CS 451 Advanced Computer Graphics, Spring 2015

Dr. Bruce A. Maxwell
Department of Computer Science
Colby College

Course Description

The course covers advanced topics in computer graphics and builds on the fundamental algorithms and concepts learned in CS 351. Topics differ with each offering and may include advanced rendering and modeling techniques, ray tracing, radiosity, 3D computer game design, or advanced graphics hardware programming. Students learn to read and analyze primary source material and regularly present their work in class. Students work in groups to undertake extensive multi-week programming projects where they implement algorithms from the primary source material.

Prerequisites: CS 351 or permission of instructor. Linear algebra recommended.

Desired Course Outcomes

A. Students understand and can implement modern computer graphics algorithms.

B. Students understand and implement algorithms from academic research papers.

C. Students work in a group to design and develop 3D modeling and rendering software.

D. Students present methods, algorithms, results, and designs in an organized and competently written manner.

E. Students write, organize and manage a large software project.

1 OpenGL

API to a graphics library

State-based API: set an attribute and it remains until changed (similar to ours)

Two matrices: Projection and ModelView

- Modelview combines both the LTM and GTM and part of the VTM
- Projectionview is just the perspective matrix and screen scaling
- New operations get right multiplied onto the modelview matrix
- The last commend in the code is the first executed on the object

OpenGL types: lots of types, use them to make the code portable

Primitives

- glBegin - starts a primitive built from vertices
- glEnd - ends a primitive built from vertices
- glBegin takes as an argument how to interpret the vertices
- After glEnd, the object draws

GLU - openGL utility package, includes routines for make standard objects

GLUT - openGL toolkit package, includes routines for creating and managing windows

many other APIs built on top of OpenGL to make creating things easier.

A typical OpenGL program

- Main function that sets up the window, initializes callbacks and openGL states, then goes into a main loop
- Initialization code
- Callbacks for keys, resizing, mouse clicks
- Callback for display

Most of the important code will go in the display function, although some may be in the initialization.

Display lists are the OpenGL modules

- glGenLists() - returns an integer to be used to identify a display list
- glNewList() - starts a new display list
- glEndList() - ends a display list
- Any primitives or state changes made inside a display list get stored
- glCallList() - executes the given display list

Sample OpenGL program for drawing a lit sphere and responding to resize and keypress events.

```
#include <stdio.h>
#include <stdlib.h>
#include <GL/gl.h>
#include <GL/glut.h>

// This program draws a sphere lit by a single light source

// Function for drawing the contents of the screen
void display(void) {
  GLfloat position[] = {10.0, 5.0, 20.0, 1.0};
  int i;

  // clear screen
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  glColor3f(1.0, 1.0, 1.0);

  // reset the modelview matrix
  glLoadIdentity();

  // set up the viewing transformation
  gluLookAt(0.0, 0.0, 8.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);

  // set up the light
  glLightfv(GL_LIGHT0, GL_POSITION, position);

  // set up a sphere
  glutSolidSphere(1.0, 32, 32);

  // draw everything
  glFlush();
}

//  Function called when the window is resized
void reshape(int w, int h) {
  // set the viewport to the updated window size
  glViewport(0, 0, (GLsizei)w, (GLsizei)h);

  // move to projection matrix mode
  glMatrixMode(GL_PROJECTION);

  // reset the projection to identity
  glLoadIdentity();

  // create the new perspective matrix
  gluPerspective((GLdouble)30, (GLdouble)w / (GLdouble)h, (GLdouble)2, (GLdouble)100);

  // go back to model view matrix mode
  glMatrixMode(GL_MODELVIEW);

  // reset the modeview matrix
  glLoadIdentity();
}

// initialize the lighting and material color values
void initlights(void) {
  GLfloat ambient[] = {0.1, 0.1, 0.1, 1.0};
  GLfloat diffuse[] = {0.9, 0.9, 0.9, 1.0};
  GLfloat specular[] = {0.5, 0.5, 0.5, 1.0};
  GLfloat mat_diffuse[] = {0.3, 0.9, 0.5, 1.0};
  GLfloat mat_specular[] = {0.1, 0.1, 0.1, 1.0};
  GLfloat mat_shininess[] = {50.0};

  // material values
  glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse);
  glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
  glMaterialfv(GL_FRONT, GL_SHININESS, mat_shininess);

  // generic lighting values
  glLightModeli(GL_LIGHT_MODEL_LOCAL_VIEWER, 1);
  glLightModelfv(GL_LIGHT_MODEL_AMBIENT, ambient);

  // specific light source
  glLightfv(GL_LIGHT0, GL_DIFFUSE, diffuse);
  glLightfv(GL_LIGHT0, GL_SPECULAR, specular );

  // enable lighting, light0 and depth testing
  glEnable(GL_LIGHTING);
  glEnable(GL_LIGHT0);
  glEnable(GL_DEPTH_TEST); // important, or you can see through the sphere

}
```

```
// init function
void init(void) {
  // background color
  glClearColor(0.0, 0.0, 0.0, 0.0);

  // whether shading is smooth or flat
  glShadeModel(GL_SMOOTH);

  initlights();
}

//  This function is called whenever a key event occurs.
//  The key value is in "key" and "x" and "y" hold the current mouse position
void keyboard(unsigned char key, int x, int y)
{
   switch( key) {
   case 'q': // quit
     exit(0);
     break;
   default:
     break;
   }
}

// main function
int main(int argc, char **argv) {

  // initialize
  glutInit(&argc, argv);
  glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB | GLUT_DEPTH);
  glutInitWindowSize(500, 500);
  glutInitWindowPosition(100, 100);
  glutCreateWindow("Sphere");
  init();

  // set up the callback functions
  glutDisplayFunc(display);
  glutReshapeFunc(reshape);
  glutKeyboardFunc(keyboard);

\// run the main loop
  glutMainLoop();

  return(0);
}
```

```
#MAKEFILE

CC=gcc
CFLAGS=-I/usr/X11R6/include -L/usr/X11R6/lib -I/opt/local/include -L/opt/local/lib -I./include -bind_at_load

gltest: glTest.c
gcc $(CFLAGS) -o $@ $^ -lglut -lGLU -lGL -lX11 -lm -lXmu

clean:
rm -f *.o *~ a.out
```

