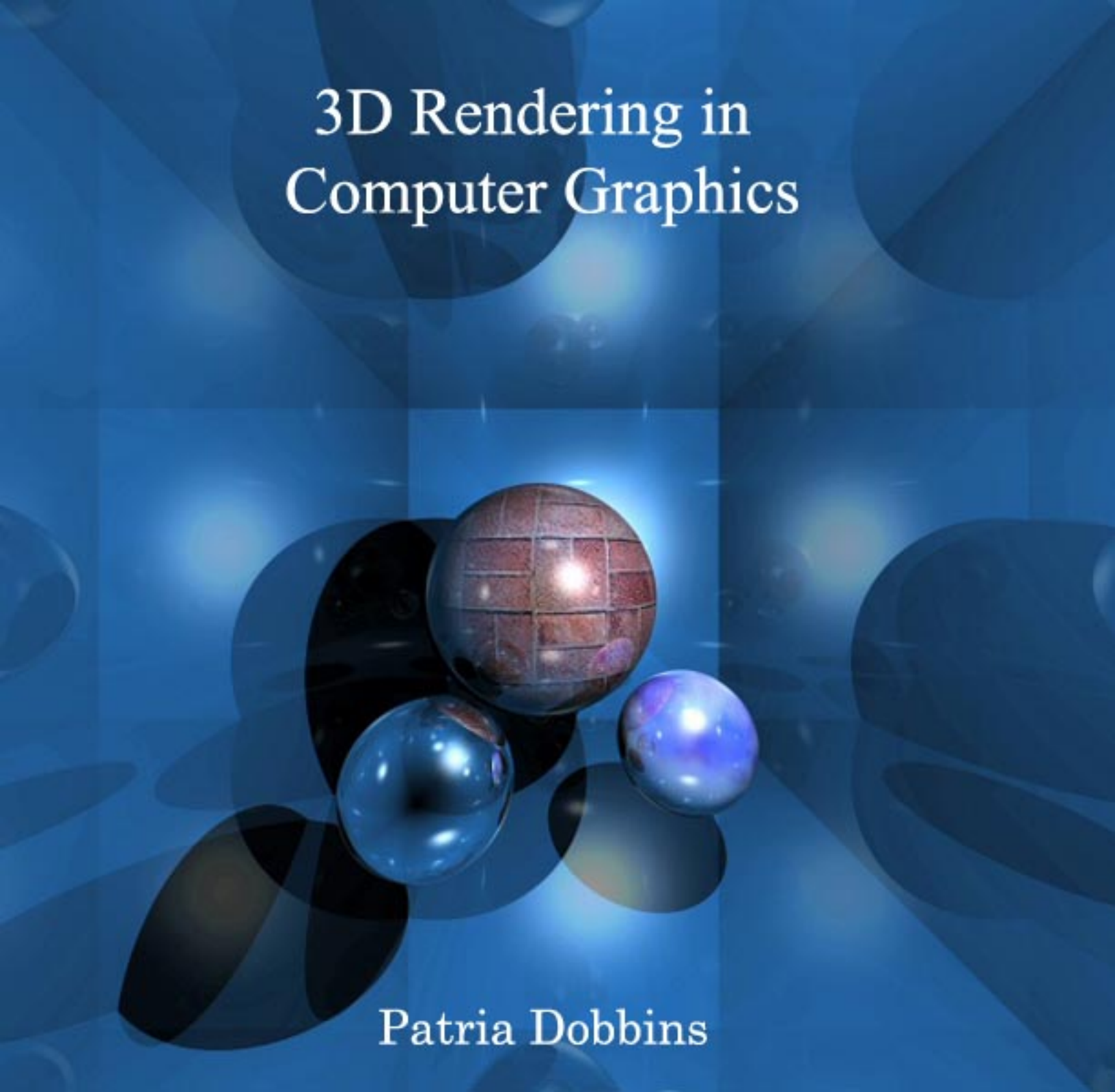
3D Rendering in Computer Graphics



Patria Dobbins

First Edition, 2012

ISBN 978-81-323-4264-9

Published by:
White Word Publications
4735/22 Prakashdeep Bldg,
Ansari Road, Darya Ganj,
Delhi - 110002
Email: info@wtbooks.com

Table of Contents

Introduction

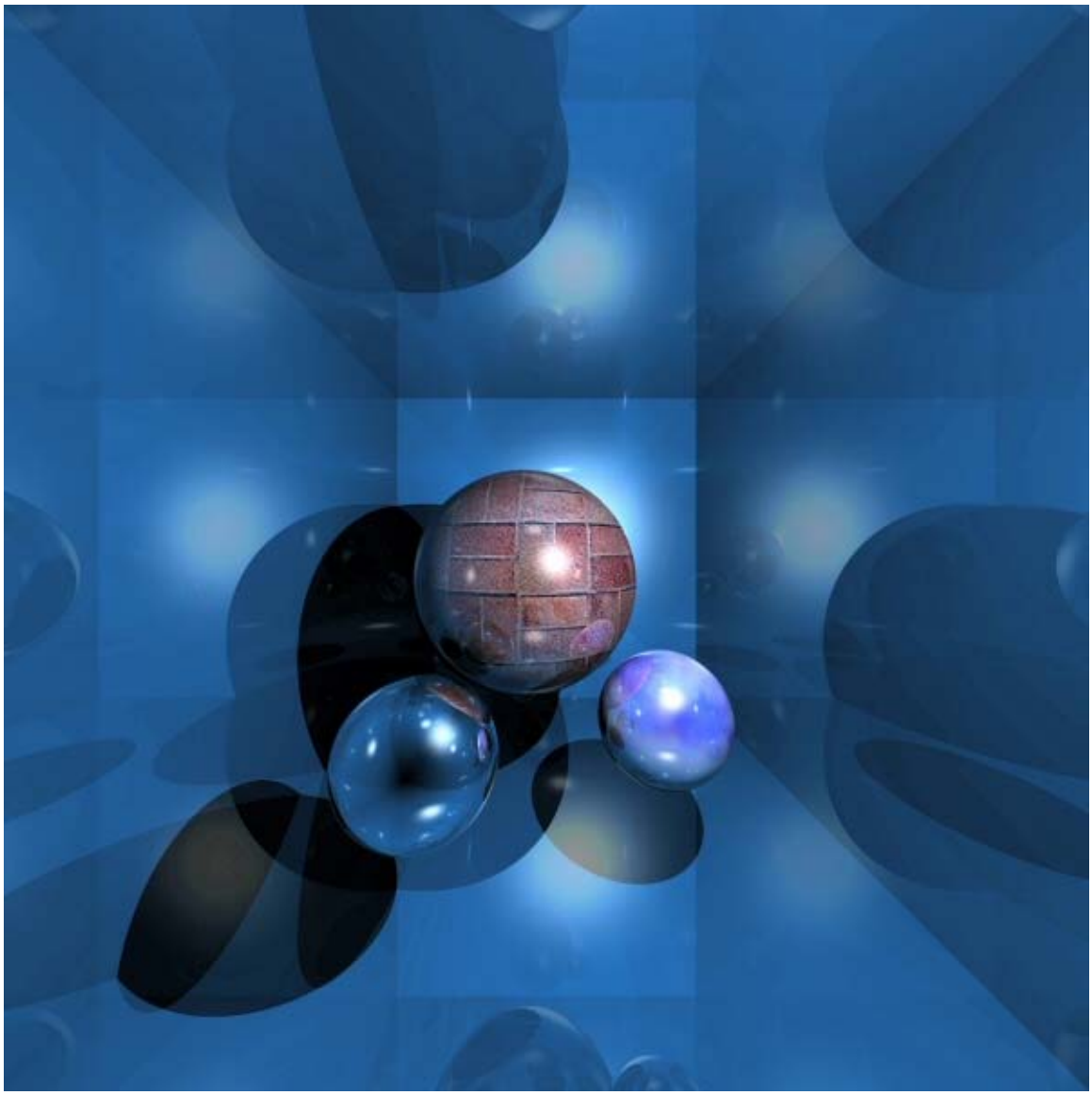
3D computer graphics

3D rendering is the 3D computer graphics process of automatically converting 3D wire frame models into 2D images with 3D photorealistic effects on a computer.

Rendering methods

Rendering is the final process of creating the actual 2D image or animation from the prepared scene. This can be compared to taking a photo or filming the scene after the setup is finished in real life. Several different, and often specialized, rendering methods have been developed. These range from the distinctly non-realistic wireframe rendering through polygon-based rendering, to more advanced techniques such as: scanline rendering, ray tracing, or radiosity. Rendering may take from fractions of a second to days for a single image/frame. In general, different methods are better suited for either photo-realistic rendering, or real-time rendering.

Real-time



An example of a ray-traced image that typically takes seconds or minutes to render.

Rendering for interactive media, such as games and simulations, is calculated and displayed in real time, at rates of approximately 20 to 120 frames per second. In real-time rendering, the goal is to show as much information as possible as the eye can process in a 30th of a second (or one frame, in the case of 30 frame-per-second animation). The goal here is primarily speed and not photo-realism. In fact, exploitations can be applied in the way the eye 'perceives' the world, and as a result the final image presented is not necessarily that of the real-world, but one close enough for the human eye to tolerate. Rendering software may simulate such visual effects as lens flares, depth of field or motion blur. These are attempts to simulate visual phenomena resulting from the optical characteristics of cameras and of the human eye. These effects can lend an element of realism to a scene, even if the effect is merely a simulated artifact of a camera. This is the

basic method employed in games, interactive worlds and VRML. The rapid increase in computer processing power has allowed a progressively higher degree of realism even for real-time rendering, including techniques such as HDR rendering. Real-time rendering is often polygonal and aided by the computer's GPU.

Non real-time

Computer-generated image created by Gilles Tran.

Animations for non-interactive media, such as feature films and video, are rendered much more slowly. Non-real time rendering enables the leveraging of limited processing power in order to obtain higher image quality. Rendering times for individual frames may vary from a few seconds to several days for complex scenes. Rendered frames are stored on a hard disk then can be transferred to other media such as motion picture film or optical disk. These frames are then displayed sequentially at high frame rates, typically 24, 25, or 30 frames per second, to achieve the illusion of movement.

When the goal is photo-realism, techniques such as ray tracing or radiosity are employed. This is the basic method employed in digital media and artistic works. Techniques have been developed for the purpose of simulating other naturally-occurring effects, such as the interaction of light with various forms of matter. Examples of such techniques include particle systems (which can simulate rain, smoke, or fire), volumetric sampling (to simulate fog, dust and other spatial atmospheric effects), caustics (to simulate light

focusing by uneven light-refracting surfaces, such as the light ripples seen on the bottom of a swimming pool), and subsurface scattering (to simulate light reflecting inside the volumes of solid objects such as human skin).

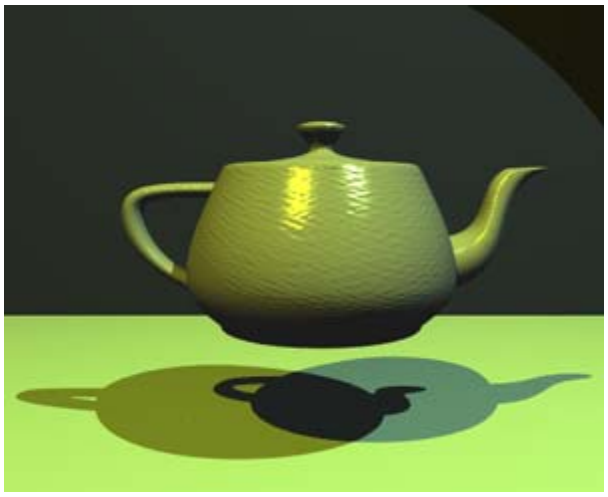
The rendering process is computationally expensive, given the complex variety of physical processes being simulated. Computer processing power has increased rapidly over the years, allowing for a progressively higher degree of realistic rendering. Film studios that produce computer-generated animations typically make use of a render farm to generate images in a timely manner. However, falling hardware costs mean that it is entirely possible to create small amounts of 3D animation on a home computer system. The output of the renderer is often used as only one small part of a completed motion-picture scene. Many layers of material may be rendered separately and integrated into the final shot using compositing software.

Reflection and shading models

Models of *reflection/scattering* and *shading* are used to describe the appearance of a surface. Although these issues may seem like problems all on their own, they are studied almost exclusively within the context of rendering. Modern 3D computer graphics rely heavily on a simplified reflection model called *Phong reflection model* (not to be confused with Phong shading). In refraction of light, an important concept is the refractive index. In most 3D programming implementations, the term for this value is "index of refraction," usually abbreviated "IOR." Shading can be broken down into two orthogonal issues, which are often studied independently:

- **Reflection/Scattering** - How light interacts with the surface *at a given point*
- **Shading** - How material properties vary across the surface

Reflection



The Utah teapot

Reflection or scattering is the relationship between incoming and outgoing illumination at a given point. Descriptions of scattering are usually given in terms of a bidirectional scattering distribution function or BSDF. Popular reflection rendering techniques in 3D computer graphics include:

- Flat shading: A technique that shades each polygon of an object based on the polygon's "normal" and the position and intensity of a light source.
- Gouraud shading: Invented by H. Gouraud in 1971, a fast and resource-conscious vertex shading technique used to simulate smoothly shaded surfaces.
- Texture mapping: A technique for simulating a large amount of surface detail by mapping images (textures) onto polygons.
- Phong shading: Invented by Bui Tuong Phong, used to simulate specular highlights and smooth shaded surfaces.
- Bump mapping: Invented by Jim Blinn, a normal-perturbation technique used to simulate wrinkled surfaces.
- Cel shading: A technique used to imitate the look of hand-drawn animation.

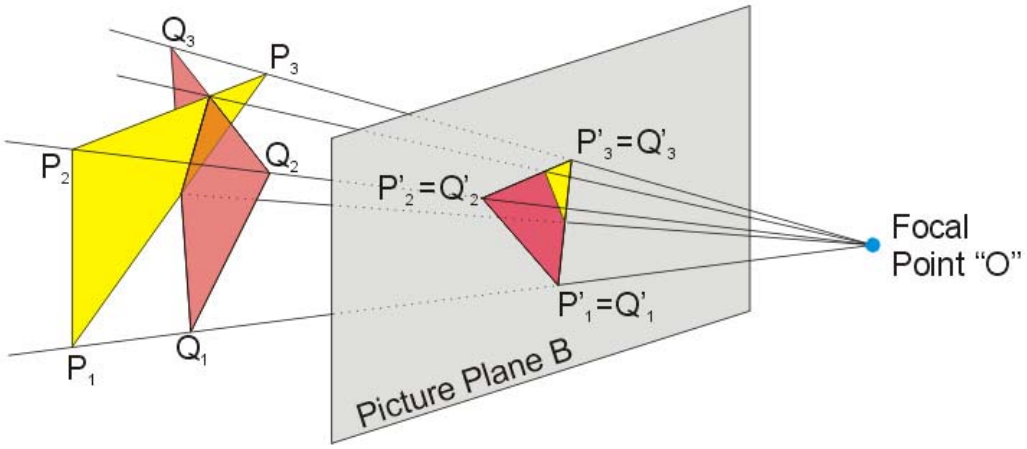
Shading

Shading addresses how different types of scattering are distributed across the surface (i.e., which scattering function applies where). Descriptions of this kind are typically expressed with a program called a shader. (Note that there is some confusion since the word "shader" is sometimes used for programs that describe local *geometric* variation.) A simple example of shading is texture mapping, which uses an image to specify the diffuse color at each point on a surface, giving it more apparent detail.

Transport

Transport describes how illumination in a scene gets from one place to another. Visibility is a major component of light transport.

Projection



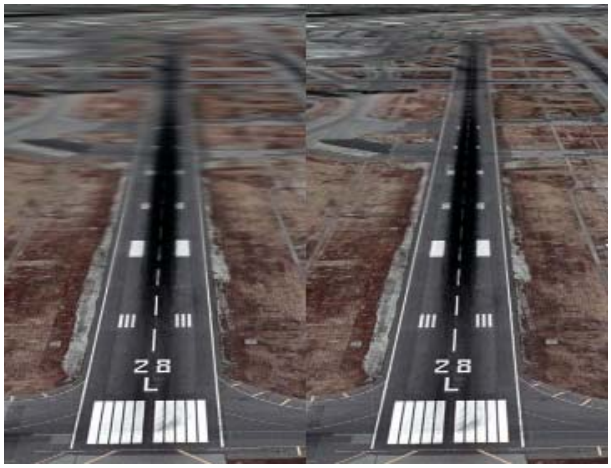
Perspective Projection

The shaded three-dimensional objects must be flattened so that the display device - namely a monitor - can display it in only two dimensions, this process is called 3D projection. This is done using projection and, for most applications, perspective projection. The basic idea behind perspective projection is that objects that are further away are made smaller in relation to those that are closer to the eye. Programs produce perspective by multiplying a dilation constant raised to the power of the negative of the distance from the observer. A dilation constant of one means that there is no perspective. High dilation constants can cause a "fish-eye" effect in which image distortion begins to occur. Orthographic projection is used mainly in CAD or CAM applications where scientific modeling requires precise measurements and preservation of the third dimension.

Chapter 1

Anisotropic Filtering & Ambient Occlusion

Anisotropic Filtering



An illustration of texture filtering methods showing a trilinear mipmapped texture on the left and the same texture enhanced with anisotropic texture filtering on the right.

In 3D computer graphics, **anisotropic filtering** (abbreviated **AF**) is a method of enhancing the image quality of textures on surfaces that are at oblique viewing angles with respect to the camera where the projection of the texture (not the polygon or other primitive on which it is rendered) appears to be non-orthogonal (thus the origin of the word: "an" for *not*, "iso" for *same*, and "tropic" from tropism, relating to direction; anisotropic filtering does not filter the same in every direction).

Like bilinear and trilinear filtering Anisotropic filtering eliminates aliasing effects, but improves on these other techniques by reducing blur and preserving detail at extreme viewing angles.

Anisotropic filtering is relatively intensive (primarily memory bandwidth and to some degree computationally, though the standard space-time tradeoff rules apply) and only became a standard feature of consumer-level graphics cards in the late 1990s. Anisotropic filtering is now common in modern graphics hardware (and video driver software) and is enabled either by users through driver settings or by graphics applications and video games through programming interfaces.

An improvement on isotropic MIP mapping

Hereafter, it is assumed the reader is familiar with MIP mapping.

If we were to explore a more approximate anisotropic algorithm, RIP mapping (rectim in parvo) as an extension from MIP mapping, we can understand how anisotropic filtering gains so much texture mapping quality. If we need to texture a horizontal plane which is at an oblique angle to the camera, traditional MIP map minification would give us insufficient horizontal resolution due to the reduction of image frequency in the vertical axis. This is because in MIP mapping each MIP level is isotropic, so a 256 × 256 texture is downsized to a 128 × 128 image, then a 64 × 64 image and so on, so resolution halves on each axis simultaneously, so a MIP map texture probe to an image will always sample an image that is of equal frequency in each axis. Thus, when sampling to avoid aliasing on a high-frequency axis, the other texture axes will be similarly downsampled and therefore potentially blurred.

With RIP map anisotropic filtering, in addition to downsampling to 128 × 128, images are also sampled to 256 × 128 and 32 × 128 etc. These anisotropically downsampled images can be probed when the texture-mapped image frequency is different for each texture axis and therefore one axis need not blur due to the screen frequency of another axis and aliasing is still avoided. Unlike more general anisotropic filtering, the RIP mapping described for illustration has a limitation in that it only supports anisotropic probes that are axis-aligned in texture space, so diagonal anisotropy still presents a problem even though real-use cases of anisotropic texture commonly have such screenspace mappings.

In layman's terms, anisotropic filtering retains the "sharpness" of a texture normally lost by MIP map texture's attempts to avoid aliasing. Anisotropic filtering can therefore be said to maintain crisp texture detail at all viewing orientations while providing fast anti-aliased texture filtering.

Degree of anisotropy supported

Different degrees or ratios of anisotropic filtering can be applied during rendering and current hardware rendering implementations set an upper bound on this ratio. This degree refers to the maximum ratio of anisotropy supported by the filtering process. So, for example 4:1 (pronounced 4 to 1) anisotropic filtering will continue to sharpen more oblique textures beyond the range sharpened by 2:1.

In practice what this means is that in highly oblique texturing situations a 4:1 filter will be twice as sharp as a 2:1 filter (it will display frequencies double that of the 2:1 filter). However, most of the scene will not require the 4:1 filter; only the more oblique and usually more distant pixels will require the sharper filtering. This means that as the degree of anisotropic filtering continues to double there are diminishing returns in terms of visible quality with fewer and fewer rendered pixels affected, and the results become less obvious to the viewer.

When one compares the rendered results of an 8:1 anisotropically filtered scene to a 16:1 filtered scene, only a relatively few highly oblique pixels, mostly on more distant geometry, will display visibly sharper textures in the scene with the higher degree of anisotropic filtering, and the frequency information on these few 16:1 filtered pixels will only be double that of the 8:1 filter. The performance penalty also diminishes because fewer pixels require the data fetches of greater anisotropy.

In the end it is the additional hardware complexity vs. these diminishing returns, which causes an upper bound to be set on the anisotropic quality in a hardware design. Applications and users are then free to adjust this trade-off through driver and software settings up to this threshold.

Implementation

True anisotropic filtering probes the texture anisotropically on the fly on a per-pixel basis for any orientation of anisotropy.

In graphics hardware, typically when the texture is sampled anisotropically, several probes (texel samples) of the texture around the center point are taken, but on a sample pattern mapped according to the projected shape of the texture at that pixel.

Each anisotropic filtering probe is often in itself a filtered MIP map sample, which adds more sampling to the process. Sixteen trilinear anisotropic samples might require 128 samples from the stored texture, as trilinear MIP map filtering needs to take four samples times two MIP levels and then anisotropic sampling (at 16-tap) needs to take sixteen of these trilinear filtered probes.

However, this level of filtering complexity is not required all the time. There are commonly available methods to reduce the amount of work the video rendering hardware and must do.

Performance and optimization

The sample count required can make anisotropic filtering extremely bandwidth-intensive. Multiple textures are common; each texture sample could be four bytes or more, so each anisotropic pixel could require 512 bytes from texture memory, although texture compression is commonly used to reduce this.

As a video display device can easily contain over a million pixels, and as the desired frame rate can be as high as 30–60 frames per second (or more) the texture memory bandwidth can become very high very quickly. Ranges of hundreds of gigabytes per second of pipeline bandwidth for texture rendering operations is not unusual where anisotropic filtering operations are involved.

Fortunately, several factors mitigate in favor of better performance. The probes themselves share cached texture samples, both inter-pixel and intra-pixel. Even with 16-tap anisotropic filtering, not all 16 taps are always needed. This tapping simplification method works because only distant *highly oblique* pixel fills tend to be highly anisotropic.

- Such anisotropic pixel fills tends to cover small regions of the screen (ie generally under 10%).
- Texture magnification filters (as a general rule) require no anisotropic filtering.

Ambient Occlusion

Ambient occlusion is a shading method used in 3D computer graphics which helps add realism to local reflection models by taking into account attenuation of light due to occlusion. Ambient occlusion attempts to approximate the way light radiates in real life, especially off what are normally considered non-reflective surfaces.

Unlike local methods like Phong shading, ambient occlusion is a global method, meaning the illumination at each point is a function of other geometry in the scene. However, it is a very crude approximation to full global illumination. The soft appearance achieved by ambient occlusion alone is similar to the way an object appears on an overcast day.

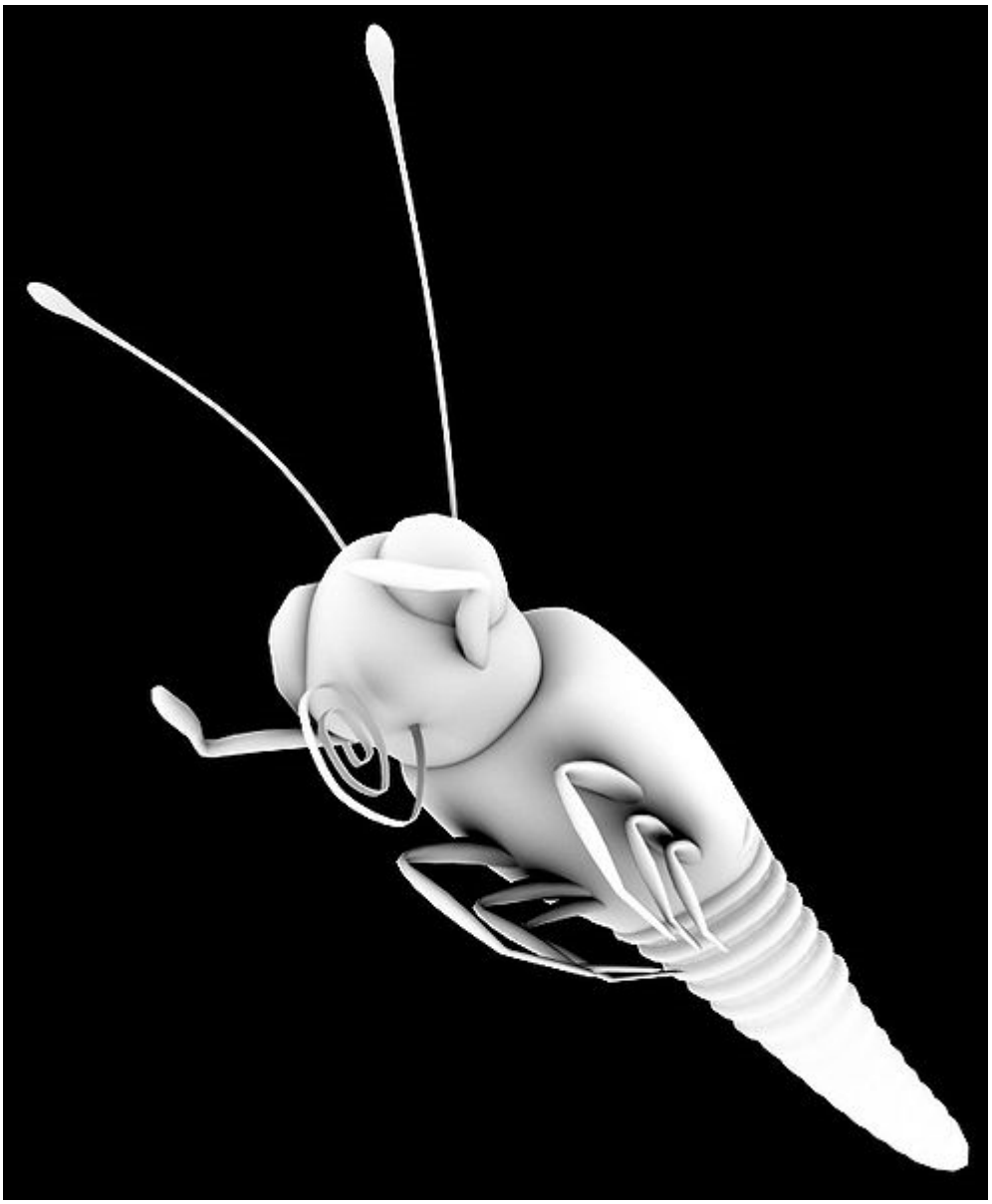
Method of implementation

Ambient occlusion is most often calculated by casting rays in every direction from the surface. Rays which reach the background or “sky” increase the brightness of the surface, whereas a ray which hits any other object contributes no illumination. As a result, points surrounded by a large amount of geometry are rendered dark, whereas points with little geometry on the visible hemisphere appear light.

