3D Rendering and Types of Mapping in Computer Graphics

Patria Dobbins
Harriett Badillo

First Edition, 2012

ISBN 978-81-323-1263-5

Published by:
College Publishing House
4735/22 Prakashdeep Bldg,
Ansari Road, Darya Ganj,
Delhi - 110002
Email: info@wtbooks.com

Table of Contents

Chapter 1 - Anisotropic Filtering & Ambient Occlusion

Chapter 2 - Binary Space Partitioning

Chapter 3 - Bump Mapping

Chapter 4 - Global Illumination & Catmull–Clark Subdivision Surface

Chapter 5 - Level of Detail

Chapter 6 - Non-Uniform Rational B-Spline

Chapter 7 - Normal Mapping & Mipmap

Chapter 8 - Particle System & Painter's Algorithm

Chapter 9 - Phong Shading

Chapter 10 - Path Tracing

Chapter 11 - Photon Mapping

Chapter 12 - Texture Mapping

Chapter 13 - Cube Mapping

Chapter 14 - Displacement Mapping and Parallax Mapping

Chapter 15 - UV Mapping, Sphere Mapping and UVW Mapping

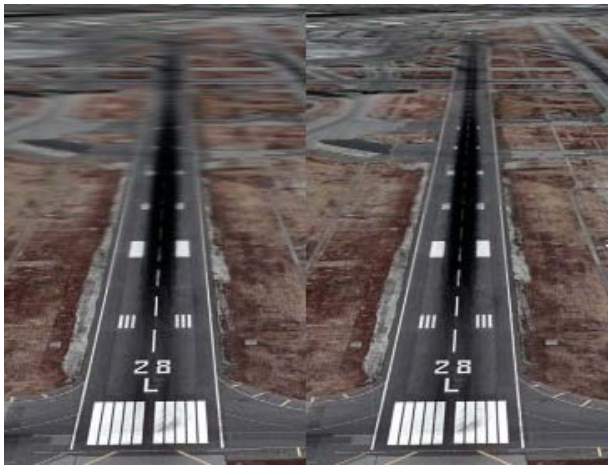
Chapter 16 - 3D Projection and Texture Filtering

Chapter 17 - Image Resolution

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.

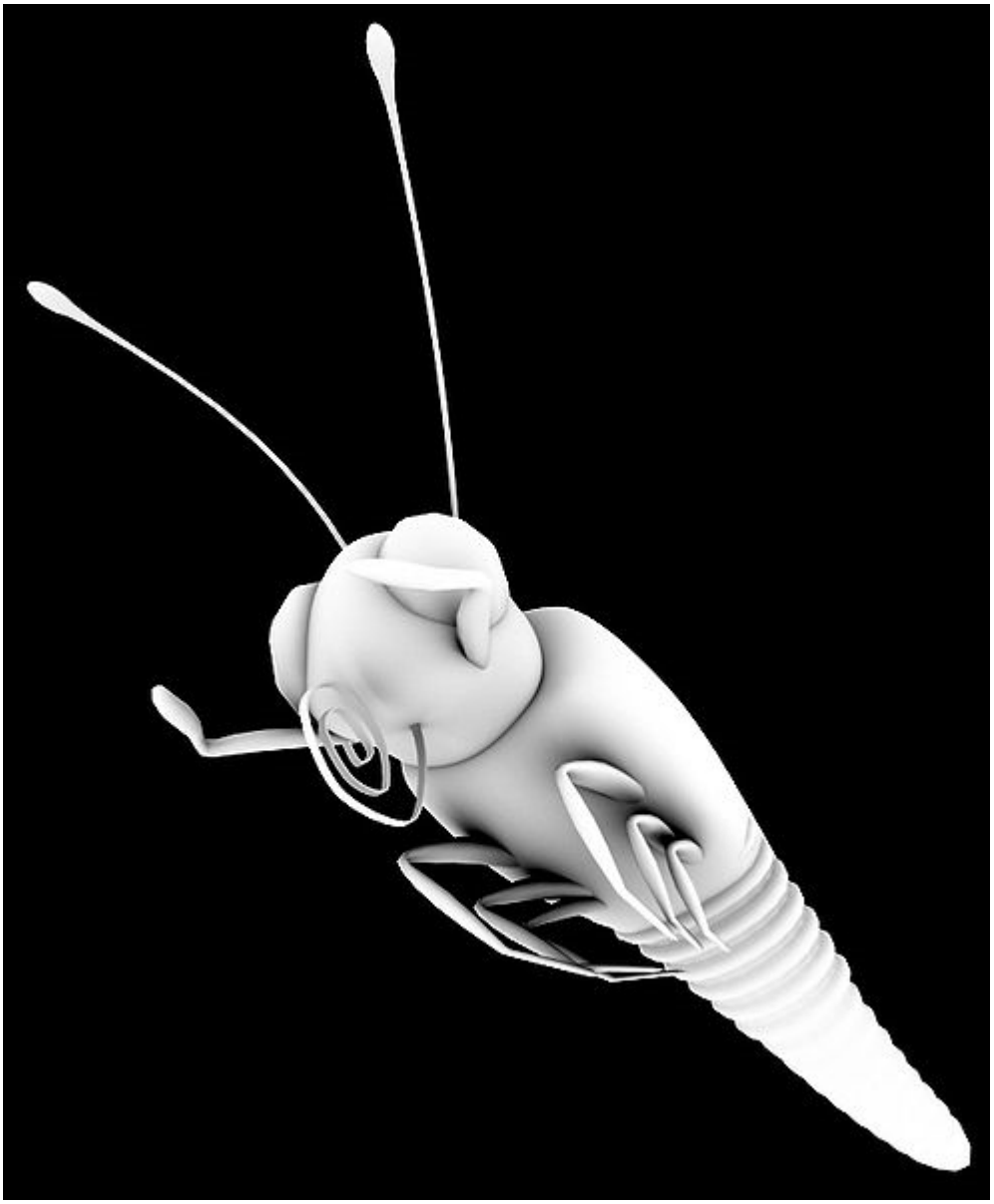
Unlike local methods like Phong shading, ambient occlusion is a global method, meaning the illumination at each point is a function of other geometry in the scene. However, it is a very crude approximation to full global illumination. The soft appearance achieved by ambient occlusion alone is similar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

Method of implementation

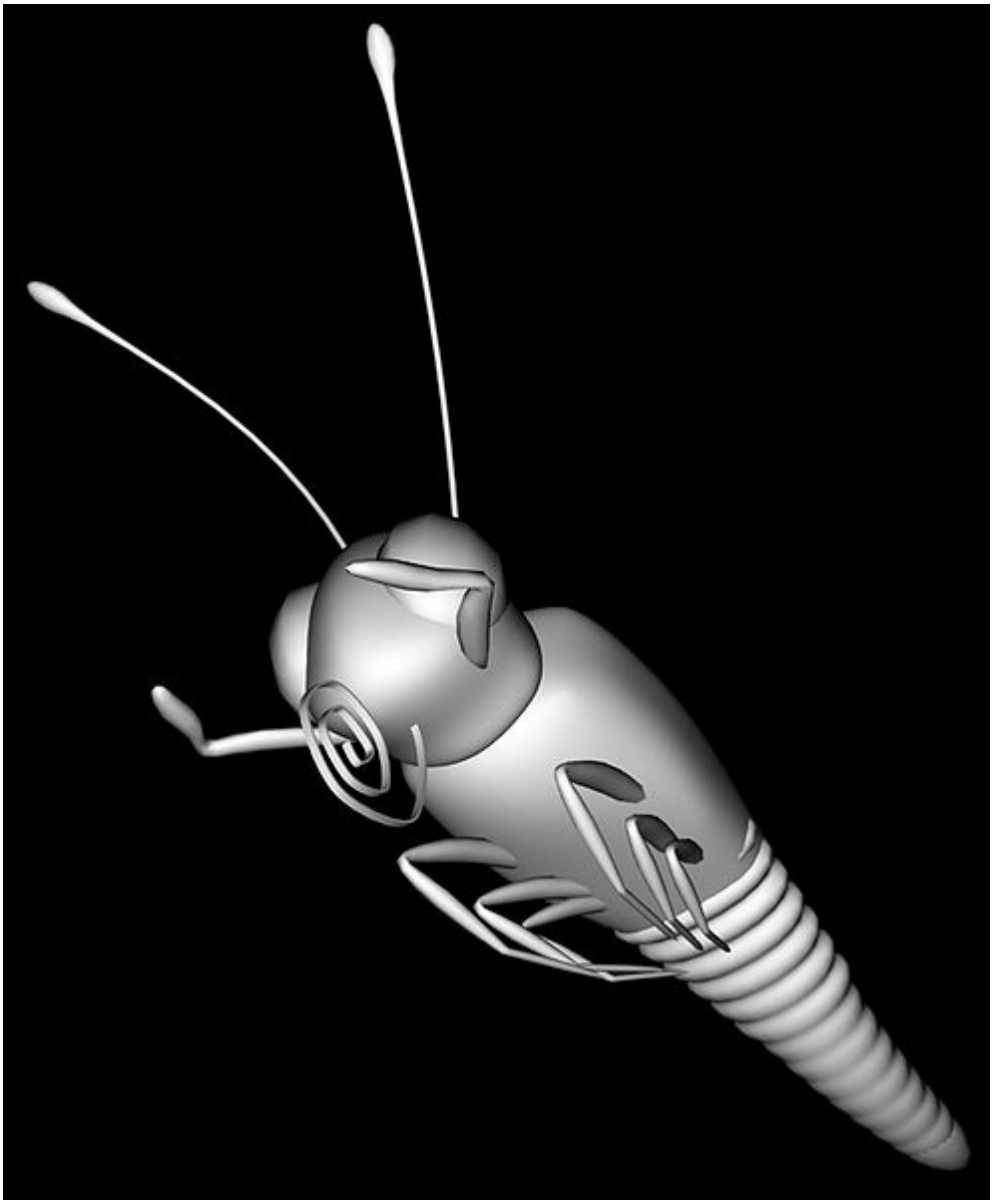
Ambient occlusion is most often calculated by casting rays in every direction from the surface. Rays which reach the background or “sky” increase the brightness of the surface, whereas a ray which hits any other object contributes no illumination. As a result, points surrounded by a large amount of geometry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

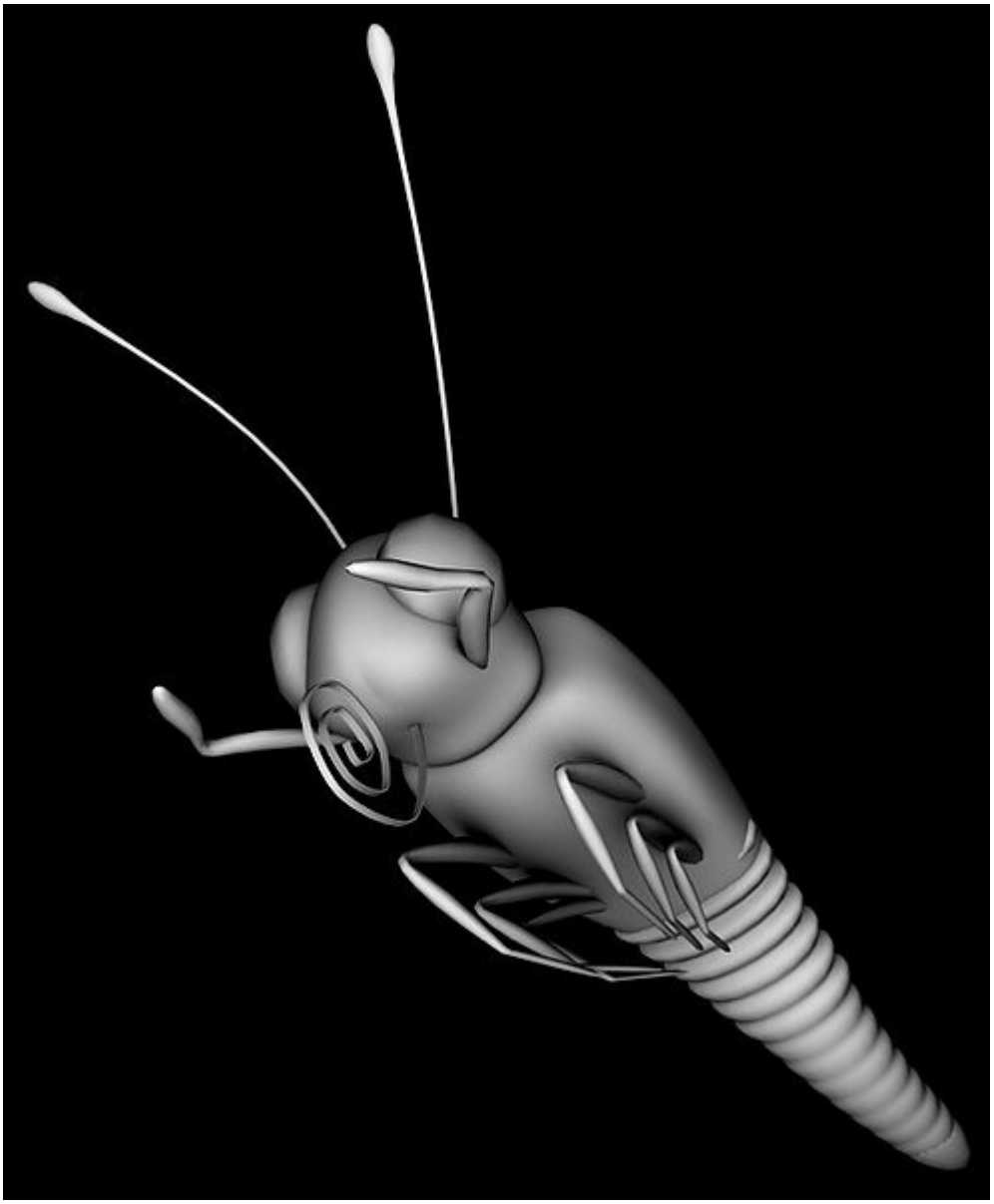
The ambient occlusion shading model has the nice property of offering a better perception of the 3d shape of the displayed objects. This was shown in a paper where the authors report the results of perceptual experiments showing that depth discrimination under diffuse uniform sky lighting is superior to that predicted by a direct lighting model.



ambient occlusion



diffuse only



combined ambient and diffuse

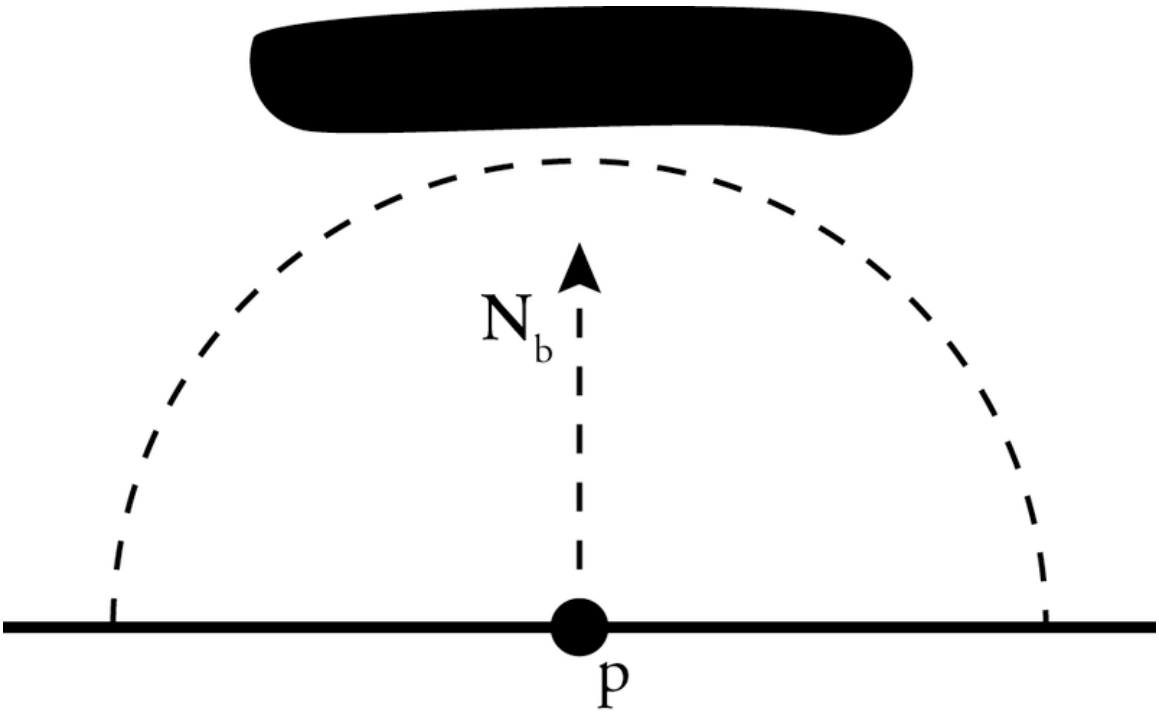
The occlusion $A_{\bar{p}}$ at a point $\bar{p}$ on a surface with normal $\hat{n}$ can be computed by integrating the visibility function over the hemisphere Ω with respect to projected solid angle:

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

where $V_{\bar{p},\omega}$ 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.

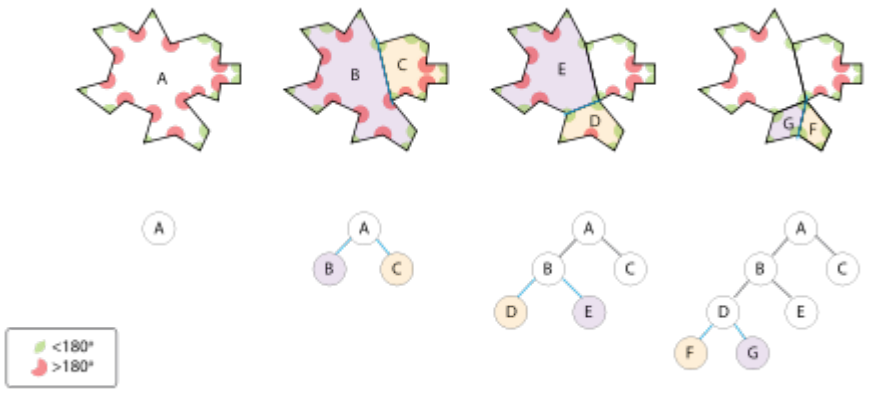
BSP trees are often used by 3D computer games, particularly first-person shooters and those with indoor environments. Probably the earliest game to use a BSP data structure was *Doom*. Other uses include ray tracing and collision detection.

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*.

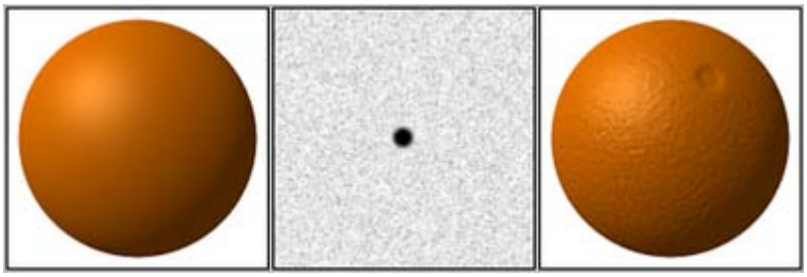
- 1992 Teller's PhD thesis described the efficient generation of potentially visible sets as a pre-processing step to acceleration real-time visible surface determination in arbitrary 3D polygonal environments. This was used in *Quake* and contributed significantly to that game's performance.