2 Photorealistic Rendering

Photorealistic rendering encompasses a wide variety of techniques, including shading models, global illumination models, physically realistic animation, and physically correct models of natural phenomena.

2.1 Ray Tracing

The most common method of rendering a scene is some type of rendering algorithm that takes advantage of the coherence of objects. Hardware pipelines are optimized to project polygons, usually triangles, from a 3D space into a 2D space and draw them individually onto the screen. The hardware rendering pipelines further divide the process into one of making shading calculations at vertices and then again on fragments, which are triangles or sub-areas of triangles.

The default rendering pipeline using a z-buffer for hidden surface removal does not inherently include shadows, transparency, reflection, or the interreflection of light between surfaces. Efforts to include these physically realistic effects are normally add-on modules that are designed to either fit within the hardware pipeline structure or are design elements that provide a sufficiently realistic result but do not represent an actual physical model of the effect.

Shadows, for example, can be built into a rendering pipeline by adding either shadow polygons or shadow volumes to the scene, enabling light sources to be turned on or off for a particular surface. Likewise, reflections can be modeled by generating textures that consist of scene renderings from the point of view of the object: environment mapping. Both of these achieve realistic effects but the rendering pipeline does not explicitly model the physical processes underlying the phenomena.

Ray tracing was the first rendering methodology developed to integrate hidden surface removal, shadows, reflection, and transparency in a single algorithm (Whitted, 1980). The basic concept of ray tracing is to send a ray from the eye location through each pixel of the image plane out into the scene. Intersecting the ray with each object in the scene indicates which surface is visible. Given the point of intersection, the algorithm can use additional rays to test the visibility of each light source and throw more rays into the scene in the perfect reflection direction and/or through the surface in the case of transparency.

The end result of throwing a ray into the scene is a tree of rays, some representing reflections, some representing transmissions. In addition, at each intersection point the algorithm knows which light sources are available. By traversing the tree of rays and collecting their contributions in a physically correct manner, the algorithm identifies the intensity and color at the source pixel. Repeating the process for each pixel generates an image of the scene that includes hidden surface removal, shadows, reflection, and transmission.

The general algorithm for each pixel is the following.