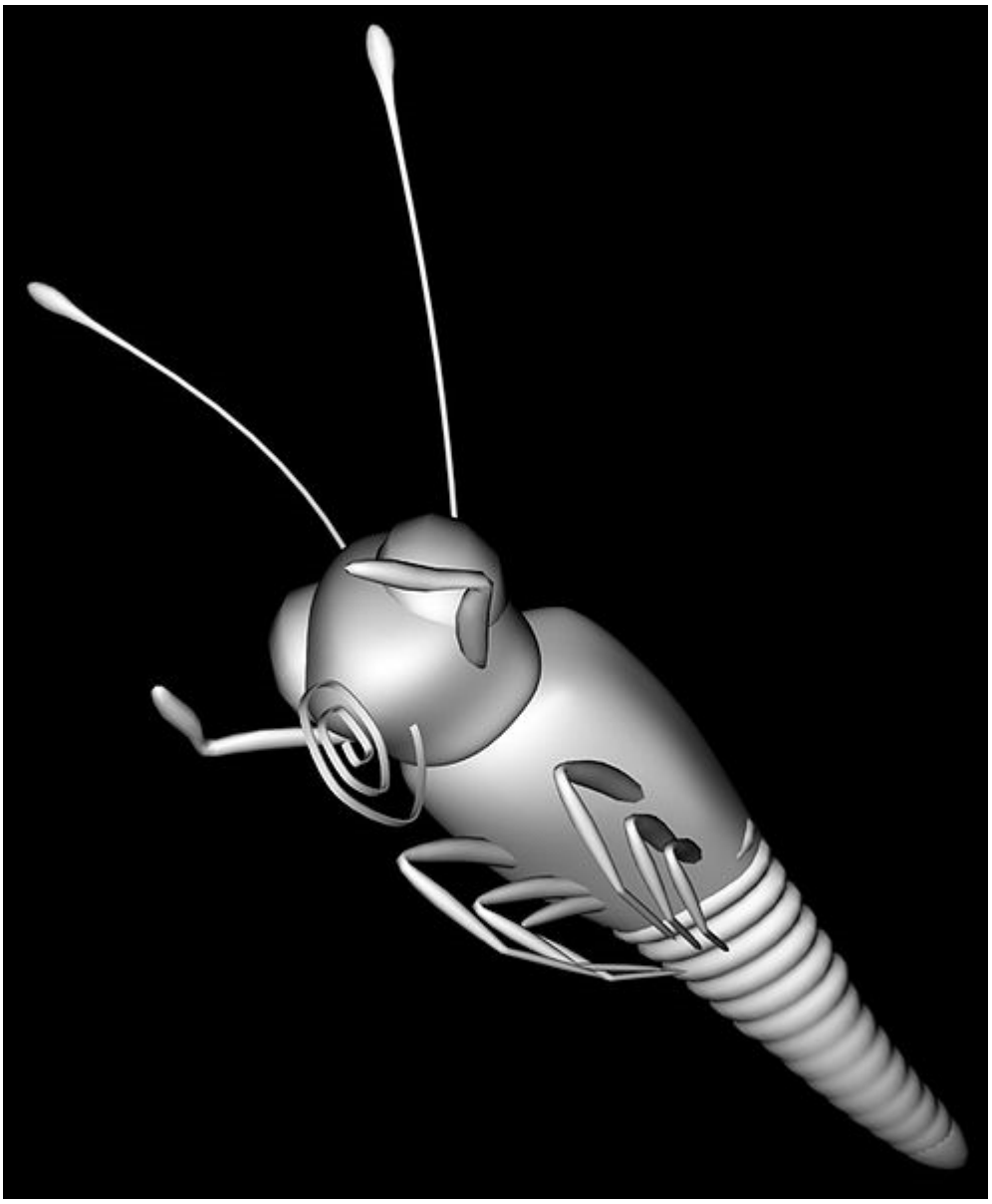
Ambient occlusion is related to accessibility shading, which determines appearance based on how easy it is for a surface to be touched by various elements (e.g., dirt, light, etc.). It has been popularized in production animation due to its relative simplicity and efficiency. In the industry, ambient occlusion is often referred to as "sky light."

The ambient occlusion shading model has the nice property of offering a better perception of the 3d shape of the displayed objects. This was shown in a paper where the

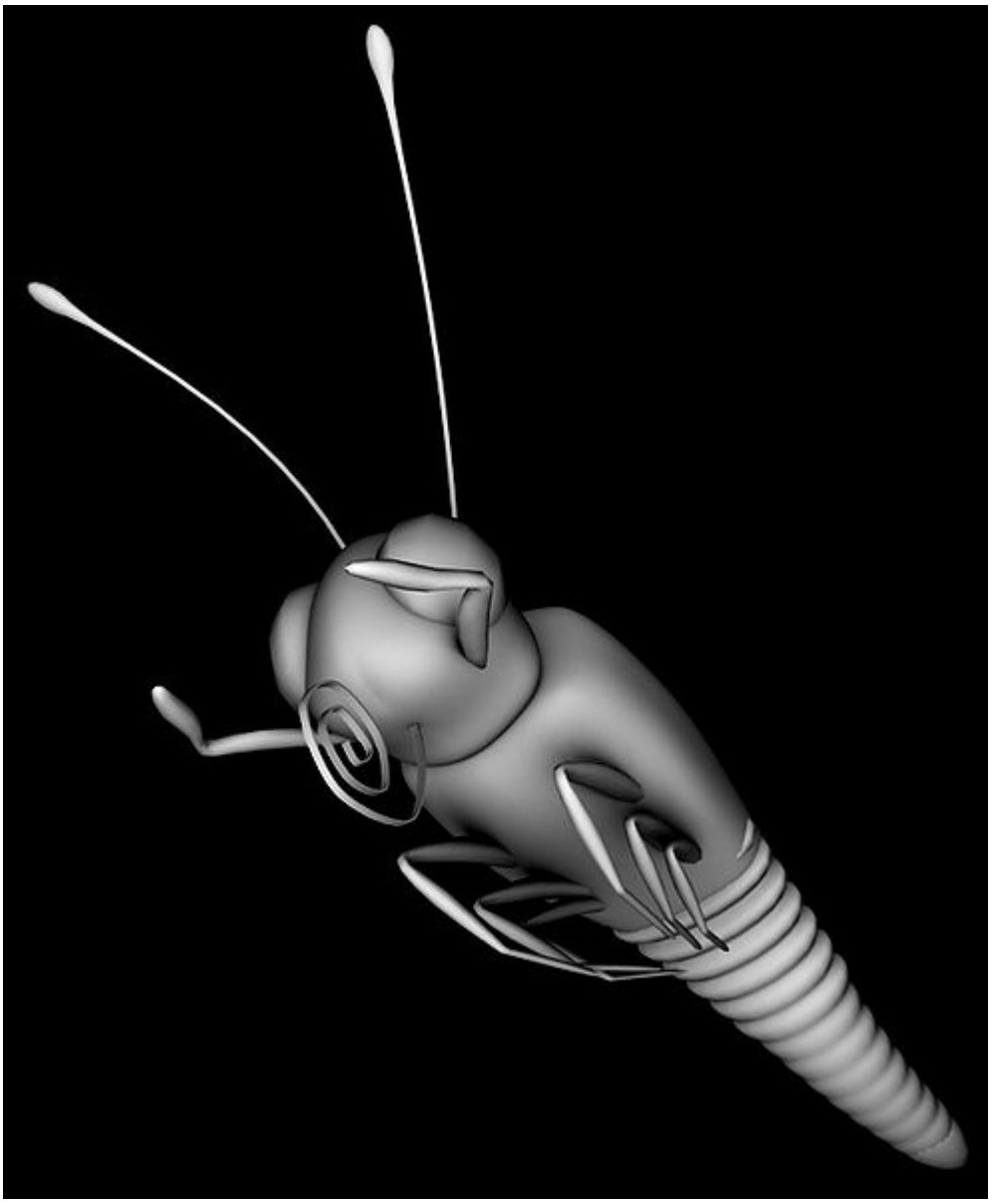
authors report the results of perceptual experiments showing that depth discrimination under diffuse uniform sky lighting is superior to that predicted by a direct lighting model.



ambient occlusion



diffuse only



combined ambient and diffuse

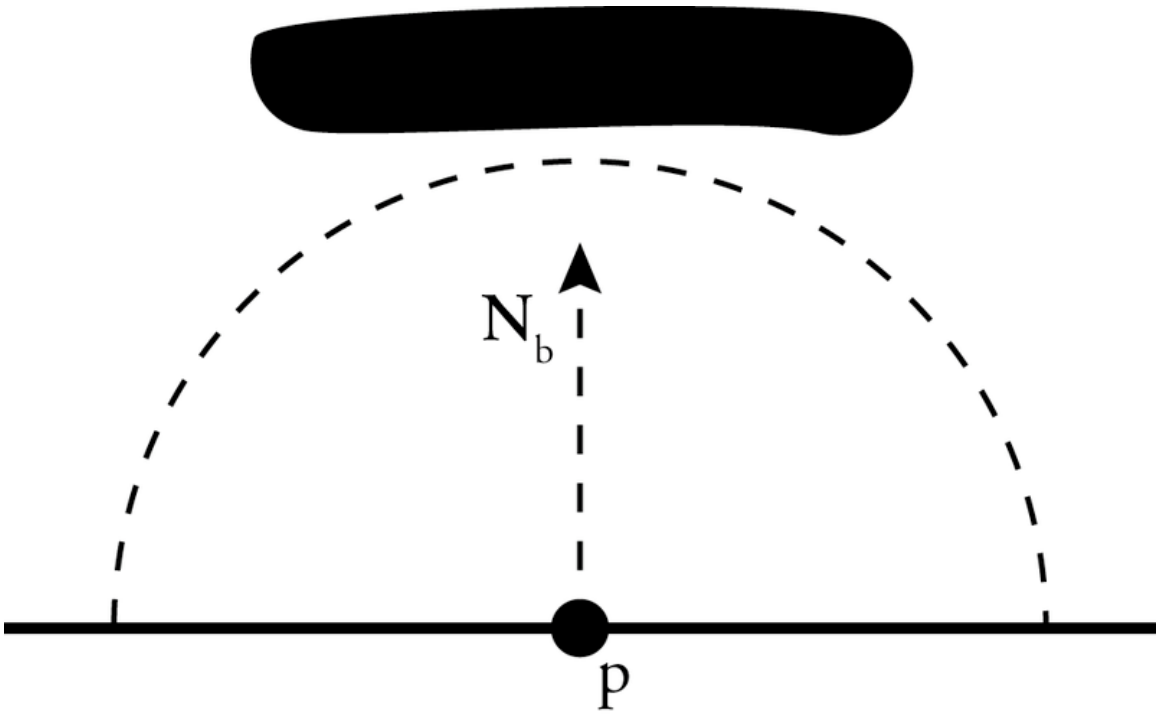
The occlusion $A_{\bar{p}}$ at a point $\bar{p}$ on a surface with normal $\hat{n}$ can be computed by integrating the visibility function over the hemisphere Ω with respect to projected solid angle:

$$A_{\bar{p}} = \frac{1}{\pi} \int_{\Omega} V_{\bar{p},\hat{\omega}} (\hat{n} \cdot \hat{\omega}) \, \mathrm{d}\omega$$

where $V_{\bar{p},\hat{\omega}}$ is the visibility function at $\bar{p}$, defined to be zero if $\bar{p}$ is occluded in the direction $\hat{\omega}$ and one otherwise, and $\mathrm{d}\omega$ is the infinitesimal solid angle step of the integration variable $\hat{\omega}$. A variety of techniques are used to approximate this integral in practice: perhaps the most straightforward way is to use the Monte Carlo method by

casting rays from the point $\bar{p}$ and testing for intersection with other scene geometry (i.e., ray casting). Another approach (more suited to hardware acceleration) is to render the view from $\bar{p}$ by rasterizing black geometry against a white background and taking the (cosine-weighted) average of rasterized fragments. This approach is an example of a "gathering" or "inside-out" approach, whereas other algorithms (such as depth-map ambient occlusion) employ "scattering" or "outside-in" techniques.

In addition to the ambient occlusion value, a "bent normal" vector $\hat{n}_b$ is often generated, which points in the average direction of unoccluded samples. The bent normal can be used to look up incident radiance from an environment map to approximate image-based lighting. However, there are some situations in which the direction of the bent normal is a misrepresentation of the dominant direction of illumination, e.g.,



In this example the bent normal N_b has an unfortunate direction, since it is pointing at an occluded surface.

In this example, light may reach the point p only from the left or right sides, but the bent normal points to the average of those two sources, which is, unfortunately, directly toward the obstruction.

Awards

In 2010, Hayden Landis, Ken McGaugh and Hilmar Koch were awarded a Scientific and Technical Academy Award for their work on ambient occlusion rendering.

Chapter 2

Binary Space Partitioning

Binary space partitioning (**BSP**) is a method for recursively subdividing a space into convex sets by hyperplanes. This subdivision gives rise to a representation of the scene by means of a tree data structure known as a **BSP tree**.

Originally, this approach was proposed in 3D computer graphics to increase the rendering efficiency by precomputing the BSP tree prior to low-level rendering operations. Some other applications include performing geometrical operations with shapes (constructive solid geometry) in CAD, collision detection in robotics and 3D computer games, and other computer applications that involve handling of complex spatial scenes.

Overview

In computer graphics it is desirable that the drawing of a scene be both correct and quick. A simple way to draw a scene is the painter's algorithm: draw it from back to front painting over the background with each closer object. However, that approach is quite limited, since time is wasted drawing objects that will be overdrawn later, and not all objects will be drawn correctly.

Z-buffering can ensure that scenes are drawn correctly and eliminate the ordering step of the painter's algorithm, but it is expensive in terms of memory use. BSP trees will split up objects so that the painter's algorithm will draw them correctly without need of a Z-buffer and eliminate the need to sort the objects; as a simple tree traversal will yield them in the correct order. It also serves as a basis for other algorithms, such as visibility lists, which attempt to reduce overdraw.

The downside is the requirement for a time consuming pre-processing of the scene, which makes it difficult and inefficient to directly implement moving objects into a BSP tree. This is often overcome by using the BSP tree together with a Z-buffer, and using the Z-buffer to correctly merge movable objects such as doors and characters onto the background scene.

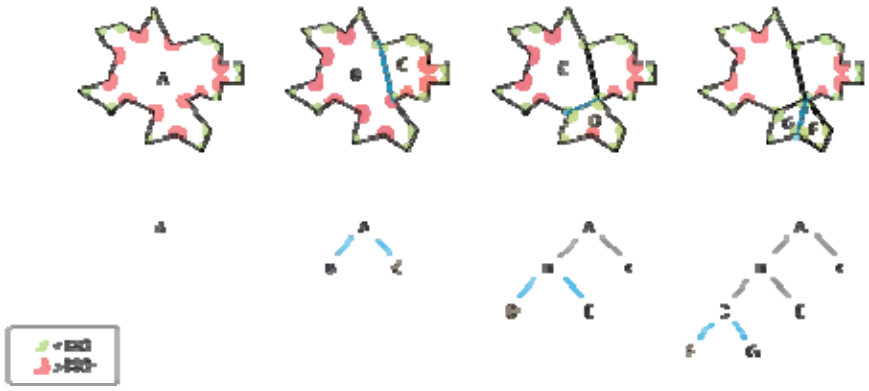
BSP trees are often used by 3D computer games, particularly first-person shooters and those with indoor environments. Probably the earliest game to use a BSP data structure was *Doom*. Other uses include ray tracing and collision detection.

Generation

Binary space partitioning is a generic process of recursively dividing a scene into two until the partitioning satisfies one or more requirements. The specific method of division varies depending on its final purpose. For instance, in a BSP tree used for collision detection, the original object would be partitioned until each part becomes simple enough to be individually tested, and in rendering it is desirable that each part be convex so that the painter's algorithm can be used.

The final number of objects will inevitably increase since lines or faces that cross the partitioning plane must be split into two, and it is also desirable that the final tree remains reasonably balanced. Therefore the algorithm for correctly and efficiently creating a good BSP tree is the most difficult part of an implementation. In 3D space, planes are used to partition and split an object's faces; in 2D space lines split an object's segments.

The following picture illustrates the process of partitioning an irregular polygon into a series of convex ones. Notice how each step produces polygons with fewer segments until arriving at G and F, which are convex and require no further partitioning. In this particular case, the partitioning line was picked between existing vertices of the polygon and intersected none of its segments. If the partitioning line intersects a segment, or face in a 3D model, the offending segment(s) or face(s) have to be split into two at the line/plane because each resulting partition must be a full, independent object.



1. A is the root of the tree and the entire polygon
2. A is split into B and C
3. B is split into D and E.
4. D is split into F and G, which are convex and hence become leaves on the tree.

Since the usefulness of a BSP tree depends upon how well it was generated, a good algorithm is essential. Most algorithms will test many possibilities for each partition until they find a good compromise. They might also keep backtracking information in memory, so that if a branch of the tree is found to be unsatisfactory, other alternative partitions may be tried. Thus producing a tree usually requires long computations.

BSP trees are also used to represent natural images. Construction methods for BSP trees representing images were first introduced as efficient representations in which only a few hundred nodes can represent an image that normally requires hundreds of thousands of pixels. Fast algorithms have also been developed to construct BSP trees of images using computer vision and signal processing algorithms. These algorithms, in conjunction with advanced entropy coding and signal approximation approaches, were used to develop image compression methods.

Rendering a scene with visibility information from the BSP tree

BSP trees are used to improve rendering performance in calculating visible triangles for the painter's algorithm for instance. The tree can be traversed in linear time from an arbitrary viewpoint.

Since a painter's algorithm works by drawing polygons farthest from the eye first, the following code recurses to the bottom of the tree and draws the polygons. As the recursion unwinds, polygons closer to the eye are drawn over far polygons. Because the BSP tree already splits polygons into trivial pieces, the hardest part of the painter's algorithm is already solved - code for back to front tree traversal.

```
traverse_tree(bsp_tree* tree, point eye)
{
  location = tree->find_location(eye);

  if(tree->empty())
    return;

  if(location > 0)      // if eye in front of location
  {
    traverse_tree(tree->back, eye);
    display(tree->polygon_list);
    traverse_tree(tree->front, eye);
  }
  else if(location < 0) // eye behind location
  {
    traverse_tree(tree->front, eye);
    display(tree->polygon_list);
    traverse_tree(tree->back, eye);
  }
  else                  // eye coincidental with partition hyperplane
  {
    traverse_tree(tree->front, eye);
    traverse_tree(tree->back, eye);
  }
}
```

Other space partitioning structures

BSP trees divide a region of space into two subregions at each node. They are related to quadtrees and octrees, which divide each region into four or eight subregions, respectively.

Relationship Table

Name	***p***	***s***
Binary Space Partition	1	2
Quadtree	2	4
Octree	3	8

where *p* is the number of dividing planes used, and *s* is the number of subregions formed.

BSP trees can be used in spaces with any number of dimensions, but quadtrees and octrees are most useful in subdividing 2- and 3-dimensional spaces, respectively. Another kind of tree that behaves somewhat like a quadtree or octree, but is useful in any number of dimensions, is the *k*d-tree.

Timeline

- 1969 Schumacker et al. published a report that described how carefully positioned planes in a virtual environment could be used to accelerate polygon ordering. The technique made use of depth coherence, which states that a polygon on the far side of the plane cannot, in any way, obstruct a closer polygon. This was used in flight simulators made by GE as well as Evans and Sutherland. However, creation of the polygonal data organization was performed manually by scene designer.

- 1980 Fuchs et al. [FUCH80] extended Schumacker’s idea to the representation of 3D objects in a virtual environment by using planes that lie coincident with polygons to recursively partition the 3D space. This provided a fully automated and algorithmic generation of a hierarchical polygonal data structure known as a Binary Space Partitioning Tree (BSP Tree). The process took place as an off-line preprocessing step that was performed once per environment/object. At run-time, the view-dependent visibility ordering was generated by traversing the tree.

- 1981 Naylor's Ph.D thesis containing a full development of both BSP trees and a graph-theoretic approach using strongly connected components for pre-computing visibility, as well as the connection between the two methods. BSP trees as a dimension independent spatial search structure was emphasized, with applications to visible surface determination. The thesis also included the first empirical data demonstrating that the size of the tree and the number of new polygons was reasonable (using a model of the Space Shuttle).

- 1983 Fuchs et al. describe a micro-code implementation of the BSP tree algorithm on an Ikonas frame buffer system. This was the first demonstration of real-time visible surface determination using BSP trees.

- 1987 Thibault and Naylor described how arbitrary polyhedra may be represented using a BSP tree as opposed to the traditional b-rep (boundary representation). This provided a solid representation vs. a surface based-representation. Set operations on polyhedra were described using a tool, enabling Constructive Solid Geometry (CSG) in real-time. This was the fore runner of BSP level design using brushes, introduced in the Quake editor and picked up in the Unreal Editor.

- 1990 Naylor, Amanatides, and Thibault provide an algorithm for merging two bsp trees to form a new bsp tree from the two original trees. This provides many benefits including: combining moving objects represented by BSP trees with a static environment (also represented by a BSP tree), very efficient CSG operations on polyhedra, exact collisions detection in O(log n * log n), and proper ordering of transparent surfaces contained in two interpenetrating objects (has been used for an x-ray vision effect).

- 1990 Teller and Séquin proposed the offline generation of potentially visible sets to accelerate visible surface determination in orthogonal 2D environments.

- 1991 Gordon and Chen [CHEN91] described an efficient method of performing front-to-back rendering from a BSP tree, rather than the traditional back-to-front approach. They utilised a special data structure to record, efficiently, parts of the screen that have been drawn, and those yet to be rendered. This algorithm, together with the description of BSP Trees in the standard computer graphics textbook of the day (Foley, Van Dam, Feiner and Hughes) was used by John Carmack in the making of *Doom*.

- 1992 Teller's PhD thesis described the efficient generation of potentially visible sets as a pre-processing step to acceleration real-time visible surface determination in arbitrary 3D polygonal environments. This was used in *Quake* and contributed significantly to that game's performance.

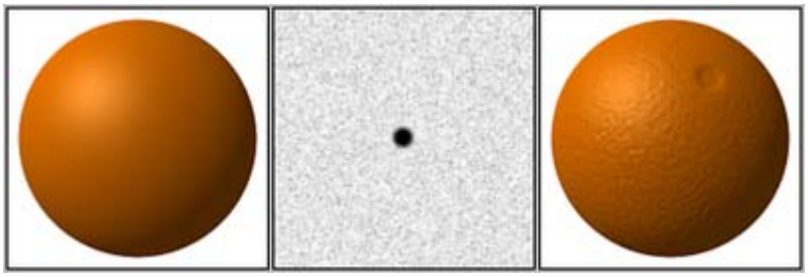
- 1993 Naylor answers the question of what characterizes a good BSP tree. He used expected case models (rather than worst case analysis) to mathematically measure the expected cost of searching a tree and used this measure to build good BSP trees. Intuitively, the tree represents an object in a multi-resolution fashion (more exactly, as a tree of approximations). Parallels with Huffman codes and probabilistic binary search trees are drawn.

- 1993 Hayder Radha's PhD thesis described (natural) image representation methods using BSP trees. This includes the development of an optimal BSP-tree construction framework for any arbitrary input image. This framework is based on a new image transform, known as the Least-Square-Error (LSE) Partitioning Line

(LPE) transform. H. Radha' thesis also developed an optimal rate-distortion (RD) image compression framework and image manipulation approaches using BSP trees.

Chapter 3

Bump Mapping

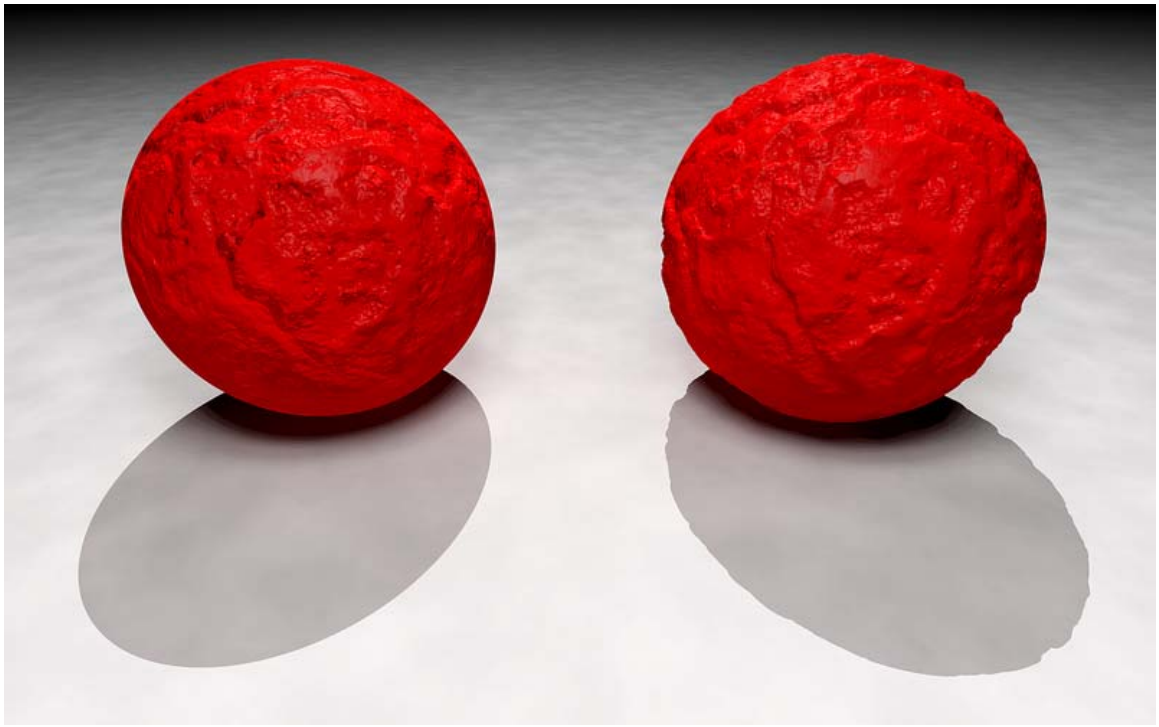


A sphere without bump mapping (left). A bump map to be applied to the sphere (middle). The sphere with the bump map applied (right) appears to have a mottled surface resembling an orange. Bump maps achieve this effect by changing how an illuminated surface reacts to light without actually modifying the size or shape of the surface

Bump mapping is a technique in computer graphics for simulating bumps and wrinkles on the surface of an object. This is achieved by perturbing the surface normals of the object and using the perturbed normal during illumination calculations. The result is an apparently bumpy surface rather than a perfectly smooth surface although the surface of the underlying object is not actually changed. Bump mapping was introduced by Blinn in 1978.

Normal and parallax mapping are the most commonly used ways of making bumps, using new techniques that makes bump mapping using a greyscale obsolete.

Bump mapping basics



Bump mapping is limited in that it does not actually modify the shape of the underlying object. On the left, a mathematical function defining a bump map simulates a crumbling surface on a sphere, but the object's outline and shadow remain those of a perfect sphere. On the right, the same function is used to modify the surface of a sphere by generating an isosurface. This actually models a sphere with a bumpy surface with the result that both its outline and its shadow are rendered realistically.

Bump mapping is a technique in computer graphics to make a rendered surface look more realistic by modeling the interaction of a bumpy surface texture with lights in the environment. Bump mapping does this by changing the brightness of the pixels on the surface in response to a heightmap that is specified for each surface.

When rendering a 3D scene, the brightness and color of the pixels are determined by the interaction of a 3D model with lights in the scene. After it is determined that an object is visible, trigonometry is used to calculate the "geometric" surface normal of the object, defined as a vector at each pixel position on the object.

The geometric surface normal then defines how strongly the object interacts with light coming from a given direction using Phong shading or a similar lighting algorithm. Light traveling perpendicular to a surface interacts more strongly than light that is more parallel to the surface. After the initial geometry calculations, a colored texture is often applied to the model to make the object appear more realistic.

After texturing, a calculation is performed for each pixel on the object's surface:

1. Look up the position on the heightmap that corresponds to the position on the surface.
2. Calculate the surface normal of the heightmap.
3. Add the surface normal from step two to the geometric surface normal so that the normal points in a new direction.
4. Calculate the interaction of the new "bumpy" surface with lights in the scene using, for example, the Phong shading.

The result is a surface that appears to have real depth. The algorithm also ensures that the surface appearance changes as lights in the scene are moved around. Normal mapping is the most commonly used bump mapping technique, but there are other alternatives, such as parallax mapping.

A limitation with bump mapping is that it perturbs only the surface normals without changing the underlying surface itself. Silhouettes and shadows therefore remain unaffected. This limitation can be overcome by techniques including the displacement mapping where bumps are actually applied to the surface or using an isosurface.

For the purposes of rendering in real-time, bump mapping is often referred to as a "pass", as in *multi-pass rendering*, and can be implemented as multiple passes (often three or four) to reduce the number of trigonometric calculations that are required.