- 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

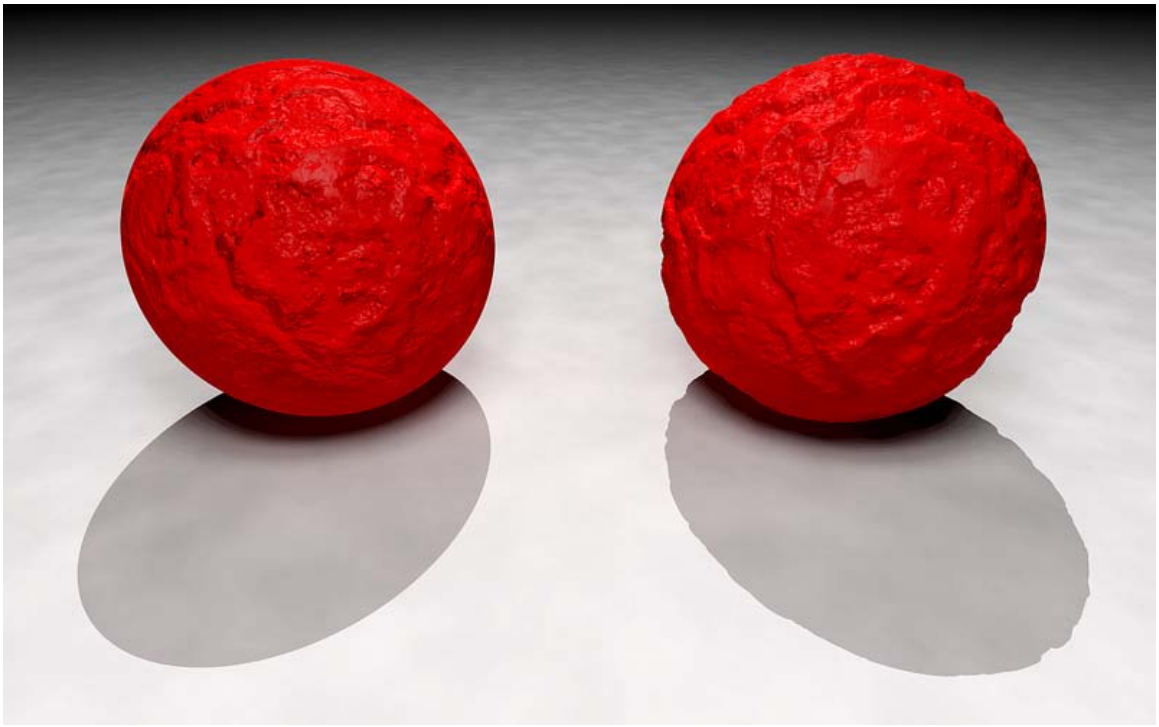


A sphere without bump mapping (left). A bump map to be applied to the sphere (middle). The sphere with the bump map applied (right) appears to have a mottled surface resembling an orange. Bump maps achieve this effect by changing how an illuminated surface reacts to light without actually modifying the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

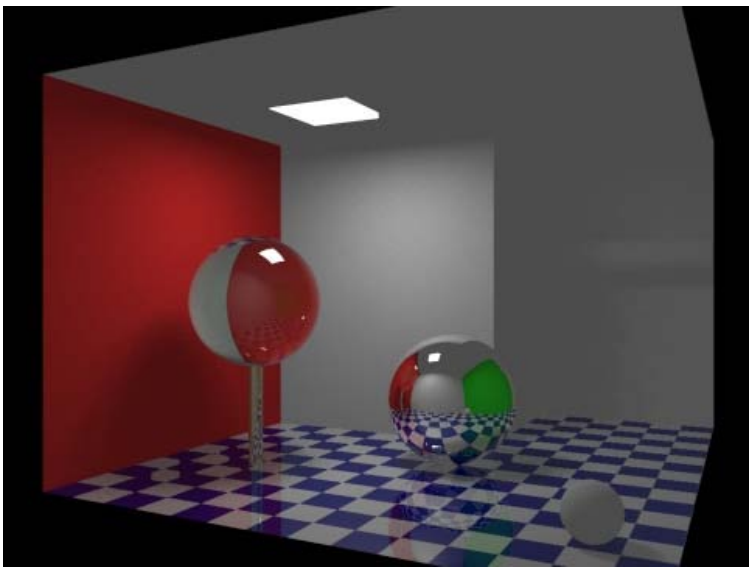
The Radeon 7200 also includes hardware support for EMBM, which was demonstrated in the technical demonstration "Radeon's Ark". However, EMBM was not supported by other graphics chips, such as NVIDIA's GeForce 256 through to the GeForce 2, which only supported the simpler Dot-3 BM. Due to this lack of 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.

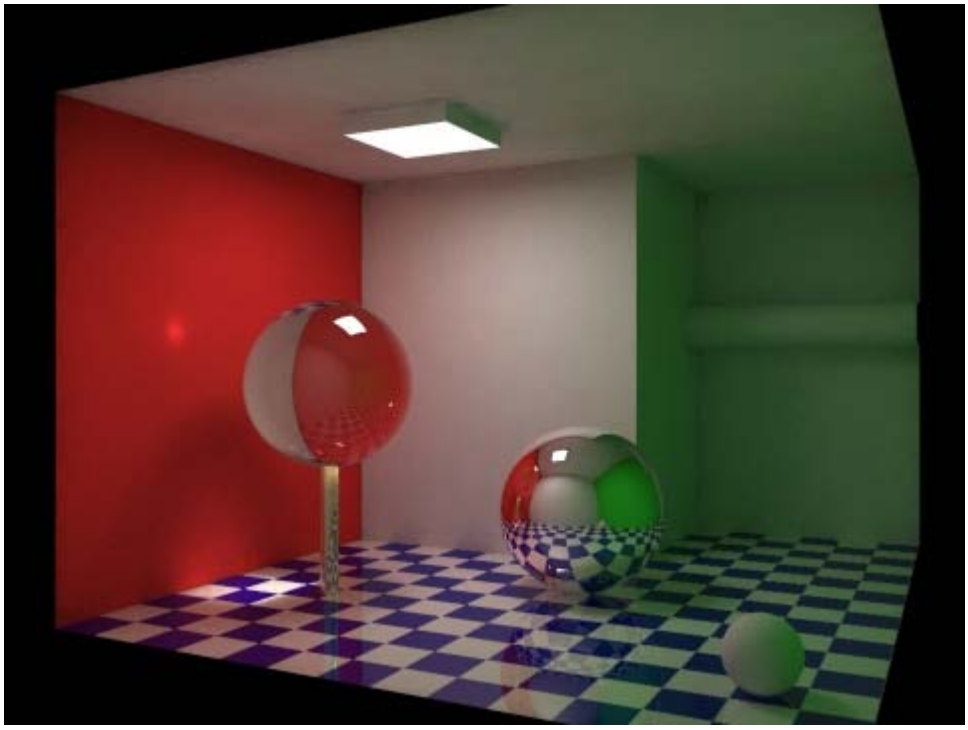
Chapter 4

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:

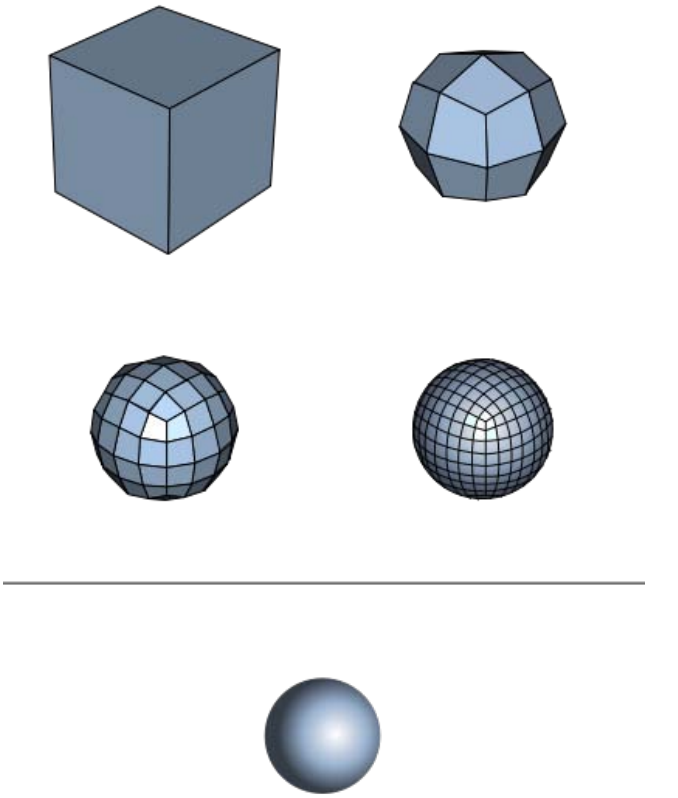
- Inversion: $L = (1 - T)^{-1} L^e$
  - is not applied in practice
- Expansion: $$L = \sum_{i=0}^{\infty} T^i L^e$$
  - bi-directional approach: Photon Mapping + Distributed ray tracing, Bi-directional path tracing, Metropolis light transport
- Iteration: $L_n tl_e + = L^{(n-1)}$
  - Radiosity

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 $\mathbb{C}^1$at extraordinary vertices and $\mathbb{C}^2$everywhere else (when n indicates how many derivatives are continuous, we speak of $\mathbb{C}^n$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

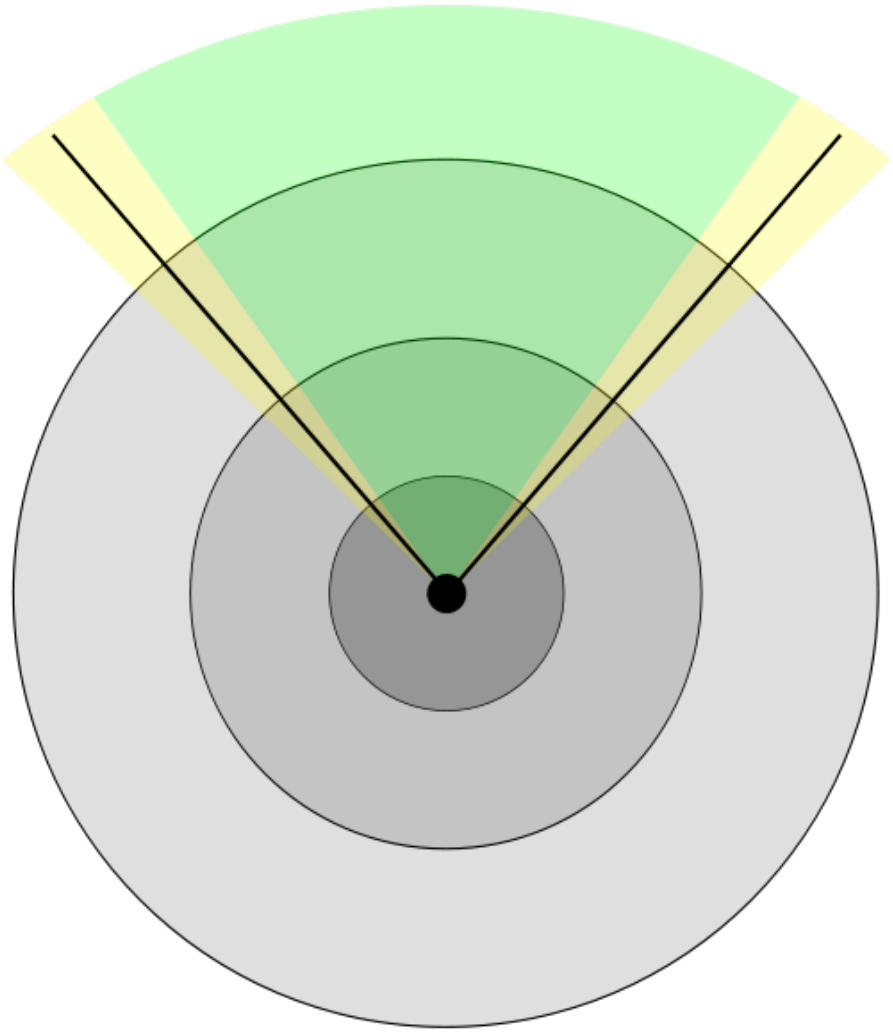
Although the algorithm introduced above covers a whole range of level of detail management techniques, real world applications usually employ different methods according the information being rendered. Because of the appearance of the considered objects, two main algorithm families are used.

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

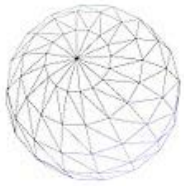
algorithm is usually easier to get working, providing both faster performance and lower CPU usage because of the few operations involved.

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.

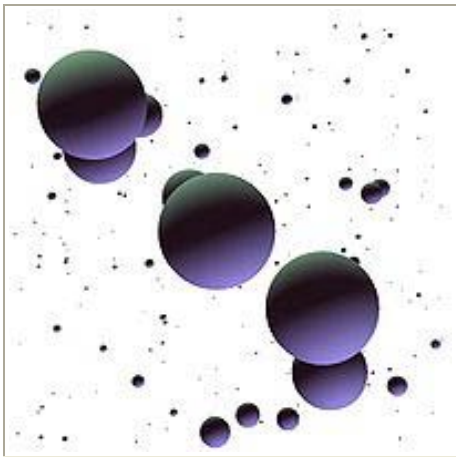
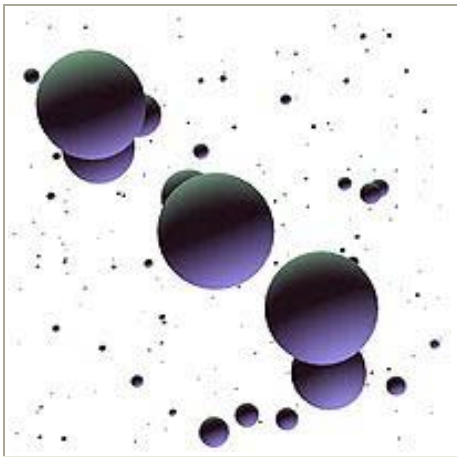
Visual impact comparisons and measurements

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.

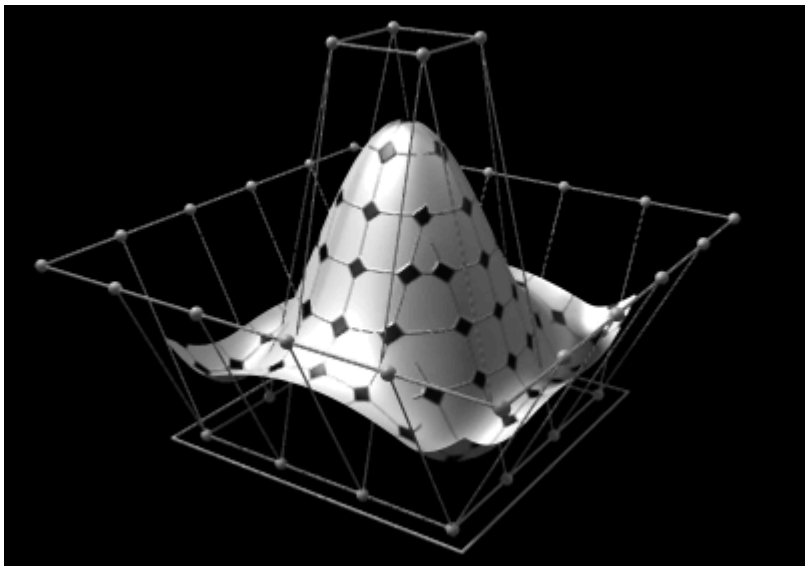
Visual impact comparisons and measurements		
	Brute	**DLOD**
Rendered images		
Render time	27.27 ms	1.29 ms
Scene vertices (thousands)	2328.48	109.44

Hierarchical LOD

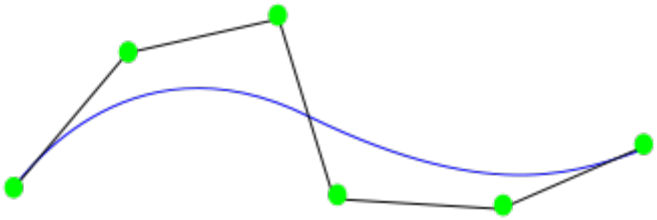
Because hardware is geared towards large amounts of detail, rendering low polygon objects may score sub-optimal performances. HLOD avoids the problem by grouping different objects together. This allows for higher efficiency as well as taking advantage of proximity considerations.

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

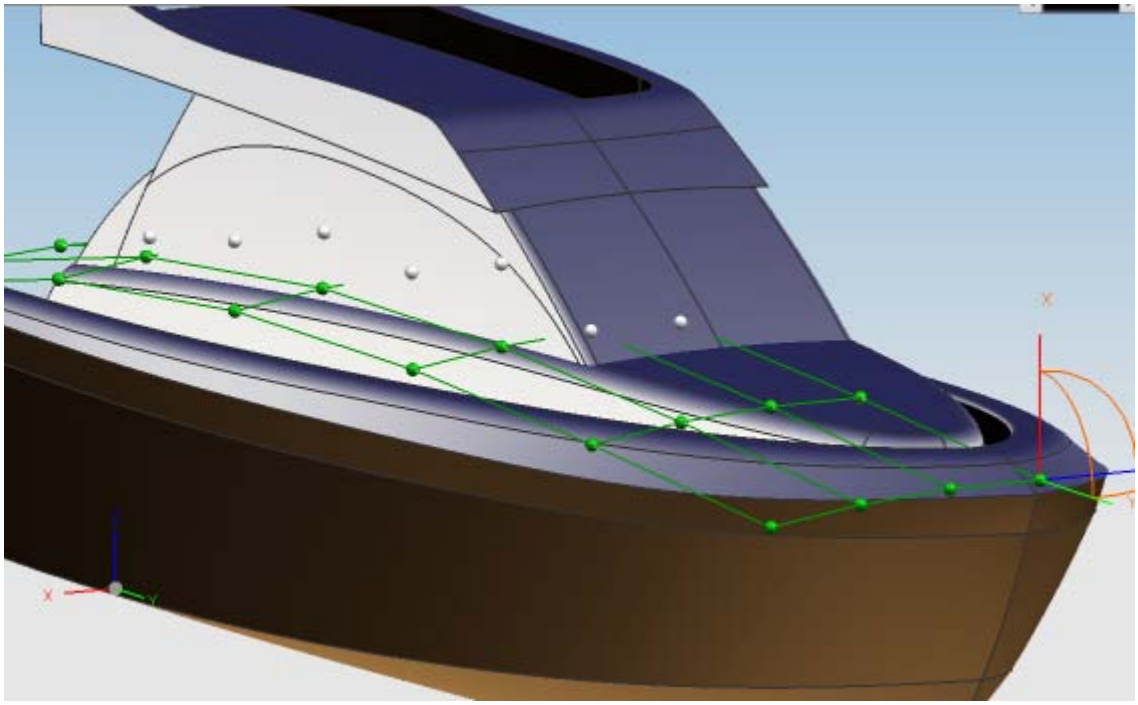
Development of NURBS began in the 1950s by engineers who were in need of a mathematically precise representation of freeform surfaces like those used for ship hulls, aerospace exterior surfaces, and car bodies, which could be exactly reproduced whenever technically needed. Prior representations of this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

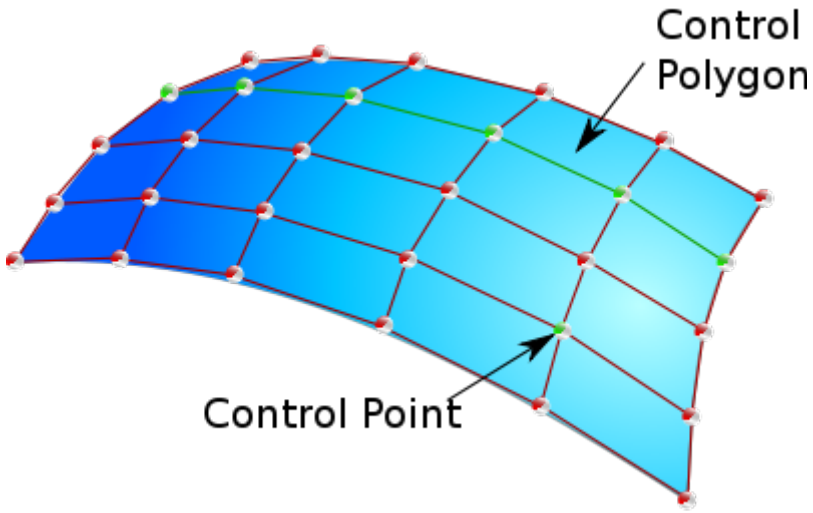
The definition of the continuity 'Cn' requires that the n^{th} derivative of the curve/surface ($d^n C(u) / du^n$) are equal at a joint. Note that the (partial) derivatives of curves and surfaces are vectors that have a direction and a magnitude. Both should be equal.

