CS380 Introduction to Computer Graphics
Homework #8

20160042 Inyong Koo

Describe the following terms with respect to computer graphics.

Chapter 11

1. Shadow rays

   In ray tracing, rather than immediately applying our reflection model, we first check whether the point of intersection between the cast ray and the surface is illuminated. We compute **shadow rays** from the point on the surface to each source. If a shadow ray intersects a surface before it meets the source, the light is blocked from reaching the point under consideration, and this point is in shadow, at least from this source.

2. Recursive ray tracing

   A ray tracer returns a color `c` for the ray with a starting point `p` and a direction `d`. We first check whether the ray intersects with an object. If the ray intersects with a light source, ray tracer returns the color of the light source. If the ray doesn't intersect with any object, it returns the background color. If it intersects with an object, it creates a reflection ray and a transmitted ray. We determine the color of the intersection, and recursively trace ray for the reflection and the transmitted ray. We should count the depth of recursion and set its limit in order to make the recursion doesn't happen infinitely. This is called **recursive ray tracing**.

3. Photon mapping

   Photon mapping follows individual photons, the carriers of light energy, from when they are produced at the light sources to when they are finally absorbed by surfaces in the scene. Photons typically go through multiple reflections and transmissions from creation to final absorption. The approach can handle complecx lighting of the sort that characterizes real-world scenes.

4. Radiosity

   The **radiosity** method can approximate the diffuse-diffuse interactions using an energy approach that was originally used for solving problems in heat transfer.

   The basic radiosity method breaks up the scene into small flat polygons, or patches, each of which can be assumed to be perfectly diffuse and renders in a constant shade. First we must find these shades. Once we have found them, we have effectively assigned a color to each patch that is independent of the viewer. Effectively, we have assigned colors to set of polygon patches in a three-dimensional environment. We can now place the viewere wherever we wish and render the scene in a conventional manner, using a pipe renderer.

   When we assume our scene consists of n patches, The radiosity of patch i, b_i, is the light intensity per unit area leaving the patch. As we are measuring intensity at a fixed wavelength, we can think of a radiosity function $b_i(\lambda)$ that determines the color of patch i.The patch radiosity can be calculated by the radiosity equation:

   $$b_i = e_i + \rho_i \sum_{j=0}^{n} f_{ij} b_j$$

   where e is the emitted intensity and f is the form factor.

5. Form factor

   The **form factor** represents the fraction of the energy leaving one patch that reaches another patch. The form factor depends on how the two patches are oriented relative to each other, how far they are from each other, and whether any other patches occlude the light from one patch and prevent it from reaching the other patch.

   The reciprocity equation shows the relationship between two patches.

$$f_{ij}a_i = f_{ji}a_j$$

6. Parallel rendering

   In many applications, particularly in the scientific visualization of large geometric data sets, we may face some challenges due to the display and the processor. In order to handle this problem, there are multiple ways to distribute the work that must be done to render a scene among the various processors.

   The simpleset approach is to execute the same application on each processor but have each use a different window that corresponds to where the processor's display is located in the output array. Another approach is to assign, or sort, primitives to the correct areas of the display in the rendering process.

7. Volume rendering

   We've been handling the display of surfaces, but we have to come up with different rendering scheme if we have a set of data in which each value represents a value at a point within a three-dimensional region. Visualization of three-dimensional scalar fields is more difficult, since it has more data to work with, and we lack the dimension to use for display. The field of **volume rendering** deals with these problems, mostly by extending our previously developed methods.

8. Marching cubes

   When we seek an approximate isosurface using the voxel values of the sclar field, A method, called **marching cubes**, approximates a surface by generating a set of three-dimensional triangles, each of which is an approximation to a piece of the isosurface.

9. Image-based rendering

   If we had a three-dimensional model, we would simply move the viewer or the object to construct the new image. But if we have only two-dimensional information, we need to accomodate using **image-based rendering**.

10. Light-field rendering

    Light-field rendering looks at the mathematical relationship between two-dimensional images and the light distribution in the three-dimensional environment. Each two-dimensional image is a sample of a four-dimensional light field. In a manner akin to how three-dimensional images are constructed from two-dimensional projections in computerized axial tomography, two-dimenisional projections from multiple cameras can be used to reconstruct the three-dimensional world.

Modeling

1. Scene graph

   In order to compose a scene, we can extend our use of tree data structures to describe the relationships among geometric objects, cameras, lights, and attributes. A graphical application program traverses the graph, and render the scenary as we push and pop the components to the pipeline. *The scene graph is equivalent to an OpenGl program* in the sense that we can use the tree to generate the proram in a totally Mechanical fashion. This approach was taken by Open Inventor and later by Open Scene Graph (OSG), both object-oriented APIs that were built on top of OpenGL.

2. Procedural methods in modeling

   We introduce four of many possible approaches to procedural modeling.

   (a) *Working with particles that obey Newton's law.*

   We can design systems of particles that are capable of ocmplex behaviors that arise from solving sets of differential equations - a routine numerical task for up to thousands of particles.

   (b) *Language-based models*

   We can control complexity by replacing polygonal models with models similar to those used for both natural and computer languages, to approximate many natural objects with a few rules that generate the required graphical entities.

   (c) *Fractal geometry*

   Fractal geometry is based on self-similarity that we see in many natural phenomena. It gives us a way of generating models at any desired level of detail.

   (d) *Procedural noise*

   It is a method of introducing a controlled amount of randomness into our models. It has been used to create texture maps, turbulent behavior in fluid models, realistic motion in animations, and fuzzy objects such as clouds.

3. Cubic parametric polynomial curve

   We can write a cubic parametric polynomial using a row and column matrix as

$$\mathbf{p}(u) = \mathbf{c}_0 + \mathbf{c}_1 u + \mathbf{c}_2 u^2 + \mathbf{c}_3 u^3 = \mathbf{u}^T \mathbf{c}$$

   where $\mathbf{c}$ is column vector of $\mathbf{c}_0, \mathbf{c}_1, \mathbf{c}_2, \mathbf{c}_3$, $\mathbf{u}$ is column vector of $1, u, u^2, u^3$.

   We can choose the degree of the curve, but the cubic polynomial curves may be sufficient to allow us to produce the desired shape in a small region, with reasonable amount of parameters to work with.