Realtime bump mapping techniques

3D graphics programmers sometimes use a lower quality, faster bump mapping technique in order to simulate bump mapping. One such method uses texel index alteration instead of altering surface normals. As of GeForce 2 class cards this technique is implemented in graphics accelerator hardware.

Full-screen bump mapping, which could be easily implemented with a very simple and fast rendering loop, was a common visual effect when bump-mapping was first introduced.

Emboss bump mapping

This technique uses texture maps to generate bump mapping effects without requiring a custom renderer. This multi-pass algorithm is an extension and refinement of texture embossing. This process duplicates the first texture image, shifts it over to the desired amount of bump, darkens the texture underneath, cuts out the appropriate shape from the texture on top, and blends the two textures into one. This is called two-pass emboss bump mapping because it requires two textures.

It is simple to implement and requires no custom hardware, and is therefore limited by the speed of the CPU. However, it only affects diffuse lighting, and the illusion is broken depending on the angle of the light.

Environment mapped bump mapping

Matrox G400 Tech Demo with EMBM

The Matrox G400 chip supports a texture-based surface detailing method called **Environment Mapped Bump Mapping** (EMBM). It was originally developed by BitBoys Oy and licensed to Matrox. EMBM was first introduced in DirectX 6.0.

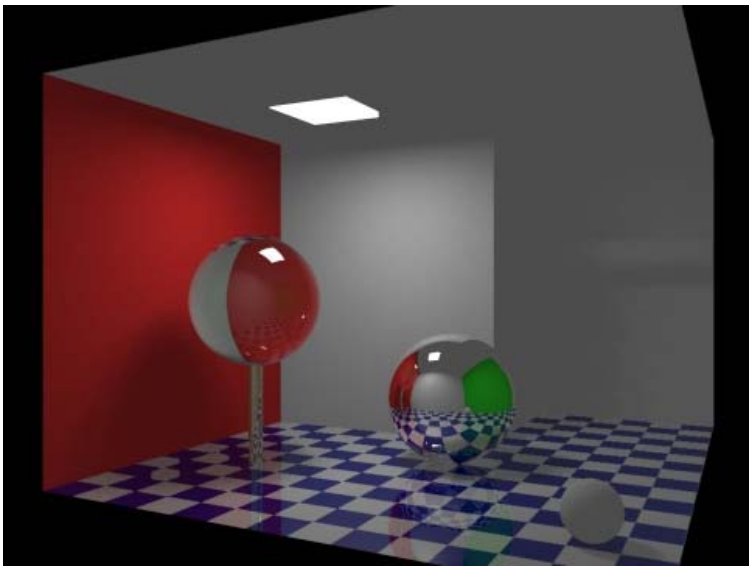
The Radeon 7200 also includes hardware support for EMBM, which was demonstrated in the technical demonstration "Radeon's Ark". However, EMBM was not supported by other graphics chips, such as NVIDIA's GeForce 256 through to the GeForce 2, which only supported the simpler Dot-3 BM. Due to this lack of industry-wide support, and its toll on the limited graphics hardware of the time, EMBM only saw limited use during G400's time. Only a few games supported the feature, such as Dungeon Keeper 2 and Millennium Soldier: Expendable.

EMBM initially required specialized hardware within the chip for its calculations, such as the Matrox G400 or Radeon 7200. It could also be rendered by the programmable pixel shaders of later DirectX 8.0 accelerators like the GeForce 3 and Radeon 8500.

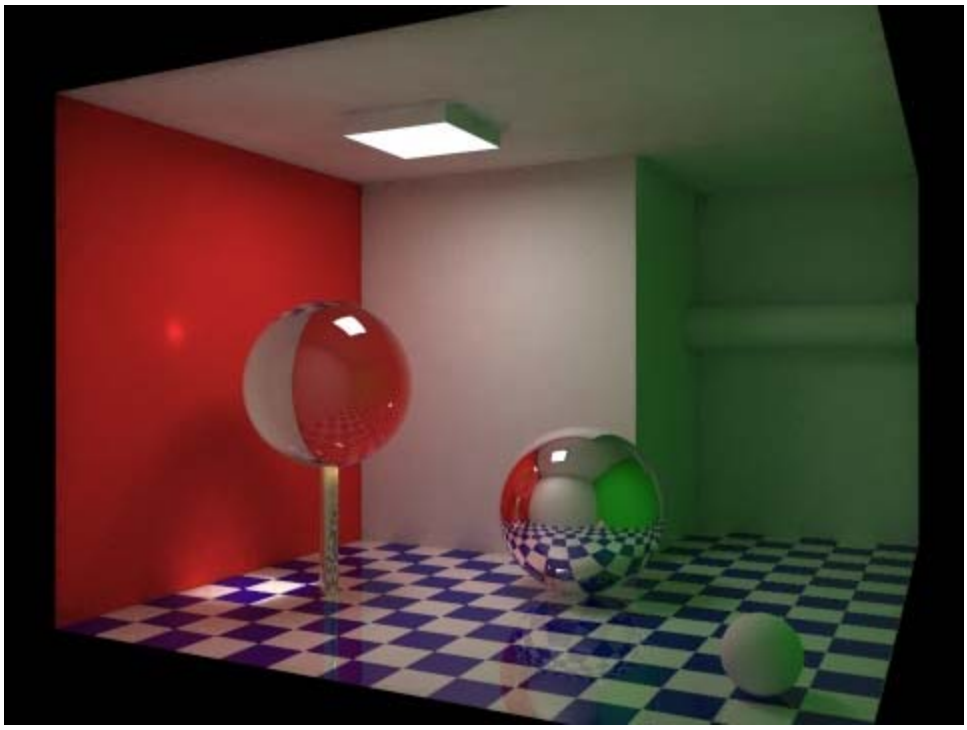
Chapter 4

Global Illumination & Catmull–Clark Subdivision Surface

Global Illumination



Rendering without global illumination. Areas that lie outside of the ceiling lamp's direct light lack definition. For example, the lamp's housing appears completely uniform. Without the ambient light added into the render, it would appear uniformly black.



Rendering with global illumination. Light is reflected by surfaces, and colored light transfers from one surface to another. Notice how color from the red wall and green wall (not visible) reflects onto other surfaces in the scene. Also notable is the caustic projected onto the red wall from light passing through the glass sphere.

Global illumination is a general name for a group of algorithms used in 3D computer graphics that are meant to add more realistic lighting to 3D scenes. Such algorithms take into account not only the light which comes directly from a light source (*direct illumination*), but also subsequent cases in which light rays from the same source are reflected by other surfaces in the scene, whether reflective or non (*indirect illumination*).

Theoretically reflections, refractions, and shadows are all examples of global illumination, because when simulating them, one object affects the rendering of another object (as opposed to an object being affected only by a direct light). In practice, however, only the simulation of diffuse inter-reflection or caustics is called global illumination.

Images rendered using global illumination algorithms often appear more photorealistic than images rendered using only direct illumination algorithms. However, such images are computationally more expensive and consequently much slower to generate. One common approach is to compute the global illumination of a scene and store that information with the geometry, i.e., radiosity. That stored data can then be used to generate images from different viewpoints for generating walkthroughs of a scene without having to go through expensive lighting calculations repeatedly.

Radiosity, ray tracing, beam tracing, cone tracing, path tracing, Metropolis light transport, ambient occlusion, photon mapping, and image based lighting are examples of algorithms used in global illumination, some of which may be used together to yield results that are not fast, but accurate.

These algorithms model diffuse inter-reflection which is a very important part of global illumination; however most of these (excluding radiosity) also model specular reflection, which makes them more accurate algorithms to solve the lighting equation and provide a more realistically illuminated scene.

The algorithms used to calculate the distribution of light energy between surfaces of a scene are closely related to heat transfer simulations performed using finite-element methods in engineering design.

In real-time 3D graphics, the diffuse inter-reflection component of global illumination is sometimes approximated by an "ambient" term in the lighting equation, which is also called "ambient lighting" or "ambient color" in 3D software packages. Though this method of approximation (also known as a "cheat" because it's not really a global illumination method) is easy to perform computationally, when used alone it does not provide an adequately realistic effect. Ambient lighting is known to "flatten" shadows in 3D scenes, making the overall visual effect more bland. However, used properly, ambient lighting can be an efficient way to make up for a lack of processing power.

Procedure

For the simulation of global illumination are used in 3D programs, more and more specialized algorithms that can effectively simulate the global illumination. These are, for example, path tracing or photon mapping, under certain conditions, including radiosity. These are always methods to try to solve the rendering equation.

The following approaches can be distinguished here:

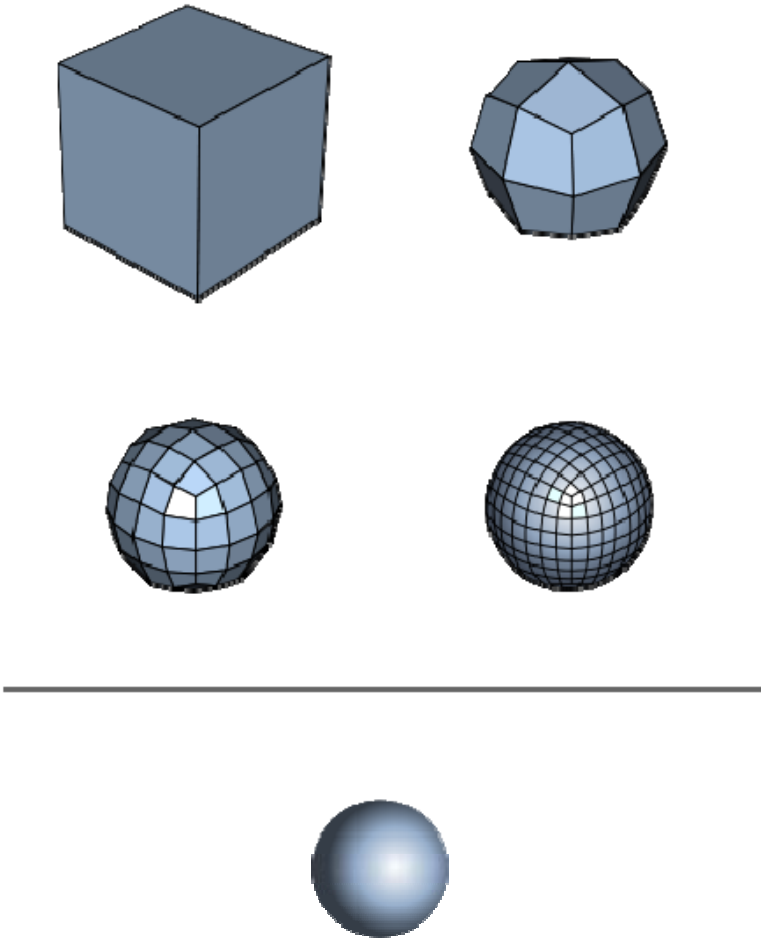
- Inversion: $L = (1 - T)^{-1} L^e$
  - is not applied in practice
- Expansion: $$L = \sum_{i=0}^{\infty} T^i L^e$$
  - bi-directional approach: Photon Mapping + Distributed ray tracing, Bi-directional path tracing, Metropolis light transport
- Iteration: $L_n tl_e + = L^{(n-1)}$
  - Radiosity

In Light path notation global lighting the paths of the type L (D | S) corresponds * E.

Image-based lighting

Another way to simulate real global illumination, is the use of High dynamic range images (HDRIs), also known as environment maps, which encircle the scene, and they illuminate. This process is known as image-based lighting.

Catmull–Clark Subdivision Surface



First three steps of Catmull–Clark subdivision of a cube with subdivision surface below

The **Catmull–Clark** algorithm is used in computer graphics to create smooth surfaces by subdivision surface modeling. It was devised by Edwin Catmull and Jim Clark in 1978 as a generalization of bi-cubic uniform B-spline surfaces to arbitrary topology. In 2005, Edwin Catmull received an Academy Award for Technical Achievement together with Tony DeRose and Jos Stam for their invention and application of subdivision surfaces.

Recursive evaluation

Catmull–Clark surfaces are defined recursively, using the following refinement scheme:

Start with a mesh of an arbitrary polyhedron. All the vertices in this mesh shall be called original points.

- For each face, add a *face point*
  - Set each face point to be the *centroid of all original points for the respective face*.
- For each edge, add an *edge point*.
  - Set each edge point to be the *average of the two neighbouring face points and its two original endpoints*.
- For each *face point*, add an edge for every edge of the face, connecting the *face point* to each *edge point* for the face.
- For each original point *P*, take the average *F* of all *n* face points for faces touching *P*, and take the average *R* of all *n* edge midpoints for edges touching *P*, where each edge midpoint is the average of its two endpoint vertices. *Move each original point* to the point

$$\frac{F + 2R + (n-3)P}{n}$$

(This is the barycenter of P, R and F with respective weights (n-3), 2 and 1. This arbitrary-looking formula was chosen by Catmull and Clark based on the aesthetic appearance of the resulting surfaces rather than on a mathematical derivation.)

The new mesh will consist only of quadrilaterals, which won't in general be planar. The new mesh will generally look smoother than the old mesh.

Repeated subdivision results in smoother meshes. It can be shown that the limit surface obtained by this refinement process is at least $\mathbf{C^1}$ at extraordinary vertices and $\mathbf{C^2}$ everywhere else (when n indicates how many derivatives are continuous, we speak of $\mathbf{C^n}$ continuity). After one iteration, the number of extraordinary points on the surface remains constant.

Exact evaluation

The limit surface of Catmull–Clark subdivision surfaces can also be evaluated directly, without any recursive refinement. This can be accomplished by means of the technique of Jos Stam . This method reformulates the recursive refinement process into a matrix exponential problem, which can be solved directly by means of matrix diagonalization.

Chapter 5

Level of Detail

In computer graphics, accounting for **level of detail** involves decreasing the complexity of a 3D object representation as it moves away from the viewer or according other metrics such as object importance, eye-space speed or position. Level of detail techniques increases the efficiency of rendering by decreasing the workload on graphics pipeline stages, usually vertex transformations. The reduced visual quality of the model is often unnoticed because of the small effect on object appearance when distant or moving fast.

Although most of the time LOD is applied to geometry detail only, the basic concept can be generalized. Recently, LOD techniques included also shader management to keep control of pixel complexity. A form of level of detail management has been applied to textures for years, under the name of mipmapping, also providing higher rendering quality.

It is commonplace to say that "an object has been *LOD'd*" when the object is simplified by the underlying *LODding algorithm*.

Historical reference

The origin of all the LoD algorithms for 3D computer graphics can be traced back to an article by James H. Clark in the October 1976 issue of *Communications of the ACM*. At the time, computers were monolithic and rare, and graphics was being driven by researchers. The hardware itself was completely different, both architecturally and performance-wise. As such, many differences could be observed with regard to today's algorithms but also many common points.

The original algorithm presented a much more generic approach to what will be discussed here. After introducing some available algorithms for geometry management, it is stated that most fruitful gains came from *"...structuring the environments being rendered"*, allowing to exploit faster transformations and clipping operations.

The same environment structuring is now proposed as a way to control varying detail thus avoiding unnecessary computations, yet delivering adequate visual quality:

> For example, a dodecahedron looks like a sphere from a sufficiently large distance and thus can be used to model it so long as it is viewed from that or a greater distance. However, if it must ever be viewed more closely, it will look like a dodecahedron. One solution to this is simply to define it with the most detail that will ever be necessary. However, then it might have far more detail than is needed to represent it at large distances, and in a complex environment with many such objects, there would be too many polygons (or other geometric primitives) for the visible surface algorithms to efficiently handle.

The proposed algorithm envisions a tree data structure which encodes in its arcs both transformations and transitions to more detailed objects. In this way, each node encodes an object and according to a fast heuristic, the tree is descended to the leafs which provide each object with more detail. When a leaf is reached, other methods could be used when higher detail is needed, such as Catmull's recursive subdivision.

> The significant point, however, is that in a complex environment, the amount of information presented about the various objects in the environment varies according to the fraction of the field of view occupied by those objects.

The paper then introduces clipping (not to be confused with culling (computer graphics), although often similar), various considerations on the *graphical working set* and its impact on performance, interactions between the proposed algorithm and others to improve rendering speed. Interested readers are encouraged in checking the references for further details on the topic.

Well known approaches

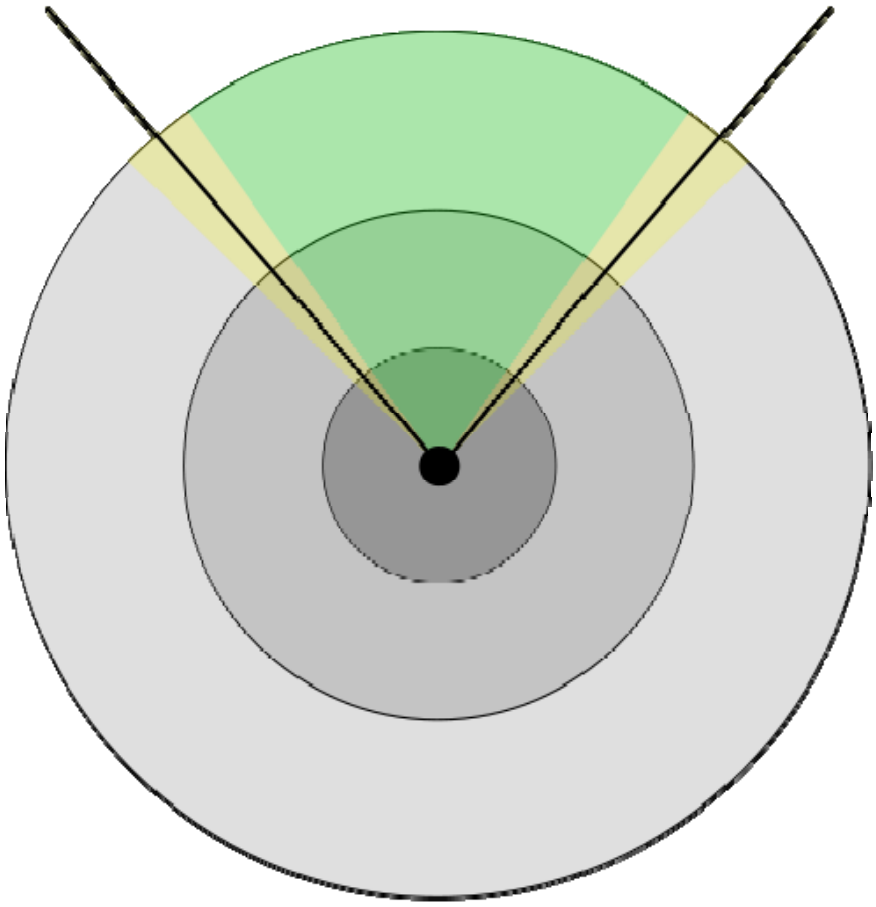
Although the algorithm introduced above covers a whole range of level of detail management techniques, real world applications usually employ different methods according the information being rendered. Because of the appearance of the considered objects, two main algorithm families are used.

The first is based on subdividing the space in a finite amount of regions, each with a certain level of detail. The result is discrete amount of detail levels, from which the name *Discrete LoD* (DLOD). There's no way to support a smooth transition between LOD levels at this level, although alpha blending or morphing can be used to avoid visual popping.

The latter considers the polygon mesh being rendered as a function which must be evaluated requiring to avoid excessive errors which are a function of some heuristic (usually distance) themselves. The given "mesh" function is then continuously evaluated

and an optimized version is produced according to a tradeoff between visual quality and performance. Those kind of algorithms are usually referred as *Continuous LOD* (CLOD).

Details on Discrete LOD



An example of various DLOD ranges. Darker areas are meant to be rendered with higher detail. An additional culling operation is run, discarding all the information outside the frustum (colored areas).

The basic concept of discrete LOD (DLOD) is to provide various models to represent the same object. Obtaining those models requires an external algorithm which is often non-trivial and subject of many polygon reduction techniques. Successive LODding algorithms will simply assume those models are available.

DLOD algorithms are often used in performance-intensive applications with small data sets which can easily fit in memory. Although out of core algorithms could be used, the information granularity is not well suited to this kind of application. This kind of

algorithm is usually easier to get working, providing both faster performance and lower CPU usage because of the few operations involved.

DLOD methods are often used for "stand-alone" moving objects, possibly including complex animation methods. A different approach is used for geomipmapping , a popular terrain rendering algorithm because this applies to terrain meshes which are both graphically and topologically different from "object" meshes. Instead of computing an error and simplify the mesh according to this, geomipmapping takes a fixed reduction method, evaluates the error introduced and computes a distance at which the error is acceptable. Although straightforward, the algorithm provides decent performance.

A discrete LOD example

As a simple example, consider the following sphere. A discrete LOD approach would cache a certain number of models to be used at different distances. Because the model can trivially be procedurally generated by its mathematical formulation, using a different amount of sample points distributed on the surface is sufficient to generate the various models required. This pass is not a LODding algorithm.

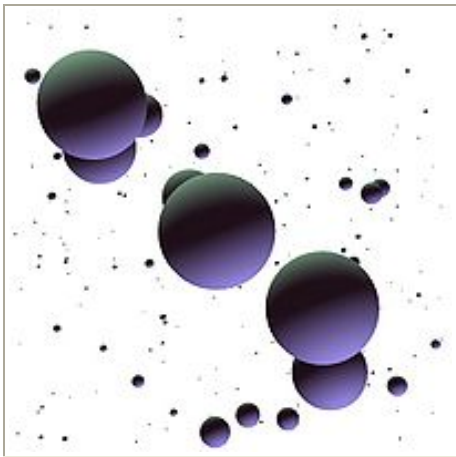
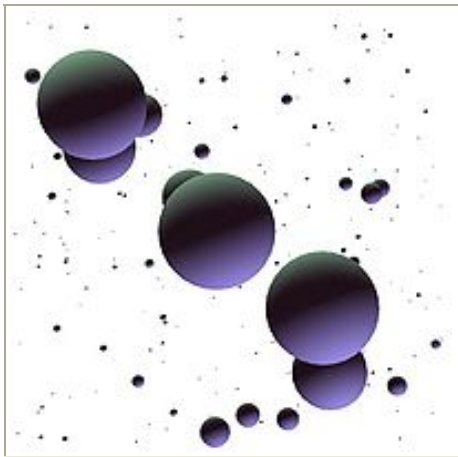
Visual impact comparisons and measurements

Image					
Vertices	~5500	~2880	~1580	~670	140
Notes	Maximum detail, for closeups.				Minimum detail, very far objects.

To simulate a realistic transform bound scenario, we'll use an ad-hoc written application. We'll make sure we're not CPU bound by using simple algorithms and minimum fragment operations. Each frame, the program will compute each sphere's distance and choose a model from a pool according to this information. To easily show the concept, the distance at which each model is used is hard coded in the source. A more involved method would compute adequate models according to the usage distance chosen.

We use OpenGL for rendering because its high efficiency in managing small batches, storing each model in a display list thus avoiding communication overheads. Additional vertex load is given by applying two directional light sources ideally located infinitely far away.

The following table compares the performance of LoD aware rendering and a full detail (*brute force*) method.

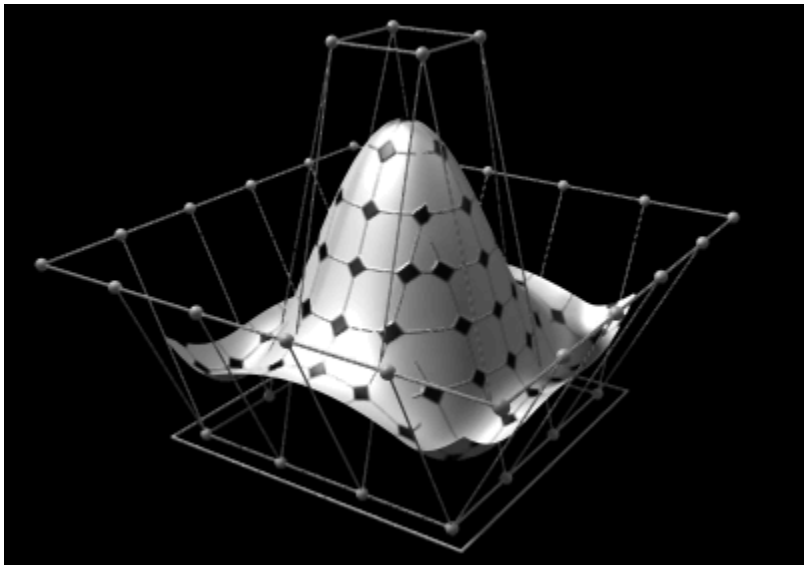
Visual impact comparisons and measurements		
	Brute	**DLOD**
Rendered images		
Render time	27.27 ms	1.29 ms
Scene vertices (thousands)	2328.48	109.44

Hierarchical LOD

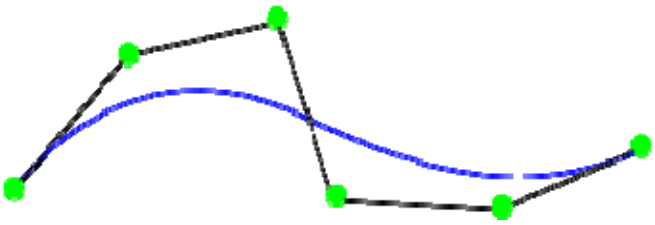
Because hardware is geared towards large amounts of detail, rendering low polygon objects may score sub-optimal performances. HLOD avoids the problem by grouping different objects together. This allows for higher efficiency as well as taking advantage of proximity considerations.

Chapter 6

Non-Uniform Rational B-Spline



Three-dimensional NURBS surfaces can have complex, organic shapes. Control points influence the directions the surface takes. The outermost square below delineates the X/Y extents of the surface.



A NURBS curve.

Non-uniform rational basis spline (**NURBS**) is a mathematical model commonly used in computer graphics for generating and representing curves and surfaces which offers great flexibility and precision for handling both analytic and freeform shapes.

History

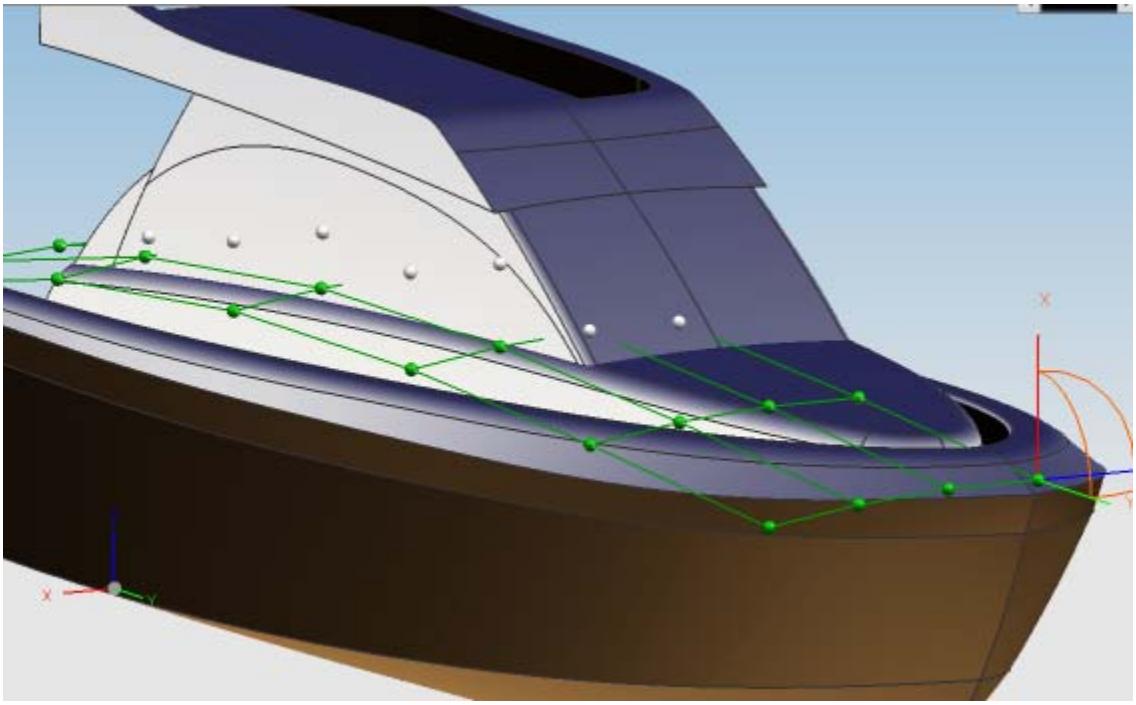
Development of NURBS began in the 1950s by engineers who were in need of a mathematically precise representation of freeform surfaces like those used for ship hulls, aerospace exterior surfaces, and car bodies, which could be exactly reproduced whenever technically needed. Prior representations of this kind of surface only existed as a single physical model created by a designer.