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.

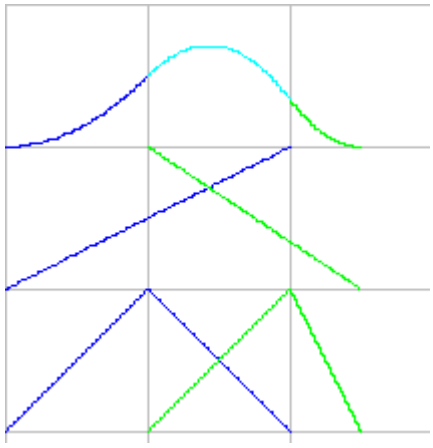
Order

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

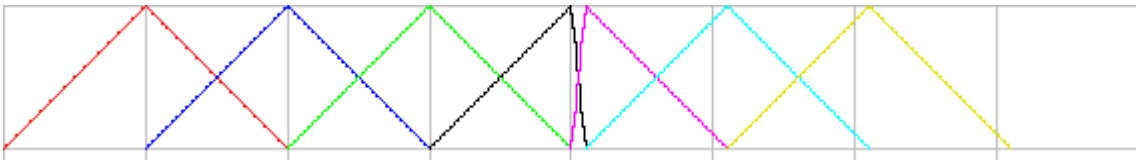
$$f_{i,n}(u) = \frac{u - k_i}{k_{i+n} - k_i}$$

and

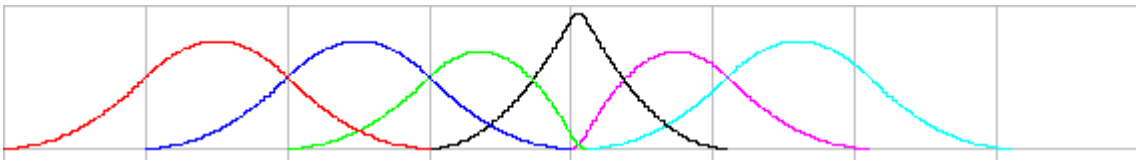
$$g_{i,n}(u) = \frac{k_{i+n} - u}{k_{i+n} - k_i}$$

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

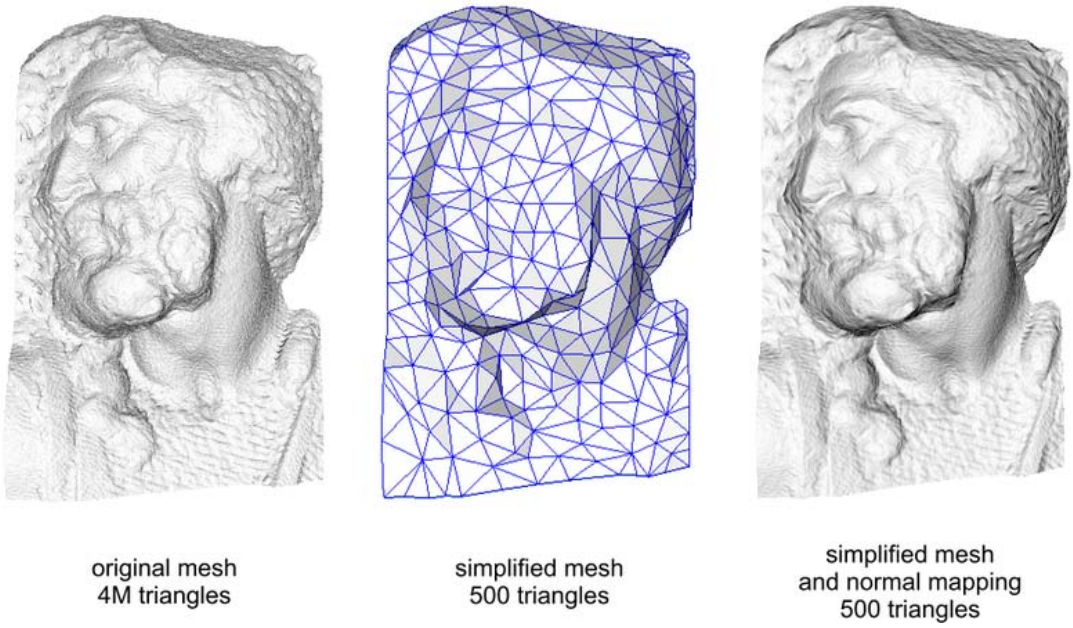
control points are positioned in such a way that the first derivative is continuous. (In fact, the curve is infinitely differentiable everywhere, as it must be if it exactly represents a circle.)

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.

$$
\begin{aligned}
t' &= t \times M_{3x3} \times V_{3x3} \\
n' &= n \times (M_{3x3} \times V_{3x3})^{-1T} = n \times M_{3x3}^{-1T} \times V_{3x3}^{-1T} \\
b' &= b \times M_{3x3} \times V_{3x3}
\end{aligned}
$$

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.

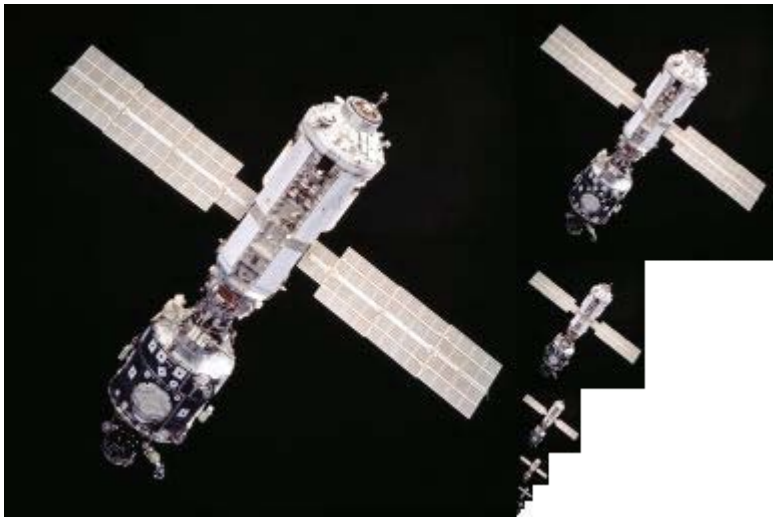
Mipmap

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.

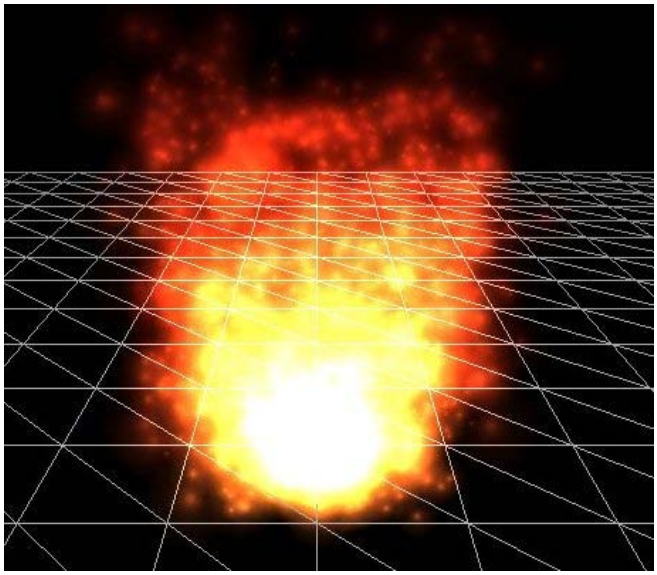
To reduce the memory requirement, and simultaneously give more resolutions to work with, summed-area tables were conceived. However, this approach tends to exhibit poor cache behavior. Also, a summed area table needs to have wider types to store the partial sums than the word size used to store the texture. 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.

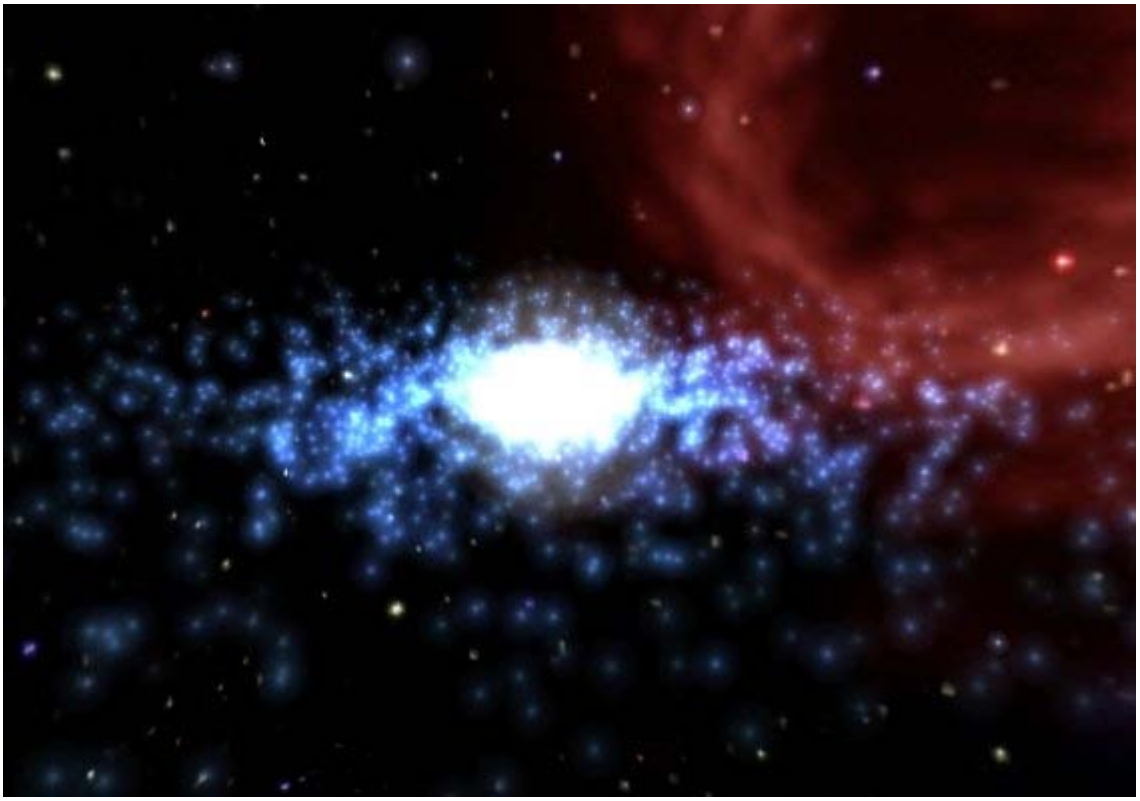
Chapter 8

Particle System & Painter's Algorithm

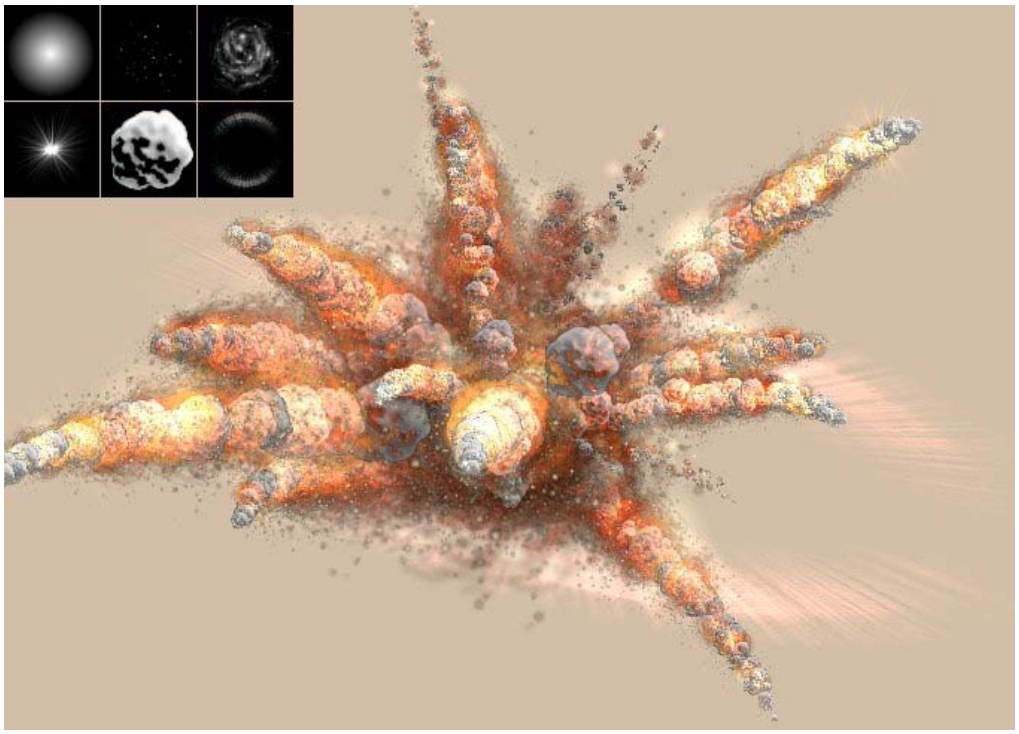
Particle System



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

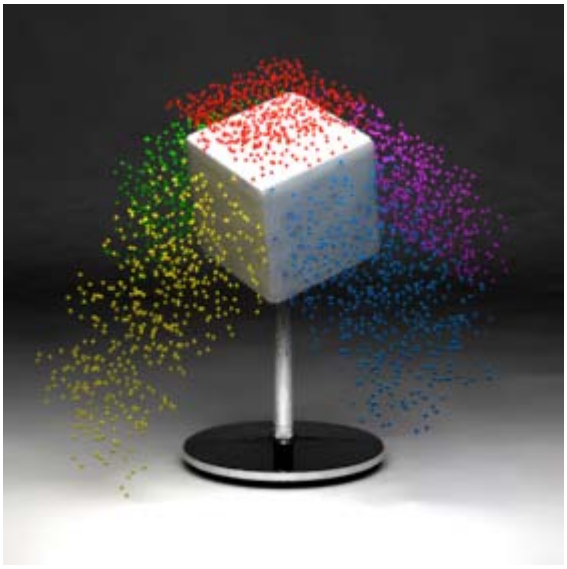
After the update is complete, each particle is rendered, usually in the form of a textured billboarded quad (i.e. a quadrilateral that is always facing the viewer). However, this is not necessary; a particle may be rendered as a single pixel in small resolution/limited processing power environments. 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.

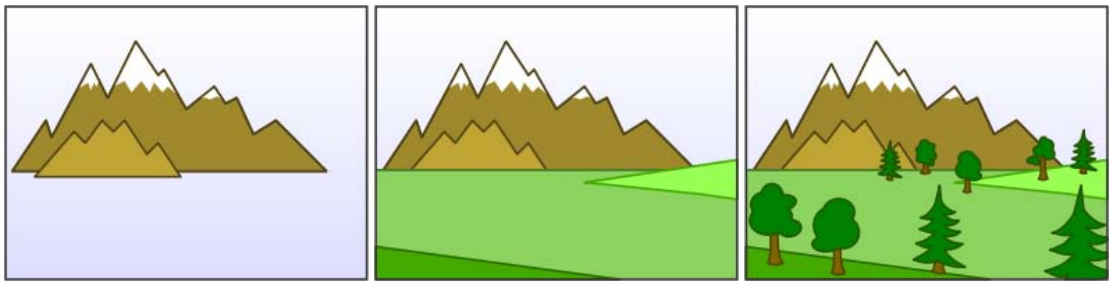
Developer-friendly particle system tools

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.

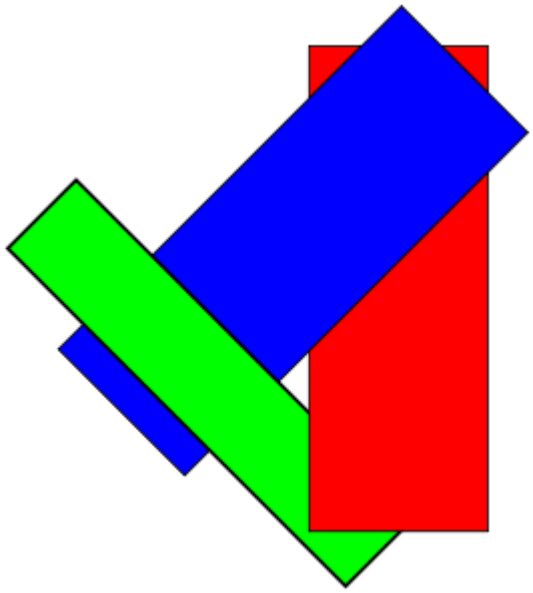
Painter's Algorithm

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.

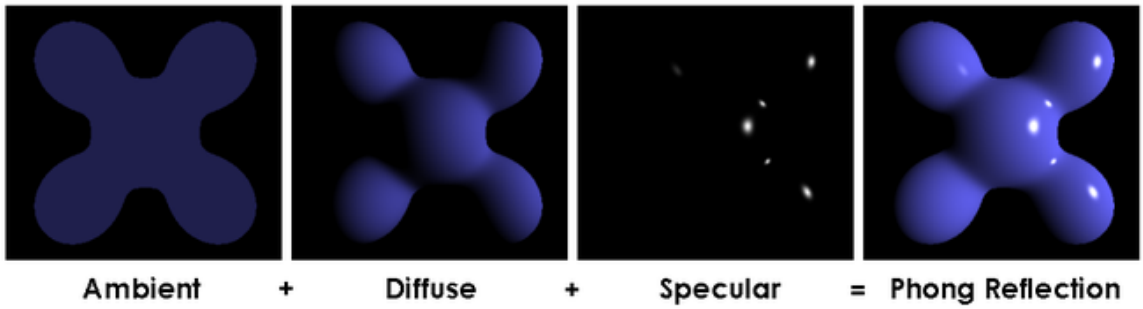
The model of reflection may also be referred to as the **Phong reflection model**, **Phong illumination** or **Phong lighting**. It may be called Phong shading in the context of pixel shaders or other places where a lighting calculation can be referred to as "shading". The interpolation method may also 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:

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

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:

$$(I_p - C_a) / C_d = L_xN_x + L_zN_z$$

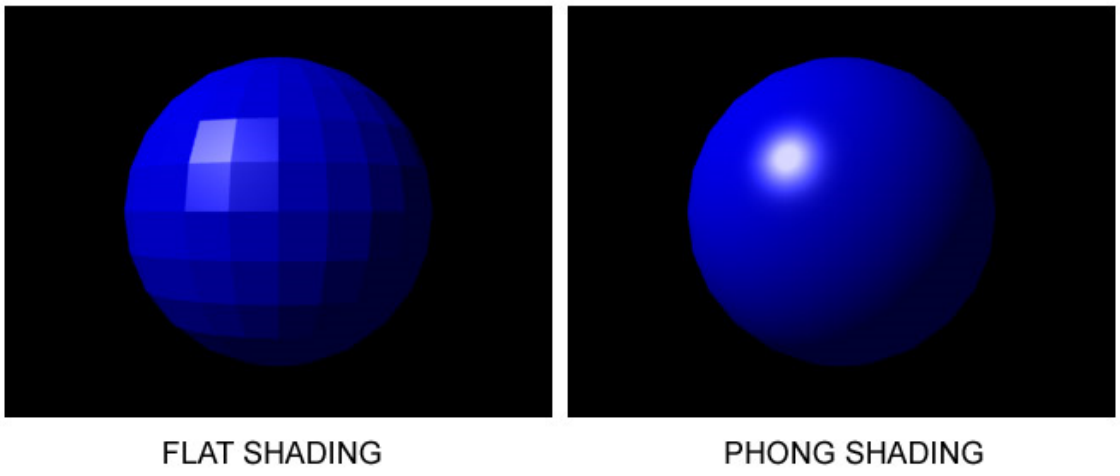
For instance if the light direction is 45 degrees above the object $L = [0.71,0.71]$ we get two equations with two unknowns.