1. Calculate the ray $v_{i,j}$ from the eye through pixel (i, j).
2. Color the pixel with the result of calling RayIntersect($v_{i,j}$, PolygonDatabase)

function RayIntersect(vector, database) returns Color

1. if $\beta <$ Cutoff return Black
2. Intersect the ray with the polygons in the scene, identifying the closest intersection.
3. If there is no intersection, return BackgroundColor
4. Calculate the surface normal at the intersection point on the closest intersecting polygon.
5. Set the return color C to black.
6. For each light source L_i
   (a) Send a ray (or many, if an area source) towards L_i
   (b) Intersect the ray with each polygon in the scene
   (c) If the ray intersects an opaque polygon, L_i is blocked.
   (d) If the ray intersects only transparent polygons, L_i is partially blocked.
   (e) Add the contribution of shading the surface with L_i to C.
7. Calculate the perfect reflection/transmission direction v_r
8. Calculate the magnitude of the surface reflection/transmission α
9. return $C + \alpha$RayIntersect$(v_r, \text{PolygonDatabase})$

The primary task of a ray tracer is to intersect a ray with object models. Object models may be mathematical functions, polygons, splines, or even volumetric objects representing amorphous phenomena like gasses or clouds. Most of the ray tracer's time is spent figuring out which object the ray hits first. The ray tracer spends a much smaller amount of time computing shading values, the proper direction in which to send more rays, and the overall shading value of a pixel.

Because the bulk of a ray tracer's time–and code–are spent doing ray-object intersections, there are two primary methods of speeding up a ray tracer. First, you can try to reduce the number of ray-object intersection tests. The primary method of reducing the number of tests is to organize the scene in a hierarchy of simple bounding surfaces. If a ray does not intersect one of the enclosing bounding surfaces, then it does not intersect any of the objects within it. Bounding surfaces that have fast ray-object intersection calculations include circles and boxes.

The second approach to speeding up a ray tracer is to make the ray-object intersection calculation faster. There are a number of different techniques that fall in this category.

- Use object models that enable fast ray-object computations. Implicit algebraic surfaces, for example, tend to have fast solutions.
- Organize the computations to optimize the use of computer hardware like memory caches.
- Parallelize the computations across CPUs.

2.2 Radiosity

One of the major perceptual issues with computer graphics is that it is too clean. Standard z-buffer rending systems can use more complex shading and shadow algorithms to capture surface properties, shadows, and highlights, but they do not incorporate any of the interaction–interreflection–between nearby surfaces: light bouncing off one surface and illuminating a nearby surface. Ray tracing starts to address this issue by using a comprehensive rendering algorithm that incorporates specular reflection. Distributed ray tracing that shoots additional rays according to the overall surface reflection function–which includes both body and surface reflection–addresses the issue of diffuse-diffuse reflection directly, but at tremendous computational cost.

Radiosity is a different algorithm for solving the global illumination function that explicitly models how each patch of the world is related to every other patch (Goral et. al., 1984). The fundamental idea of the radiosity algorithm is that the illumination arriving at a patch–the irradiance–and the illumination leaving a patch–the radiance–are related by the albedo of the surface, modified by any self-emittance the surface may possess. The albedo is the percent of the incoming light reflected back to the world. The radiosity of a patch is the energy per unit surface leaving the patch. In rendering, this is proportional to the visible color of the patch from the point of view of the camera.

The incoming light is represented by an integral: it is the sum of the light arriving on the patch from all directions. That light can come from emitters–light sources–or from surfaces reflecting their own incoming light. If we assume a Lambertian reflectance function, which models reflected light as leaving equally in all directions, then the amount of light leaving the surface per solid angle is a constant and independent of the viewing direction. Therefore, the amount of light leaving one patch and arriving at another is related only to the geometry of the two patches. If they are both large and close together, much of the energy leaving one patch will arrive at the other patch. On the other hand, if the patches are distant and/or small, very little energy leaving one of the patches will arrive at the other patch.

If we create all of the relationships between all of the patches in the scene, we arrive at a large number of linear equations that represent the flow of energy around the system. Given a fixed set of emitting surfaces, there is a steady state solution to the system that defines the radiosity of each patch. The radiosity algorithm for solving for the steady state values comes from the domain of heat flow.