The pioneers of this development were Pierre Bézier who worked as an engineer at Renault, and Paul de Casteljau who worked at Citroën, both in France. Bézier worked nearly parallel to de Casteljau, neither knowing about the work of the other. But because Bézier published the results of his work, the average computer graphics user today recognizes splines — which are represented with control points lying off the curve itself — as Bézier splines, while de Casteljau's name is only known and used for the algorithms he developed to evaluate parametric surfaces. In the 1960s it became clear that non-uniform, rational B-splines are a generalization of Bézier splines, which can be regarded as uniform, non-rational B-splines.

At first NURBS were only used in the proprietary CAD packages of car companies. Later they became part of standard computer graphics packages.

Real-time, interactive rendering of NURBS curves and surfaces was first made available on Silicon Graphics workstations in 1989. In 1993, the first interactive NURBS modeller for PCs, called NöRBS, was developed by CAS Berlin, a small startup company cooperating with the Technical University of Berlin. Today most professional computer graphics applications available for desktop use offer NURBS technology, which is most often realized by integrating a NURBS engine from a specialized company.

Use



NURBS are commonly used in computer-aided design (CAD), manufacturing (CAM), and engineering (CAE) and are part of numerous industry wide used standards, such as IGES, STEP, ACIS, and PHIGS. NURBS tools are also found in various 3D modeling and animation software packages, such as form•Z, Blender, Maya, Rhino3D, Cinema 4D, Cobalt, Shark FX, and Solid Modeling Solutions. Other than this there are specialized NURBS modeling software packages such as Autodesk Alias Surface, solidThinking and ICEM Surf.

They allow representation of geometrical shapes in a compact form. They can be efficiently handled by the computer programs and yet allow for easy human interaction. NURBS surfaces are functions of two parameters mapping to a surface in three-dimensional space. The shape of the surface is determined by control points.

In general, editing NURBS curves and surfaces is highly intuitive and predictable. Control points are always either connected directly to the curve/surface, or act as if they were connected by a rubber band. Depending on the type of user interface, editing can be realized via an element's control points, which are most obvious and common for Bézier curves, or via higher level tools such as spline modeling or hierarchical editing.

A surface under construction, e.g. the hull of a motor yacht, is usually composed of several NURBS surfaces known as *patches*. These patches should be fitted together in such a way that the boundaries are invisible. This is mathematically expressed by the concept of geometric continuity.

Higher-level tools exist which benefit from the ability of NURBS to create and establish geometric continuity of different levels:

Positional continuity (G0)
: holds whenever the end positions of two curves or surfaces are coincidental. The curves or surfaces may still meet at an angle, giving rise to a sharp corner or edge and causing broken highlights.

Tangential continuity (G1)
: requires the end vectors of the curves or surfaces to be parallel, ruling out sharp edges. Because highlights falling on a tangentially continuous edge are always continuous and thus look natural, this level of continuity can often be sufficient.

Curvature continuity (G2)
: further requires the end vectors to be of the same length and rate of length change. Highlights falling on a curvature-continuous edge do not display any change, causing the two surfaces to appear as one. This can be visually recognized as "perfectly smooth". This level of continuity is very useful in the creation of models that require many bi-cubic patches composing one continuous surface.

Geometric continuity mainly refers to the shape of the resulting surface; since NURBS surfaces are functions, it is also possible to discuss the derivatives of the surface with respect to the parameters. This is known as parametric continuity. Parametric continuity of a given degree implies geometric continuity of that degree.

First- and second-level parametric continuity (C0 and C1) are for practical purposes identical to positional and tangential (G0 and G1) continuity. Third-level parametric continuity (C2), however, differs from curvature continuity in that its parameterization is also continuous. In practice, C2 continuity is easier to achieve if uniform B-splines are used.

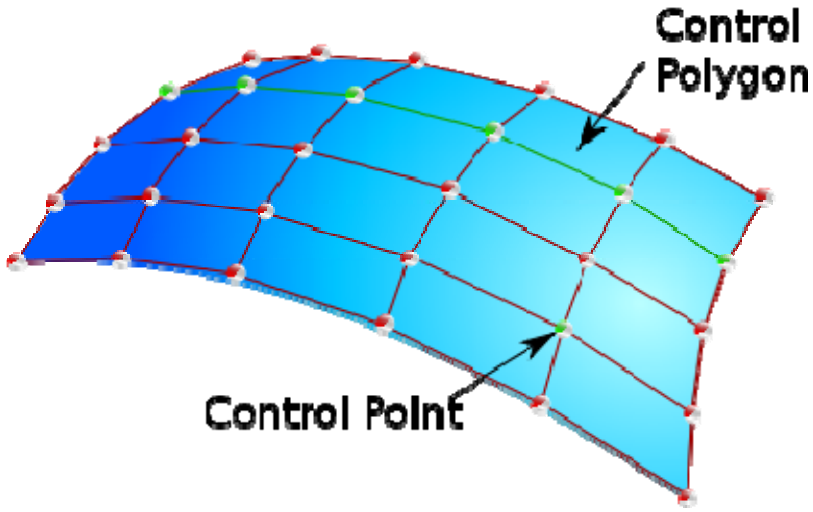
The definition of the continuity 'C*n*' requires that the n^{th} derivative of the curve/surface ($d^n C(u) / du^n$) are equal at a joint. Note that the (partial) derivatives of curves and surfaces are vectors that have a direction and a magnitude. Both should be equal.

Highlights and reflections can reveal the perfect smoothing, which is otherwise practically impossible to achieve without NURBS surfaces that have at least G2 continuity. This same principle is used as one of the surface evaluation methods whereby a ray-traced or reflection-mapped image of a surface with white stripes reflecting on it will show even the smallest deviations on a surface or set of surfaces. This method is derived from car prototyping wherein surface quality is inspected by checking the quality of reflections of a neon-light ceiling on the car surface. This method is also known as "Zebra analysis".

Technical specifications

A **NURBS curve** is defined by its *order*, a set of weighted *control points*, and a *knot vector*. NURBS curves and surfaces are generalizations of both B-splines and Bézier

curves and surfaces, the primary difference being the weighting of the control points which makes NURBS curves *rational* (non-rational B-splines are a special case of rational B-splines). Whereas Bézier curves evolve into only one parametric direction, usually called *s* or *u*, NURBS surfaces evolve into two parametric directions, called *s* and *t* or *u* and *v*.



By evaluating a NURBS curve at various values of the parameter, the curve can be represented in cartesian two- or three-dimensional space. Likewise, by evaluating a NURBS surface at various values of the two parameters, the surface can be represented in cartesian space.

NURBS curves and surfaces are useful for a number of reasons:

- They are invariant under affine as well as perspective transformations: operations like rotations and translations can be applied to NURBS curves and surfaces by applying them to their control points.
- They offer one common mathematical form for both standard analytical shapes (e.g., conics) and free-form shapes.
- They provide the flexibility to design a large variety of shapes.
- They reduce the memory consumption when storing shapes (compared to simpler methods).
- They can be evaluated reasonably quickly by numerically stable and accurate algorithms.

In the next sections, NURBS is discussed in one dimension (curves). It should be noted that all of it can be generalized to two or even more dimensions.

Control points

The control points determine the shape of the curve. Typically, each point of the curve is computed by taking a weighted sum of a number of control points. The weight of each

point varies according to the governing parameter. For a curve of degree d, the weight of any control point is only nonzero in d+1 intervals of the parameter space. Within those intervals, the weight changes according to a polynomial function (*basis functions*) of degree d. At the boundaries of the intervals, the basis functions go smoothly to zero, the smoothness being determined by the degree of the polynomial.

As an example, the basis function of degree one is a triangle function. It rises from zero to one, then falls to zero again. While it rises, the basis function of the previous control point falls. In that way, the curve interpolates between the two points, and the resulting curve is a polygon, which is continuous, but not differentiable at the interval boundaries, or **knots**. Higher degree polynomials have correspondingly more continuous derivatives. Note that within the interval the polynomial nature of the basis functions and the linearity of the construction make the curve perfectly smooth, so it is only at the knots that discontinuity can arise.

The fact that a single control point only influences those intervals where it is active is a highly desirable property, known as **local support**. In modelling, it allows the changing of one part of a surface while keeping other parts equal.

Adding more control points allows better approximation to a given curve, although only a certain class of curves can be represented exactly with a finite number of control points. NURBS curves also feature a scalar **weight** for each control point. This allows for more control over the shape of the curve without unduly raising the number of control points. In particular, it adds conic sections like circles and ellipses to the set of curves that can be represented exactly. The term *rational* in NURBS refers to these weights.

The control points can have any dimensionality. One-dimensional points just define a scalar function of the parameter. These are typically used in image processing programs to tune the brightness and color curves. Three-dimensional control points are used abundantly in 3D modelling, where they are used in the everyday meaning of the word 'point', a location in 3D space. Multi-dimensional points might be used to control sets of time-driven values, e.g. the different positional and rotational settings of a robot arm. NURBS surfaces are just an application of this. Each control 'point' is actually a full vector of control points, defining a curve. These curves share their degree and the number of control points, and span one dimension of the parameter space. By interpolating these control vectors over the other dimension of the parameter space, a continuous set of curves is obtained, defining the surface.

The knot vector

The knot vector is a sequence of parameter values that determines where and how the control points affect the NURBS curve. The number of knots is always equal to the number of control points plus curve degree plus one. The knot vector divides the parametric space in the intervals mentioned before, usually referred to as *knot spans*. Each time the parameter value enters a new knot span, a new control point becomes

active, while an old control point is discarded. It follows that the values in the knot vector should be in nondecreasing order, so (0, 0, 1, 2, 3, 3) is valid while (0, 0, 2, 1, 3, 3) is not.

Consecutive knots can have the same value. This then defines a knot span of zero length, which implies that two control points are activated at the same time (and of course two control points become deactivated). This has impact on continuity of the resulting curve or its higher derivatives; for instance, it allows to create corners in an otherwise smooth NURBS curve. A number of coinciding knots is sometimes referred to as a knot with a certain **multiplicity**. Knots with multiplicity two or three are known as double or triple knots. The multiplicity of a knot is limited to the degree of the curve; since a higher multiplicity would split the curve into disjoint parts and it would leave control points unused. For first-degree NURBS, each knot is paired with a control point.

The knot vector usually starts with a knot that has multiplicity equal to the order. This makes sense, since this activates the control points that have influence on the first knot span. Similarly, the knot vector usually ends with a knot of that multiplicity. Curves with such knot vectors start and end in a control point.

The individual knot values are not meaningful by themselves; only the ratios of the difference between the knot values matter. Hence, the knot vectors (0, 0, 1, 2, 3, 3) and (0, 0, 2, 4, 6, 6) produce the same curve. The positions of the knot values influences the mapping of parameter space to curve space. Rendering a NURBS curve is usually done by stepping with a fixed stride through the parameter range. By changing the knot span lengths, more sample points can be used in regions where the curvature is high. Another use is in situations where the parameter value has some physical significance, for instance if the parameter is time and the curve describes the motion of a robot arm. The knot span lengths then translate into velocity and acceleration, which are essential to get right to prevent damage to the robot arm or its environment. This flexibility in the mapping is what the phrase *non uniform* in NURBS refers to.

Necessary only for internal calculations, knots are usually not helpful to the users of modeling software. Therefore, many modeling applications do not make the knots editable or even visible. It's usually possible to establish reasonable knot vectors by looking at the variation in the control points. More recent versions of NURBS software (e.g., Autodesk Maya and Rhinoceros 3D) allow for interactive editing of knot positions, but this is significantly less intuitive than the editing of control points.

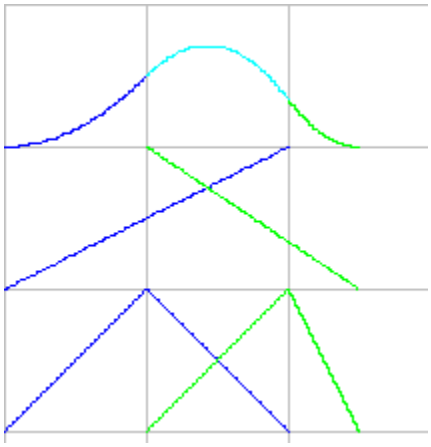
Order

The *order* of a NURBS curve defines the number of nearby control points that influence any given point on the curve. The curve is represented mathematically by a polynomial of degree one less than the order of the curve. Hence, second-order curves (which are represented by linear polynomials) are called linear curves, third-order curves are called quadratic curves, and fourth-order curves are called cubic curves. The number of control points must be greater than or equal to the order of the curve.

In practice, cubic curves are the ones most commonly used. Fifth- and sixth-order curves are sometimes useful, especially for obtaining continuous higher order derivatives, but curves of higher orders are practically never used because they lead to internal numerical problems and tend to require disproportionately large calculation times.

Construction of the basis functions

The basis functions used in NURBS curves are usually denoted as $N_{i,n}(u)$, in which i corresponds to the i-th control point, and n corresponds with the degree of the basis function. The parameter dependence is frequently left out, so we can write $N_{i,n}$. The definition of these basis functions is recursive in n. The degree-0 functions $N_{i,0}$ are piecewise constant functions. They are one on the corresponding knot span and zero everywhere else. Effectively, $N_{i,n}$ is a linear interpolation of $N_{i,n-1}$ and $N_{i+1,n-1}$. The latter two functions are non-zero for n knot spans, overlapping for $n-1$ knot spans. The function $N_{i,n}$ is computed as



From bottom to top: Linear basis functions $N_{1,1}$ (blue) and $N_{2,1}$ (green), their weight functions f and g and the resulting quadratic basis function. The knots are 0, 1, 2 and 2.5

$$N_{i,n} = f_{i,n}N_{i,n-1} + g_{i+1,n}N_{i+1,n-1}$$

f_i rises linearly from zero to one on the interval where $N_{i,n-1}$ is non-zero, while g_{i+1} falls from one to zero on the interval where $N_{i+1,n-1}$ is non-zero. As mentioned before, $N_{i,1}$ is a triangular function, nonzero over two knot spans rising from zero to one on the first, and falling to zero on the second knot span. Higher order basis functions are non-zero over corresponding more knot spans and have correspondingly higher degree. If u is the parameter, and k_i is the i-th knot, we can write the functions f and g as

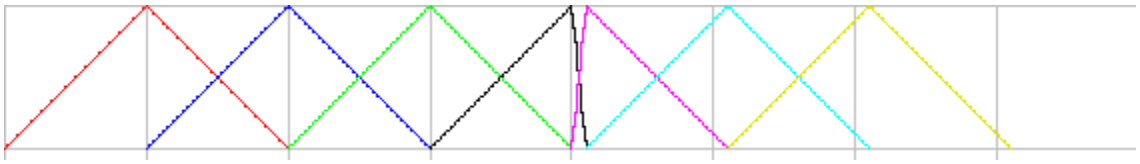
$$f_{i,n}(u) = \frac{u - k_i}{k_{i+n} - k_i}$$

and

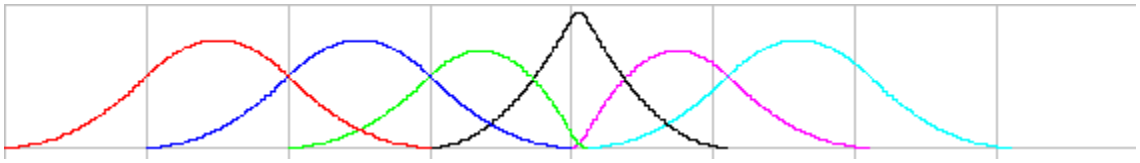
$$g_{i,n}(u) = \frac{k_{i+n} - u}{k_{i+n} - k_i}$$

The functions *f* and *g* are positive when the corresponding lower order basis functions are non-zero. By induction on n it follows that the basis functions are non-negative for all values of *n* and *u*. This makes the computation of the basis functions numerically stable.

Again by induction, it can be proved that the sum of the basis functions for a particular value of the parameter is unity. This is known as the **partition of unity** property of the basis functions.



Linear basis functions



Quadratic basis functions

The figures show the linear and the quadratic basis functions for the knots {..., 0, 1, 2, 3, 4, 4.1, 5.1, 6.1, 7.1, ...}

One knot span is considerably shorter than the others. On that knot span, the peak in the quadratic basis function is more distinct, reaching almost one. Conversely, the adjoining basis functions fall to zero more quickly. In the geometrical interpretation, this means that the curve approaches the corresponding control point closely. In case of a double knot, the length of the knot span becomes zero and the peak reaches one exactly. The basis function is no longer differentiable at that point. The curve will have a sharp corner if the neighbour control points are not collinear.

General form of a NURBS curve

Using the definitions of the basis functions $N_{i,n}$ from the previous paragraph, a NURBS curve takes the following form :

$$C(u) = \sum_{i=1}^{k} \frac{N_{i,n} w_i}{\sum_{j=1}^{k} N_{j,n} w_j} \mathbf{P}_i = \frac{\sum_{i=1}^{k} N_{i,n} w_i \mathbf{P}_i}{\sum_{i=1}^{k} N_{i,n} w_i}$$

In this, *k* is the number of control points $\mathbf{P}_i$ and w_i are the corresponding weights. The denominator is a normalizing factor that evaluates to one if all weights are one. This can

be seen from the partition of unity property of the basis functions. It is customary to write this as

$$C(u) = \sum_{i=1}^{k} R_{i,n} \mathbf{P}_i$$

in which the functions

$$R_{i,n} = \frac{N_{i,n} w_i}{\sum_{j=1}^{k} N_{j,n} w_j}$$

are known as the *rational basis functions*.

Manipulating NURBS objects

A number of transformations can be applied to a NURBS object. For instance, if some curve is defined using a certain degree and N control points, the same curve can be expressed using the same degree and N+1 control points. In the process a number of control points change position and a knot is inserted in the knot vector. These manipulations are used extensively during interactive design. When adding a control point, the shape of the curve should stay the same, forming the starting point for further adjustments. A number of these operations are discussed below.

Knot insertion

As the term suggests, **knot insertion** inserts a knot into the knot vector. If the degree of the curve is *n*, then $n - 1$ control points are replaced by *n* new ones. The shape of the curve stays the same.

A knot can be inserted multiple times, up to the maximum multiplicity of the knot. This is sometimes referred to as **knot refinement** and can be achieved by an algorithm that is more efficient than repeated knot insertion.

Knot removal

Knot removal is the reverse of knot insertion. Its purpose is to remove knots and the associated control points in order to get a more compact representation. Obviously, this is not always possible while retaining the exact shape of the curve. In practice, a tolerance in the accuracy is used to determine whether a knot can be removed. The process is used to clean up after an interactive session in which control points may have been added manually, or after importing a curve from a different representation, where a straightforward conversion process leads to redundant control points.

Degree elevation

A NURBS curve of a particular degree can always be represented by a NURBS curve of higher degree. This is frequently used when combining separate NURBS curves, e.g. when creating a NURBS surface interpolating between a set of NURBS curves or when unifying adjacent curves. In the process, the different curves should be brought to the same degree, usually the maximum degree of the set of curves. The process is known as **degree elevation**.

Curvature

The most important property in differential geometry is the curvature κ. It describes the local properties (edges, corners, etc.) and relations between the first and second derivative, and thus, the precise curve shape. Having determined the derivatives it is easy to compute $\kappa = \frac{|r'(t) \times r''(t)|}{|r'(t)|^3}$ or approximated as the arclength from the second derivate $\kappa = |\, r''(s_o)\,|$. The direct computation of the curvature κ with these equations is the big advantage of parameterized curves against their polygonal representations.

Example: a circle

Non-rational splines or Bézier curves may approximate a circle, but they cannot represent it exactly. Rational splines can represent any conic section, including the circle, exactly. This representation is not unique, but one possibility appears below:

x	***y***	***z***	**weight**
1	0	0	1
$\sqrt{2}/2$	$\sqrt{2}/2$	0	$\sqrt{2}/2$
0	1	0	1
$-\sqrt{2}/2$	$\sqrt{2}/2$	0	$\sqrt{2}/2$
−1	0	0	1
$-\sqrt{2}/2$	$-\sqrt{2}/2$	0	$\sqrt{2}/2$
0	−1	0	1
$\sqrt{2}/2$	$-\sqrt{2}/2$	0	$\sqrt{2}/2$
1	0	0	1

The order is three, since a circle is a quadratic curve and the spline's order is one more than the degree of its piecewise polynomial segments. The knot vector is $\{0, 0, 0, \pi/2, \pi/2, \pi, \pi, 3\pi/2, 3\pi/2, 2\pi, 2\pi, 2\pi\}$. The circle is composed of four quarter circles, tied together with double knots. Although double knots in a third order NURBS curve would normally result in loss of continuity in the first derivative, the

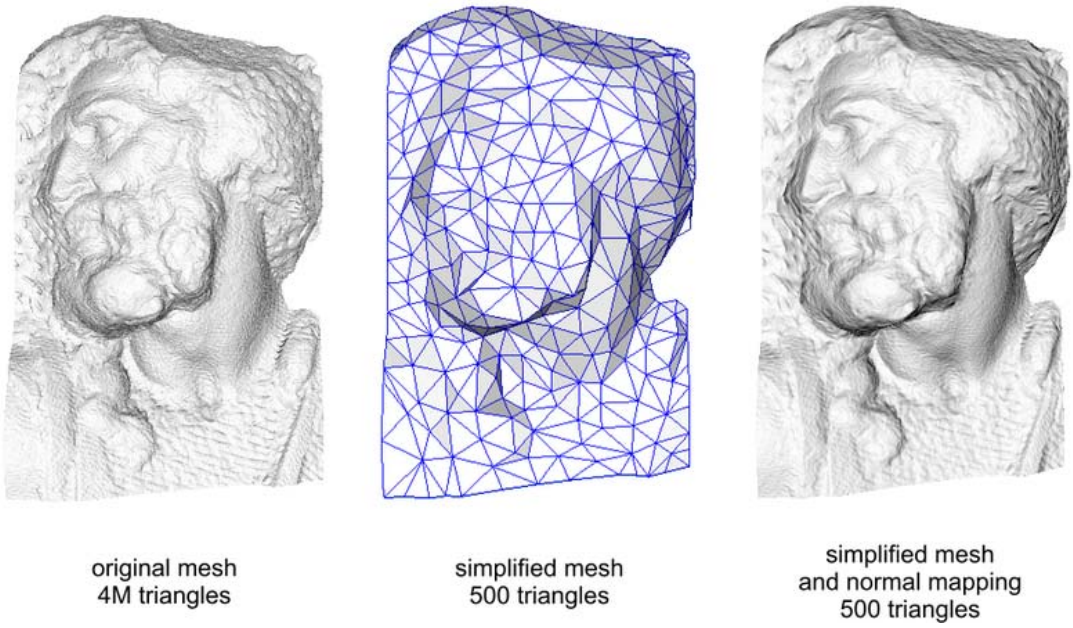
control points are positioned in such a way that the first derivative is continuous. (In fact, the curve is infinitely differentiable everywhere, as it must be if it exactly represents a circle.)

The curve represents a circle exactly, but it is not exactly parametrized in the circle's arc length. This means, for example, that the point at t does not lie at $(\sin(t),\cos(t))$ (except for the start, middle and end point of each quarter circle, since the representation is symmetrical). This is obvious; the x coordinate of the circle would otherwise provide an exact rational polynomial expression for $\cos(t)$, which is impossible. The circle does make one full revolution as its parameter t goes from 0 to 2π, but this is only because the knot vector was arbitrarily chosen as multiples of $\pi / 2$.

Chapter 7

Normal Mapping & Mipmap

Normal Mapping



Normal mapping used to re-detail simplified meshes.

In 3D computer graphics, **normal mapping**, or "Dot3 bump mapping", is a technique used for faking the lighting of bumps and dents. It is used to add details without using more polygons. A normal map is usually an RGB image that corresponds to the X, Y, and Z coordinates of a surface normal from a more detailed version of the object. A common use of this technique is to greatly enhance the appearance and details of a low polygon model by generating a normal map from a high polygon model.

History

The idea of taking geometric details from a high polygon model was introduced in "Fitting Smooth Surfaces to Dense Polygon Meshes" by Krishnamurthy and Levoy, Proc. SIGGRAPH 1996, where this approach was used for creating displacement maps over nurbs. In 1998, two papers were presented with key ideas for transferring details with normal maps from high to low polygon meshes: "Appearance Preserving Simplification", by Cohen et al. SIGGRAPH 1998, and "A general method for preserving attribute values on simplified meshes" by Cignoni et al. IEEE Visualization '98. The former introduced the idea of storing surface normals directly in a texture, rather than displacements, though it required the low-detail model to be generated by a particular constrained simplification algorithm. The latter presented a simpler approach that decouples the high and low polygonal mesh and allows the recreation of any attributes of the high-detail model (color, texture coordinates, displacements, etc.) in a way that is not dependent on how the low-detail model was created. The combination of storing normals in a texture, with the more general creation process is still used by most currently available tools.

How it works

To calculate the Lambertian (diffuse) lighting of a surface, the unit vector from the shading point to the light source is dotted with the unit vector normal to that surface, and the result is the intensity of the light on that surface. Imagine a polygonal model of a sphere - you can only approximate the shape of the surface. By using a 3-channel bitmap textured across the model, more detailed normal vector information can be encoded. Each channel in the bitmap corresponds to a spatial dimension (X, Y and Z). These spatial dimensions are relative to a constant coordinate system for object-space normal maps, or to a smoothly varying coordinate system (based on the derivatives of position with respect to texture coordinates) in the case of tangent-space normal maps. This adds much more detail to the surface of a model, especially in conjunction with advanced lighting techniques.

Calculating Tangent Space

In order to find the perturbation in the normal the tangent space must be correctly calculated. Most often the normal is perturbed in a fragment shader after applying the model and view matrices. Typically the geometry provides a normal and tangent. The tangent is part of the tangent plane and can be transformed simply with the linear part of the matrix (the upper 3x3). However, the normal needs to be transformed by the inverse transpose. Most applications will want bitangent to match the transformed geometry (and associated uv's). So instead of enforcing the bitanget to be normal to the tangent, it is generally preferable to transform the bitangent just like the tangent. Let *t* be tangent, *n* be normal, *b* be bitangent, M_{3x3} be linear part of model matrix, and V_{3x3} be the linear part of the view matrix.

$$t' = t \times M_{3x3} \times V_{3x3}$$
$$n' = n \times (M_{3x3} \times V_{3x3})^{-1T} = n \times M_{3x3}^{-1T} \times V_{3x3}^{-1T}$$

$$b' = b \times M_{3x3} \times V_{3x3}$$

Normal mapping in video games

Interactive normal map rendering was originally only possible on PixelFlow, a parallel rendering machine built at the University of North Carolina at Chapel Hill. It was later possible to perform normal mapping on high-end SGI workstations using multi-pass rendering and framebuffer operations or on low end PC hardware with some tricks using paletted textures. However, with the advent of shaders in personal computers and game consoles, normal mapping became widely used in proprietary commercial video games starting in late 2003, and followed by open source games in later years. Normal mapping's popularity for real-time rendering is due to its good quality to processing requirements ratio versus other methods of producing similar effects. Much of this efficiency is made possible by distance-indexed detail scaling, a technique which selectively decreases the detail of the normal map of a given texture (cf. mipmapping), meaning that more distant surfaces require less complex lighting simulation.

Basic normal mapping can be implemented in any hardware that supports palettized textures. The first game console to have specialized normal mapping hardware was the Sega Dreamcast. However, Microsoft's Xbox was the first console to widely use the effect in retail games. Out of the sixth generation consoles, only the PlayStation 2's GPU lacks built-in normal mapping support. Games for the Xbox 360 and the PlayStation 3 rely heavily on normal mapping and are beginning to implement parallax mapping. The Nintendo 3DS has been shown to support normal mapping, as demonstrated by Resident Evil: Revelations and Metal Gear Solid: Snake Eater.