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

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.

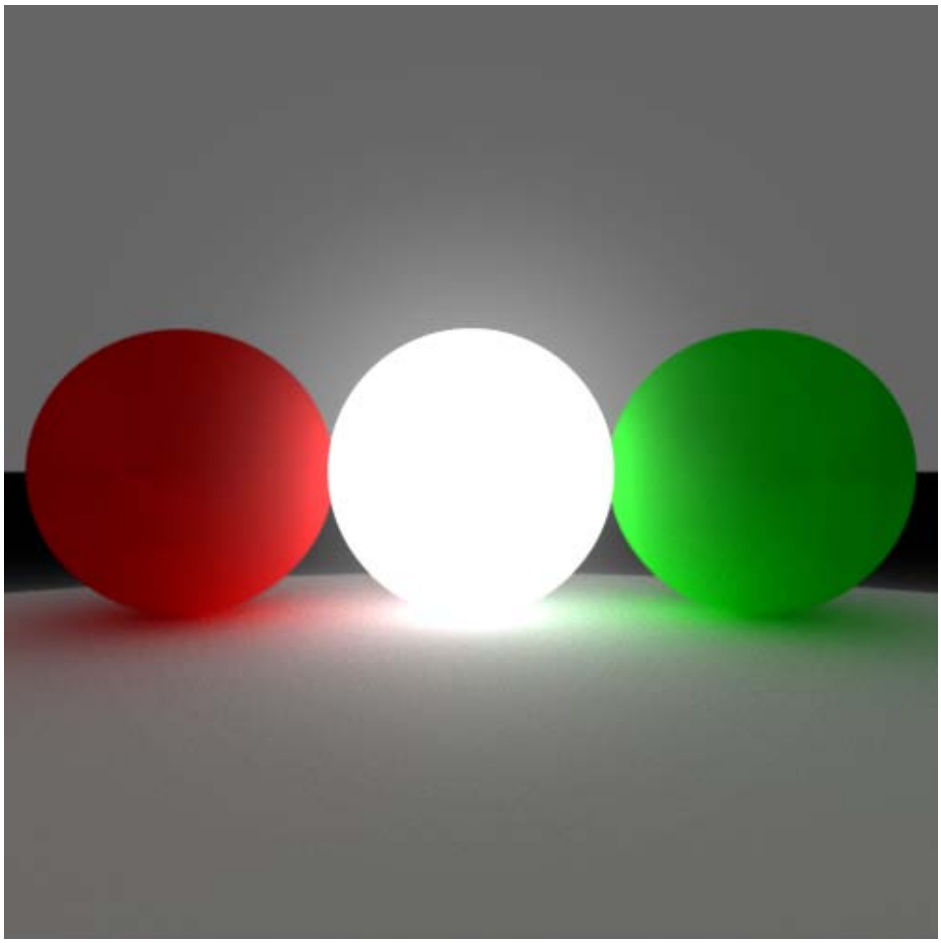
The most serious problem with Gouraud shading occurs when specular highlights are found in the middle of a large polygon. Since these specular highlights are absent from the polygon's vertices and Gouraud shading interpolates based on the vertex colors, the specular highlight will be missing from the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

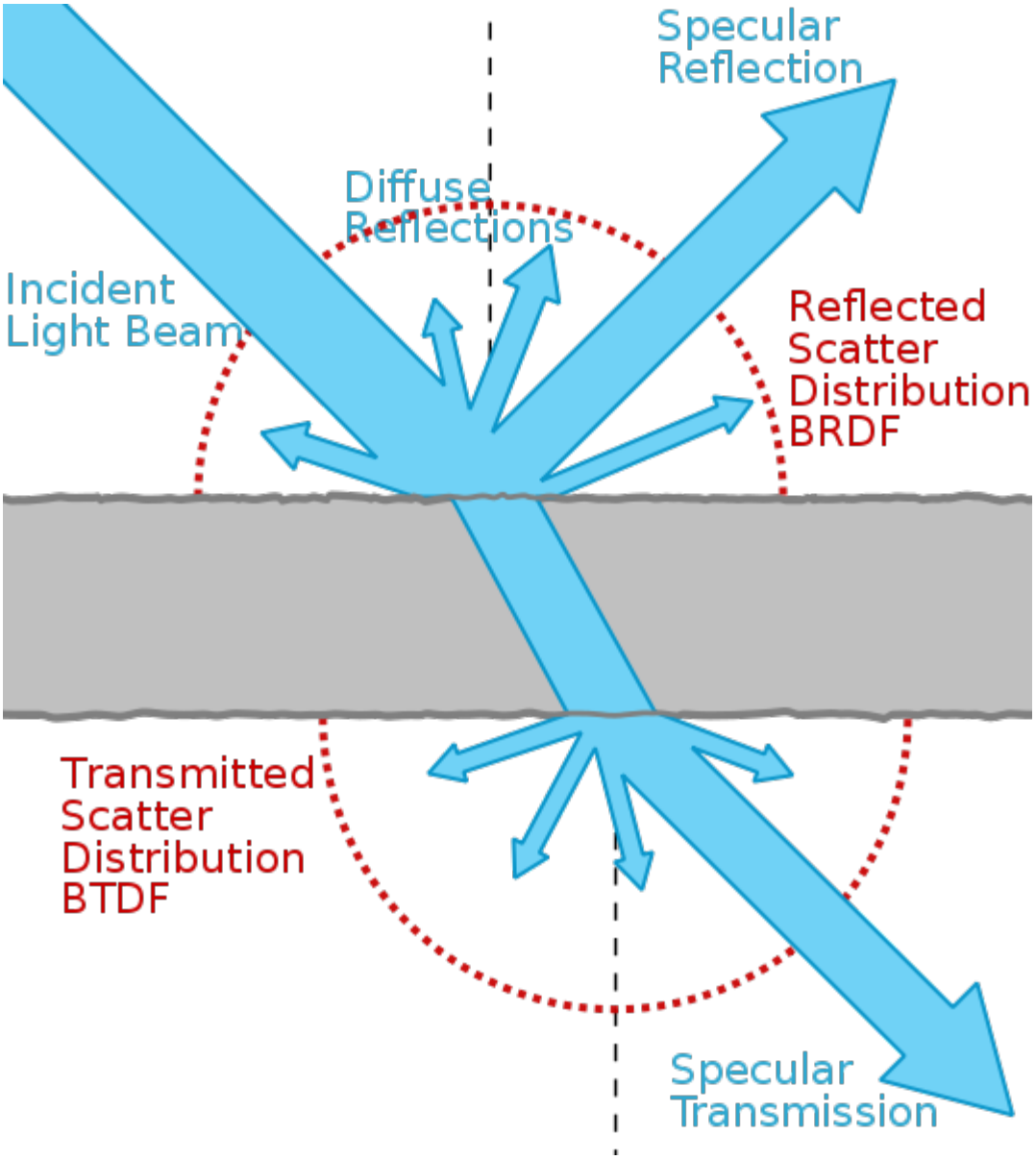
Performance

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.

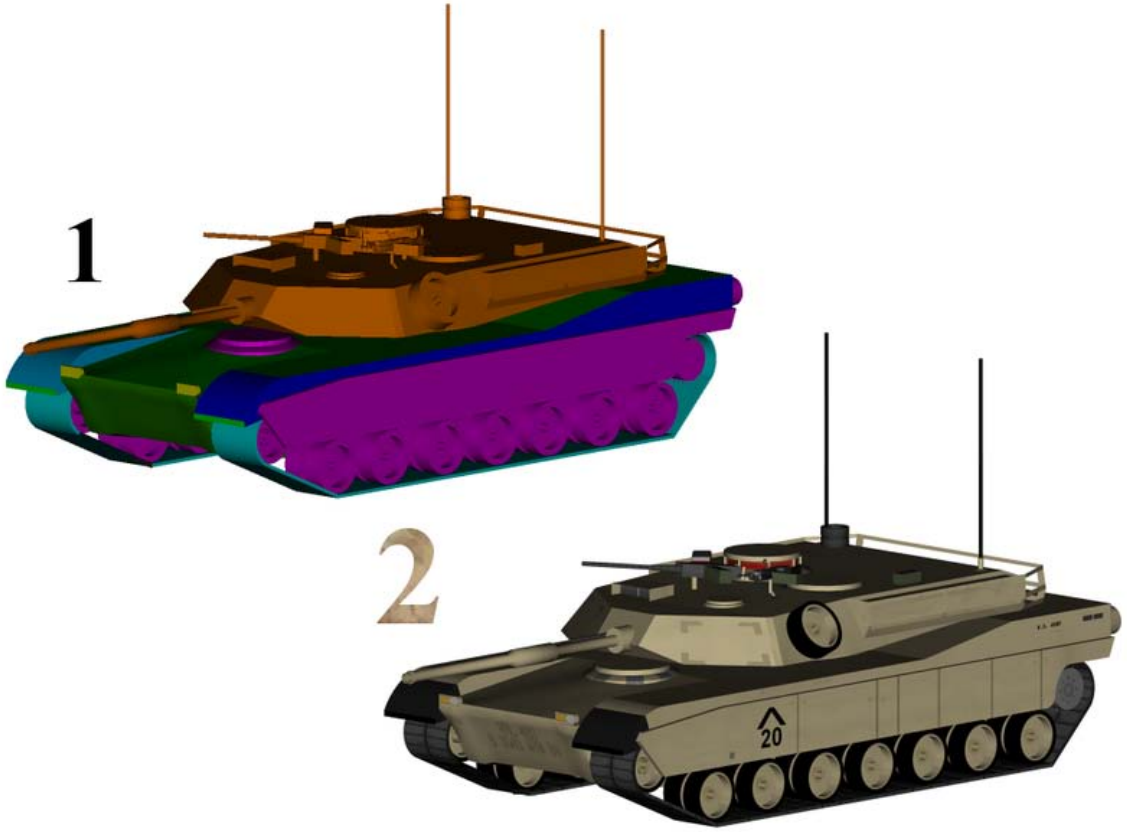
- Image space photon mapping achieves real-time performance by computing the first and last scattering using a GPU rasterizer.

Variations

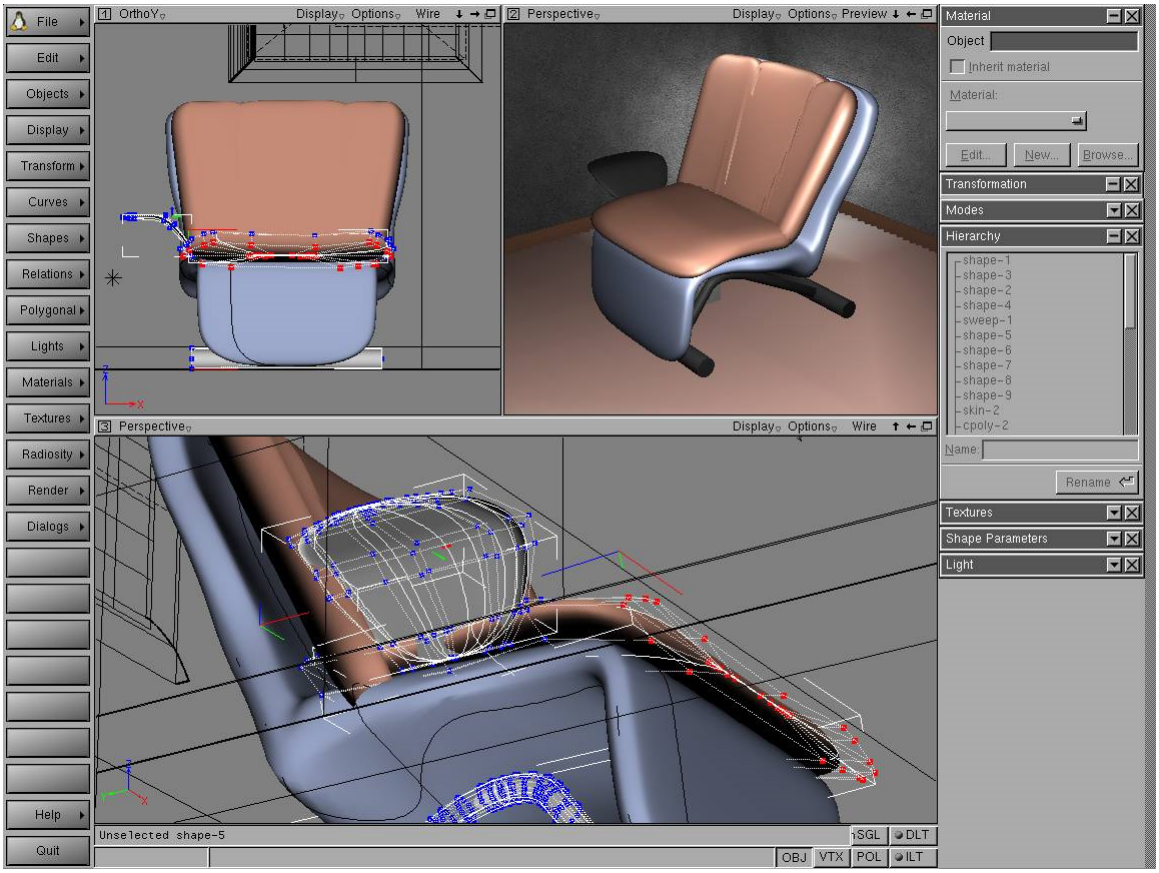
- Although photon mapping was designed to work primarily with ray tracers, it can also be extended for use with scanline renderers.