The radiosity of a surface consists of two parts: the energy emitted by a surface and the energy reflected by the surface.

$$B_i = E_i + \rho_i H_i \tag{1}$$

- B_i : the radiosity, which is the energy per unit surface area leaving surface i
- E_i: the self-emittance of the surface, which is non-zero only for light sources
- ρ_i : the albedo of the surface, or the percent of incoming energy that is reflected
- H_i : the irradiance, or radiant energy incident on the surface

The incident radiant energy on a surface patch is the sum of the radiant intensities of all surfaces j that are visible from surface i, modified by the geometry of the relationship between the surfaces. We can assume that the radiosity of a surface does not vary across a small patch.

$$H_i = \sum_{j=1}^{N} B_j \frac{A_j F_{ji}}{A_i} \tag{2}$$

- B_j : the radiosity of patch j
- A_j : the area of patch j
- A_i : the area of patch i
- F_{ji} : the fraction of energy leaving surface j that hits surface i

Equation (2) defines the relationship between patches in the scene. The incident energy from patch j is proportional to its radiosity and area, and it is inversely proportional to the area of patch i.

The form factor between two patches is reciprocal: if you know the relationship in one direction, then you know it in the other direction.

$$A_i F_{ij} = A_j F_{ji} \tag{3}$$

By substituting the definition of H into the radiosity equation and making use of the reciprocity relationship, we can redefine the radiosity equation and then solve for the emittance, putting all of the radiosity values together as in (5).

$$\begin{aligned} B_i &= E_i + \rho_i \sum_{j=1}^{N} B_j \frac{A_j F_{ji}}{A_i} \\ B_i &= E_i + \rho_i \sum_{j=1}^{N} B_j F_{ij} \end{aligned} \tag{4}$$

$$E_i = B_i - \rho_i \sum_{j=1}^{N} B_j F_{ij} \tag{5}$$

Writing (5) for each surface patch in the scene creates a large set of linear equations that relate each patch to each other patch.

$$\begin{bmatrix} 1-\rho_1 F_{11} & -\rho_1 F_{12} & \dots & -\rho_1 F_{1N} \\ -\rho_2 F_{21} & 1-\rho_2 F_{22} & \dots & -\rho_2 F_{2N} \\ \dots & \dots & \dots & \dots \\ -\rho_N F_{N1} & -\rho_N F_{N2} & \dots & 1-\rho_N F_{NN} \end{bmatrix} \begin{bmatrix} B_1 \\ B_2 \\ \dots \\ B_N \end{bmatrix} = \begin{bmatrix} E_1 \\ E_2 \\ \dots \\ E_N \end{bmatrix} \tag{6}$$

Note the following properties of the radiosity matrix equation.

- All of the F_{ii} form factors are zero, making all of the diagonal terms 1.
- The E_i are all zero except for light sources.
- The sum of the form factors in a row is ≤ 1

Applying the equation in all three color channels–wavelengths–gives the radiosities for each color channel, which are proportional to the apparent intensity of each surface patch in a scene. Once calculated, for a static scene the form factors are fixed, which means they can be re-used if the colors, lighting, or albedos of surfaces change.

The radiosity matrix also has the property that the Gauss-Seidel iterative solution technique is a useful, and potentially fast, method of solving this set of equations. Note that the matrix may be very large, as radiosity is only effective if each patch in the scene is small. Modeling a flat wall, for example, must be done with many small patches, not a single large polygon.

2.3 Radiosity Algorithm

The complete radiosity algorithm is as follows.

1. Discretize the environment into small patches.
   - The patch size must be small enough to catch changes in the illumination field.
   - Large patches relative to the changing illumination conditions will cause aliasing.
   - Adaptive subdivision during the radiosity process is possible to implement.
2. Calculate the form factors between all of the patches.
   - There are $\frac{N^2}{2}$ form factors.
   - The algorithm can calculate the form factors on demand (lazy evaluation).
3. Solve the radiosity matrix using Gauss-Seidel iteration.
4. Use the radiosities in a standard z-buffer rendering pipeline.
   - Render the scene using Gauraud shading
   - Can add direct surface reflection to the scene with Phong shading.
   - The color of a vertex is the average of adjoining patch radiosities.

2.4 Radiosity Form Factors

Calculating the radiosity form factors is, in general, an order of magnitude more expensive than solving the radiosity matrix. Gauss-Seidel iteration is an $O(N)$ process, in practice, but there are $O(N^2)$ form factors, given N patches in the environment.

The generic method for calculating form factors is to use their geometric relationship and some calculus to calculate the flow of energy from one patch to another. For a convex space with no interior obstacles, such an approach is reasonable. However, in a general environment with obstacles, it quickly becomes a challenge to develop a closed form solution.