Mipmap

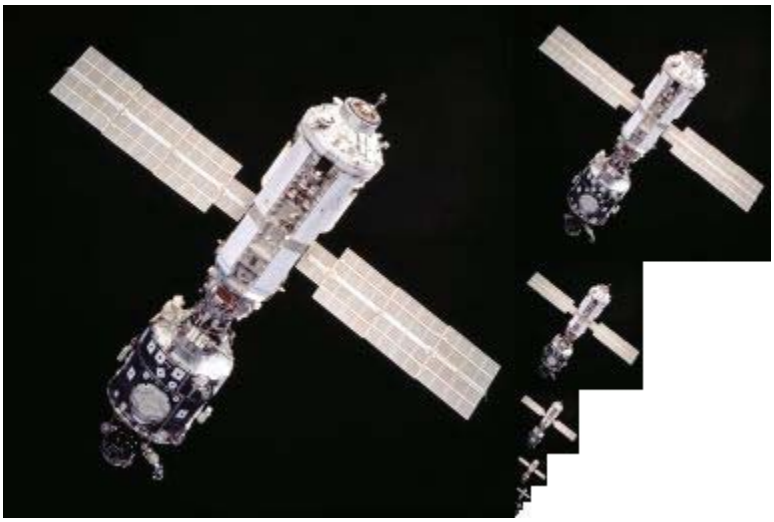
In 3D computer graphics texture filtering, **MIP maps** (also **mipmaps**) are pre-calculated, optimized collections of images that accompany a main texture, intended to increase rendering speed and reduce aliasing artifacts. They are widely used in 3D computer games, flight simulators and other 3D imaging systems. The technique is known as **mipmapping**. The letters "MIP" in the name are an acronym of the Latin phrase *multum in parvo*, meaning "much in a small space". Mipmaps need more space in memory. They also form the basis of wavelet compression.

Origin

Mipmapping was invented by Lance Williams in 1983 and is described in his paper *Pyramidal parametrics*. From the abstract: "This paper advances a 'pyramidal parametric' prefiltering and sampling geometry which minimizes aliasing effects and assures

continuity within and between target images." The "pyramid" can be imagined as the set of mipmaps stacked on top of each other.

How it works



An example of mipmap image storage: the principal image on the left is accompanied by filtered copies of reduced size.

Each bitmap image of the mipmap set is a version of the main texture, but at a certain reduced level of detail. Although the main texture would still be used when the view is sufficient to render it in full detail, the renderer will switch to a suitable mipmap image (or in fact, interpolate between the two nearest, if trilinear filtering is activated) when the texture is viewed from a distance or at a small size. Rendering speed increases since the number of texture pixels ("texels") being processed can be much lower than with simple textures. Artifacts are reduced since the mipmap images are effectively already anti-aliased, taking some of the burden off the real-time renderer. Scaling down and up is made more efficient with mipmaps as well.

If the texture has a basic size of 256 by 256 pixels, then the associated mipmap set may contain a series of 8 images, each one-fourth the total area of the previous one: 128×128 pixels, 64×64, 32×32, 16×16, 8×8, 4×4, 2×2, 1×1 (a single pixel). If, for example, a scene is rendering this texture in a space of 40×40 pixels, then either a scaled up version of the 32×32 (without trilinear interpolation) or an interpolation of the 64×64 and the 32×32 mipmaps (with trilinear interpolation) would be used. The simplest way to generate these textures is by successive averaging; however, more sophisticated algorithms (perhaps based on signal processing and Fourier transforms) can also be used.

The increase in storage space required for all of these mipmaps is a third of the original texture, because the sum of the areas $1/4 + 1/16 + 1/64 + 1/256 + \cdots$ converges to 1/3. In the case of an RGB image with three channels stored as separate planes, the total mipmap can be visualized as fitting neatly into a square area twice as large as the dimensions of the original image on each side (four times the original area - one square for each

channel, then increase subtotal that by a third). This is the inspiration for the tag "multum in parvo".

In many instances, the filtering should not be uniform in each direction (it should be anisotropic, as opposed to isotropic), and a compromise resolution is used. If a higher resolution is used, the cache coherence goes down, and the aliasing is increased in one direction, but the image tends to be clearer. If a lower resolution is used, the cache coherence is improved, but the image is overly blurry, to the point where it becomes difficult to identify.

To help with this problem, nonuniform mipmaps (also known as rip-maps) are sometimes used. With a 16×16 base texture map, the rip-map resolutions would be 16×8, 16×4, 16×2, 16×1, 8×16, 8×8, 8×4, 8×2, 8×1, 4×16, 4×8, 4×4, 4×2, 4×1, 2×16, 2×8, 2×4, 2×2, 2×1, 1×16, 1×8, 1×4, 1×2 and 1×1.

A trade off : anisotropic mip-mapping

The unfortunate problem with this approach is that rip-maps require four times as much memory as the base texture map, and so rip-maps have been very unpopular. Also for 1×4 and more extreme 4 maps each rotated by 45° would be needed and the real memory requirement is growing more than linearly.

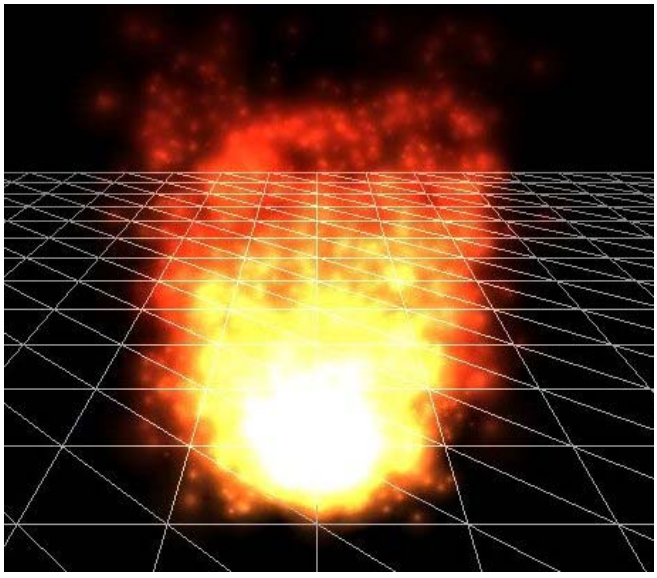
To reduce the memory requirement, and simultaneously give more resolutions to work with, summed-area tables were conceived. However, this approach tends to exhibit poor cache behavior. Also, a summed area table needs to have wider types to store the partial sums than the word size used to store the texture. For these reasons, there isn't any hardware that implements summed-area tables today.

A compromise has been reached today, called anisotropic mip-mapping. In the case where an anisotropic filter is needed, a higher resolution mipmap is used, and several texels are averaged in one direction to get more filtering in that direction. This has a somewhat detrimental effect on the cache, but greatly improves image quality.

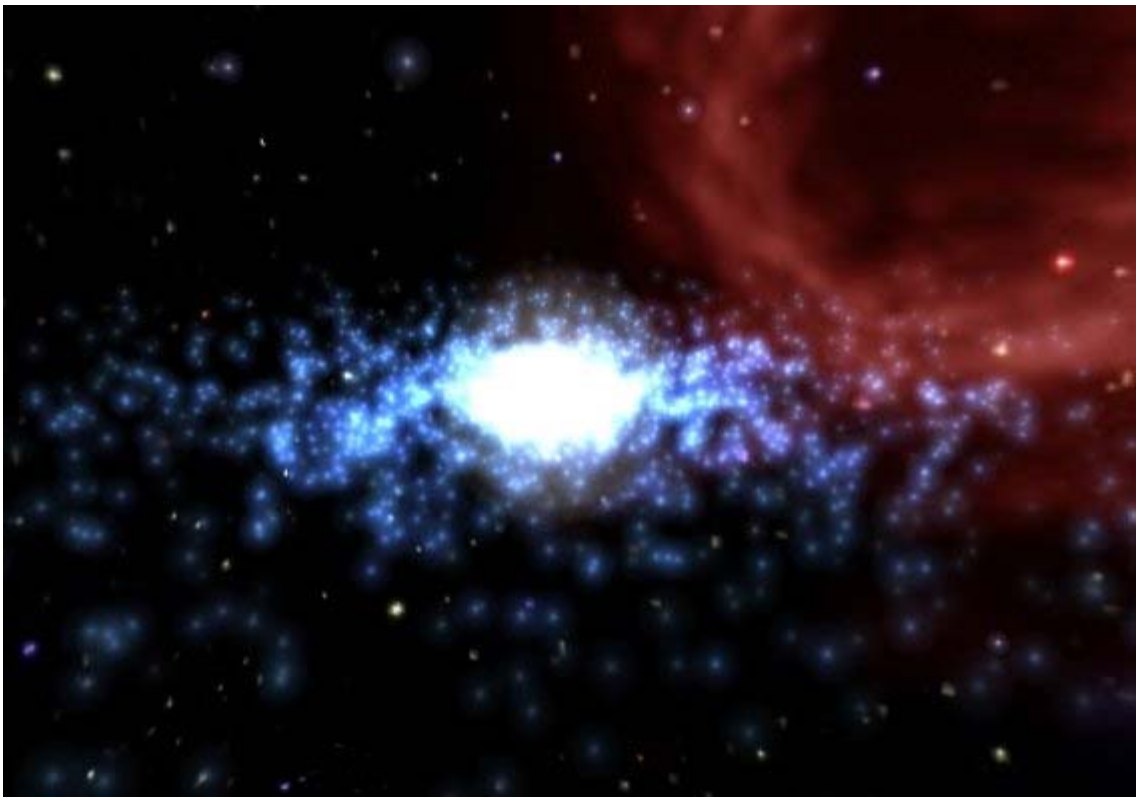
Chapter 8

Particle System & Painter's Algorithm

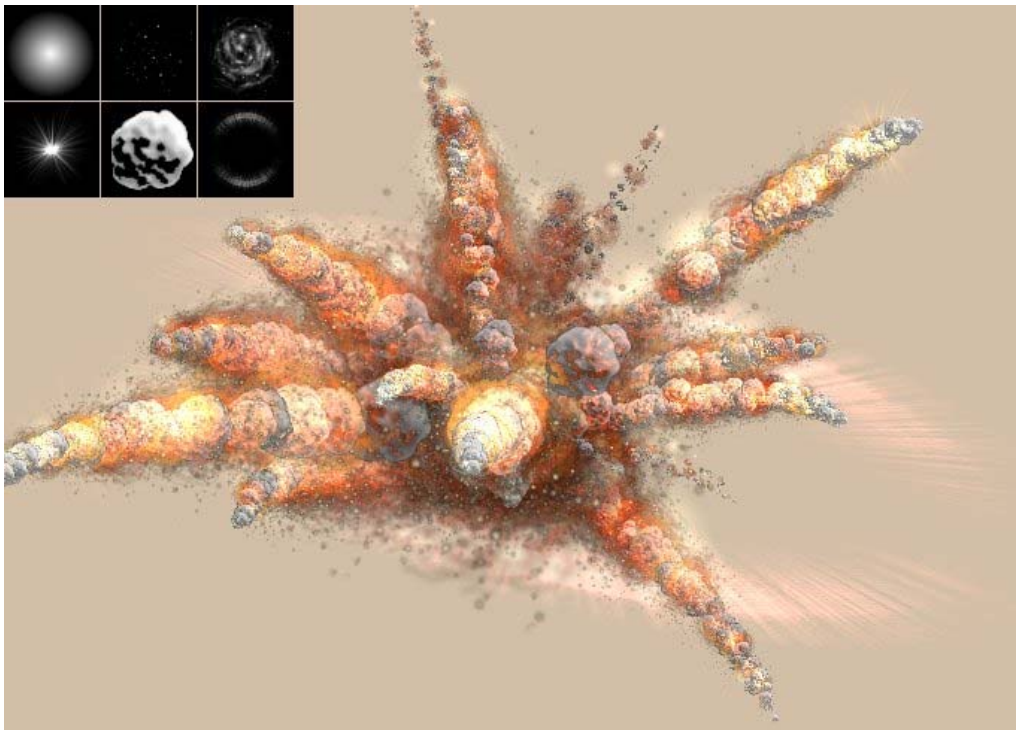
Particle System



A particle system used to simulate a fire, created in 3dengfx.



Ad-hoc particle system used to simulate a galaxy, created in 3dengfx.



A particle system used to simulate a bomb explosion, created in particleIllusion.

The term **particle system** refers to a computer graphics technique to simulate certain fuzzy phenomena, which are otherwise very hard to reproduce with conventional rendering techniques. Examples of such phenomena which are commonly replicated using particle systems include fire, explosions, smoke, moving water, sparks, falling leaves, clouds, fog, snow, dust, meteor tails, hair, fur, grass, or abstract visual effects like glowing trails, magic spells, etc.

While in most cases particle systems are implemented in three dimensional graphics systems, two dimensional particle systems may also be used under some circumstances.

Typical implementation

Typically a particle system's position and motion in 3D space are controlled by what is referred to as an **emitter**. The emitter acts as the source of the particles, and its location in 3D space determines where they are generated and whence they proceed. A regular 3D mesh object, such as a cube or a plane, can be used as an emitter. The emitter has attached to it a set of particle behavior parameters. These parameters can include the spawning rate (how many particles are generated per unit of time), the particles' initial velocity vector (the direction they are emitted upon creation), particle lifetime (the length of time each individual particle exists before disappearing), particle color, and many more. It is common for all or most of these parameters to be "fuzzy" — instead of a precise numeric value, the artist specifies a central value and the degree of randomness allowable on either side of the center (i.e. the average particle's lifetime might be 50 frames ±20%). When using a mesh object as an emitter, the initial velocity vector is often set to be normal to the individual face(s) of the object, making the particles appear to "spray" directly from each face.

A typical particle system's update loop (which is performed for each frame of animation) can be separated into two distinct stages, the **parameter update/simulation** stage and the **rendering** stage.

Simulation stage

During the **simulation** stage, the number of new particles that must be created is calculated based on spawning rates and the interval between updates, and each of them is spawned in a specific position in 3D space based on the emitter's position and the spawning area specified. Each of the particle's parameters (i.e. velocity, color, etc.) is initialized according to the emitter's parameters. At each update, all existing particles are checked to see if they have exceeded their lifetime, in which case they are removed from the simulation. Otherwise, the particles' position and other characteristics are advanced based on some sort of physical simulation, which can be as simple as translating their current position, or as complicated as performing physically-accurate trajectory calculations which take into account external forces (gravity, friction, wind, etc.). It is common to perform some sort of collision detection between particles and specified 3D objects in the scene to make the particles bounce off of or otherwise interact with

obstacles in the environment. Collisions between particles are rarely used, as they are computationally expensive and not really useful for most simulations.

Rendering stage

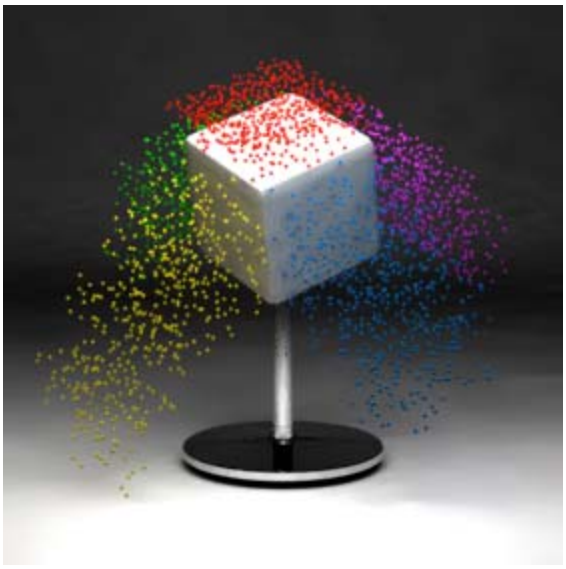
After the update is complete, each particle is rendered, usually in the form of a textured billboarded quad (i.e. a quadrilateral that is always facing the viewer). However, this is not necessary; a particle may be rendered as a single pixel in small resolution/limited processing power environments. Particles can be rendered as Metaballs in off-line rendering; isosurfaces computed from particle-metaballs make quite convincing liquids. Finally, 3D mesh objects can "stand in" for the particles — a snowstorm might consist of a single 3D snowflake mesh being duplicated and rotated to match the positions of thousands or millions of particles.

Snowflakes versus hair

Particle systems can be either animated or static; that is, the lifetime of each particle can either be distributed over time or rendered all at once. The consequence of this distinction is the difference between the appearance of "snow" and the appearance of "hair."

The term "particle system" itself often brings to mind only the animated aspect, which is commonly used to create moving particulate simulations — sparks, rain, fire, etc. In these implementations, each frame of the animation contains each particle at a specific position in its life cycle, and each particle occupies a single point position in space.

However, if the entire life cycle of the each particle is rendered simultaneously, the result is **static** particles — strands of material that show the particles' overall trajectory, rather than point particles. These strands can be used to simulate hair, fur, grass, and similar materials. The strands can be controlled with the same velocity vectors, force fields, spawning rates, and deflection parameters that animated particles obey. In addition, the rendered thickness of the strands can be controlled and in some implementations may be varied along the length of the strand. Different combinations of parameters can impart stiffness, limpness, heaviness, bristliness, or any number of other properties. The strands may also use texture mapping to vary the strands' color, length, or other properties across the emitter surface.



A cube emitting 5000 animated particles, obeying a "gravitational" force in the negative Y direction.

The same cube emitter rendered using static particles, or strands.

Artist-friendly particle system tools

Particle systems can be created and modified natively in many 3D modeling and rendering packages including Lightwave, Houdini, Maya, XSI, 3D Studio Max and Blender. These editing programs allow artists to have instant feedback on how a particle system will look with properties and constraints that they specify. There is also plug-in software available that provides enhanced particle effects; examples include AfterBurn and RealFlow (for liquids). Compositing software such as Combustion or specialized,

particle-only software such as Particle Studio and particleIllusion can be used for the creation of particle systems for film and video.

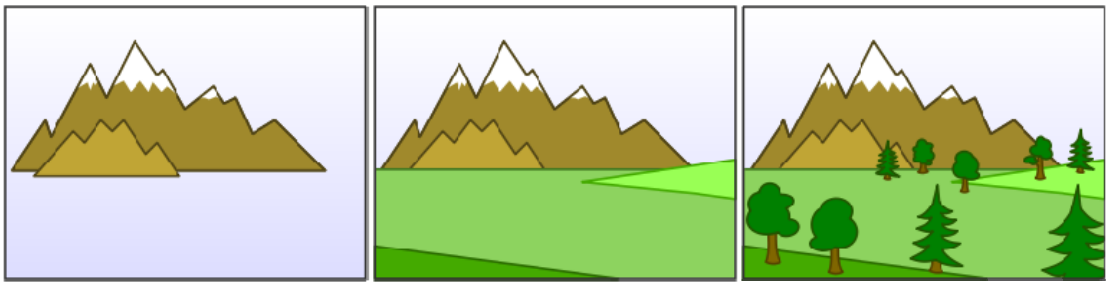
Developer-friendly particle system tools

Particle systems code that can be included in game engines, digital content creation systems, and effects applications can be written from scratch or downloaded. One free implementation is *The Particle Systems API*. Another for the XNA framework is the *Dynamic Particle System Framework*. Havok provides multiple particle system APIs. Their Havok FX API focuses especially on particle system effects. Ageia provides a particle system and other game physics API that is used in many games, including Unreal Engine 3 games. In February 2008, Ageia was bought by Nvidia.

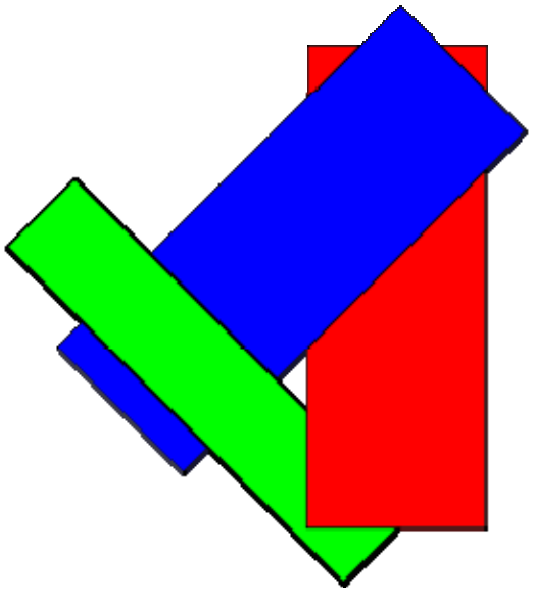
Painter's Algorithm

The **painter's algorithm**, also known as a **priority fill**, is one of the simplest solutions to the visibility problem in 3D computer graphics. When projecting a 3D scene onto a 2D plane, it is necessary at some point to decide which polygons are visible, and which are hidden.

The name "painter's algorithm" refers to the technique employed by many painters of painting distant parts of a scene before parts which are nearer thereby covering some areas of distant parts. The painter's algorithm sorts all the polygons in a scene by their depth and then paints them in this order, farthest to closest. It will paint over the parts that are normally not visible — thus solving the visibility problem — at the cost of having painted redundant areas of distant objects.



The distant mountains are painted first, followed by the closer meadows; finally, the closest objects in this scene, the trees, are painted.



Overlapping polygons can cause the algorithm to fail

The algorithm can fail in some cases, including cyclic overlap or piercing polygons. In the case of cyclic overlap, as shown in the figure to the right, Polygons A, B, and C overlap each other in such a way that it is impossible to determine which polygon is above the others. In this case, the offending polygons must be cut to allow sorting. Newell's algorithm, proposed in 1972, provides a method for cutting such polygons. Numerous methods have also been proposed in the field of computational geometry.

The case of piercing polygons arises when one polygon intersects another. As with cyclic overlap, this problem may be resolved by cutting the offending polygons.

In basic implementations, the painter's algorithm can be inefficient. It forces the system to render each point on every polygon in the visible set, even if that polygon is occluded in the finished scene. This means that, for detailed scenes, the painter's algorithm can overly tax the computer hardware.

A **reverse painter's algorithm** is sometimes used, in which objects nearest to the viewer are painted first — with the rule that paint must never be applied to parts of the image that are already painted. In a computer graphic system, this can be very efficient, since it is not necessary to calculate the colors (using lighting, texturing and such) for parts of the more distant scene that are hidden by nearby objects. However, the reverse algorithm suffers from many of the same problems as the standard version.

These and other flaws with the algorithm led to the development of Z-buffer techniques, which can be viewed as a development of the painter's algorithm, by resolving depth conflicts on a pixel-by-pixel basis, reducing the need for a depth-based rendering order. Even in such systems, a variant of the painter's algorithm is sometimes employed. As Z-buffer implementations generally rely on fixed-precision depth-buffer registers implemented in hardware, there is scope for visibility problems due to rounding error.

These are overlaps or gaps at joins between polygons. To avoid this, some graphics engine implementations "overrender", drawing the affected edges of both polygons in the order given by painter's algorithm. This means that some pixels are actually drawn twice (as in the full painters algorithm) but this happens on only small parts of the image and has a negligible performance effect.

Chapter 9

Phong Shading

Phong shading refers to a set of techniques in 3D computer graphics. Phong shading includes a model for the reflection of light from surfaces and a compatible method of estimating pixel colors by interpolating surface normals across rasterized polygons.

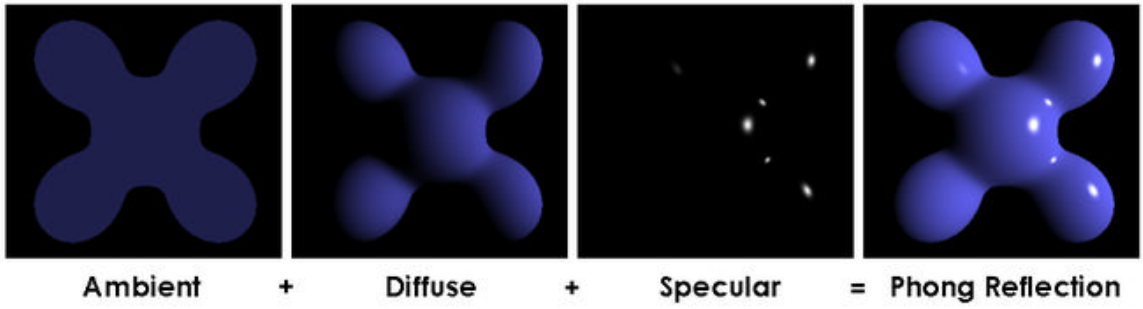
The model of reflection may also be referred to as the **Phong reflection model**, **Phong illumination** or **Phong lighting**. It may be called Phong shading in the context of pixel shaders or other places where a lighting calculation can be referred to as "shading". The interpolation method may also be called **Phong interpolation**, which is usually referred to by "per-pixel lighting". Typically it is called "shading" when contrasted with other interpolation methods such as Gouraud shading or flat shading. The Phong reflection model may be used in conjunction with any of these interpolation methods.

History

These methods were developed by Bui Tuong Phong at the University of Utah, who published them in his 1973 Ph.D. dissertation. Phong's shading methods were considered radical at the time of their introduction, but have evolved into a baseline shading method for many rendering applications. Phong's methods have proven popular due to their generally parsimonious use of CPU time per rendered pixel.

Phong reflection model

Phong reflection is an empirical model of local illumination. It describes the way a surface reflects light as a combination of the diffuse reflection of rough surfaces with the specular reflection of shiny surfaces. It is based on Bui Tuong Phong's informal observation that shiny surfaces have small intense specular highlights, while dull surfaces have large highlights that fall off more gradually. The reflection model also includes an *ambient* term to account for the small amount of light that is scattered about the entire scene.



Visual illustration of the Phong equation: here the light is white, the ambient and diffuse colors are both blue, and the specular color is white, reflecting a small part of the light hitting the surface, but only in very narrow highlights. The intensity of the diffuse component varies with the direction of the surface, and the ambient component is uniform (independent of direction).

For each light source in the scene, we define the components i_s and i_d as the intensities (often as RGB values) of the specular and diffuse components of the light sources respectively. A single term i_a controls the ambient lighting; it is sometimes computed as a sum of contributions from all light sources.

For each *material* in the scene, we define:

> k_s: specular reflection constant, the ratio of reflection of the specular term of incoming light
> k_d: diffuse reflection constant, the ratio of reflection of the diffuse term of incoming light (Lambertian reflectance)
> k_a: ambient reflection constant, the ratio of reflection of the ambient term present in all points in the scene rendered
> α: is a *shininess* constant for this material, which is larger for surfaces that are smoother and more mirror-like. When this constant is large the specular highlight is small.

We further define lights as the set of all light sources, L as the direction vector from the point on the surface toward each light source, N as the normal at this point on the surface, R as the direction that a perfectly reflected ray of light would take from this point on the surface, and V as the direction pointing towards the viewer (such as a virtual camera).

Then the *Phong reflection model* provides an equation for computing the shading value of each surface point I_p:

$$I_p = k_a i_a + \sum_{m \in \text{lights}} \left(k_d (L_m \cdot N) i_{m,d} + k_s (R_m \cdot V)^{\alpha} i_{m,s} \right).$$

where the direction vector R_m is calculated as the reflection of $-L_m$ (the direction from the light source to the surface) on the surface using a Householder transformation:

$$R_m = 2(L_m \cdot N)N - L_m$$

The diffuse term is not affected by the viewer direction (*V*). The specular term is large only when the viewer direction (*V*) is aligned with the reflection direction *R*. Their alignment is measured by the α power of the cosine of the angle between them. The cosine of the angle between the normalized vectors *R* and *V* is equal to their dot product. When α is large, in the case of a nearly mirror-like reflection, the specular highlight will be small, because any viewpoint not aligned with the reflection will have a cosine less than one which rapidly approaches zero when raised to a high power.

When we have color representations as RGB values, this equation will typically be calculated separately for R, G and B intensities.

Although the above formulation is the common way of presenting the Phong model, each term should only be included if the term's dot product is positive.

Computational approximations

When implementing the Phong reflection model in graphics software, there are a number of methods for approximating the model, rather than implementing the exact formulas, which can speed up the calculation.