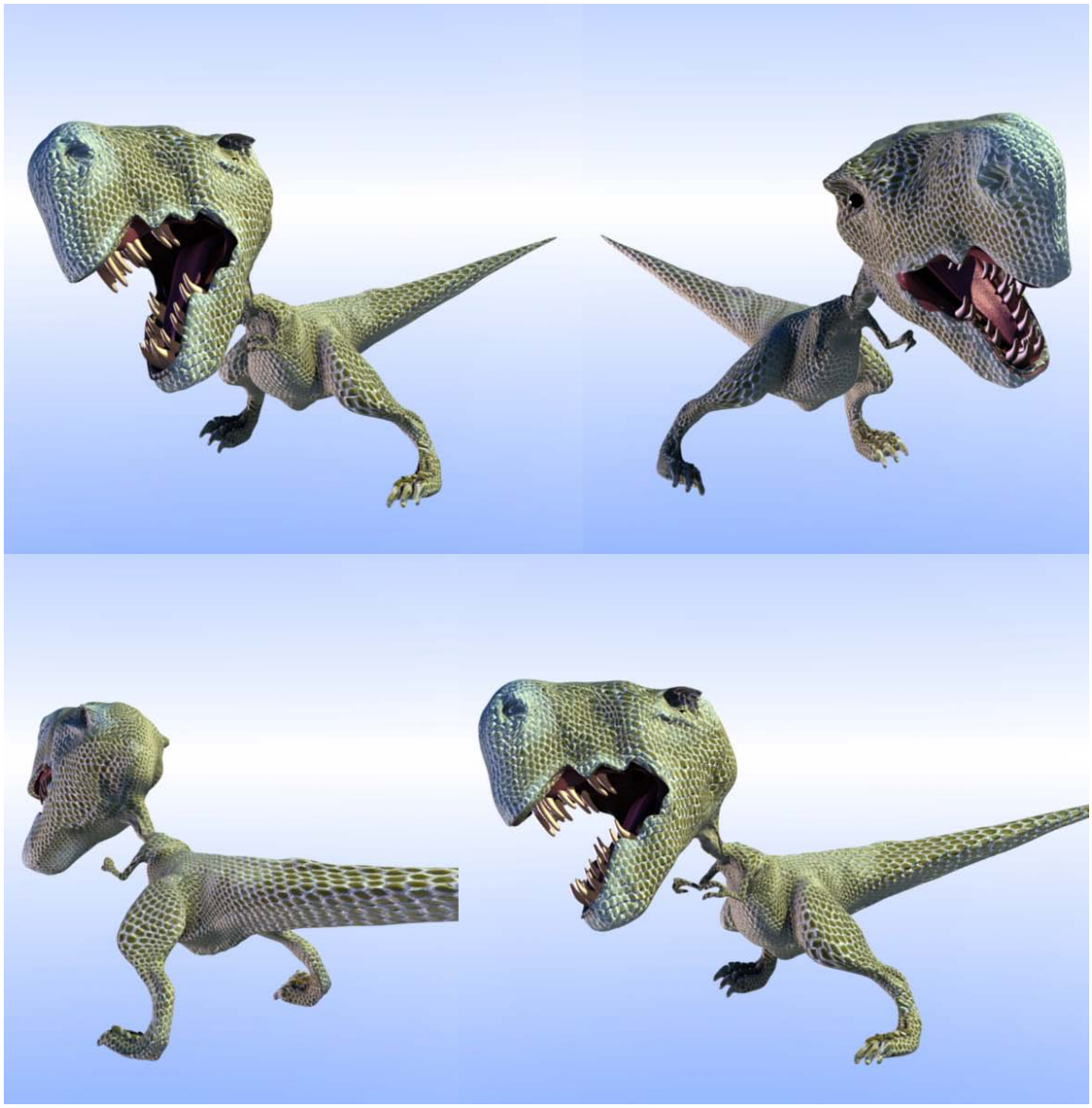
Chapter 12

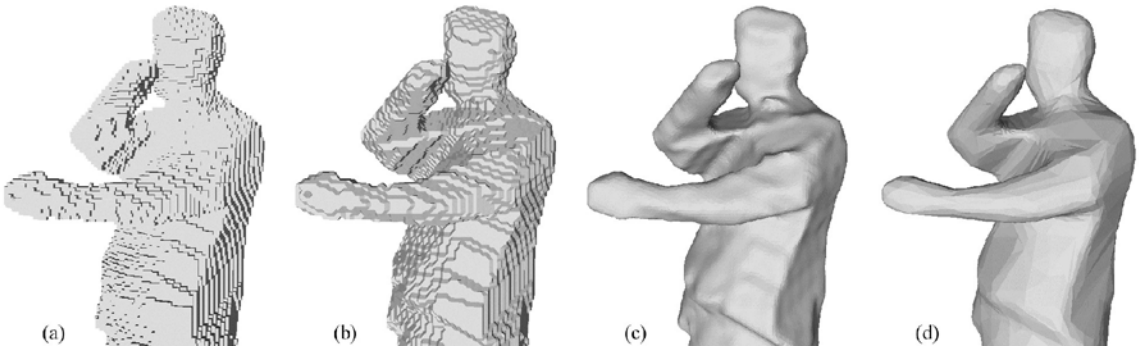
Texture Mapping



1 = 3D model without textures
2 = 3D model with textures

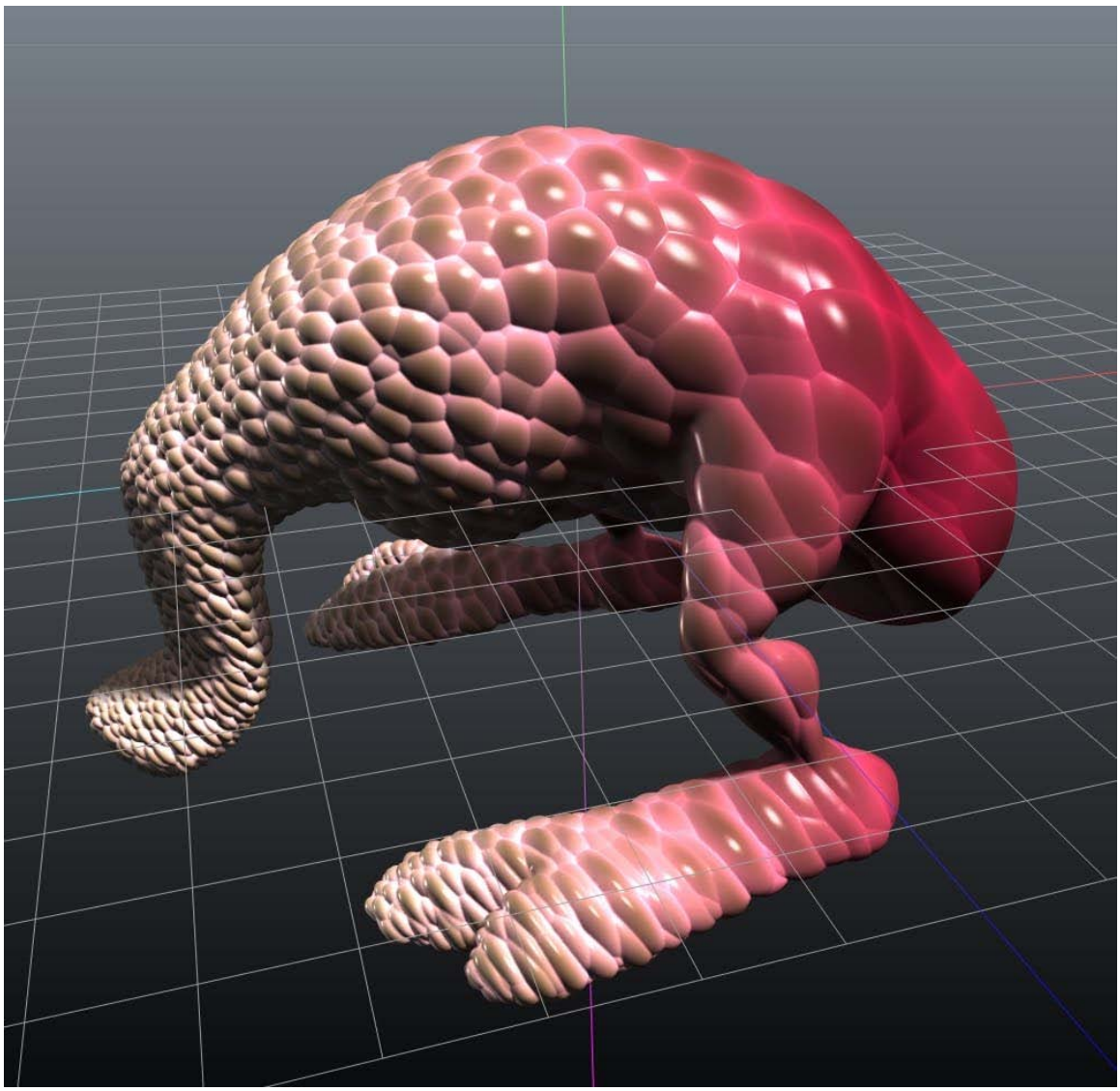

File
Edit
Objects
Display
Transform
Curves
Shapes
Relations
Polygonal
Lights
Materials
Textures
Radiosity
Render
Dialogs
Help
Quit
OrthoY
Display Options Wire
Perspective
Display Options Preview
Perspective
Display Options Wire
Material
Object
Inherit material
Material:
Edit... New... Browse...
Transformation
Modes
Hierarchy
shape-1
shape-3
shape-2
shape-4
sweep-1
shape-5
shape-6
shape-7
shape-8
shape-9
skin-2
cpoly-2
Name:
Rename
Textures
Shape Parameters
Light
Unselected shape-5
SGL
DLT
OBJ
VTX
POL
ILT

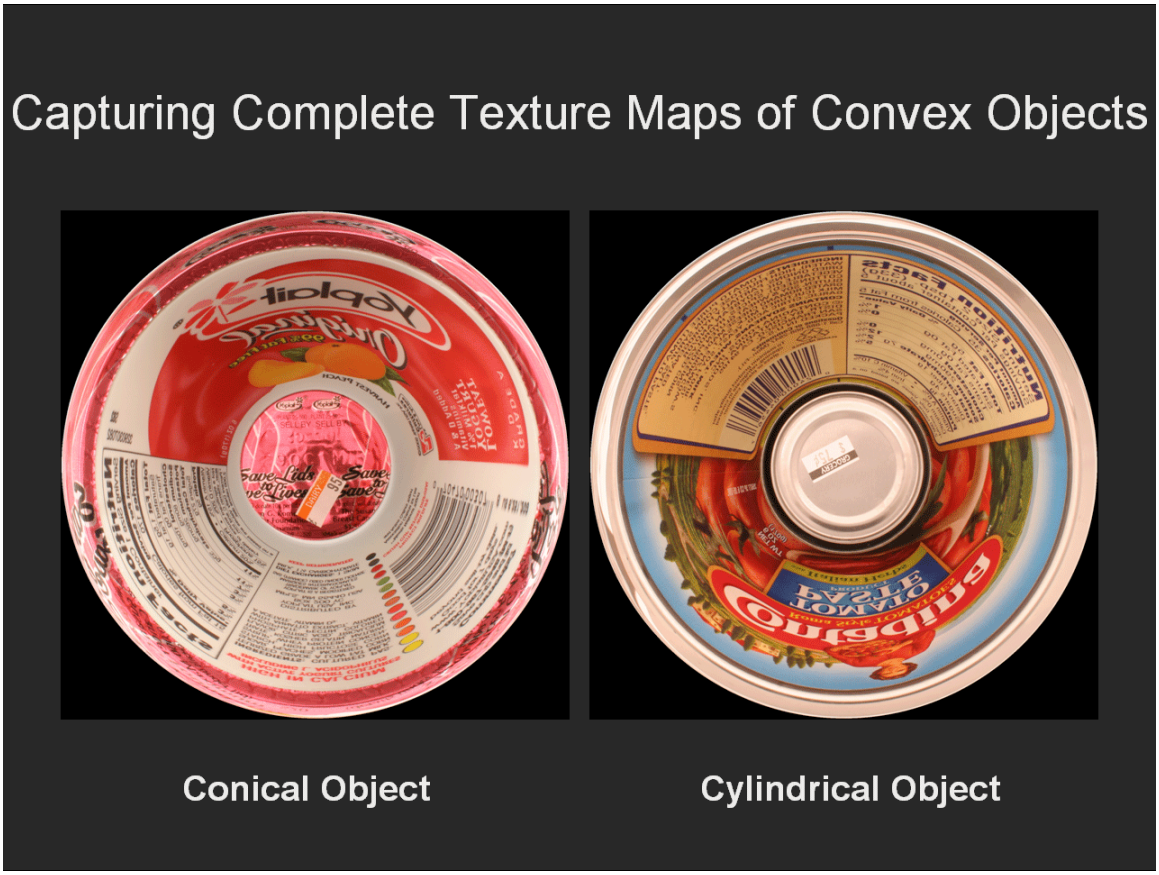


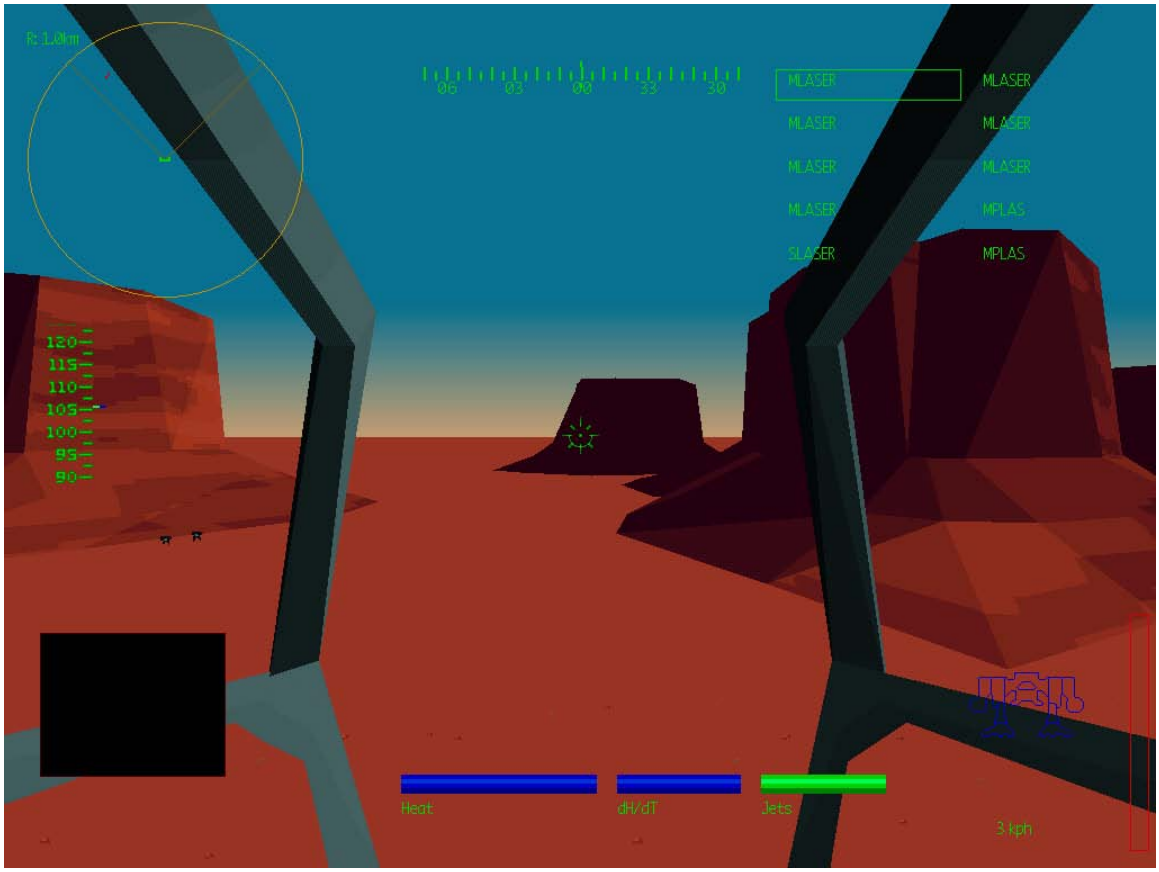

(a)
(b)
(c)
(d)

Texture mapping is a method for adding detail, surface texture (a bitmap or raster image), or color to a computer-generated graphic or 3D model. Its application to 3D graphics was pioneered by Dr Edwin Catmull in his Ph.D. thesis of 1974.

Texture mapping

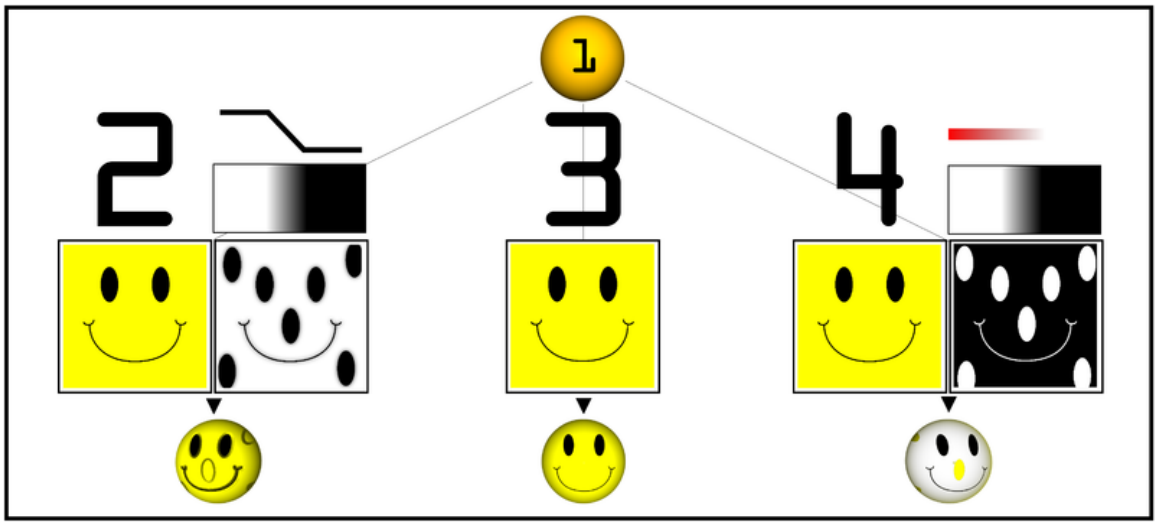



Capturing Complete Texture Maps of Convex Objects
Conical Object
Cylindrical Object



A **texture map** is applied (mapped) to the surface of a shape or polygon. This process is akin to applying patterned paper to a plain white box.

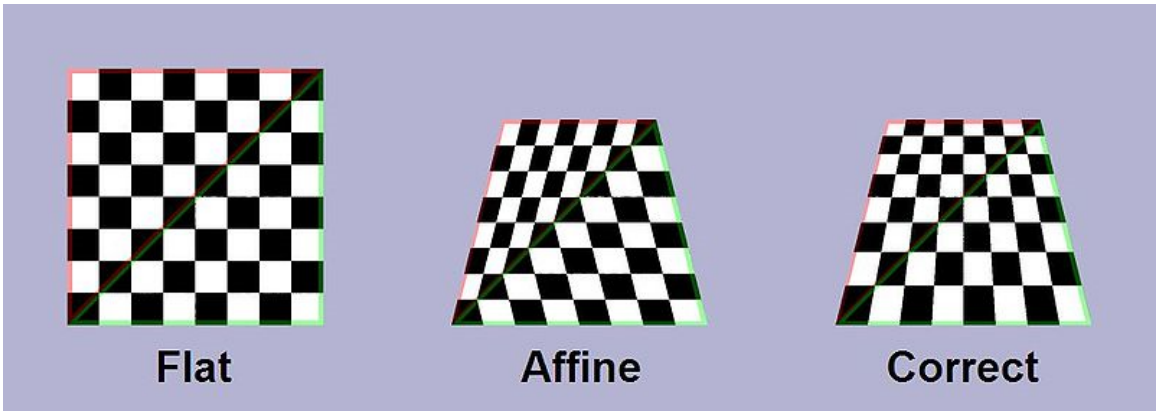
Multitexturing is the use of more than one texture at a time on a polygon. For instance, a light map texture may be used to light a surface as an alternative to recalculating that lighting every time the surface is rendered. Another multitexture technique is bump mapping, which allows a texture to directly control the facing direction of a surface for the purposes of its lighting calculations; it can give a very good appearance of a complex surface, such as tree bark or rough concrete, that takes on lighting detail in addition to the usual detailed coloring. Bump mapping has become popular in recent video games as graphics hardware has become powerful enough to accommodate it in real-time.



Examples of multitexturing
1. Untextured sphere 2. Texture and bump maps 3. Texture map only 4. Opacity and texture maps

The way the resulting pixels on the screen are calculated from the texels (texture pixels) is governed by texture filtering. The fastest method is to use the nearest-neighbour interpolation, but bilinear interpolation or trilinear interpolation between mipmaps are two commonly used alternatives which reduce aliasing or jaggies. In the event of a texture coordinate being outside the texture, it is either clamped or wrapped.

Perspective correctness



Because affine texture mapping does not take into account the depth information about a polygon's vertices, where the polygon is not perpendicular to the viewer it produces a noticeable defect.

Texture coordinates are specified at each vertex of a given triangle, and these coordinates are interpolated using an extended Bresenham's line algorithm. If these texture coordinates are linearly interpolated across the screen, the result is **affine texture**

mapping. This is a fast calculation, but there can be a noticeable discontinuity between adjacent triangles when these triangles are at an angle to the plane of the screen.

Perspective correct texturing accounts for the vertices' positions in 3D space, rather than simply interpolating a 2D triangle. This achieves the correct visual effect, but it is slower to calculate. Instead of interpolating the texture coordinates directly, the coordinates are divided by their depth (relative to the viewer), and the reciprocal of the depth value is also interpolated and used to recover the perspective-correct coordinate. This correction makes it so that in parts of the polygon that are closer to the viewer the difference from pixel to pixel between texture coordinates is smaller (stretching the texture wider), and in parts that are farther away this difference is larger (compressing the texture).

Affine texture mapping directly interpolates a texture coordinate $u_\alpha$between two endpoints $u_0$and u_1:

$$u_\alpha = (1-\alpha)u_0 + \alpha u_1 \text{where } 0 \leq \alpha \leq 1$$

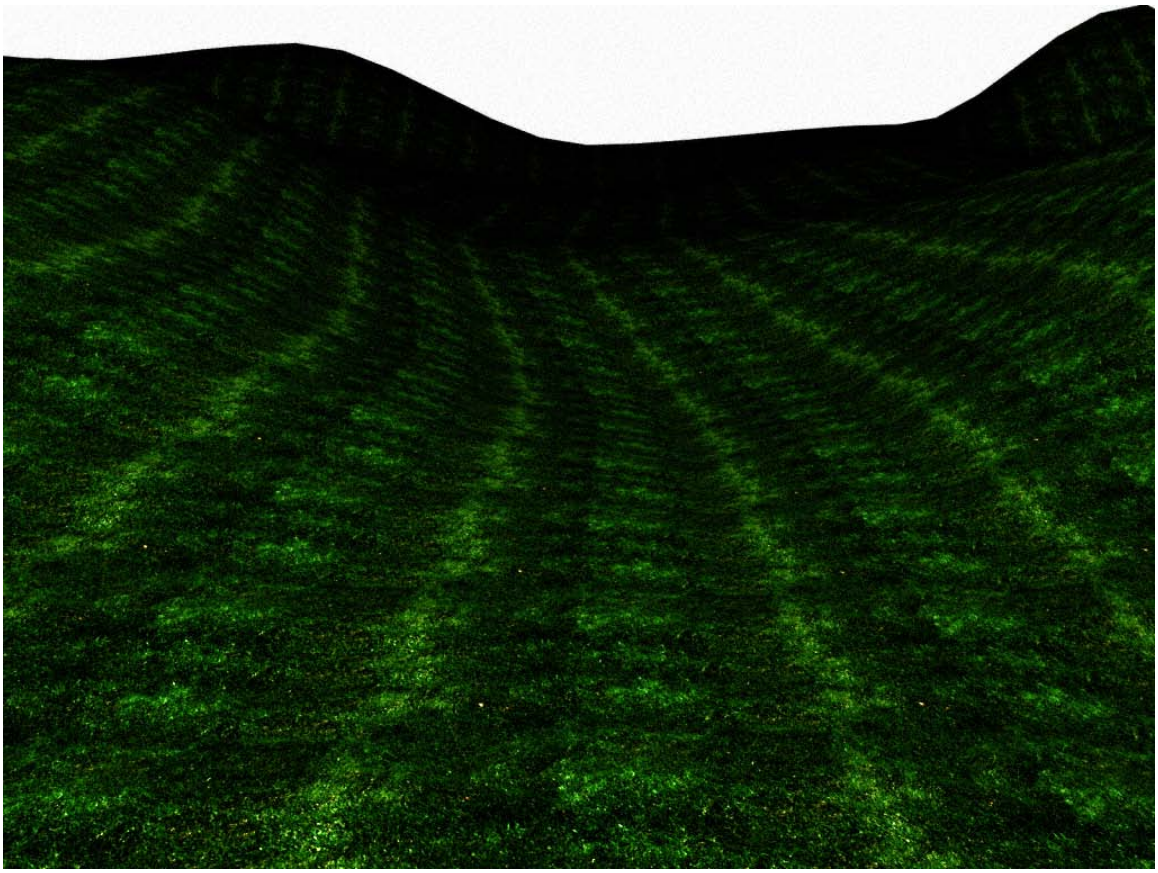
Perspective correct mapping interpolates after dividing by depth z, then uses its interpolated reciprocal to recover the correct coordinate:

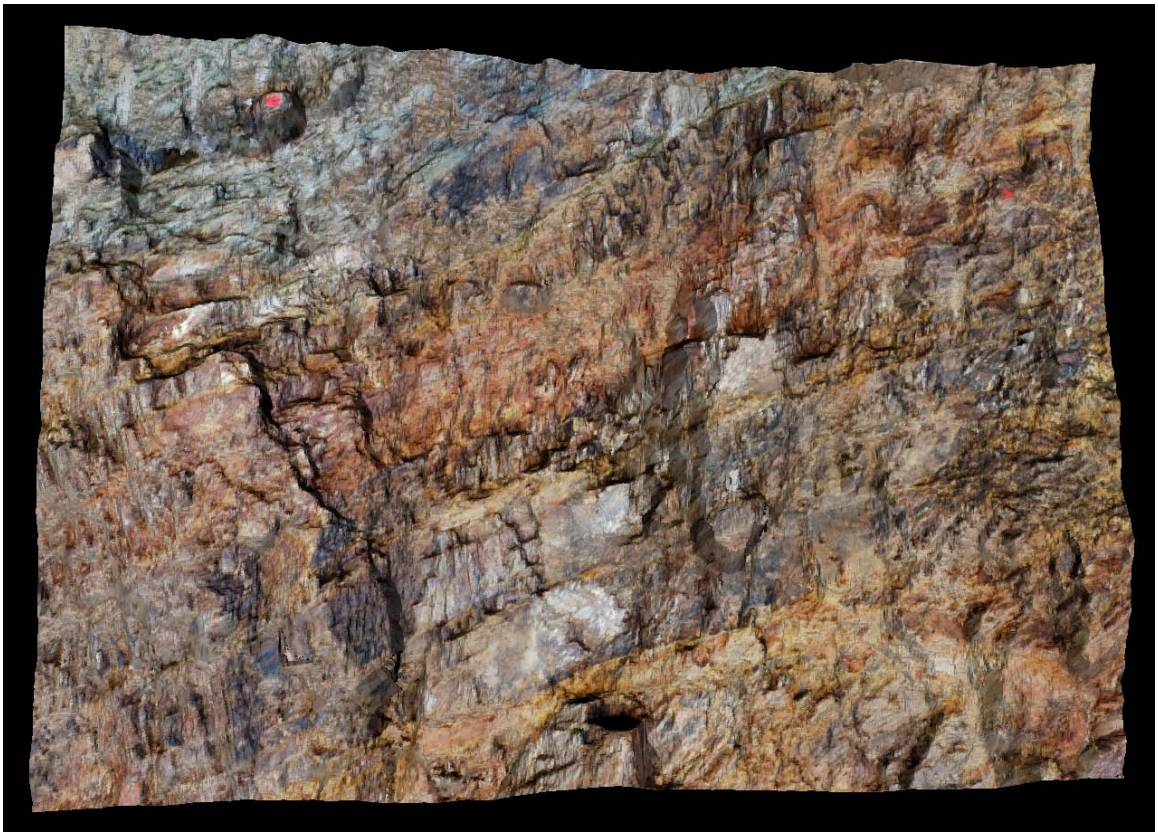
$$u_\alpha = \frac{(1-\alpha)\frac{u_0}{z_0} + \alpha\frac{u_1}{z_1}}{(1-\alpha)\frac{1}{z_0} + \alpha\frac{1}{z_1}}$$

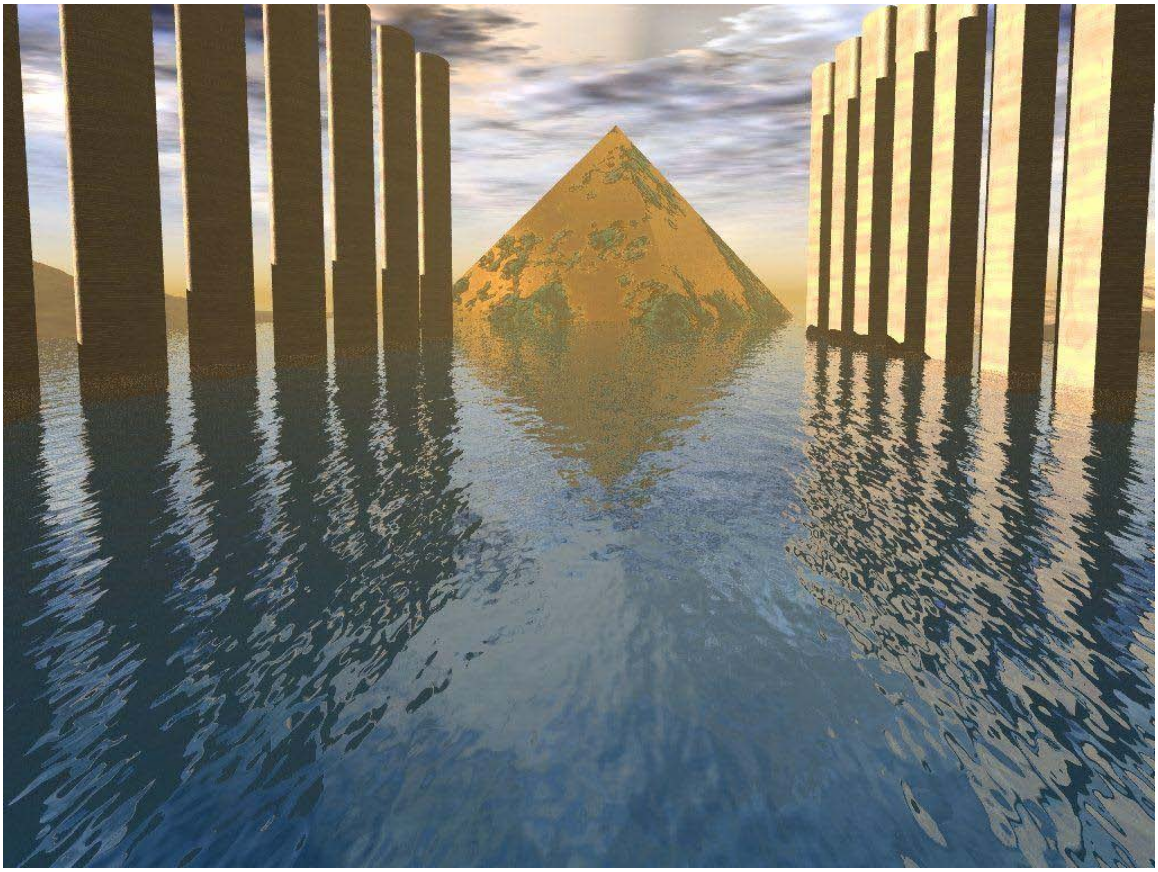
All modern graphics hardware implements perspective correct texturing.

Doom renders vertical spans (walls) with affine texture mapping

Screen space sub division techniques. Top left: Quake like, top right: bilinear, bottom left: const-z





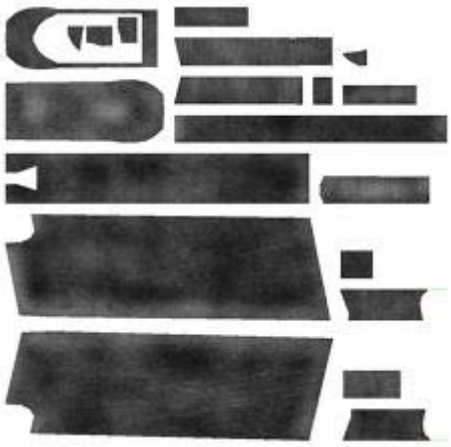


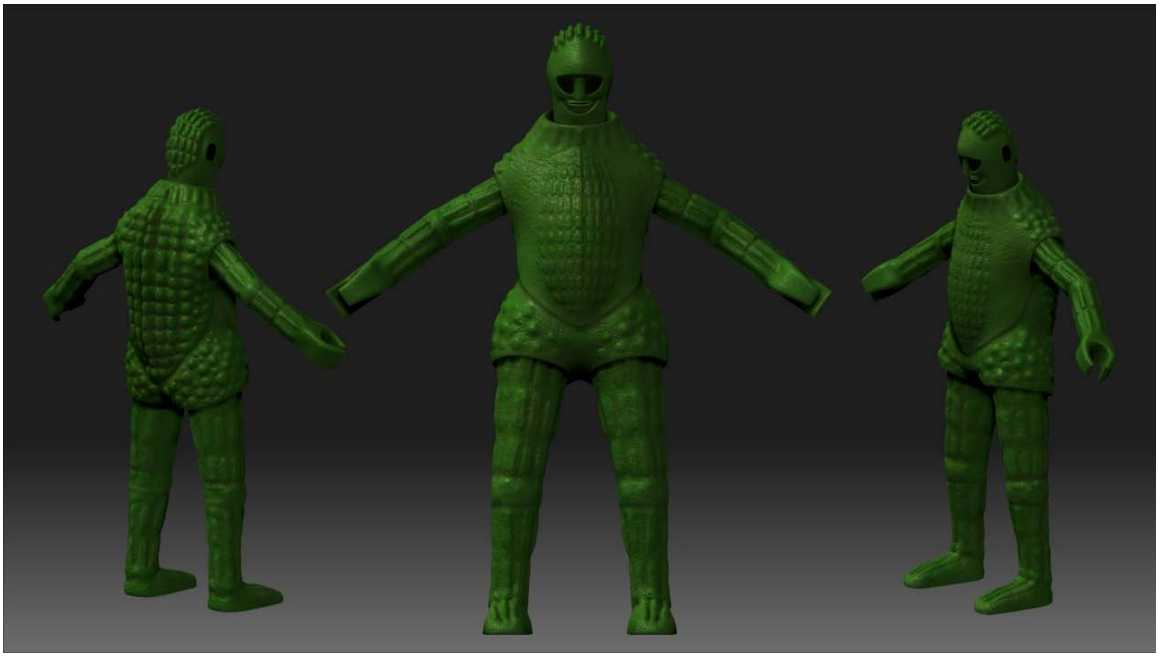
Classic texture mappers generally did only simple mapping with at most one lighting effect and the perspective correctness was about 16 times more expensive. Thus to on the one hand not wait on the divisions and on the other hand not let the division and multiplication circuits run idle every triangle is subdivided in subgroups of about 16 pixels. For perspective texture mapping ignorant hardware a triangle is broken down into smaller triangles for rendering, which in non architectural applications has some synergy with level of detail. Software renderers generally preferred screen subdivision because it has less overhead. Additionally they try to do linear interpolation along a line of pixels to simplify the set-up (compared to 2d affine interpolation) and thus again the overhead (also affine texture-mapping does not fit into the low number of registers of the 086 CPU; the 68000 or any RISC is much more suited). For instance, *Doom* restricted the world to vertical walls and horizontal floors/ceilings. This meant the walls would be a constant distance along a vertical line and the floors/ceilings would be a constant distance along a horizontal line. A fast affine mapping could be used along those lines because it would be correct. A different approach was taken for *Quake*, which would calculate perspective correct coordinates only once every 16 pixels of a scanline and linearly interpolate between them, effectively running at the speed of linear interpolation because the perspective correct calculation runs in parallel on the co-processor . The polygons are rendered independently, hence it may be possible to switch between spans and columns or diagonal directions depending on the orientation of the polygon normal to achieve a more constant z, but the effort seems not to be worth it.

Another technique was subdividing the polygons into smaller polygons, like triangles in 3d-space or squares in screen space, and using an affine mapping on them. The distortion of affine mapping becomes much less noticeable on smaller polygons. Yet another technique was approximating the perspective with a faster calculation, such as a polynomial. Still another technique uses 1/z value of the last two drawn pixels to linearly extrapolate the next value. The division is then done starting from those values so that only a small remainder has to be divided , but the amount of bookkeeping makes this method too slow on most systems. Finally, some programmers extended the constant distance trick used for Doom by finding the line of constant distance for arbitrary polygons and rendering along it.

Resolution

The resolution of a texture map is usually given in terms of pixel width, assuming the map is square: as an example, a 1K texture has a resolution of 1024x1024, that is 1048576 pixels.

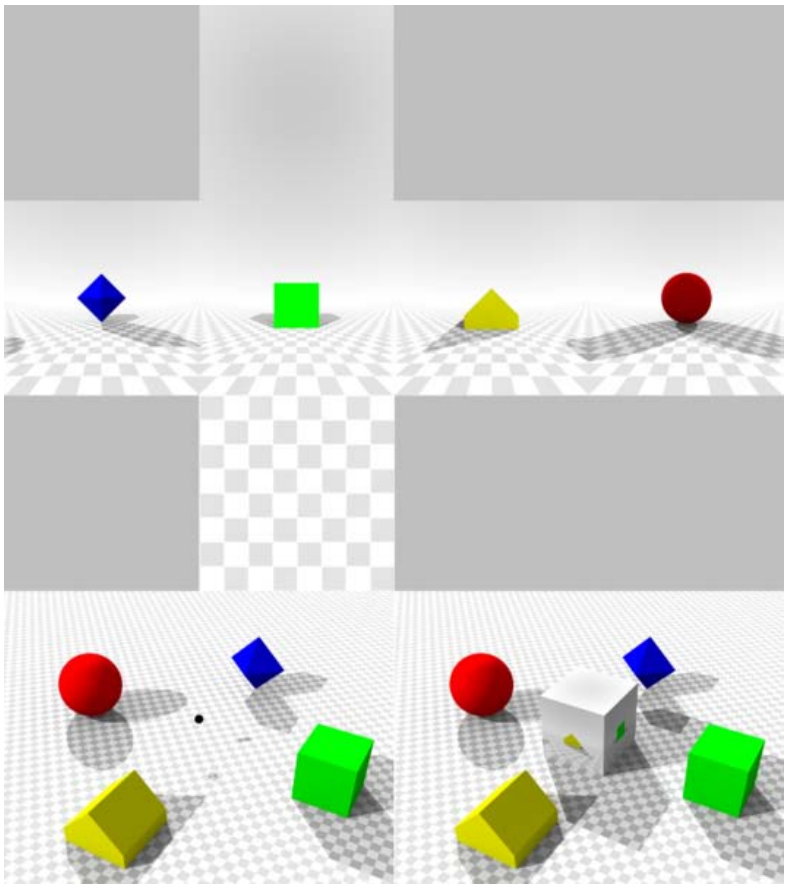




Graphics cards cannot render texture maps beyond a threshold that depends on their technical specifications, mostly in terms of RAM.

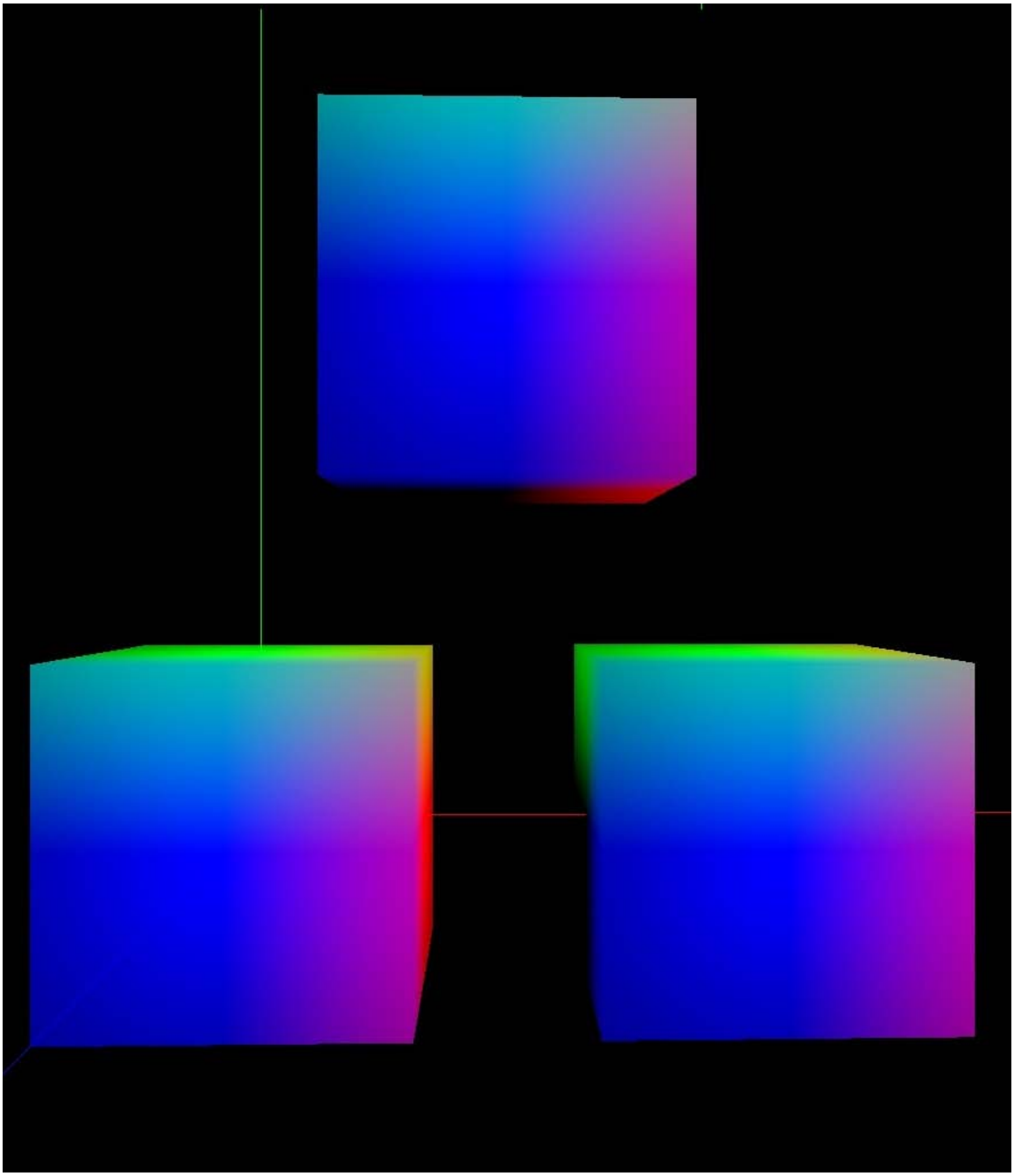
Chapter 13

Cube Mapping

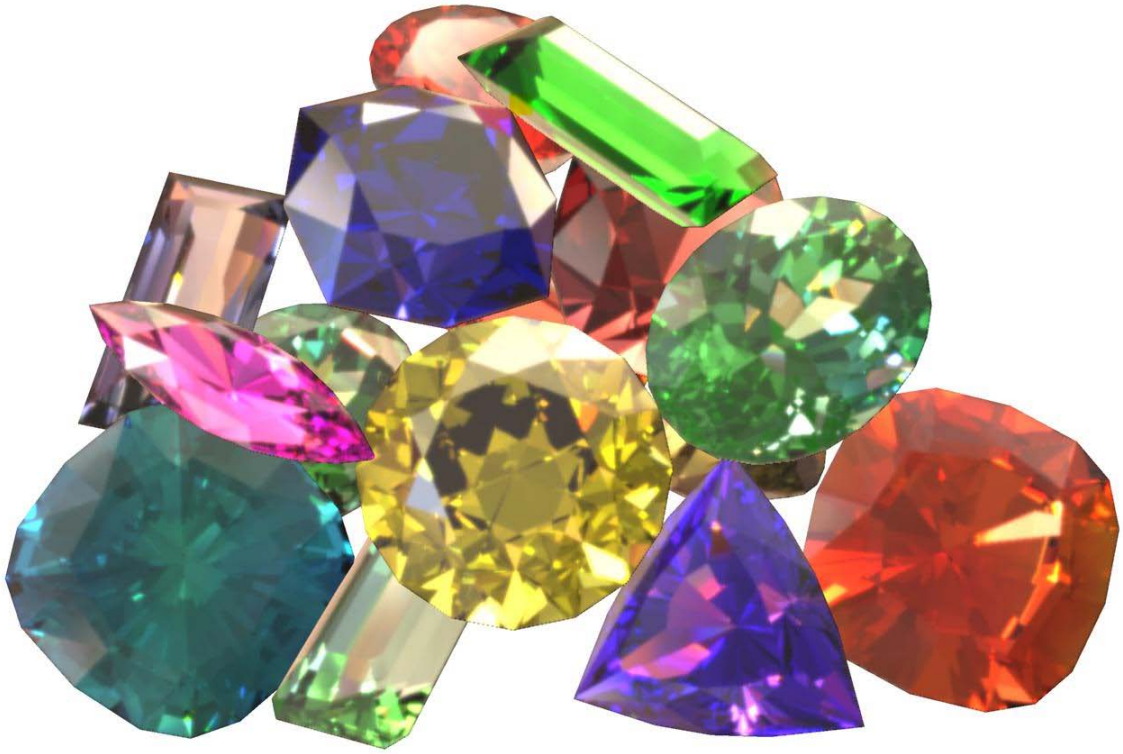


The lower left image shows a scene with a viewpoint marked with a black dot. The upper image shows the net of the cube mapping as seen from that viewpoint, and the lower right image shows the cube superimposed on the original scene.