One of the most important developments in making radiosity practical was the development of the hemi-cube solution to the generation of form factors (Cohen and Greenberg, 1985). A hemi-cube is the half of a cube enclosing the visible side of a surface patch. The hemi-cube represents the complete illumination environment of the target surface patch. To discover the visibility of each surface patch from the target patch we can use a z-buffer algorithm to render the scene onto the hemi-cube from the point of view of the target.

Each pixel of the rendered images on the sides of the hemi-cube represents a solid angle of space, and the polygon inside that pixel is the surface contributing to incoming light from that direction. Therefore, the form factor between the target polygon i, and a projected polygon j is proportional to the number of pixels on the hemi-cube covered by polygon j. We can pre-compute the form factor contribution of each pixel on the hemi-cube, enabling us to compute the form factor F_{ji} (and F_{ij} by reciprocity) by summing the pixels covered by polygon j.

$$\text{X-Y plane (top)} : \Delta F = \frac{1}{\pi(x^2 + y^2 + 1)^2}\Delta A \tag{7}$$

$$\text{Y-Z plane (side)} : \Delta F = \frac{z}{\pi(y^2 + z^2 + 1)^2}\Delta A \tag{8}$$

$$\text{X-Z plane (side)} : \Delta F = \frac{z}{\pi(x^2 + z^2 + 1)^2}\Delta A \tag{9}$$

Note that one hemi-cube operation calculates the form factors between the target patch and all other patches. Therefore, one hemi-cube rendering generates all of the form factors for a row of the radiosity matrix.

2.5 Re-ordering the Radiosity Solution

The major drawback to radiosity is the computational cost of both the form factors and the computation. Each row of the radiosity matrix is represented by (10), which defines the value of the radiosity of patch B_i in terms of its emittance, albedo, and incident energy.

$$B_i = E_i + \rho_i \sum_{j=1}^{N} B_j F_{ij} \tag{10}$$

The Gauss-Seidel iteration method of solving for the radiosity values updates each row of the matrix in turn. The new estimate of the radiosity is adjusted closer to its true value using a gradient-descent style solution. Generically, Gauss-Seidel iteration uses (11) to solve the equation $Ax = b$, where $x_i^{(k+1)}$ is the new value of the unknown variable, $x_i^{(k)}$ is the current value, a_{ij} is element ij of A and b_i is element i of b.

$$x_i^{(k+1)} = \frac{1}{a_{ii}}\left[b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{(k+1)} - \sum_{j=i+1}^{N} a_{ij}x_{(k)}\right] \tag{11}$$

Re-written for radiosity, the iteration process looks like (12), since the diagonals all have value 1. The initial values of the radiosities B_i are set to the emittance values of the patch E_i. Note that most of the initial values will be zero; only light sources will have non-zero values.

$$B_i^{(k+1)} = E_i - \sum_{j=1}^{i-1} \rho_i F_{ij} B_j^{(k+1)} - \sum_{j=i+1}^{N} \rho_i F_{ij} B_{(k)} \tag{12}$$

Conceptually, (12) says that the new radiosity value is a function of all the patches sending light to the target patch i. Since Gauss-Seidel iteration starts with most radiosity values at zero, only emitting patches

will modify the radiosities on the first iteration, and then only the surfaces visible from the light sources (non-zero form factors). The concept is that each patch is gathering light to it. However, it is a lot of computational effort in the first few iterations with very little progress.

Computationally, it is possible modify the G-S process to update columns of the matrix instead of rows. Conceptually, this reverses the distribution of energy so that each patch is shooting energy to its neighbors rather than gathering. While this does not seem like an immediate win, since most patches will be shooting zero energy to start, it is also possible re-order the order in which columns get updated so that patches emitting a lot of energy send that energy to other patches first. Re-ordering the computations and using a shooting concept is called progressive radiosity (Cohen, Chen, Wallace, and Greenberg, 1988). It significantly speeds up the radiosity computation and also enables quicker previews of the final scene.

Progressive Radiosity
(Cohen, Chen, Wallace, and Greenberg, 1988)

set all $B_i = 0$ and $\Delta B_i = 0$ except for emitting surfaces, which are set to E_i
for each Gauss-Seidel iteration
    build a max-heap based on $\Delta B_i A_i$
    for each patch i in the max-heap
        calculate form factors F_{ij} with a hemi-cube at patch i
        for each patch j
            $\Delta Rad = \rho_j \Delta B_i F_{ij} A_i / A_j$
            $\Delta B_j = \Delta B_j + \Delta Rad$
            $B_j = B_j + \Delta Rad$
        $\Delta B_i = 0$