If α is large, the calculation of the power term may be computationally expensive since it requires a large number of multiplications; it can be approximated by realizing that

$$(R_m \cdot V)^\alpha = (1-\lambda)^\alpha = (1-\lambda)^{\beta\gamma} = ((1-\lambda)^\beta)^\gamma \approx \begin{cases} (1-\beta\lambda)^\gamma, & 1-\beta\lambda > 0 \\ 0, & 1-\beta\lambda \leq 0 \end{cases}$$

for a sufficiently large integer γ (typically 4 will be enough), where $\lambda = 1 - R_m \cdot V$, which can be approximated as $\lambda = ||R_m - V||^2/2$, and $\beta = \alpha/\gamma$ is a real number (not necessarily an integer). This method substitutes a few multiplications for a variable exponentiation, and if using the difference vector $R_m - V$ instead of the dot product doesn't require as accurate a normalization of the interpolated normal vector in computing the reflection vector.

Inverse Phong reflection model

The Phong shading reflection model is an approximation of shading of objects in real life. This means that the Phong equation can relate the shading seen in a photograph with the surface normals of the visible object. Inverse refers to the wish to estimate the surface normals given a rendered image, natural or computer-made.

The Phong reflection model contains many parameters, such as the surface diffuse reflection parameter (albedo) which may vary within the object. Thus the normals of an

object in a photograph can only be determined, by introducing additive information such as the number of lights, light directions and reflection parameters.

For example we have a cylindrical object for instance a finger and like to calculate the normal $N = [N_x,N_z]$ on a line on the object. We assume only one light, no specular reflection, and uniform known (approximated) reflection parameters. We can then simplify the Phong equation to:

$$I_p(x) = C_a + C_d(L(x) \cdot N(x))$$

With C_a a constant equal to the ambient light and C_d a constant equal to the diffusion reflection. We can re-write the equation to:

$$(I_p(x) - C_a)/C_d = L(x) \cdot N(x)$$

Which can be rewritten for a line through the cylindrical object as:

$$(I_p - C_a) / C_d = L_x N_x + L_z N_z$$

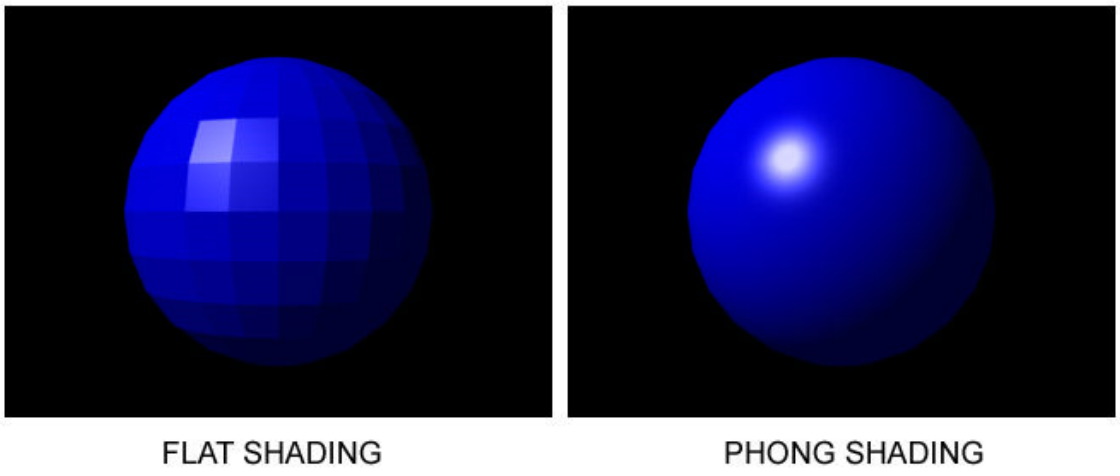
For instance if the light direction is 45 degrees above the object $L = [0.71,0.71]$ we get two equations with two unknowns.

$$(I_p - C_a) / C_d = 0.71N_x + 0.71N_z$$
$$1 = \sqrt{(N_x^2 + N_z^2)}$$

Because of the powers of two in the equation there are two possible solutions for the normal direction. Thus some prior information of the geometry is needed to define the correct normal direction. The normals are directly related to angles of inclination of the line on the object surface. Thus the normals allow the calculation of the relative surface heights of the line on the object using a line integral, if we assume a continuous surface.

If the object is not cylindrical, we have three unknown normal values $N = [N_x,N_y,N_z]$. Then the two equations still allow the normal to rotate around the view vector, thus additional constraints are needed from prior geometric information. For instance in face recognition those geometric constraints can be obtained using principal component analysis (PCA) on a database of depth-maps of faces, allowing only surface normals solutions which are found in a normal population.

Phong interpolation



Phong shading interpolation example

Phong shading improves upon Gouraud shading and provides a better approximation of the shading of a smooth surface. Phong shading assumes a smoothly varying surface normal vector. The Phong interpolation method works better than Gouraud shading when applied to a reflection model that has small specular highlights such as the Phong reflection model.

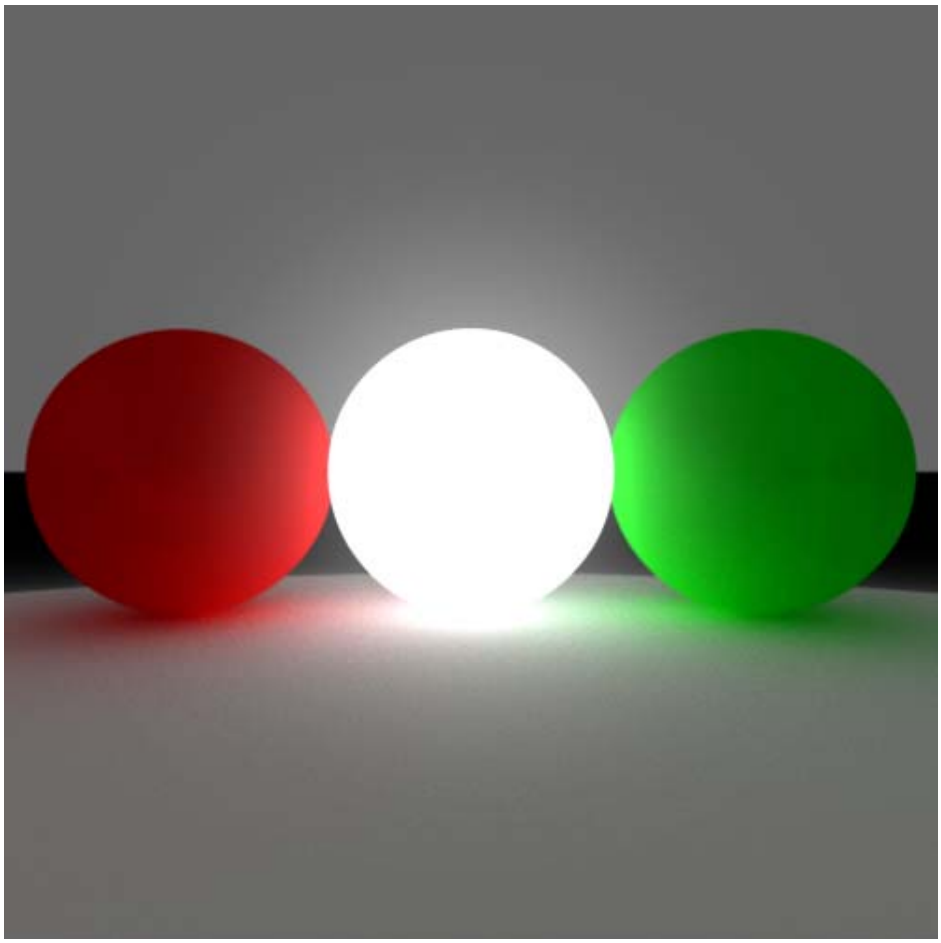
The most serious problem with Gouraud shading occurs when specular highlights are found in the middle of a large polygon. Since these specular highlights are absent from the polygon's vertices and Gouraud shading interpolates based on the vertex colors, the specular highlight will be missing from the polygon's interior. This problem is fixed by Phong shading.

Unlike Gouraud shading, which interpolates colors across polygons, in Phong shading a normal vector is linearly interpolated across the surface of the polygon from the polygon's vertex normals. The surface normal is interpolated and normalized at each pixel and then used in the Phong reflection model to obtain the final pixel color. Phong shading is more computationally expensive than Gouraud shading since the reflection model must be computed at each pixel instead of at each vertex.

In some modern hardware, variants of this algorithm are implemented using pixel or fragment shaders. This can be accomplished by coding normal vectors as secondary colors for each polygon, have the rasterizer use Gouraud shading to interpolate them and interpret them appropriately in the pixel or fragment shader to calculate the light for each pixel based on this normal information.

Chapter 10

Path Tracing



A simple scene showing the soft phenomena simulated with path tracing.

Path tracing is a computer graphics rendering technique that attempts to simulate the physical behaviour of light as closely as possible. It is a generalisation of conventional ray tracing, tracing rays from the virtual camera through several bounces on or through objects. The image quality provided by path tracing is usually superior to that of images

produced using conventional rendering methods at the cost of much greater computation requirements.

Path tracing naturally simulates many effects that have to be specifically added to other methods (ray tracing or scanline rendering), such as soft shadows, depth of field, motion blur, caustics, ambient occlusion, and indirect lighting. Implementation of a renderer including these effects is correspondingly simpler.

Due to its accuracy and unbiased nature, path tracing is used to generate reference images when testing the quality of other rendering algorithms. In order to get high quality images from path tracing, a large number of rays must be traced to avoid visible artifacts in the form of noise.

History

The rendering equation and its use in computer graphics was presented by James Kajiya in 1986. This presentation contained what was probably the first description of the path tracing algorithm. A decade later, Lafortune suggested many refinements, including bidirectional path tracing.

Metropolis light transport, a method of perturbing previously found paths in order to increase performance for difficult scenes, was introduced in 1997 by Eric Veach and Leonidas J. Guibas.

More recently, computers and GPUs have become powerful enough to render images more quickly, causing more widespread interest in path tracing algorithms. Tim Purcell first presented a global illumination algorithm running on a GPU in 2002. In 2009, Vladimir Koylazov demonstrated the first commercial implementation of a path tracer running on a GPU, and other implementations have followed. This was aided by the maturing of GPGPU programming toolkits such as CUDA and OpenCL.

Description

In the real world, many small amounts of light are emitted from light sources, and travel in straight lines (rays) from object to object, changing colour and intensity, until they are absorbed (possibly by an eye or camera). This process is simulated by path tracing, except that the paths are traced backwards, from the camera to the light. The inefficiency arises in the random nature of the bounces from many surfaces, as it is usually quite unlikely that a path will intersect a light. As a result, most traced paths do not contribute to the final image.

This behaviour is described mathematically by the rendering equation, which is the equation that path tracing algorithms try to solve.

Path tracing is not simply ray tracing with infinite recursion depth. In conventional ray tracing, lights are sampled directly when a diffuse surface is hit by a ray. In path tracing,

a new ray is *randomly generated within the hemisphere of the object* and then traced until it hits a light — possibly never. This type of path can hit many diffuse surfaces before interacting with a light.

A simple path tracing pseudocode might look something like this:

```
 Color TracePath(Ray r,depth) {
   if(depth == MaxDepth)
     return Black;  // bounced enough times

   r.FindNearestObject();
   if(r.hitSomething == false)
     return Black;  // nothing was hit

   Material m = r.thingHit->material;
   Color emittance = m.emittance;

   // pick a random direction from here and keep going
   Ray newRay;
   newRay.origin = r.pointWhereObjWasHit;
   newRay.direction =
RandomUnitVectorInHemisphereOf(r.normalWhereObjWasHit);
   float cos_omega = DotProduct(newRay.direction,
r.normalWhereObjWasHit);

   Color BDRF = m.reflectance*cos_omega;
   Color reflected = TracePath(newRay,depth+1);

   return emittance + ( BDRF * cos_omega * reflected );
 }
```

In the above example if every surface of a closed space emitted and reflected (0.5,0.5,0.5) then every pixel in the image would be white.

Bidirectional path tracing

In order to accelerate the convergence of images, bidirectional algorithms trace paths in both directions. In the forward direction, rays are traced from light sources until they are too faint to be seen or strike the camera. In the reverse direction (the usual one), rays are traced from the camera until they strike a light or too many bounces ("depth") have occurred. This approach normally results in an image that converges much more quickly than using only one direction.

Veach and Guibas give a more accurate description:

These methods generate one subpath starting at a light source and another starting at the lens, then they consider all the paths obtained by joining every prefix of one subpath to every suffix of the other. This leads to a family of different importance sampling techniques for paths, which are then combined to minimize variance.

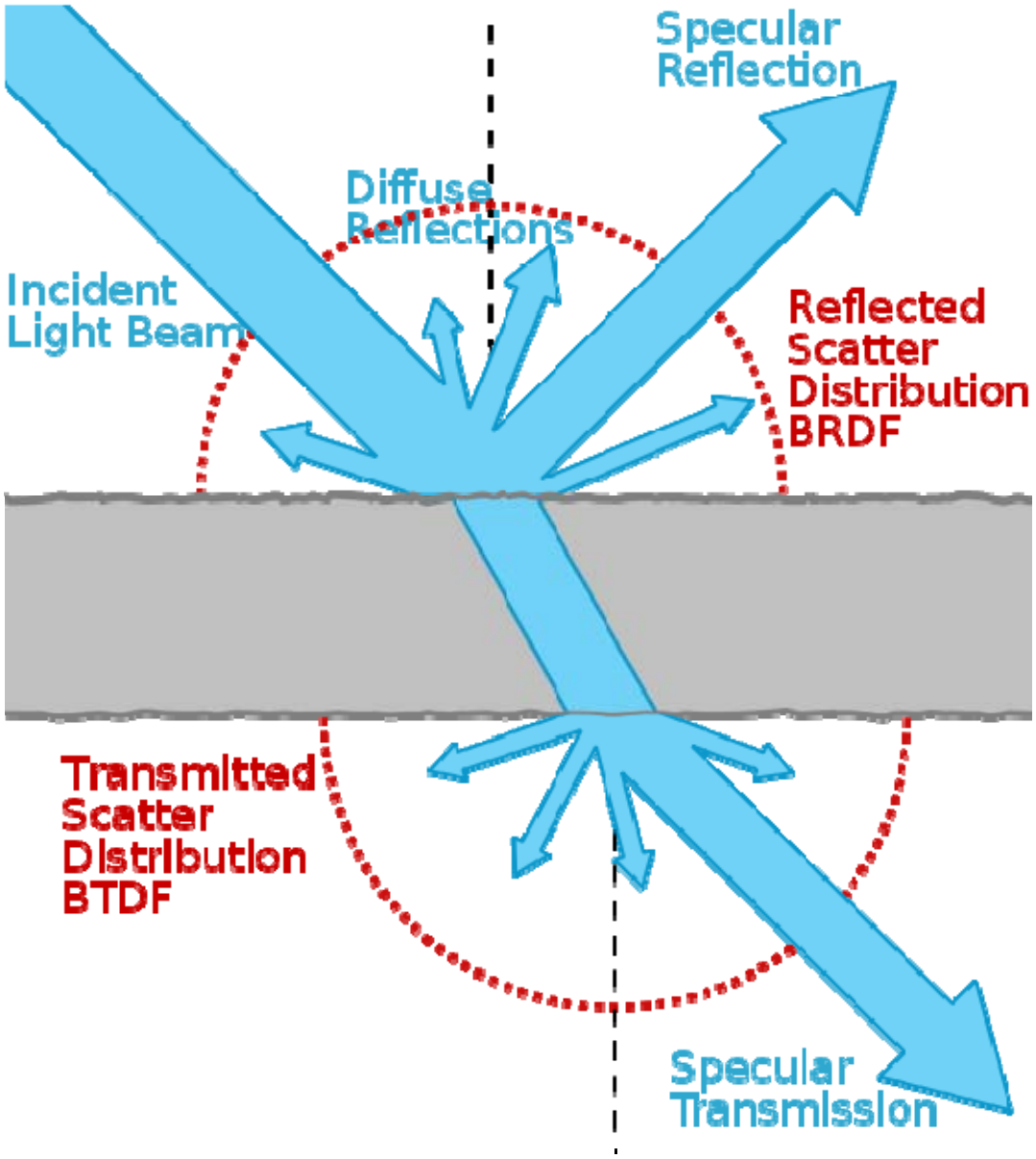
Performance

A path tracer continuously samples pixels of an image. The image starts to become recognisable after only a few samples per pixel, perhaps 100. However, for the image to "converge" and reduce noise to acceptable levels usually takes around 5000 samples for most images, and many more for pathological cases. This can take hours or days depending on scene complexity and hardware and software performance. Newer GPU implementations are promising from 1-10 million samples per second on modern hardware, producing acceptably noise-free images in seconds or minutes. Noise is particularly a problem for animations, giving them a normally-unwanted "film-grain" quality of random speckling.

Metropolis light transport obtains more important samples first, by slightly modifying previously-traced successful paths. This can result in a lower-noise image with fewer samples.

Renderer performance is quite difficult to measure fairly. One approach is to measure "Samples per second", or the number of paths that can be traced and added to the image each second. This varies considerably between scenes and also depends on the "path depth", or how many times a ray is allowed to bounce before it is abandoned. It also depends heavily on the hardware used. Finally, one renderer may generate many low quality samples, while another may converge faster using fewer high-quality samples.

Scattering distribution functions



Scattering distribution functions

The reflective properties (amount, direction and colour) of surfaces are modelled using BRDFs. The equivalent for transmitted light (light that goes through the object) are BTDFs. A path tracer can take full advantage of complex, carefully modelled or measured distribution functions, which controls the appearance ("material", "texture" or "shading" in computer graphics terms) of an object.

Chapter 11

Photon Mapping

In computer graphics, **photon mapping** is a two-pass global illumination algorithm developed by Henrik Wann Jensen that solves the rendering equation. Rays from the light source and rays from the camera are traced independently until some termination criterion is met, then they are connected in a second step to produce a radiance value. It is used to realistically simulate the interaction of light with different objects. Specifically, it is capable of simulating the refraction of light through a transparent substance such as glass or water, diffuse interreflection between illuminated objects, the subsurface scattering of light in translucent materials, and some of the effects caused by particulate matter such as smoke or water vapor. It can also be extended to more accurate simulations of light such as spectral rendering.

Unlike path tracing, bidirectional path tracing and Metropolis light transport, photon mapping is a "biased" rendering algorithm, which means that averaging many renders using this method does not converge to a correct solution to the rendering equation. However, since it is a consistent method, a correct solution can be achieved by increasing the number of photons.

Effects

Caustics

A model of a wine glass ray traced with photon mapping to show caustics.

Light refracted or reflected causes patterns called caustics, usually visible as concentrated patches of light on nearby surfaces. For example, as light rays pass through a wine glass sitting on a table, they are refracted and patterns of light are visible on the table. Photon mapping can trace the paths of individual photons to model where these concentrated patches of light will appear.

Diffuse interreflection

Diffuse interreflection is apparent when light from one diffuse object is reflected onto another. Photon mapping is particularly adept at handling this effect because the algorithm reflects photons from one surface to another based on that surface's bidirectional reflectance distribution function (BRDF), and thus light from one object striking another is a natural result of the method. Diffuse interreflection was first modeled using radiosity solutions. Photon mapping differs though in that it separates the light transport from the nature of the geometry in the scene. Color bleed is an example of diffuse interreflection.

Subsurface scattering

Subsurface scattering is the effect evident when light enters a material and is scattered before being absorbed or reflected in a different direction. Subsurface scattering can accurately be modeled using photon mapping. This was the original way Jensen implemented it; however, the method becomes slow for highly scattering materials, and bidirectional surface scattering reflectance distribution functions (BSSRDFs) are more efficient in these situations.

Usage

Construction of the photon map (1st pass)

With photon mapping, light packets called *photons* are sent out into the scene from the light sources. Whenever a photon intersects with a surface, the intersection point and incoming direction are stored in a cache called the *photon map*. Typically, two photon maps are created for a scene: one especially for caustics and a global one for other light. After intersecting the surface, a probability for either reflecting, absorbing, or transmitting/refracting is given by the material. A Monte Carlo method called *Russian roulette* is used to choose one of these actions. If the photon is absorbed, no new direction is given, and tracing for that photon ends. If the photon reflects, the surface's BRDF is used to determine a new direction. Finally, if the photon is transmitting, a different function for its direction is given depending upon the nature of the transmission.

Once the photon map is constructed (or during construction), it is typically arranged in a manner that is optimal for the k-nearest neighbor algorithm, as photon look-up time depends on the spatial distribution of the photons. Jensen advocates the usage of kd-trees. The photon map is then stored on disk or in memory for later usage.

Rendering (2nd pass)

In this step of the algorithm, the photon map created in the first pass is used to estimate the radiance of every pixel of the output image. For each pixel, the scene is ray traced until the closest surface of intersection is found.

At this point, the rendering equation is used to calculate the surface radiance leaving the point of intersection in the direction of the ray that struck it. To facilitate efficiency, the equation is decomposed into four separate factors: direct illumination, specular reflection, caustics, and soft indirect illumination.

For an accurate estimate of direct illumination, a ray is traced from the point of intersection to each light source. As long as a ray does not intersect another object, the light source is used to calculate the direct illumination. For an approximate estimate of indirect illumination, the photon map is used to calculate the radiance contribution.

Specular reflection can be, in most cases, calculated using ray tracing procedures (as it handles reflections well).

The contribution to the surface radiance from caustics is calculated using the caustics photon map directly. The number of photons in this map must be sufficiently large, as the map is the only source for caustics information in the scene.

For soft indirect illumination, radiance is calculated using the photon map directly. This contribution, however, does not need to be as accurate as the caustics contribution and thus uses the global photon map.

Calculating radiance using the photon map

In order to calculate surface radiance at an intersection point, one of the cached photon maps is used. The steps are:

1. Gather the N nearest photons using the nearest neighbor search function on the photon map.
2. Let S be the sphere that contains these N photons.
3. For each photon, divide the amount of flux (real photons) that the photon represents by the area of S and multiply by the BRDF applied to that photon.
4. The sum of those results for each photon represents total surface radiance returned by the surface intersection in the direction of the ray that struck it.

Optimizations

- To avoid emitting unneeded photons, the initial direction of the outgoing photons is often constrained. Instead of simply sending out photons in random directions, they are sent in the direction of a known object that is a desired photon manipulator to either focus or diffuse the light. There are many other refinements that can be made to the algorithm: for example, choosing the number of photons to send, and where and in what pattern to send them. It would seem that emitting more photons in a specific direction would cause a higher density of photons to be stored in the photon map around the position where the photons hit, and thus measuring this density would give an inaccurate value for irradiance. This is true; however, the algorithm used to compute radiance does *not* depend on irradiance estimates.

- For soft indirect illumination, if the surface is Lambertian, then a technique known as irradiance caching may be used to interpolate values from previous calculations.

- To avoid unnecessary collision testing in direct illumination, shadow photons can be used. During the photon mapping process, when a photon strikes a surface, in addition to the usual operations performed, a shadow photon is emitted in the same direction the original photon came from that goes all the way through the

object. The next object it collides with causes a shadow photon to be stored in the photon map. Then during the direct illumination calculation, instead of sending out a ray from the surface to the light that tests collisions with objects, the photon map is queried for shadow photons. If none are present, then the object has a clear line of sight to the light source and additional calculations can be avoided.

- To optimize image quality, particularly of caustics, Jensen recommends use of a cone filter. Essentially, the filter gives weight to photons' contributions to radiance depending on how far they are from ray-surface intersections. This can produce sharper images.

- Image space photon mapping achieves real-time performance by computing the first and last scattering using a GPU rasterizer.

Variations

- Although photon mapping was designed to work primarily with ray tracers, it can also be extended for use with scanline renderers.

Chapter 12

3D Projection

3D projection is any method of mapping three-dimensional points to a two-dimensional plane. As most current methods for displaying graphical data are based on planar two-dimensional media, the use of this type of projection is widespread, especially in computer graphics, engineering and drafting.

Orthographic projection

When the human eye looks at a scene, objects in the distance appear smaller than objects close by. Orthographic projection ignores this effect to allow the creation of to-scale drawings for construction and engineering.

Orthographic projections are a small set of transforms often used to show profile, detail or precise measurements of a three dimensional object. Common names for orthographic projections include plane, cross-section, bird's-eye, and elevation.

If the normal of the viewing plane (the camera direction) is parallel to one of the 3D axes, the mathematical transformation is as follows; To project the 3D point a_x, a_y, a_z onto the 2D point b_x, b_y using an orthographic projection parallel to the y axis (profile view), the following equations can be used:

$$b_x = s_x a_x + c_x$$
$$b_y = s_z a_z + c_z$$

where the vector **s** is an arbitrary scale factor, and **c** is an arbitrary offset. These constants are optional, and can be used to properly align the viewport. Using matrix multiplication, the equations become:

$$\begin{bmatrix} b_x \\ b_y \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 \\ 0 & 0 & s_z \end{bmatrix} \begin{bmatrix} a_x \\ a_y \\ a_z \end{bmatrix} + \begin{bmatrix} c_x \\ c_z \end{bmatrix}.$$

While orthographically projected images represent the three dimensional nature of the object projected, they do not represent the object as it would be recorded photographically or perceived by a viewer observing it directly. In particular, parallel lengths at all points in an orthographically projected image are of the same scale regardless of whether they are far away or near to the virtual viewer. As a result, lengths near to the viewer are not foreshortened as they would be in a perspective projection.

Perspective projection

When the human eye looks at a scene, objects in the distance appear smaller than objects close by - this is known as perspective. While orthographic projection ignores this effect to allow accurate measurements, perspective definition shows distant objects as smaller to provide additional realism.

The perspective projection requires greater definition. A conceptual aid to understanding the mechanics of this projection involves treating the 2D projection as being viewed through a camera viewfinder. The camera's position, orientation, and field of view control the behavior of the projection transformation. The following variables are defined to describe this transformation:

- $\mathbf{a}_{x,y,z}$ - the 3D position of a point *A* that is to be projected.
- $\mathbf{c}_{x,y,z}$ - the 3D position of a point *C* representing the camera.
- $\theta_{x,y,z}$ - The orientation of the camera (represented, for instance, by Tait–Bryan angles).
- $\mathbf{e}_{x,y,z}$ - the viewer's position relative to the display surface.

Which results in:

- $\mathbf{b}_{x,y}$ - the 2D projection of $\mathbf{a}$.

When $\mathbf{c}_{x,y,z} = (0, 0, 0)$ and $\theta_{x,y,z} = (0, 0, 0)$, the 3D vector $(1, 2, 0)$ is projected to the 2D vector $(1, 2)$.

Otherwise, to compute $\mathbf{b}_{x,y}$ we first define a vector $\mathbf{d}_{x,y,z}$ as the position of point *A* with respect to a coordinate system defined by the camera, with origin in *C* and rotated by θ with respect to the initial coordinate system. This is achieved by subtracting $\mathbf{c}$ from $\mathbf{a}$ and then applying a rotation by $-\theta$ to the result. This transformation is often called a **camera transform**, and can be expressed as follows, expressing the rotation in terms of rotations about the *x*, *y*, and *z* axes (these calculations assume that the axes are ordered as a left-handed system of axes):

$$\begin{bmatrix} \mathbf{d}_x \\ \mathbf{d}_y \\ \mathbf{d}_z \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta_x & -\sin\theta_x \\ 0 & \sin\theta_x & \cos\theta_x \end{bmatrix} \begin{bmatrix} \cos\theta_y & 0 & \sin\theta_y \\ 0 & 1 & 0 \\ -\sin\theta_y & 0 & \cos\theta_y \end{bmatrix} \begin{bmatrix} \cos\theta_z & -\sin\theta_z & 0 \\ \sin\theta_z & \cos\theta_z & 0 \\ 0 & 0 & 1 \end{bmatrix} \left(\begin{bmatrix} \mathbf{a}_x \\ \mathbf{a}_y \\ \mathbf{a}_z \end{bmatrix} - \begin{bmatrix} \mathbf{c}_x \\ \mathbf{c}_y \\ \mathbf{c}_z \end{bmatrix} \right)$$

This representation corresponds to rotating by three Euler angles (more properly, Tait–Bryan angles), using the *xyz* convention, which can be interpreted either as "rotate about the *extrinsic* axes (axes of the *scene*) in the order *z, y, x* (reading right-to-left)" or "rotate about the *intrinsic* axes (axes of the *camera*) in the order *x, y, z*) (reading left-to-right)". Note that if the camera is not rotated ($\theta_{x,y,z} = (0, 0, 0)$), then the matrices drop out (as identities), and this reduces to simply a shift: $\mathbf{d} = \mathbf{a} - \mathbf{c}$.