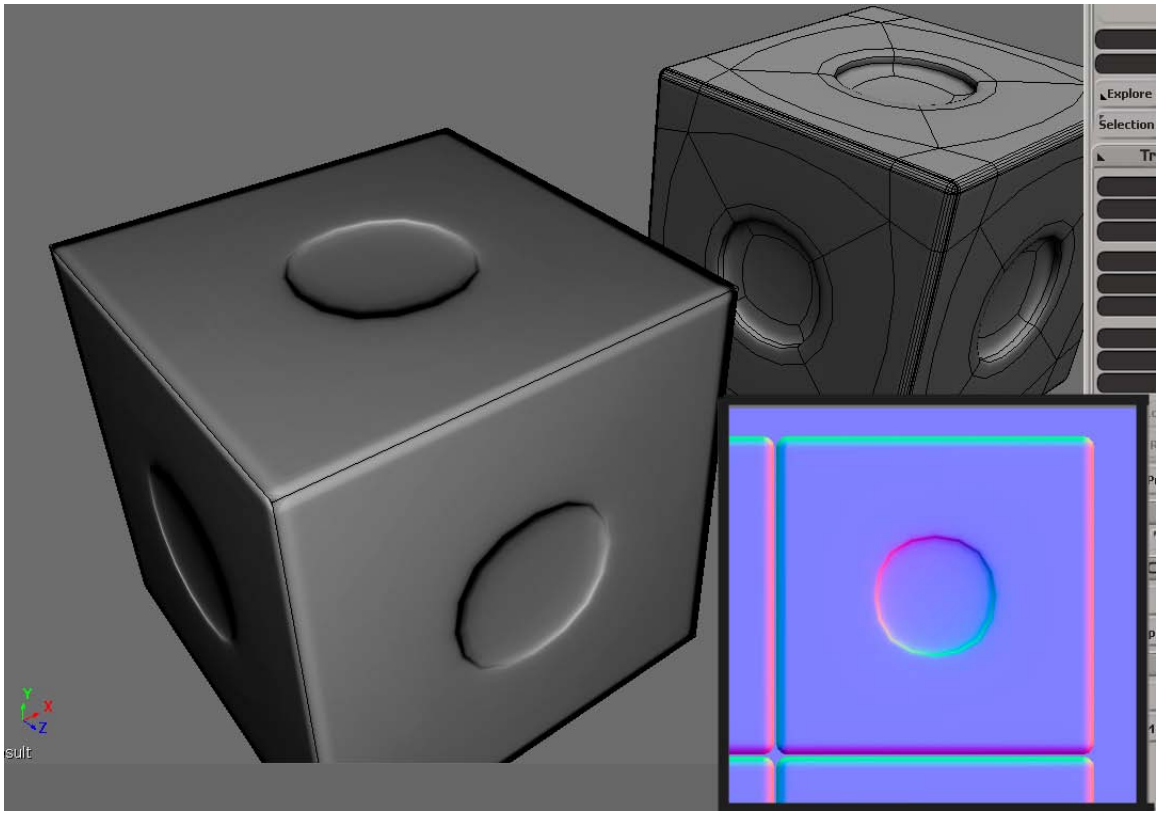

96.82



In computer graphics, cube mapping is a method of environment mapping that uses a six-sided cube as the map shape. The environment is projected onto the six faces of a cube and stored as six square textures, or unfolded into six regions of a single texture. The cube map is generated by first rendering the scene six times from a viewpoint, with the views defined by an orthogonal 90 degree view frustum representing each cube face.




59.03



In the majority of cases, cube mapping is preferred over the older method of sphere mapping because it eliminates many of the problems that are inherent in sphere mapping such as image distortion, viewpoint dependency, and computational efficiency. Also, cube mapping provides a much larger capacity to support real-time rendering of reflections relative to sphere mapping because the combination of inefficiency and viewpoint dependency severely limit the ability of sphere mapping to be applied when there is a consistently changing viewpoint.

History

Cube mapping was first proposed in 1986 by Ned Greene in his paper “Environment Mapping and Other Applications of World Projections”, ten years after environment mapping was first put forward by Jim Blinn and Martin Newell. However, hardware limitations on the ability to access six texture images simultaneously made it infeasible to implement cube mapping without further technological developments. This problem was remedied in 1999 with the release of the Nvidia GeForce 256. Nvidia touted cube mapping in hardware as “a breakthrough image quality feature of GeForce 256 that ... will allow developers to create accurate, real-time reflections. Accelerated in hardware, cube environment mapping will free up the creativity of developers to use reflections and specular lighting effects to create interesting, immersive environments.” Today, cube mapping is still used in a variety of graphical applications as a favored method of environment mapping.

Advantages

Cube mapping is preferred over other methods of environment mapping because of its relative simplicity. Also, cube mapping produces results that are similar to those obtained by ray tracing, but is much more computationally efficient – the moderate reduction in quality is compensated for by large gains in efficiency.

Predating cube mapping, sphere mapping has many inherent flaws that made it impractical for most applications. Sphere mapping is view dependent meaning that a different texture is necessary for each viewpoint. Therefore, in applications where the viewpoint is mobile, it would be necessary to dynamically generate a new sphere mapping for each new viewpoint (or, to pre-generate a mapping for every viewpoint). Also, a texture mapped onto a sphere's surface must be stretched and compressed, and warping and distortion (particularly along the edge of the sphere) are a direct consequence of this. Although these image flaws can be reduced using certain tricks and techniques like “pre-stretching”, this just adds another layer of complexity to sphere mapping.

Paraboloid mapping provides some improvement on the limitations of sphere mapping, however it requires two rendering passes in addition to special image warping operations and more involved computation.

Conversely, cube mapping requires only a single render pass, and due to its simple nature, is very easy for developers to comprehend and generate. Also, cube mapping uses the entire resolution of the texture image, compared to sphere and paraboloid mappings, which also allows it to use lower resolution images to achieve the same quality. Although handling the seams of the cube map is a problem, algorithms have been developed to handle seam behavior and result in a seamless reflection.

Applications

Computer-aided design

Computer-aided design (**CAD**), also known as **computer-aided design and drafting** (**CADD**), is the use of computer technology for the process of design and design-documentation. Computer Aided Drafting describes the process of drafting with a computer. CADD software, or environments, provide the user with input-tools for the purpose of streamlining design processes; drafting, documentation, and manufacturing processes. CADD output is often in the form of electronic files for print or machining operations. The development of CADD-based software is in direct correlation with the processes it seeks to economize; industry-based software (construction, manufacturing, etc.) typically uses vector-based (linear) environments whereas graphic-based software utilizes raster-based (pixelated) environments.

CADD environments often involve more than just shapes. As in the manual drafting of technical and engineering drawings, the output of CAD must convey information, such as

materials, processes, dimensions, and tolerances, according to application-specific conventions.

CAD may be used to design curves and figures in two-dimensional (2D) space; or curves, surfaces, and solids in three-dimensional (3D) objects.

CAD is an important industrial art extensively used in many applications, including automotive, shipbuilding, and aerospace industries, industrial and architectural design, prosthetics, and many more. CAD is also widely used to produce computer animation for special effects in movies, advertising and technical manuals. The modern ubiquity and power of computers means that even perfume bottles and shampoo dispensers are designed using techniques unheard of by engineers of the 1960s. Because of its enormous economic importance, CAD has been a major driving force for research in computational geometry, computer graphics (both hardware and software), and discrete differential geometry.

The design of geometric models for object shapes, in particular, is often called *computer-aided geometric design* (*CAGD*).

Overview

Current computer-aided design software packages range from 2D vector-based drafting systems to 3D solid and surface modellers. Modern CAD packages can also frequently allow rotations in three dimensions, allowing viewing of a designed object from any desired angle, even from the inside looking out. Some CAD software is capable of dynamic mathematic modeling, in which case it may be marketed as **CADD** — *computer-aided design and drafting*.

CAD is used in the design of tools and machinery and in the drafting and design of all types of buildings, from small residential types (houses) to the largest commercial and industrial structures (hospitals and factories).

CAD is mainly used for detailed engineering of 3D models and/or 2D drawings of physical components, but it is also used throughout the engineering process from conceptual design and layout of products, through strength and dynamic analysis of assemblies to definition of manufacturing methods of components. It can also be used to design objects.

CAD has become an especially important technology within the scope of computer-aided technologies, with benefits such as lower product development costs and a greatly shortened design cycle. CAD enables designers to lay out and develop work on screen, print it out and save it for future editing, saving time on their drawings.

Uses

Computer-aided design is one of the many tools used by engineers and designers and is used in many ways depending on the profession of the user and the type of software in question.

CAD is one part of the whole Digital Product Development (DPD) activity within the Product Lifecycle Management (PLM) process, and as such is used together with other tools, which are either integrated modules or stand-alone products, such as:

- Computer-aided engineering (CAE) and Finite element analysis (FEA)
- Computer-aided manufacturing (CAM) including instructions to Computer Numerical Control (CNC) machines
- Photo realistic rendering
- Document management and revision control using Product Data Management (PDM).

CAD is also used for the accurate creation of photo simulations that are often required in the preparation of Environmental Impact Reports, in which computer-aided designs of intended buildings are superimposed into photographs of existing environments to represent what that locale will be like were the proposed facilities allowed to be built. Potential blockage of view corridors and shadow studies are also frequently analyzed through the use of CAD.

Types

There are several different types of CAD. Each of these different types of CAD systems require the operator to think differently about how he or she will use them and he or she must design their virtual components in a different manner for each.

There are many producers of the lower-end 2D systems, including a number of free and open source programs. These provide an approach to the drawing process without all the fuss over scale and placement on the drawing sheet that accompanied hand drafting, since these can be adjusted as required during the creation of the final draft.

3D wireframe is basically an extension of 2D drafting. Each line has to be manually inserted into the drawing. The final product has no mass properties associated with it and cannot have features directly added to it, such as holes. The operator approaches these in a similar fashion to the 2D systems, although many 3D systems allow using the wireframe model to make the final engineering drawing views.

3D "dumb" solids (programs incorporating this technology include AutoCAD) are created in a way analogous to manipulations of real world objects. Basic three-dimensional geometric forms (prisms, cylinders, spheres, and so on) have solid volumes added or subtracted from them, as if assembling or cutting real-world objects. Two-dimensional projected views can easily be generated from the models. Basic 3D solids

don't usually include tools to easily allow motion of components, set limits to their motion, or identify interference between components.

3D parametric solid modeling require the operator to use what is referred to as "design intent". The objects and features created are adjustable. Any future modifications will be simple, difficult, or nearly impossible, depending on how the original part was created. One must think of this as being a "perfect world" representation of the component. If a feature was intended to be located from the center of the part, the operator needs to locate it from the center of the model, not, perhaps, from a more convenient edge or an arbitrary point, as he could when using "dumb" solids. Parametric solids require the operator to consider the consequences of his actions carefully.

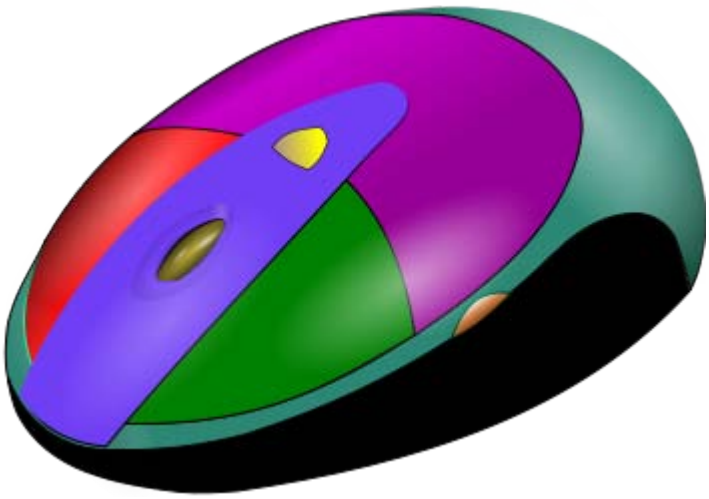
Some software packages provide the ability to edit parametric and non-parametric geometry without the need to understand or undo the design intent history of the geometry by use of direct modeling functionality. This ability may also include the additional ability to infer the correct relationships between selected geometry (e.g., tangency, concentricity) which makes the editing process less time and labor intensive while still freeing the engineer from the burden of understanding the model's design intent history. These kind of non history based systems are called Explicit Modellers or Direct CAD Modelers. The first Explicit Modeling system was introduced to the world at the end of 80's by Hewlett-Packard under the name SolidDesigner.

Draft views are able to be generated easily from the models. Assemblies usually incorporate tools to represent the motions of components, set their limits, and identify interference. The tool kits available for these systems are ever increasing; including 3D piping and injection mold designing packages.

Mid range software are integrating parametric solids more easily to the end user: integrating more intuitive functions (SketchUp), using the best of both 3D dumb solids and parametric characteristics (VectorWorks), making very real-view scenes in relative few steps (Cinema4D) or offering all-in-one (form•Z).

Top end systems offer the capabilities to incorporate more organic, aesthetics and ergonomic features into designs (Catia, GenerativeComponents). Freeform surface modelling is often combined with solids to allow the designer to create products that fit the human form and visual requirements as well as they interface with the machine.

Technology



A CAD model of a mouse

Originally software for Computer-Aided Design systems was developed with computer languages such as Fortran, but with the advancement of object-oriented programming methods this has radically changed. Typical modern parametric feature based modeler and freeform surface systems are built around a number of key C modules with their own APIs. A CAD system can be seen as built up from the interaction of a graphical user interface (GUI) with NURBS geometry and/or boundary representation (B-rep) data via a geometric modeling kernel. A geometry constraint engine may also be employed to manage the associative relationships between geometry, such as wireframe geometry in a sketch or components in an assembly.

Unexpected capabilities of these associative relationships have led to a new form of prototyping called digital prototyping. In contrast to physical prototypes, which entail manufacturing time in the design.

Today, CAD systems exist for all the major platforms (Windows, Linux, UNIX and Mac OS X); some packages even support multiple platforms.

Right now, no special hardware is required for most CAD software. However, some CAD systems can do graphically and computationally expensive tasks, so good graphics card, high speed (and possibly multiple) CPUs and large amounts of RAM are recommended.

The human-machine interface is generally via a computer mouse but can also be via a pen and digitizing graphics tablet. Manipulation of the view of the model on the screen is also sometimes done with the use of a spacemouse/SpaceBall. Some systems also support stereoscopic glasses for viewing the 3D model.

Effects

Beginning in the 1980s Computer-Aided Design programs reduced the need of draftsmen significantly, especially in small to mid-sized companies. Their affordability and ability to run on personal computers also allowed engineers to do their own drafting work, eliminating the need for entire departments. In today's world most, if not all, students in universities do not learn drafting techniques because they are not required to do so. The days of hand drawing for final drawings are almost obsolete. Universities such as New Jersey Institute of Technology no longer require the use of protractors and compasses to create drawings, instead there are several classes that focus on the use of CAD software such as Pro Engineer or IDEAS-MS.

Another consequence had been that since the latest advances were often quite expensive, small and even mid-size firms often could not compete against large firms who could use their computational edge for competitive purposes. Today, however, hardware and software costs have come down. Even high-end packages work on less expensive platforms and some even support multiple platforms. The costs associated with CAD implementation now are more heavily weighted to the costs of training in the use of these high level tools, the cost of integrating a CAD/CAM/CAE PLM using enterprise across multi-CAD and multi-platform environments and the costs of modifying design work flows to exploit the full advantage of CAD tools. CAD vendors have effectively lowered these training costs. These methods can be split into three categories:

1. Improved and simplified user interfaces. This includes the availability of “role” specific tailorable user interfaces through which commands are presented to users in a form appropriate to their function and expertise.
2. Enhancements to application software. One such example is improved design-in-context, through the ability to model/edit a design component from within the context of a large, even multi-CAD, active digital mockup.
3. User oriented modeling options. This includes the ability to free the user from the need to understand the design intent history of a complex intelligent model.

Skyboxes

Perhaps the most trivial application of cube mapping is to create pre-rendered panoramic sky images which are then rendered by the graphical engine as faces of a cube at practically infinite distance with the view point located in the center of the cube. The perspective projection of the cube faces done by the graphics engine undoes the effects of projecting the environment to create the cube map, so that the observer experiences an illusion of being surrounded by the scene which was used to generate the skybox. This technique has found a widespread use in video games since it allows designers to add complex (albeit not explorable) environments to a game at almost no performance cost.

Skylight Illumination

Cube maps can be useful for modelling outdoor illumination accurately. Simply modelling sunlight as a single infinite light oversimplifies outdoor illumination and results in unrealistic lighting. Although plenty of light does come from the sun, the scattering of rays in the atmosphere causes the whole sky to act as a light source (often referred to as skylight illumination).However, by using a cube map the diffuse contribution from skylight illumination can be captured. Unlike environment maps where the reflection vector is used, this method accesses the cube map based on the surface normal vector to provide a fast approximation of the diffuse illumination from the skylight. The one downside to this method is that computing cube maps to properly represent a skylight is very complex; however, a considerable amount of research has been done to effectively model skylight illumination.

Dynamic Reflection

Basic environment mapping uses a static cube map - although the object can be moved and distorted, the reflected environment stays consistent. However, a cube map texture can be consistently updated to represent a dynamically changing environment (for example, trees swaying in the wind). A simple yet costly way to generate dynamic reflections, involves building the cube maps at runtime for every frame. Although this is far less efficient than static mapping because of additional rendering steps, it can still be performed at interactive rates.

Unfortunately, this technique does not scale well when multiple reflective objects are present. A unique dynamic environment map is usually required for each reflective object. Also, further complications are added if reflective objects can reflect each other - dynamic cube maps can be recursively generated approximating the effects normally generated using raytracing.

