variant, the lower left image the 'shooting and sorting' variant, and the lower right image is the 'shooting and sorting and ambient' progressive radiosity variant.

The 'shooting' variants show their superiority over the 'gathering' variant here, as more of the scene is illuminated earlier. The slide demonstrates that the 'shooting and sorting' variant concentrates on those surfaces that contribute most to the overall solution.

68: **The Two-Pass Radiosity Solution.**
Several important variations on the basic diffuse radiosity solution have been developed. The first is designed to relax the restriction on the diffuse-only nature of the basic radiosity solution, by breaking the intensity solution into two steps. First, a pass with a traditional radiosity algorithm calculates the diffuse intensity of the surfaces, followed by a pass with a ray tracing algorithm, which collects the diffuse intensity information from the surfaces and adds to it specular information. This second pass is viewpoint-dependent: specular highlights on a surface are dependent on the location of both the light and the viewer relative to the surface.

69: **Participating Media.**
Another variation on the basic diffuse radiosity solution adds the contribution of light passing through a participating medium, such as smoke, fog, or water vapor in the air. In this algorithm, light energy is sent through a three-dimensional volume representing a participating medium, which both attenuates the light energy and, potentially, adds to the intensity solution through illumination of the participating medium.

70: **Advantages and Disadvantages.**
The largest single advantage of the radiosity method for computer image generation is the highly realistic quality of the resulting images. No other method accurately calculates the diffuse interreflection of light energy in an environment. Soft shadows and color bleeding are natural by-products of this method, just as hard shadows and mirror-like reflections are natural by-products of a typical ray-tracing algorithm.

In addition to being visually pleasing, the method can be quite accurate in its treatment of energy transport between surfaces. The viewpoint independence of the basic radiosity algorithm provides the opportunity for interactive 'walkthroughs' of environments, as one intensity solution for an environment will serve as the base for any particular view of the environment.

The costs associated with the radiosity method are substantial. The 'full matrix' radiosity method requires a large amount of storage and long computation times for form factor calculation and matrix solution. The 'progressive' method must also calculate a large number of form factors, many more than once.

Accuracy in the resulting intensity solution requires preprocessing the environment, subdividing large surfaces into a set of smaller surfaces, and more surfaces means more storage and computation.

71: **State of the Art and Future Work.**
More recently, several new algorithms have been developed which help to alleviate the restrictions of the basic radiosity solution. Ray tracing algorithms can be modified to handle the intricacies of accurate light transport between surfaces without explicit form factor calculation.

'Intelligent' pre-processing of environments can subdivide the surfaces of an environment based on the geometry of the environment and on the probable location of light-shadow boundaries, creating an optimal subdivision. As noted previously, the inherent diffuse nature of the basic radiosity algorithm has been relaxed with the development of multiple pass algorithms which incorporate both the diffuse and specular components of light.

Current research efforts include more accurate modeling of the characteristics of lights and surfaces, through BRDFs (bidirectional reflectance distribution functions), concentration on minimizing the cost of form factor calculation, and increasing the accuracy of form factor calculation.

72: **Consolation Room Image.**
This image suggests one treatment of a consolation room in a hospital or physician's office. It is part of a research experiment comparing the effect of different lighting on the overall appearance and perception of an environment.

73: **Conference Room Image.**
This image shows a typical conference room.

74: **Conference Room Photograph.**
For comparison, this image is an actual photograph of the conference room modelled and rendered in the previous slide.

75: **Theatre.**
This image shows a model of a proposed theater near Candlestick Park in San Francisco, and contains 1,061,543 surface elements.

76: **Theatre With Polygonal Mesh.**
This image is the same model of the proposed theater used in the previous slide, but with the surface element mesh visible.

77: **Steel Mill.**
This image of a steel rolling mill was created using progressive radiosity. The original model contains about 30,000 polygons, which were subdivided into about 55,000 elements during the solution. It was computed on a DEC VAX 8700 and displayed using a Hewlett-Packard SRX.

78: **Le Corbusier's Chapel at Ronchamp.**
This view of Le Corbusier's Chapel at Ronchamp was created using 4,000 steps of progressive radiosity. The stained glass windows were simulated using 113 area light sources. The radiosity solution was computed using Hewlett-Packard's licensable ARTCore Radiosity and Ray Tracing library running on an HP Model 835 TurboSRX workstation. The sunbeams were rendered using specially written ray tracing software run as a view-dependent post-process to the radiosity solution.

Credits:
The 'consolation room' image and the composite progressive radiosity image were modeled by German Bauer, Kevin Simon, and Virginia Weinhold, using in-house modeling and animation software and rendered with the RADIANCE global illumination package.

The 'color bleeding' image was modelled by Stephen Spencer using in-house modeling and animation software and rendered with the RADIANCE global illumination package.

The progressive variant images are courtesy of Shenchang Eric Chen, copyright 1988, Cornell University Program of Computer Graphics.

The participating medium image is courtesy of Holly Rushmeier. Copyright 1987, Cornell University Program of Computer Graphics.

The 'conference room' image is courtesy of Greg Ward. Copyright 1990 Lawrence Berkeley Laboratory, by Anat Grynberg and Greg Ward. It was rendered with the RADIANCE global illumination package.

The 'conference room' photograph is courtesy of Greg Ward.

The 'Theater' images are courtesy of Dan Baum. The radiosity algorithms and software used in the creation of these images are described in: 'Making Radiosity Usable: Automatic Preprocessing and Meshing Techniques for the Generation of Accurate Radiosity Solutions', by Daniel R. Baum, Steve Mann, Kevin P. Smith and James W. Winget. Published in SIGGRAPH '91.

The hardware used to create these images was a Silicon Graphics 4D-310 GTX.

The Candlestick Theater architect is Mark Mack Architects.

Database modeling by Charles Ehrlich, Department of Architecture, University of California at Berkeley.

The 'steel mill' image is courtesy of John Wallace, 3D/EYE Inc. It was produced by John Wallace and Stewart Feldman. Copyright 1987, Cornell University Program of Computer Graphics.

The 'Ronchamp' image is courtesy of Eric Haines, 3D/EYE Inc. It was modeled by Keith Howie and Paul Boudreau, and rendered by Eric Haines. Copyright 1991, The Hewlett-Packard Company.

Special thanks go to Wayne Carlson, Peter Carswell, Yina Chang, Erika Galvao, Carol Gigliotti, Barb Helfer, James Kent, Stephen May, Kevin Simon, Stephen Spencer, and Virginia Weinhold, all faculty, students, and/or staff members of The Ohio State University Advanced Computing Center for the Arts and Design and the Ohio Supercomputer Center, for their assistance in creating the slides.

Thanks also to Greg Ward, designer of the RADIANCE global illumination rendering system, for his assistance with debugging images and programs.