Alternatively, without using matrices, (note that the signs of angles are inconsistent with matrix form):

$$
\begin{aligned}
d_x &= \cos\theta_y \cdot (\sin\theta_z \cdot (a_y - c_y) + \cos\theta_z \cdot (a_x - c_x)) - \sin\theta_y \cdot (a_z - c_z) \\
d_y &= \sin\theta_x \cdot (\cos\theta_y \cdot (a_z - c_z) + \sin\theta_y \cdot (\sin\theta_z \cdot (a_y - c_y) + \cos\theta_z \cdot (a_x - c_x))) + \cos\theta_x \cdot (\cos\theta_z \cdot (a_y - c_y) - \sin\theta_z \cdot (a_x - c_x)) \\
d_z &= \cos\theta_x \cdot (\cos\theta_y \cdot (a_z - c_z) + \sin\theta_y \cdot (\sin\theta_z \cdot (a_y - c_y) + \cos\theta_z \cdot (a_x - c_x))) - \sin\theta_x \cdot (\cos\theta_z \cdot (a_y - c_y) - \sin\theta_z \cdot (a_x - c_x))
\end{aligned}
$$

This transformed point can then be projected onto the 2D plane using the formula (here, *x/y* is used as the projection plane, literature also may use *x/z*):

$$
\begin{aligned}
\mathbf{b}_x &= (\mathbf{d}_x - \mathbf{e}_x)(\mathbf{e}_z / \mathbf{d}_z) \\
\mathbf{b}_y &= (\mathbf{d}_y - \mathbf{e}_y)(\mathbf{e}_z / \mathbf{d}_z)
\end{aligned}
.
$$

Or, in matrix form using homogeneous coordinates, the system

$$
\begin{bmatrix} \mathbf{f}_x \\ \mathbf{f}_y \\ \mathbf{f}_z \\ \mathbf{f}_w \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & -\mathbf{e}_x \\ 0 & 1 & 0 & -\mathbf{e}_y \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/\mathbf{e}_z & 0 \end{bmatrix} \begin{bmatrix} \mathbf{d}_x \\ \mathbf{d}_y \\ \mathbf{d}_z \\ 1 \end{bmatrix}
$$

in conjunction with an argument using similar triangles, leads to division by the homogeneous coordinate, giving

$$
\begin{aligned}
\mathbf{b}_x &= \mathbf{f}_x / \mathbf{f}_w \\
\mathbf{b}_y &= \mathbf{f}_y / \mathbf{f}_w
\end{aligned}
.
$$

The distance of the viewer from the display surface, $\mathbf{e}_z$, directly relates to the field of view, where $\alpha = 2 \cdot \tan^{-1}(1/\mathbf{e}_z)$ is the viewed angle. (Note: This assumes that you map the points (-1,-1) and (1,1) to the corners of your viewing surface)

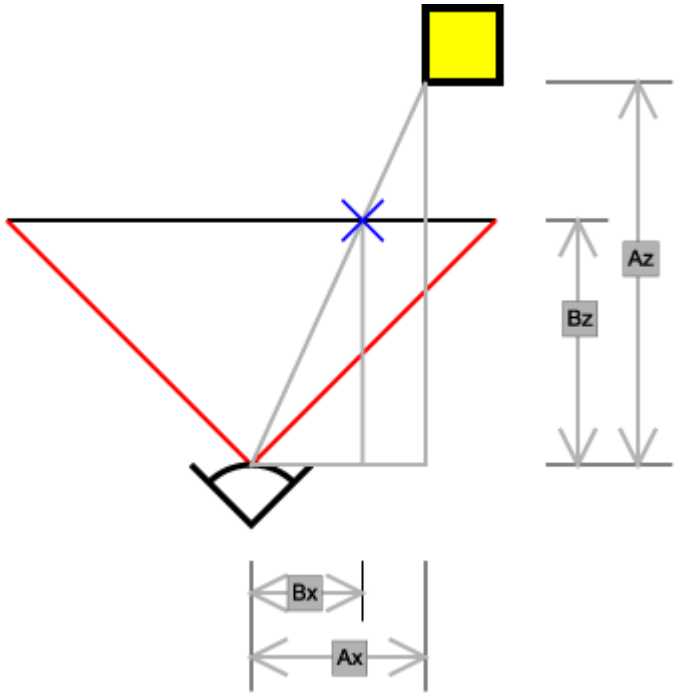
The above equations can also be rewritten as:

$$
\begin{aligned}
\mathbf{b}_x &= (\mathbf{d}_x \mathbf{s}_x) / (\mathbf{d}_z \mathbf{r}_x) \mathbf{r}_z \\
\mathbf{b}_y &= (\mathbf{d}_y \mathbf{s}_y) / (\mathbf{d}_z \mathbf{r}_y) \mathbf{r}_z
\end{aligned}
.
$$

In which $\mathbf{s}_{x,y}$ is the display size, $\mathbf{r}_{x,y}$ is the recording surface size (CCD or film), $\mathbf{r}_z$ is the distance from the recording surface to the aperture, and $\mathbf{d}_z$ is the distance from the point to the aperture.

Subsequent clipping and scaling operations may be necessary to map the 2D plane onto any particular display media.

Diagram



To determine which screen x-coordinate corresponds to a point at Ax,Az multiply the point coordinates by:

$$\text{screen x-coordinate}(Bx) = \text{model x-coordinate}(Ax) \times \frac{\text{distance from eye to screen}(Bz)}{\text{distance from eye to point}(Az)}$$

the same works for the screen y-coordinate:

$$\text{screen y-coordinate}(By) = \text{model y-coordinate}(Ay) \times \frac{\text{distance from eye to screen}(Bz)}{\text{distance from eye to point}(Az)}$$

(where Ax and Ay are coordinates occupied by the object before the perspective transform)

Chapter 13

Radiosity (3D Computer Graphics)

Screenshot of scene rendered with RRV (simple implementation of radiosity renderer based on OpenGL) 79th iteration.

Radiosity is a global illumination algorithm used in 3D computer graphics rendering. Radiosity is an application of the finite element method to solving the rendering equation for scenes with purely diffuse surfaces. Unlike Monte Carlo algorithms (such as path tracing) which handle all types of light paths, typical radiosity methods only account for paths which leave a light source and are reflected diffusely some number of times (possibly zero) before hitting the eye. Such paths are represented as "LD*E". Radiosity calculations are viewpoint independent which increases the computations involved, but makes them useful for all viewpoints.

Radiosity methods were first developed in about 1950 in the engineering field of heat transfer. They were later refined specifically for application to the problem of rendering computer graphics in 1984 by researchers at Cornell University.

Notable commercial radiosity engines are Lightscape (now incorporated into the Autodesk 3D Studio Max internal render engine), form•Z RenderZone Plus by AutoDesSys, Inc.), and ElAS (Electric Image Animation System).

Visual characteristics

Difference between standard direct illumination and radiosity

The inclusion of radiosity calculations in the rendering process often lends an added element of realism to the finished scene, because of the way it mimics real-world phenomena. Consider a simple room scene.

The image on the left was rendered with a typical **direct illumination renderer**. There are *three types* of lighting in this scene which have been specifically chosen and placed by the artist in an attempt to create realistic lighting: **spot lighting** with shadows (placed outside the window to create the light shining on the floor), **ambient lighting** (without which any part of the room not lit directly by a light source would be totally dark), and **omnidirectional lighting** without shadows (to reduce the flatness of the ambient lighting).

The image on the right was rendered using a **radiosity algorithm**. There is only **one source of light**: an image of the sky placed outside the window. The difference is marked. The room glows with light. Soft shadows are visible on the floor, and subtle lighting effects are noticeable around the room. Furthermore, the red color from the carpet has bled onto the grey walls, giving them a slightly warm appearance. None of these effects were specifically chosen or designed by the artist.

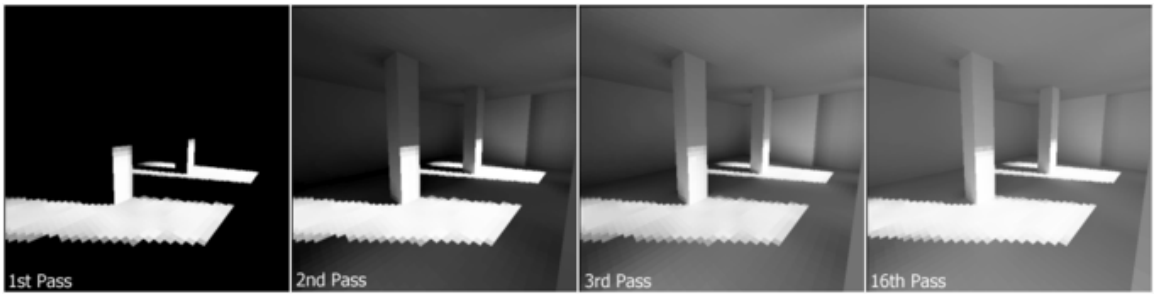
Overview of the radiosity algorithm

The surfaces of the scene to be rendered are each divided up into one or more smaller surfaces (patches). A view factor is computed for each pair of patches. View factors (also known as *form factors*) are coefficients describing how well the patches can see each other. Patches that are far away from each other, or oriented at oblique angles relative to

one another, will have smaller view factors. If other patches are in the way, the view factor will be reduced or zero, depending on whether the occlusion is partial or total.

The view factors are used as coefficients in a linearized form of the rendering equation, which yields a linear system of equations. Solving this system yields the radiosity, or brightness, of each patch, taking into account diffuse interreflections and soft shadows.

Progressive radiosity solves the system iteratively in such a way that after each iteration we have intermediate radiosity values for the patch. These intermediate values correspond to bounce levels. That is, after one iteration, we know how the scene looks after one light bounce, after two passes, two bounces, and so forth. Progressive radiosity is useful for getting an interactive preview of the scene. Also, the user can stop the iterations once the image looks good enough, rather than wait for the computation to numerically converge.



As the algorithm iterates, light can be seen to flow into the scene, as multiple bounces are computed. Individual patches are visible as squares on the walls and floor.

Another common method for solving the radiosity equation is "shooting radiosity," which iteratively solves the radiosity equation by "shooting" light from the patch with the most error at each step. After the first pass, only those patches which are in direct line of sight of a light-emitting patch will be illuminated. After the second pass, more patches will become illuminated as the light begins to bounce around the scene. The scene continues to grow brighter and eventually reaches a steady state.

Mathematical formulation

The basic radiosity method has its basis in the theory of thermal radiation, since radiosity relies on computing the amount of light energy transferred among surfaces. In order to simplify computations, the method assumes that all scattering is perfectly diffuse. Surfaces are typically discretized into quadrilateral or triangular elements over which a piecewise polynomial function is defined.

After this breakdown, the amount of light energy transfer can be computed by using the known reflectivity of the reflecting patch, combined with the view factor of the two patches. This dimensionless quantity is computed from the geometric orientation of two

patches, and can be thought of as the fraction of the total possible emitting area of the first patch which is covered by the second patch.

More correctly, radiosity is the energy leaving the patch surface per discrete time interval and is the combination of emitted and reflected energy:

$$B_i \, dA_i = E_i \, dA_i + R_i \int_j B_j F_{ji} \, dA_j,$$

where:

- B_i is the radiosity of patch i.
- E_i is emitted energy.
- R_i is the reflectivity of the patch, giving reflected energy by multiplying by the incident energy (the energy which arrives from other patches).
- All j ($j \neq i$) in the rendered environment are integrated for $B_j F_{ji} \, dA_j$, to determine the energy leaving each patch j that arrives at patch i.
- F_{ij} is the constant-valued view factor for the radiation leaving i and hitting patch j.

The reciprocity:

$$A_i F_{ij} = A_j F_{ji}$$

gives:

$$B_i = E_i + R_i \int_j B_j F_{ij}$$

For ease of use the integral is replaced and uniform radiosity is assumed over the patch, creating the simpler:

$$B_i = E_i + R_i \sum_{j=1}^{n} B_j F_{ij}$$

This equation can then be applied to each patch. The equation is monochromatic, so color radiosity rendering requires calculation for each of the required colors.

The view factor F_{ji} can be calculated in a number of ways. Early methods used a *hemicube* (an imaginary cube centered upon the first surface to which the second surface was projected, devised by Cohen and Greenberg in 1985) to approximate the form factor, which also solved the intervening patch problem. This is quite computationally expensive, because ideally form factors must be derived for every possible pair of patches, leading to a quadratic increase in computation with added geometry. New methods include adaptive integration

Reducing computation time

Although in its basic form radiosity is assumed to have a quadratic increase in computation time with added geometry (surfaces and patches), this need not be the case. The radiosity problem can be rephrased as a problem of rendering a texture mapped scene. In this case, the computation time increases only linearly with the number of patches (ignoring complex issues like cache use). Using a binary space partitioning tree can massively reduce the amount of time spent determining which patches are completely hidden from others in complex scenes.

Following the commercial enthusiasm for radiosity-enhanced imagery, but prior to the standardization of rapid radiosity calculation, many architects and graphic artists used a technique referred to loosely as false radiosity. By darkening areas of texture maps corresponding to corners, joints and recesses, and applying them via self-illumination or diffuse mapping, a radiosity-like effect of patch interaction could be created with a standard scanline renderer (cf. ambient occlusion).

Since radiosity can now be computed more effectively using texture mapping algorithms, it lends itself to acceleration using standard graphics acceleration hardware, widely available for all types of computers.

Advantages

A modern render of the iconic Utah teapot. **Radiosity** was used for all diffuse illumination in this scene.

One of the advantages of the Radiosity algorithm is that it is relatively simple to explain and implement. This makes it a useful algorithm for teaching students about global illumination algorithms. A typical direct illumination renderer already contains nearly all of the algorithms (perspective transformations, texture mapping, hidden surface removal) required to implement radiosity. A strong grasp of mathematics is not required to understand or implement this algorithm.

Limitations

Typical radiosity methods only account for light paths of the form LD*E, i.e., paths which start at a light source and make multiple diffuse bounces before reaching the eye. Although there are several approaches to integrating other illumination effects such as specular and glossy reflections, radiosity-based methods are generally not used to solve the complete rendering equation.

Basic radiosity also has trouble resolving sudden changes in visibility (e.g., hard-edged shadows) because coarse, regular discretization into piecewise constant elements corresponds to a low-pass box filter of the spatial domain. Discontinuity meshing uses knowledge of visibility events to generate a more intelligent discretization.

Confusion about terminology

Radiosity was perhaps the first rendering algorithm in widespread use which accounted for diffuse indirect lighting. Earlier rendering algorithms, such as Whitted-style ray tracing were capable of computing effects such as reflections, refractions, and shadows, but despite being highly global phenomena, these effects were not commonly referred to as "global illumination." As a consequence, the term "global illumination" became confused with "diffuse interreflection," and "Radiosity" became confused with "global illumination" in popular parlance. However, the three are distinct concepts.

The radiosity method in the current computer graphics context derives from (and is fundamentally the same as) the radiosity method in heat transfer. In this context radiosity is the total radiative flux (both reflected and re-radiated) leaving a surface, also sometimes known as radiant exitance. Calculation of Radiosity rather than surface temperatures is a key aspect of the radiosity method that permits linear matrix methods to be applied to the problem.

Chapter 14

Reflection Mapping & Reflection (Computer Graphics)

Reflection Mapping

An example of reflection mapping.

In computer graphics, **environment mapping**, or **reflection mapping**, is an efficient Image-based lighting technique for approximating the appearance of a reflective surface by means of a precomputed texture image. The texture is used to store the image of the distant environment surrounding the rendered object.

Several ways of storing the surrounding environment are employed. The first technique was **sphere mapping**, in which a single texture contains the image of the surroundings as reflected on a mirror ball. It has been almost entirely surpassed by **cube mapping**, in which the environment is projected onto the six faces of a cube and stored as six square textures or *unfolded* into six square regions of a single texture. Other projections that have some superior mathematical or computational properties include the **paraboloid mapping**, the **pyramid mapping**, the **octahedron mapping**, and the **HEALPix mapping**.

The reflection mapping approach is more efficient than the classical ray tracing approach of computing the exact reflection by tracing a ray and following its optical path. The reflection color used in the shading computation at a pixel is determined by calculating the reflection vector at the point on the object and mapping it to the texel in the environment map. This technique often produces results that are superficially similar to those generated by raytracing, but is less computationally expensive since the radiance value of the reflection comes from calculating the angles of incidence and reflection, followed by a texture lookup, rather than followed by tracing a ray against the scene geometry and computing the radiance of the ray, simplifying the GPU workload.

However in most circumstances a mapped reflection is only an approximation of the real reflection. Environment mapping relies on two assumptions that are seldom satisfied:

1) All radiance incident upon the object being shaded comes from an infinite distance. When this is not the case the reflection of nearby geometry appears in the wrong place on the reflected object. When this is the case, no parallax is seen in the reflection.

2) The object being shaded is convex, such that it contains no self-interreflections. When this is not the case the object does not appear in the reflection; only the environment does.

Reflection mapping is also a traditional Image-based lighting technique for creating reflections of real-world backgrounds on synthetic objects.

Environment mapping is generally the fastest method of rendering a reflective surface. To further increase the speed of rendering, the renderer may calculate the position of the reflected ray at each vertex. Then, the position is interpolated across polygons to which the vertex is attached. This eliminates the need for recalculating every pixel's reflection direction.

If normal mapping is used, each polygon has many face normals (the direction a given point on a polygon is facing), which can be used in tandem with an environment map to produce a more realistic reflection. In this case, the angle of reflection at a given point on a polygon will take the normal map into consideration. This technique is used to make an otherwise flat surface appear textured, for example corrugated metal, or brushed aluminium.

Types of reflection mapping

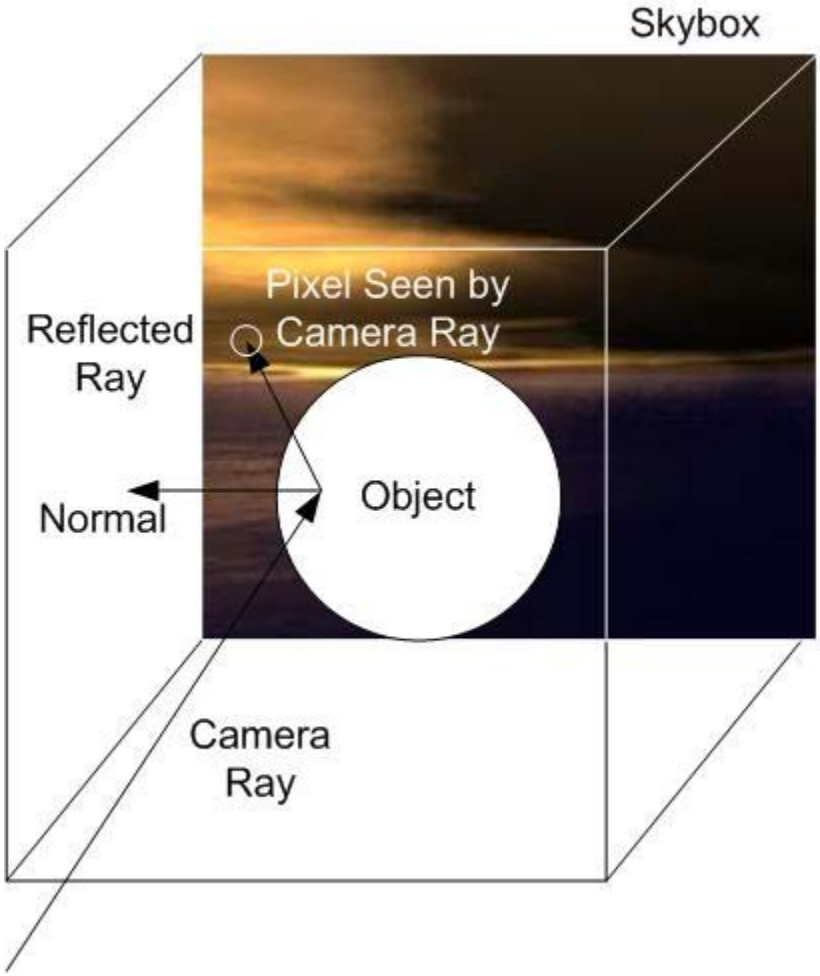
Sphere mapping

Sphere mapping represents the sphere of incident illumination as though it were seen in the reflection of a reflective sphere through an orthographic camera. The texture image can be created by approximating this ideal setup, or using a fisheye lens or via prerendering a scene with a spherical mapping.

The spherical mapping suffers from limitations that detract from the realism of resulting renderings. Because spherical maps are stored as azimuthal projections of the environments they represent, an abrupt point of singularity (a “black hole” effect) is visible in the reflection on the object where texel colors at or near the edge of the map are distorted due to inadequate resolution to represent the points accurately. The spherical mapping also wastes pixels that are in the square but not in the sphere.

The artifacts of the spherical mapping are so severe that it is effective only for viewpoints near that of the virtual orthographic camera.

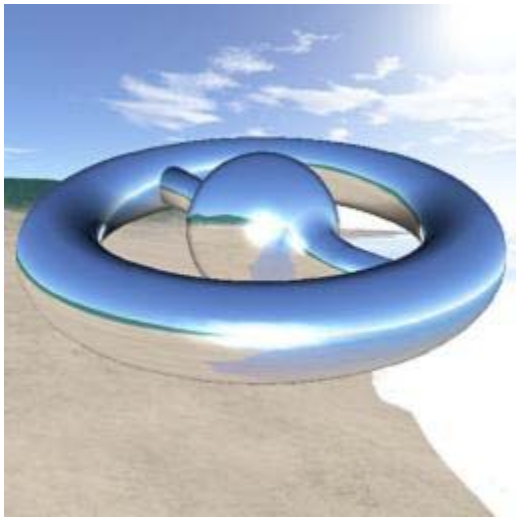
Cube mapping



A diagram depicting an apparent reflection being provided by cube mapped reflection. The map is actually projected onto the surface from the point of view of the observer. Highlights which in raytracing would be provided by tracing the ray and determining the angle made with the normal, can be 'fudged', if they are manually painted into the texture field (or if they already appear there depending on how the texture map was obtained), from where they will be projected onto the mapped object along with the rest of the texture detail.

Cube mapping and other polyhedron mappings address the severe distortion of sphere maps. If cube maps are made and filtered correctly, they have no visible seams, and can be used independent of the viewpoint of the often-virtual camera acquiring the map. Cube and other polyhedron maps have since superseded sphere maps in most computer graphics applications, with the exception of acquiring image-based lighting.

Generally, cube mapping uses the same skybox that is used in outdoor renderings. Cube mapped reflection is done by determining the vector that the object is being viewed at. This **camera ray** is reflected about the surface normal of where the camera vector intersects the object. This results in the **reflected ray** which is then passed to the cube map to get the texel which provides the radiance value used in the lighting calculation. This creates the effect that the object is reflective.



Example of a three-dimensional model using cube mapped reflection

HEALPix mapping

HEALPix environment mapping is similar to the other polyhedron mappings, but can be hierarchical, thus providing a unified framework for generating polyhedra that better approximate the sphere. This allows lower distortion at the cost of increased computation.

History

Precursor work in texture mapping had been established by Edwin Catmull, with refinements for curved surfaces by James Blinn, in 1974. Blinn went on to further refine his work, developing environment mapping by 1976.

Gene Miller experimented with spherical environment mapping in 1982 at MAGI Synthavision.

Wolfgang Heidrich introduced Paraboloid Mapping in 1998.

Emil Praun introduced Octahedron Mapping in 2003 .

Mauro Steigleder introduced Pyramid Mapping in 2005 .

Tien-Tsin Wong, et al. introduced the existing HEALPix mapping for rendering in 2006.

Reflection (Computer Graphics)

Ray traced model demonstrating specular reflection.

Reflection in computer graphics is used to emulate reflective objects like mirrors and shiny surfaces.

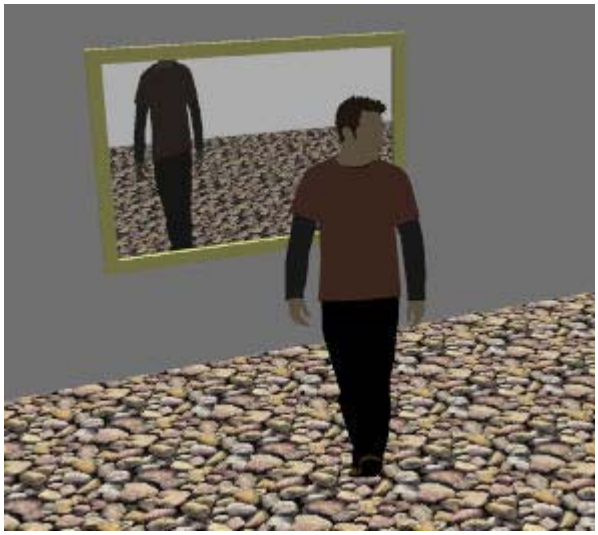
Reflection is accomplished in a ray trace renderer by following a ray from the eye to the mirror and then calculating where it bounces from, and continuing the process until no surface is found, or a non-reflective surface is found. Reflection on a shiny surface like wood or tile can add to the photorealistic effects of a 3D rendering.

- **Polished** - A Polished Reflection is an undisturbed reflection, like a mirror or chrome.
- **Blurry** - A Blurry Reflection means that tiny random bumps on the surface of the material cause the reflection to be blurry.

- **Metallic** - A reflection is Metallic if the highlights and reflections retain the color of the reflective object.
- **Glossy** - This term can be misused. Sometimes it is a setting which is the opposite of Blurry. (When "Glossiness" has a low value, the reflection is blurry.) However, some people use the term "Glossy Reflection" as a synonym for "Blurred Reflection." Glossy used in this context means that the reflection is actually blurred.

Examples

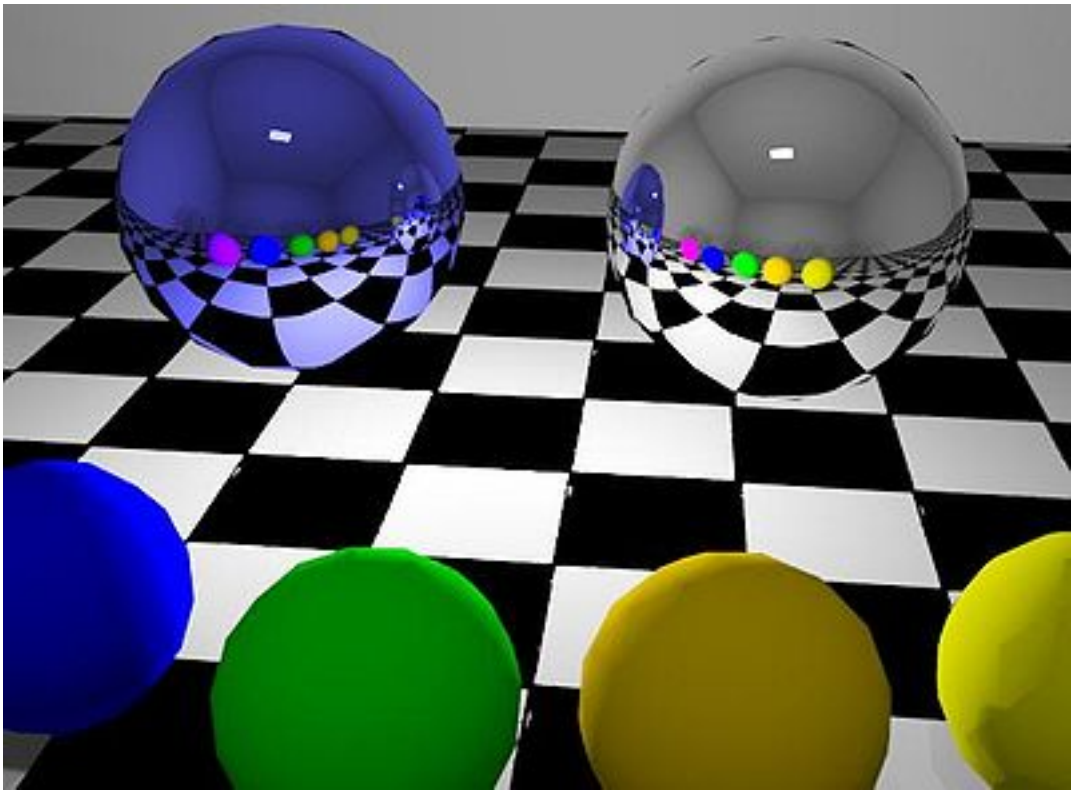
Polished or Mirror reflection



Mirror on wall rendered with 100% reflection.

Mirrors are usually almost 100% reflective.

Metallic Reflection

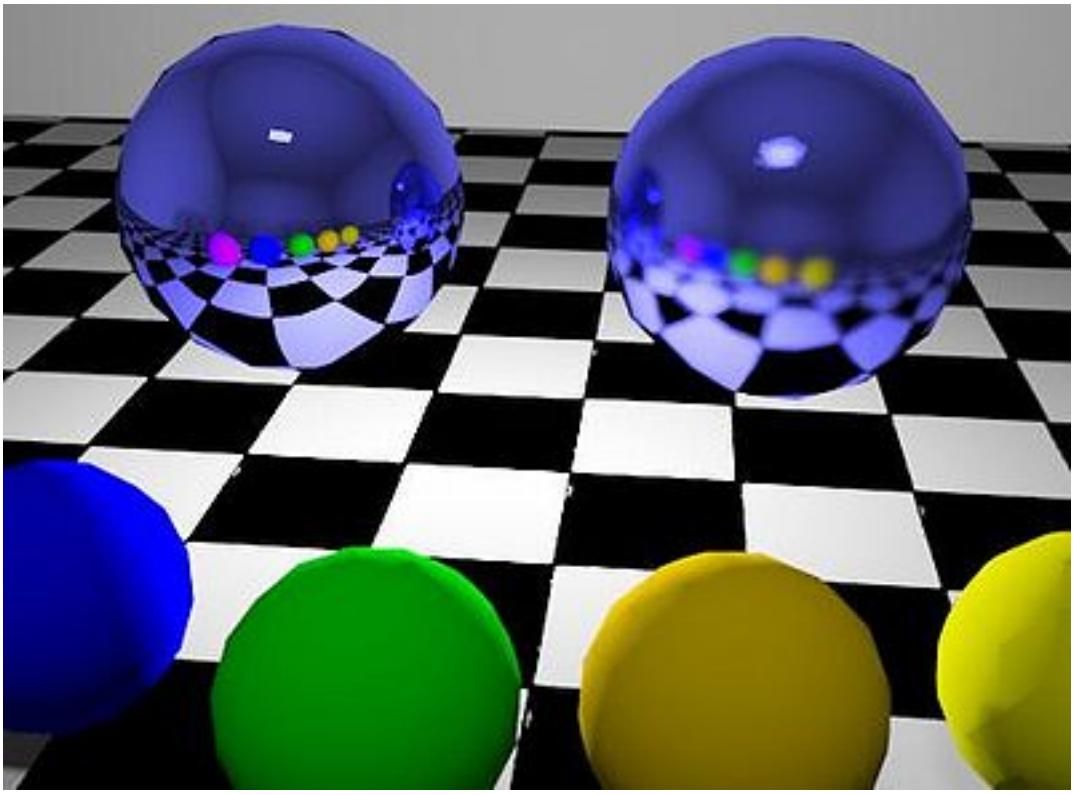


The large sphere on the left is blue with its reflection marked as metallic. The large sphere on the right is the same color but does not have the metallic property selected.

Normal, (non metallic), objects reflect light and colors in the original color of the object being reflected.

Metallic objects reflect lights and colors altered by the color of the metallic object itself.

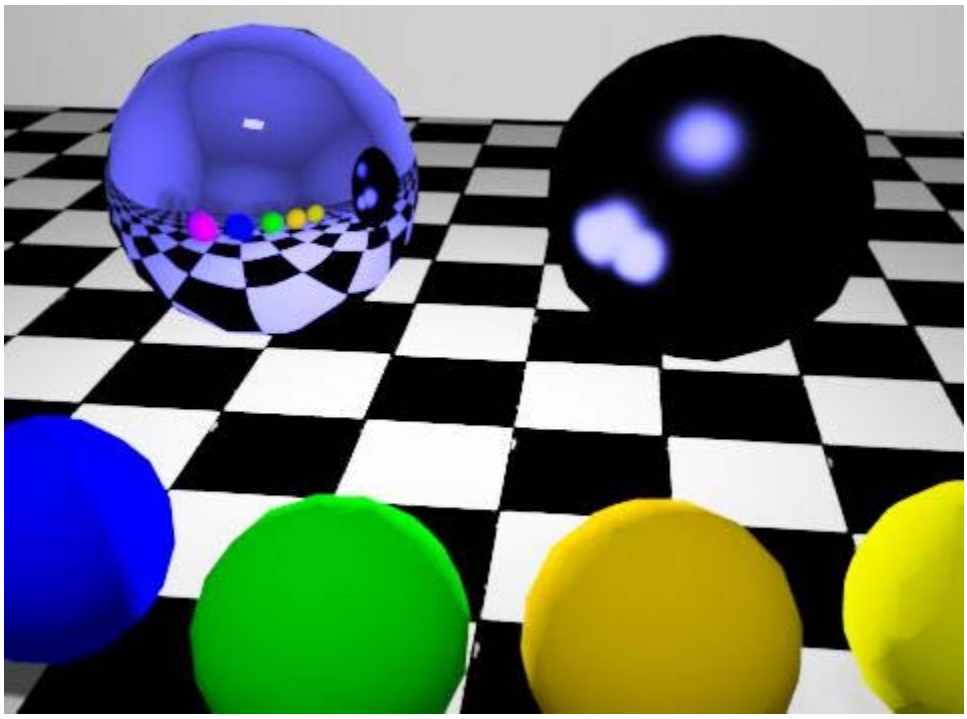
Blurry Reflection



The large sphere on the left has sharpness set to 100%. The sphere on the right has sharpness set to 50% which creates a blurry reflection.

Many materials are imperfect reflectors, where the reflections are blurred to various degrees due to surface roughness that scatters the rays of the reflections.

Glossy Reflection



The sphere on the left has normal, metallic reflection. The sphere on the right has the same parameters, except that the reflection is marked as "glossy".

Chapter 15

Rendering (Computer Graphics)

An image created by using POV-Ray 3.6.

Rendering is the process of generating an image from a model (or models in what collectively could be called a *scene* file), by means of computer programs. A scene file contains objects in a strictly defined language or data structure; it would contain geometry, viewpoint, texture, lighting, and shading information as a description of the virtual scene. The data contained in the scene file is then passed to a rendering program to be processed and output to a digital image or raster graphics image file. The term

"rendering" may be by analogy with an "artist's rendering" of a scene. Though the technical details of rendering methods vary, the general challenges to overcome in producing a 2D image from a 3D representation stored in a scene file are outlined as the graphics pipeline along a rendering device, such as a GPU. A GPU is a purpose-built device able to assist a CPU in performing complex rendering calculations. If a scene is to look relatively realistic and predictable under virtual lighting, the rendering software should solve the rendering equation. The rendering equation doesn't account for all lighting phenomena, but is a general lighting model for computer-generated imagery. 'Rendering' is also used to describe the process of calculating effects in a video editing file to produce final video output.

Rendering is one of the major sub-topics of 3D computer graphics, and in practice always connected to the others. In the graphics pipeline, it is the last major step, giving the final appearance to the models and animation. With the increasing sophistication of computer graphics since the 1970s onward, it has become a more distinct subject.

Rendering has uses in architecture, video games, simulators, movie or TV special effects, and design visualization, each employing a different balance of features and techniques. As a product, a wide variety of renderers are available. Some are integrated into larger modeling and animation packages, some are stand-alone, some are free open-source projects. On the inside, a renderer is a carefully engineered program, based on a selective mixture of disciplines related to: light physics, visual perception, mathematics and software development.

In the case of 3D graphics, rendering may be done slowly, as in pre-rendering, or in real time. Pre-rendering is a computationally intensive process that is typically used for movie creation, while real-time rendering is often done for 3D video games which rely on the use of graphics cards with 3D hardware accelerators.

Usage

When the pre-image (a wireframe sketch usually) is complete, rendering is used, which adds in bitmap textures or procedural textures, lights, bump mapping and relative position to other objects. The result is a completed image the consumer or intended viewer sees.

For movie animations, several images (frames) must be rendered, and stitched together in a program capable of making an animation of this sort. Most 3D image editing programs can do this.

Features

Image rendered with computer aided design.

A rendered image can be understood in terms of a number of visible features. Rendering research and development has been largely motivated by finding ways to simulate these efficiently. Some relate directly to particular algorithms and techniques, while others are produced together.

- shading — how the color and brightness of a surface varies with lighting
- texture-mapping — a method of applying detail to surfaces
- bump-mapping — a method of simulating small-scale bumpiness on surfaces

- fogging/participating medium — how light dims when passing through non-clear atmosphere or air
- shadows — the effect of obstructing light
- soft shadows — varying darkness caused by partially obscured light sources
- reflection — mirror-like or highly glossy reflection
- transparency (optics), transparency (graphic) or opacity — sharp transmission of light through solid objects
- translucency — highly scattered transmission of light through solid objects
- refraction — bending of light associated with transparency
- diffraction — bending, spreading and interference of light passing by an object or aperture that disrupts the ray
- indirect illumination — surfaces illuminated by light reflected off other surfaces, rather than directly from a light source (also known as global illumination)
- caustics (a form of indirect illumination) — reflection of light off a shiny object, or focusing of light through a transparent object, to produce bright highlights on another object
- depth of field — objects appear blurry or out of focus when too far in front of or behind the object in focus
- motion blur — objects appear blurry due to high-speed motion, or the motion of the camera
- non-photorealistic rendering — rendering of scenes in an artistic style, intended to look like a painting or drawing

Techniques

Many rendering algorithms have been researched, and software used for rendering may employ a number of different techniques to obtain a final image.

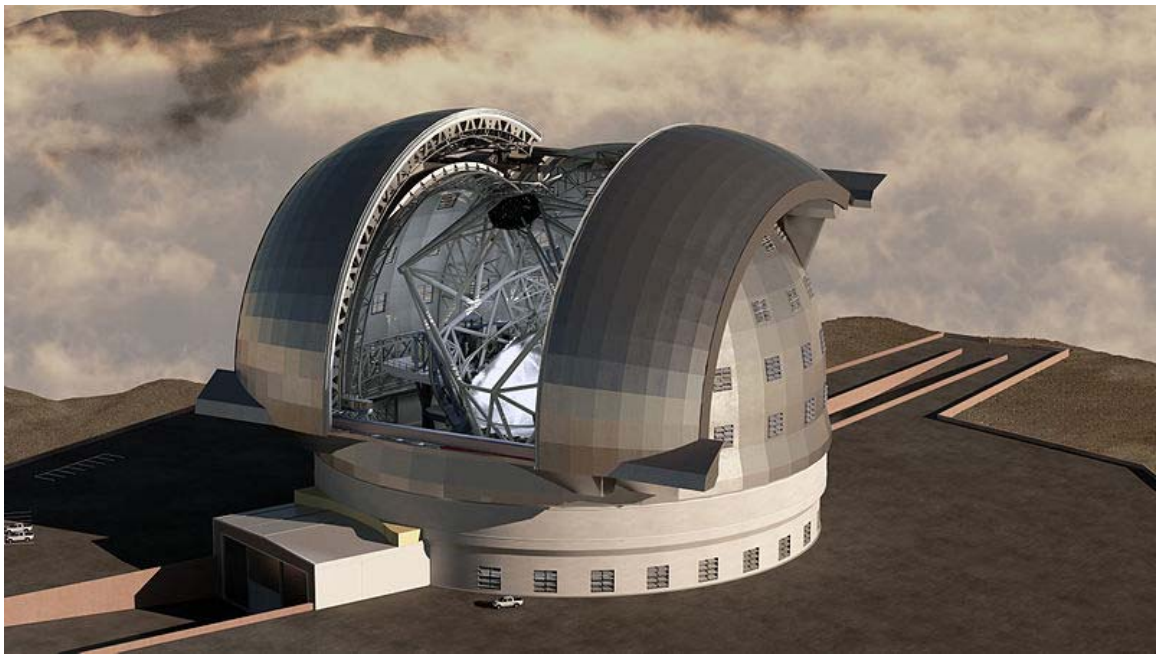
Tracing every particle of light in a scene is nearly always completely impractical and would take a stupendous amount of time. Even tracing a portion large enough to produce an image takes an inordinate amount of time if the sampling is not intelligently restricted.

Therefore, four loose families of more-efficient light transport modelling techniques have emerged: rasterization, including scanline rendering, geometrically projects objects in the scene to an image plane, without advanced optical effects; ray casting considers the scene as observed from a specific point-of-view, calculating the observed image based only on geometry and very basic optical laws of reflection intensity, and perhaps using Monte Carlo techniques to reduce artifacts; and ray tracing is similar to ray casting, but employs more advanced optical simulation, and usually uses Monte Carlo techniques to obtain more realistic results at a speed that is often orders of magnitude slower. The fourth type of light transport techique, radiosity is not usually implemented as a rendering technique, but instead calculates the passage of light as it leaves the light source and illuminates surfaces. These surfaces are usually rendered to the display using one of the other three techniques.

Most advanced software combines two or more of the techniques to obtain good-enough results at reasonable cost.

Another distinction is between image order algorithms, which iterate over pixels of the image plane, and object order algorithms, which iterate over objects in the scene. Generally object order is more efficient, as there are usually fewer objects in a scene than pixels.

Scanline rendering and rasterisation



Rendering of the European Extremely Large Telescope.

A high-level representation of an image necessarily contains elements in a different domain from pixels. These elements are referred to as primitives. In a schematic drawing, for instance, line segments and curves might be primitives. In a graphical user interface, windows and buttons might be the primitives. In 3D rendering, triangles and polygons in space might be primitives.

If a pixel-by-pixel (image order) approach to rendering is impractical or too slow for some task, then a primitive-by-primitive (object order) approach to rendering may prove useful. Here, one loops through each of the primitives, determines which pixels in the image it affects, and modifies those pixels accordingly. This is called **rasterization**, and is the rendering method used by all current graphics cards.

Rasterization is frequently faster than pixel-by-pixel rendering. First, large areas of the image may be empty of primitives; rasterization will ignore these areas, but pixel-by-pixel rendering must pass through them. Second, rasterization can improve cache coherency and reduce redundant work by taking advantage of the fact that the pixels

occupied by a single primitive tend to be contiguous in the image. For these reasons, rasterization is usually the approach of choice when interactive rendering is required; however, the pixel-by-pixel approach can often produce higher-quality images and is more versatile because it does not depend on as many assumptions about the image as rasterization.

The older form of rasterization is characterized by rendering an entire face (primitive) as a single color. Alternatively, rasterization can be done in a more complicated manner by first rendering the vertices of a face and then rendering the pixels of that face as a blending of the vertex colors. This version of rasterization has overtaken the old method as it allows the graphics to flow without complicated textures (a rasterized image when used face by face tends to have a very block-like effect if not covered in complex textures; the faces aren't smooth because there is no gradual color change from one primitive to the next). This newer method of rasterization utilizes the graphics card's more taxing shading functions and still achieves better performance because the simpler textures stored in memory use less space. Sometimes designers will use one rasterization method on some faces and the other method on others based on the angle at which that face meets other joined faces, thus increasing speed and not hurting the overall effect.

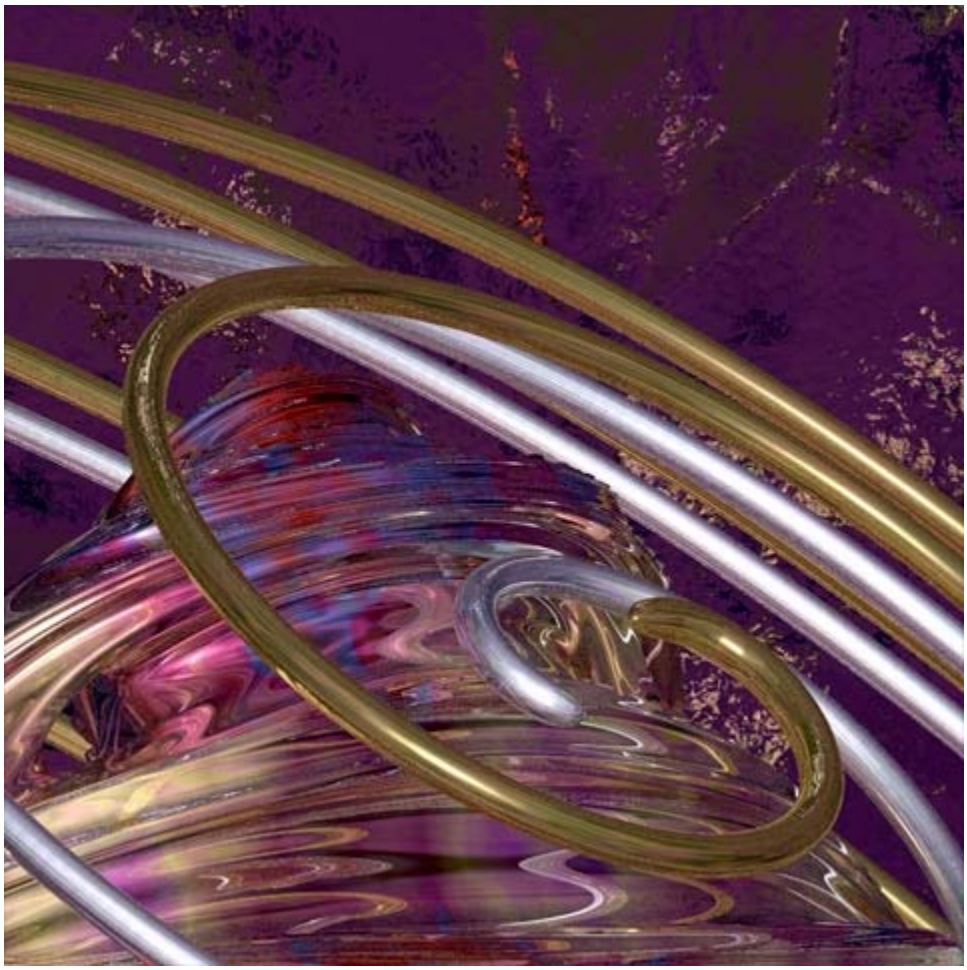
Ray casting

In **ray casting** the geometry which has been modeled is parsed pixel by pixel, line by line, from the point of view outward, as if casting rays out from the point of view. Where an object is intersected, the color value at the point may be evaluated using several methods. In the simplest, the color value of the object at the point of intersection becomes the value of that pixel. The color may be determined from a texture-map. A more sophisticated method is to modify the colour value by an illumination factor, but without calculating the relationship to a simulated light source. To reduce artifacts, a number of rays in slightly different directions may be averaged.

Rough simulations of optical properties may be additionally employed: a simple calculation of the ray from the object to the point of view is made. Another calculation is made of the angle of incidence of light rays from the light source(s), and from these as well as the specified intensities of the light sources, the value of the pixel is calculated. Another simulation uses illumination plotted from a radiosity algorithm, or a combination of these two.

Raycasting is primarily used for realtime simulations, such as those used in 3D computer games and cartoon animations, where detail is not important, or where it is more efficient to manually fake the details in order to obtain better performance in the computational stage. This is usually the case when a large number of frames need to be animated. The resulting surfaces have a characteristic 'flat' appearance when no additional tricks are used, as if objects in the scene were all painted with matte finish.

Ray tracing



Spiral Sphere and Julia, Detail, a computer-generated image created by visual artist Robert W. McGregor using only POV-Ray 3.6 and its built-in scene description language.

Ray tracing aims to simulate the natural flow of light, interpreted as particles. Often, ray tracing methods are utilized to approximate the solution to the rendering equation by applying Monte Carlo methods to it. Some of the most used methods are Path Tracing, Bidirectional Path Tracing, or Metropolis light transport, but also semi realistic methods are in use, like Whitted Style Ray Tracing, or hybrids. While most implementations let light propagate on straight lines, applications exist to simulate relativistic spacetime effects.

In a final, production quality rendering of a ray traced work, multiple rays are generally shot for each pixel, and traced not just to the first object of intersection, but rather, through a number of sequential 'bounces', using the known laws of optics such as "angle of incidence equals angle of reflection" and more advanced laws that deal with refraction and surface roughness.

Once the ray either encounters a light source, or more probably once a set limiting number of bounces has been evaluated, then the surface illumination at that final point is evaluated using techniques described above, and the changes along the way through the various bounces evaluated to estimate a value observed at the point of view. This is all repeated for each sample, for each pixel.

In distribution ray tracing, at each point of intersection, multiple rays may be spawned. In path tracing, however, only a single ray or none is fired at each intersection, utilizing the statistical nature of Monte Carlo experiments.

As a brute-force method, ray tracing has been too slow to consider for real-time, and until recently too slow even to consider for short films of any degree of quality, although it has been used for special effects sequences, and in advertising, where a short portion of high quality (perhaps even photorealistic) footage is required.

However, efforts at optimizing to reduce the number of calculations needed in portions of a work where detail is not high or does not depend on ray tracing features have led to a realistic possibility of wider use of ray tracing. There is now some hardware accelerated ray tracing equipment, at least in prototype phase, and some game demos which show use of real-time software or hardware ray tracing.

Radiosity

Radiosity is a method which attempts to simulate the way in which directly illuminated surfaces act as indirect light sources that illuminate other surfaces. This produces more realistic shading and seems to better capture the 'ambience' of an indoor scene. A classic example is the way that shadows 'hug' the corners of rooms.

The optical basis of the simulation is that some diffused light from a given point on a given surface is reflected in a large spectrum of directions and illuminates the area around it.

The simulation technique may vary in complexity. Many renderings have a very rough estimate of radiosity, simply illuminating an entire scene very slightly with a factor known as ambiance. However, when advanced radiosity estimation is coupled with a high quality ray tracing algorithim, images may exhibit convincing realism, particularly for indoor scenes.

In advanced radiosity simulation, recursive, finite-element algorithms 'bounce' light back and forth between surfaces in the model, until some recursion limit is reached. The colouring of one surface in this way influences the colouring of a neighbouring surface, and vice versa. The resulting values of illumination throughout the model (sometimes including for empty spaces) are stored and used as additional inputs when performing calculations in a ray-casting or ray-tracing model.

Due to the iterative/recursive nature of the technique, complex objects are particularly slow to emulate. Prior to the standardization of rapid radiosity calculation, some graphic artists used a technique referred to loosely as false radiosity by darkening areas of texture maps corresponding to corners, joints and recesses, and applying them via self-illumination or diffuse mapping for scanline rendering. Even now, advanced radiosity calculations may be reserved for calculating the ambiance of the room, from the light reflecting off walls, floor and ceiling, without examining the contribution that complex objects make to the radiosity—or complex objects may be replaced in the radiosity calculation with simpler objects of similar size and texture.

Radiosity calculations are viewpoint independent which increases the computations involved, but makes them useful for all viewpoints. If there is little rearrangement of radiosity objects in the scene, the same radiosity data may be reused for a number of frames, making radiosity an effective way to improve on the flatness of ray casting, without seriously impacting the overall rendering time-per-frame.

Because of this, radiosity is a prime component of leading real-time rendering methods, and has been used from beginning-to-end to create a large number of well-known recent feature-length animated 3D-cartoon films.

Sampling and filtering

One problem that any rendering system must deal with, no matter which approach it takes, is the **sampling problem**. Essentially, the rendering process tries to depict a continuous function from image space to colors by using a finite number of pixels. As a consequence of the Nyquist–Shannon sampling theorem, any spatial waveform that can be displayed must consist of at least two pixels, which is proportional to image resolution. In simpler terms, this expresses the idea that an image cannot display details, peaks or troughs in color or intensity, that are smaller than one pixel.

If a naive rendering algorithm is used without any filtering, high frequencies in the image function will cause ugly aliasing to be present in the final image. Aliasing typically manifests itself as jaggies, or jagged edges on objects where the pixel grid is visible. In order to remove aliasing, all rendering algorithms (if they are to produce good-looking images) must use some kind of low-pass filter on the image function to remove high frequencies, a process called antialiasing.

Optimization

Optimizations used by an artist when a scene is being developed

Due to the large number of calculations, a work in progress is usually only rendered in detail appropriate to the portion of the work being developed at a given time, so in the initial stages of modeling, wireframe and ray casting may be used, even where the target output is ray tracing with radiosity. It is also common to render only parts of the scene at

high detail, and to remove objects that are not important to what is currently being developed.

Common optimizations for real time rendering

For real-time, it is appropriate to simplify one or more common approximations, and tune to the exact parameters of the scenery in question, which is also tuned to the agreed parameters to get the most 'bang for the buck'.

Academic core

The implementation of a realistic renderer always has some basic element of physical simulation or emulation — some computation which resembles or abstracts a real physical process.

The term "*physically-based*" indicates the use of physical models and approximations that are more general and widely accepted outside rendering. A particular set of related techniques have gradually become established in the rendering community.

The basic concepts are moderately straightforward, but intractable to calculate; and a single elegant algorithm or approach has been elusive for more general purpose renderers. In order to meet demands of robustness, accuracy and practicality, an implementation will be a complex combination of different techniques.

Rendering research is concerned with both the adaptation of scientific models and their efficient application.

The rendering equation

This is the key academic/theoretical concept in rendering. It serves as the most abstract formal expression of the non-perceptual aspect of rendering. All more complete algorithms can be seen as solutions to particular formulations of this equation.

$$L_o(x, \vec{w}) = L_e(x, \vec{w}) + \int_\Omega f_r(x, \vec{w}', \vec{w}) L_i(x, \vec{w}')(\vec{w}' \cdot \vec{n}) d\vec{w}'$$

Meaning: at a particular position and direction, the outgoing light (L_o) is the sum of the emitted light (L_e) and the reflected light. The reflected light being the sum of the incoming light (L_i) from all directions, multiplied by the surface reflection and incoming angle. By connecting outward light to inward light, via an interaction point, this equation stands for the whole 'light transport' — all the movement of light — in a scene.

The Bidirectional Reflectance Distribution Function

The **Bidirectional Reflectance Distribution Function** (BRDF) expresses a simple model of light interaction with a surface as follows:

$$f_r(x, \vec{w}', \vec{w}) = \frac{dL_r(x, \vec{w})}{L_i(x, \vec{w}')(\vec{w}' \cdot \vec{n})d\vec{w}'}$$

Light interaction is often approximated by the even simpler models: diffuse reflection and specular reflection, although both can be BRDFs.

Geometric optics

Rendering is practically exclusively concerned with the particle aspect of light physics — known as geometric optics. Treating light, at its basic level, as particles bouncing around is a simplification, but appropriate: the wave aspects of light are negligible in most scenes, and are significantly more difficult to simulate. Notable wave aspect phenomena include diffraction (as seen in the colours of CDs and DVDs) and polarisation (as seen in LCDs). Both types of effect, if needed, are made by appearance-oriented adjustment of the reflection model.

Visual perception

Though it receives less attention, an understanding of human visual perception is valuable to rendering. This is mainly because image displays and human perception have restricted ranges. A renderer can simulate an almost infinite range of light brightness and color, but current displays — movie screen, computer monitor, etc. — cannot handle so much, and something must be discarded or compressed. Human perception also has limits, and so does not need to be given large-range images to create realism. This can help solve the problem of fitting images into displays, and, furthermore, suggest what short-cuts could be used in the rendering simulation, since certain subtleties won't be noticeable. This related subject is tone mapping.

Mathematics used in rendering includes: linear algebra, calculus, numerical mathematics, signal processing, and Monte Carlo methods.

Rendering for movies often takes place on a network of tightly connected computers known as a render farm.

The current state of the art in 3-D image description for movie creation is the Mental Ray scene description language designed at mental images and the RenderMan shading language designed at Pixar. (compare with simpler 3D fileformats such as VRML or APIs such as OpenGL and DirectX tailored for 3D hardware accelerators).

Other renderers (including proprietary ones) can and are sometimes used, but most other renderers tend to miss one or more of the often needed features like good texture filtering, texture caching, programmable shaders, highend geometry types like hair, subdivision or nurbs surfaces with tesselation on demand, geometry caching, raytracing with geometry caching, high quality shadow mapping, speed or patent-free implementations. Other highly sought features these days may include IPR and hardware rendering/shading.