Global Illumination

An algorithm for global illumination computation at interactive rates using a cube-map data structure, was presented at ICCVG 2002.

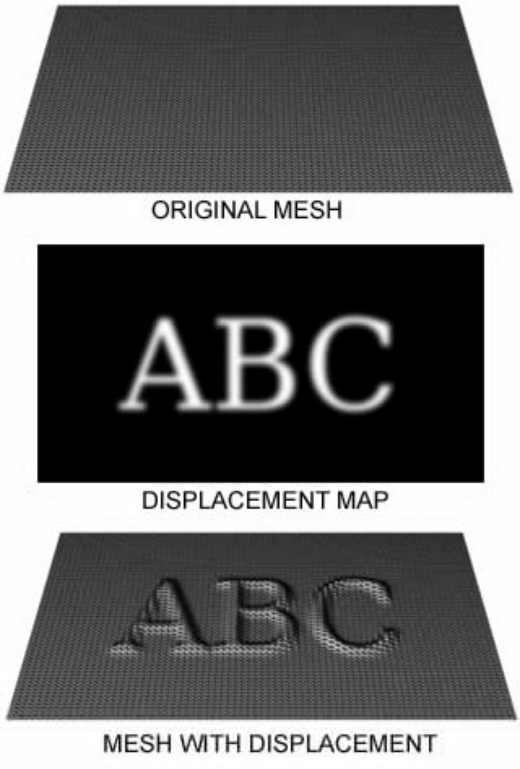
Projection textures

Another application which found widespread use in video games, projective texture mapping relies on cube maps to project images of an environment onto the surrounding scene; for example, a point light source is tied to a cube map which is a panoramic image shot from inside a lantern cage or a window frame through which the light is filtering. This enables a game designer to achieve realistic lighting without having to complicate the scene geometry or resort to expensive real-time shadow volume computations.

Chapter 14

Displacement Mapping and Parallax Mapping

Displacement mapping



Displacement mapping

Displacement mapping is an alternative computer graphics technique in contrast to bump mapping, normal mapping, and parallax mapping, using a (procedural-) texture- or height map to cause an effect where the actual geometric position of points over the textured surface are *displaced*, often along the local surface normal, according to the value the texture function evaluates to at each point on the surface. It gives surfaces a great sense of depth and detail, permitting in particular self-occlusion, self-shadowing

and silhouettes; on the other hand, it is the most costly of this class of techniques owing to the large amount of additional geometry.

For years, displacement mapping was a peculiarity of high-end rendering systems like PhotoRealistic RenderMan, while realtime APIs, like OpenGL and DirectX, are only starting to use this feature. One of the reasons for this is that the original implementation of displacement mapping required an adaptive tessellation of the surface in order to obtain enough micropolygons whose size matched the size of a pixel on the screen.

Meaning of the term in different contexts

Displacement mapping includes the term mapping which refers to a texture map being used to modulate the displacement strength. The displacement direction is usually the local surface normal. Today, many renderers allow programmable shading which can create high quality (multidimensional) procedural textures and patterns at arbitrary high frequencies. The use of the term mapping becomes arguable then, as no texture map is involved anymore. Therefore, the broader term **displacement** is often used today to refer to a super concept that also includes displacement based on a texture map.

Renderers using the REYES algorithm, or similar approaches based on micropolygons, have allowed displacement mapping at arbitrary high frequencies since they became available almost 20 years ago.

The first commercially available renderer to implement a micropolygon displacement mapping approach through REYES was Pixar's PhotoRealistic RenderMan. Micropolygon renderers commonly tessellate geometry themselves at a granularity suitable for the image being rendered. That is: the modeling application delivers high-level primitives to the renderer. Examples include true NURBS- or subdivision surfaces. The renderer then tessellates this geometry into micropolygons at render time using view-based constraints derived from the image being rendered.

Other renderers that require the modeling application to deliver objects pre-tessellated into arbitrary polygons or even triangles have defined the term displacement mapping as moving the vertices of these polygons. Often the displacement direction is also limited to the surface normal at the vertex. While conceptually similar, those polygons are usually a lot larger than micropolygons. The quality achieved from this approach is thus limited by the geometry's tessellation density a long time before the renderer gets access to it.

This difference between displacement mapping in micropolygon renderers vs. displacement mapping in a non-tessellating (macro)polygon renderers can often lead to confusion in conversations between people whose exposure to each technology or implementation is limited. Even more so, as in recent years, many non-micropolygon renderers have added the ability to do displacement mapping of a quality similar to what a micropolygon renderer is able to deliver, naturally. To distinguish between the crude pre-tessellation-based displacement these renderers did before, the term **sub-pixel displacement** got introduced to describe this feature.

Sub-pixel displacement commonly refers to finer re-tessellation of geometry that was already tessellated into polygons. This re-tessellation results in micropolygons or often microtriangles. The vertices of these then get moved along their normals to archive the displacement mapping.

True micropolygon renderers have always been able to do what sub-pixel-displacement achieved only recently, but at a higher quality and in arbitrary displacement directions.

Recent developments seem to indicate that some of the renderers which use sub-pixel displacement move towards supporting higher level geometry too. As the vendors of these renderers are likely to keep using the term sub-pixel displacement, this will probably lead to more obfuscation of what displacement mapping really stands for, in 3D computer graphics.

In reference to Microsoft's proprietary High Level Shader Language, displacement mapping can be interpreted as a kind of "vertex-texture mapping" where the values of the texture map do not alter pixel colors (as is much more common), but instead change the position of vertices. Unlike bump, normal and parallax mapping, all of which can be said to "fake" the behavior of displacement mapping, in this way a genuinely *rough* surface can be produced from a texture. It has to be used in conjunction with adaptive tessellation techniques (that increases the number of rendered polygons according to current viewing settings) to produce highly detailed meshes.

Parallax mapping

Example of parallax mapping. The walls are textured with parallax maps. Screenshot taken from one of the base examples of the open source Irrlicht 3d engine.

Parallax mapping (also called **offset mapping** or **virtual displacement mapping**) is an enhancement of the bump mapping or normal mapping techniques applied to textures in 3D rendering applications such as video games. To the end user, this means that textures such as stone walls will have more apparent depth and thus greater realism with less of an influence on the performance of the simulation. Parallax mapping was introduced by Tomomichi Kaneko et al. in 2001.

Parallax mapping is implemented by displacing the texture coordinates at a point on the rendered polygon by a function of the view angle in tangent space (the angle relative to the surface normal) and the value of the height map at that point. At steeper view-angles, the texture coordinates are displaced more, giving the illusion of depth due to parallax effects as the view changes.

Parallax mapping described by Kaneko is a single step process that does not account for occlusion. Subsequent enhancements have been made to the algorithm incorporating iterative approaches to allow for occlusion and accurate silhouette rendering.

Steep parallax mapping

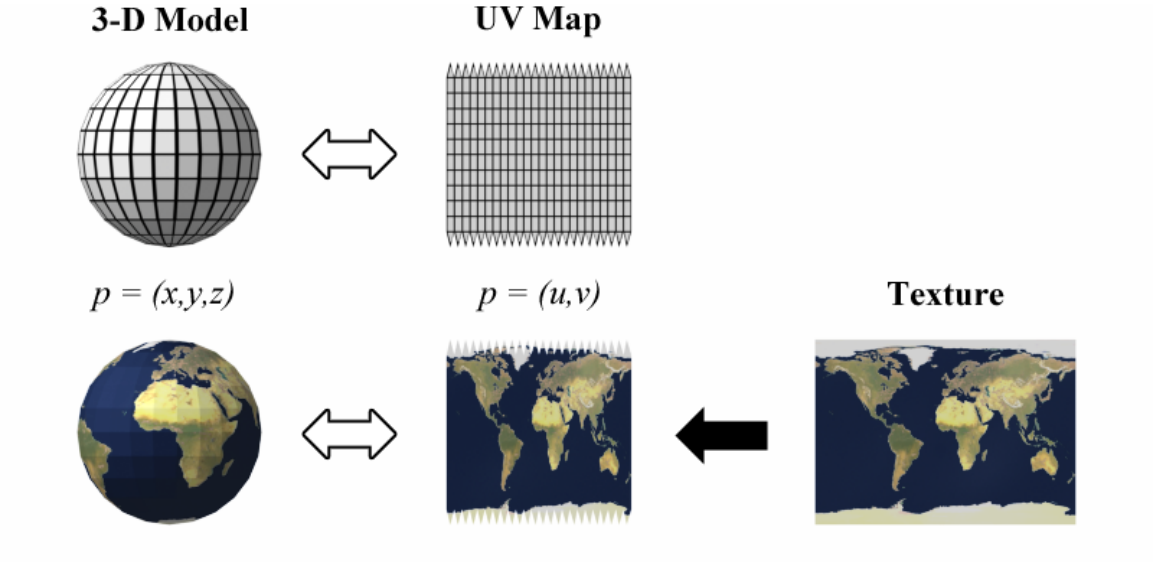
Steep parallax mapping is one name for the class of algorithms that trace rays against heightfields. The idea is to walk along a ray that has entered the heightfield's volume, finding the intersection point of the ray with the heightfield. This closest intersection is what part of the heightfield is truly visible. **Relief mapping** and **parallax occlusion mapping** are other common names for these techniques.

Interval mapping improves on the usual binary search done in relief mapping by creating a line between known inside and outside points and choosing the next sample point by intersecting this line with a ray, rather than using the midpoint as in a traditional binary search.

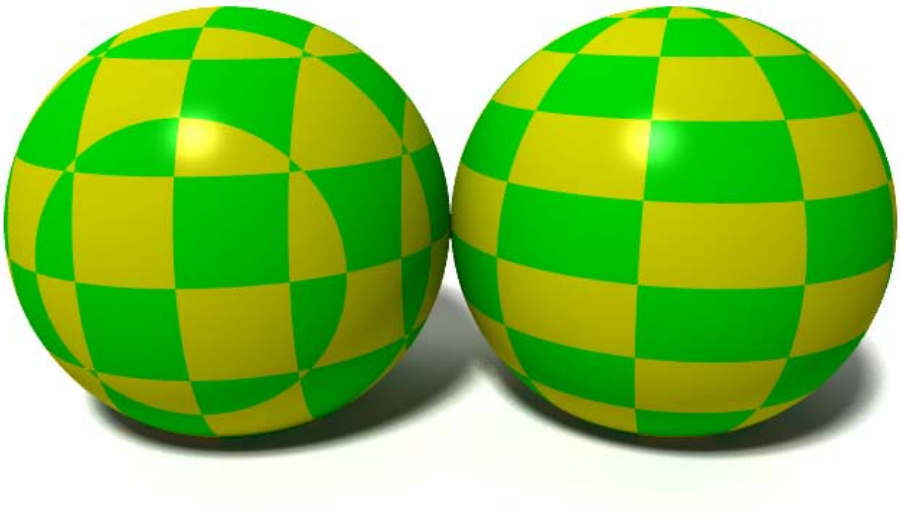
Chapter 15

UV Mapping, Sphere Mapping and UVW Mapping

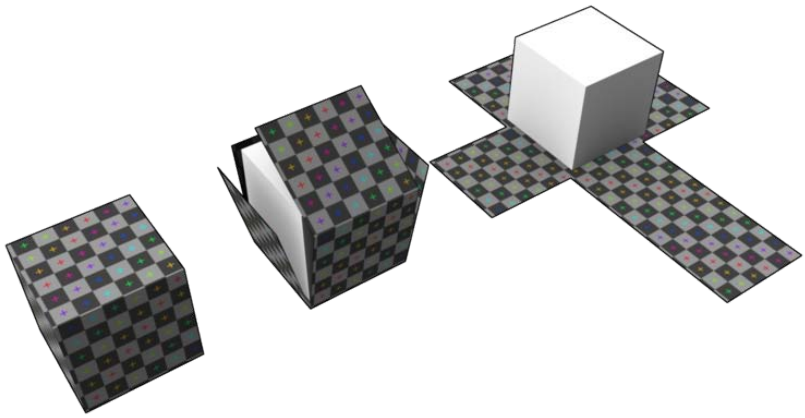
UV mapping



The application of a texture in the UV space related to the effect in 3D



A checkered sphere, without and with UV mapping (3D checkered or 2D checkered)



A representation of the UV mapping of a cube. The flattened cube net may then be textured to texture the cube.

UV mapping is the 3D modeling process of making a 2D image representation of a 3D model.

UV mapping

The process of UV Mapping transforms the texture map onto the 3D object. In contrast to "X", "Y" and "Z" Cartesian Coordinates, which are the coordinates for the original 3D object in the modeling space, another set of coordinates is required to describe the surface of the mesh, so the letters "U" and "V" are used.

UV texturing permits polygons that make up a 3D object to be painted with color from an image. The image is called a UV texture map, but it's just an ordinary image. The UV mapping process involves assigning pixels in the image to surface mappings on the polygon, usually done by "programmatically" copying a triangle shaped piece of the image map and pasting it onto a triangle on the object. UV is the alternative to XY, it only maps into a texture space than into the geometric space of the object. But the rendering computation uses the UV texture coordinates to determine how to paint the three dimensional surface.

In the example to the right, a sphere is given a checkered texture, first without and then with UV mapping. Without UV mapping, the checkers tile XYZ space and the texture is carved out of the sphere. With UV mapping, the checkers tile UV space and points on the sphere map to this space according to their latitude and longitude.

When a model is created as a polygon mesh using a 3D modeler, UV coordinates can be generated for each vertex in the mesh. One way is for the 3D modeler to unfold the triangle mesh at the seams, automatically laying out the triangles on a flat page. If the mesh is a UV sphere, for example, the modeler might transform it into a equirectangular projection. Once the model is unwrapped, the artist can paint a texture on each triangle individually, using the unwrapped mesh as a template. When the scene is rendered, each triangle will map to the appropriate texture from the "decal sheet".

A UV map can either be generated automatically by the software application, made manually by the artist, or some combination of both. Often a UV map will be generated, and then the artist will adjust and optimize it to minimize seams and overlaps. If the model is symmetric, the artist might overlap opposite triangles to allow painting both sides simultaneously.

UV coordinates are applied per face, not per vertex. This means a shared vertex can have different UV coordinates in each of its triangles, so adjacent triangles can be cut apart and positioned on different areas of the texture map.

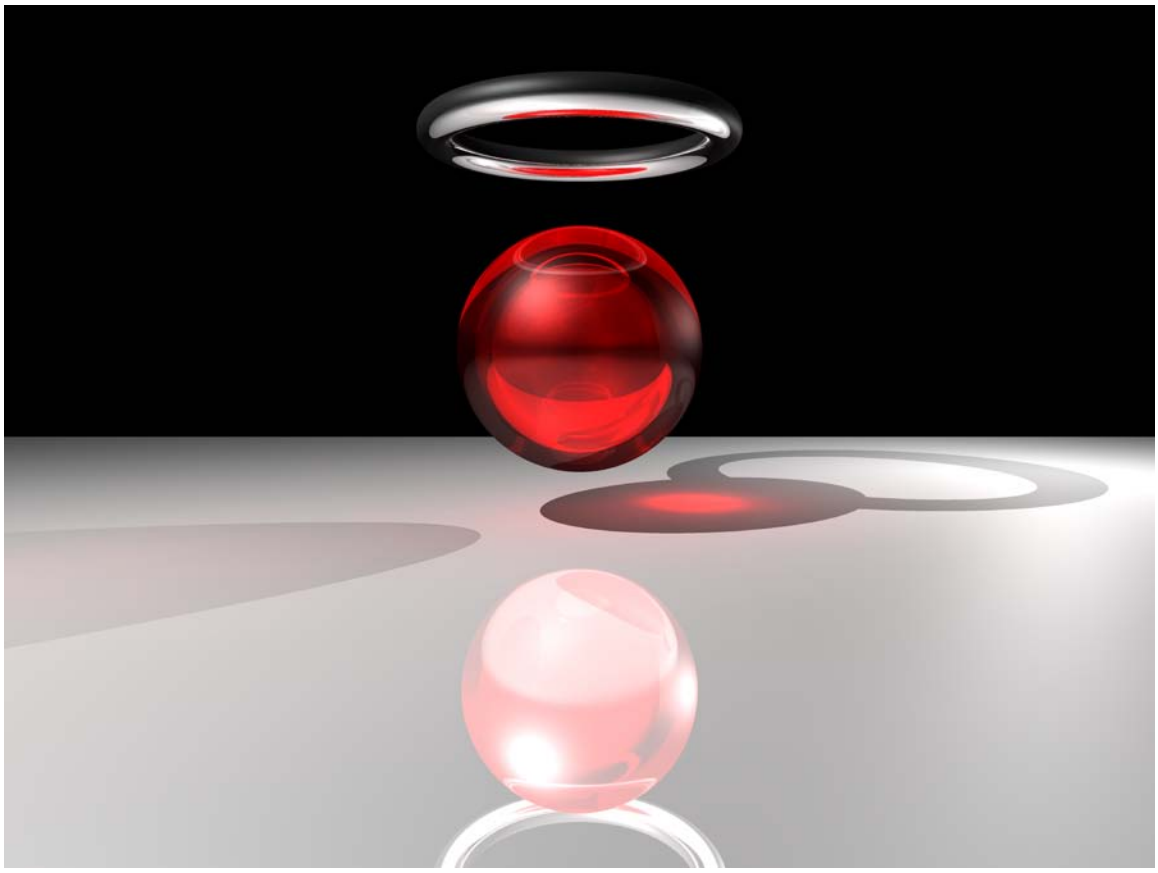
The UV Mapping process at its simplest requires three steps: unwrapping the mesh, creating the texture, and applying the texture.

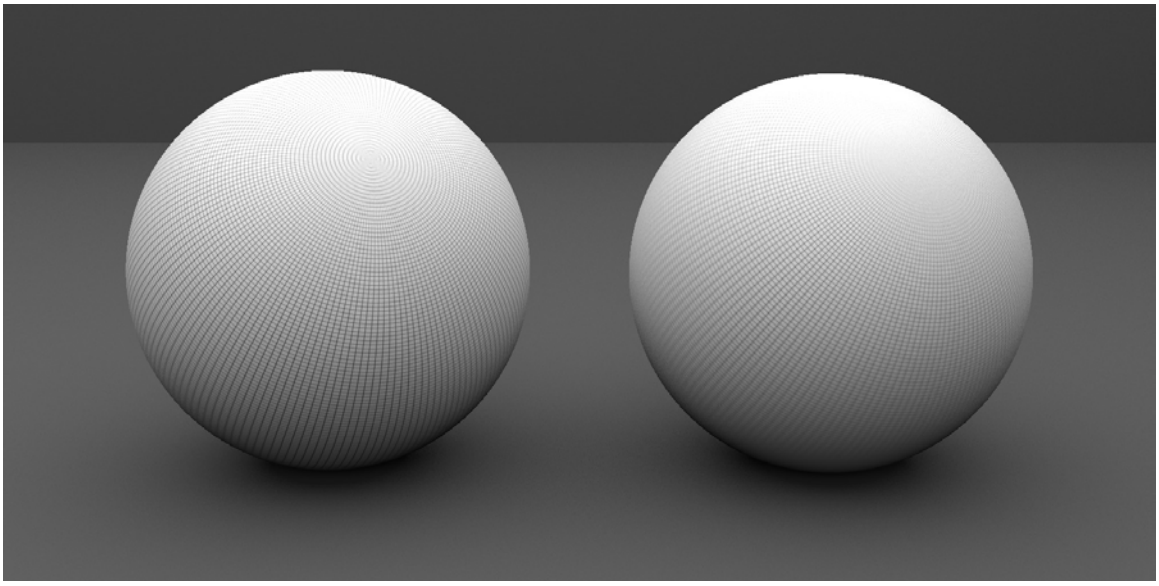
Finding UV on a sphere

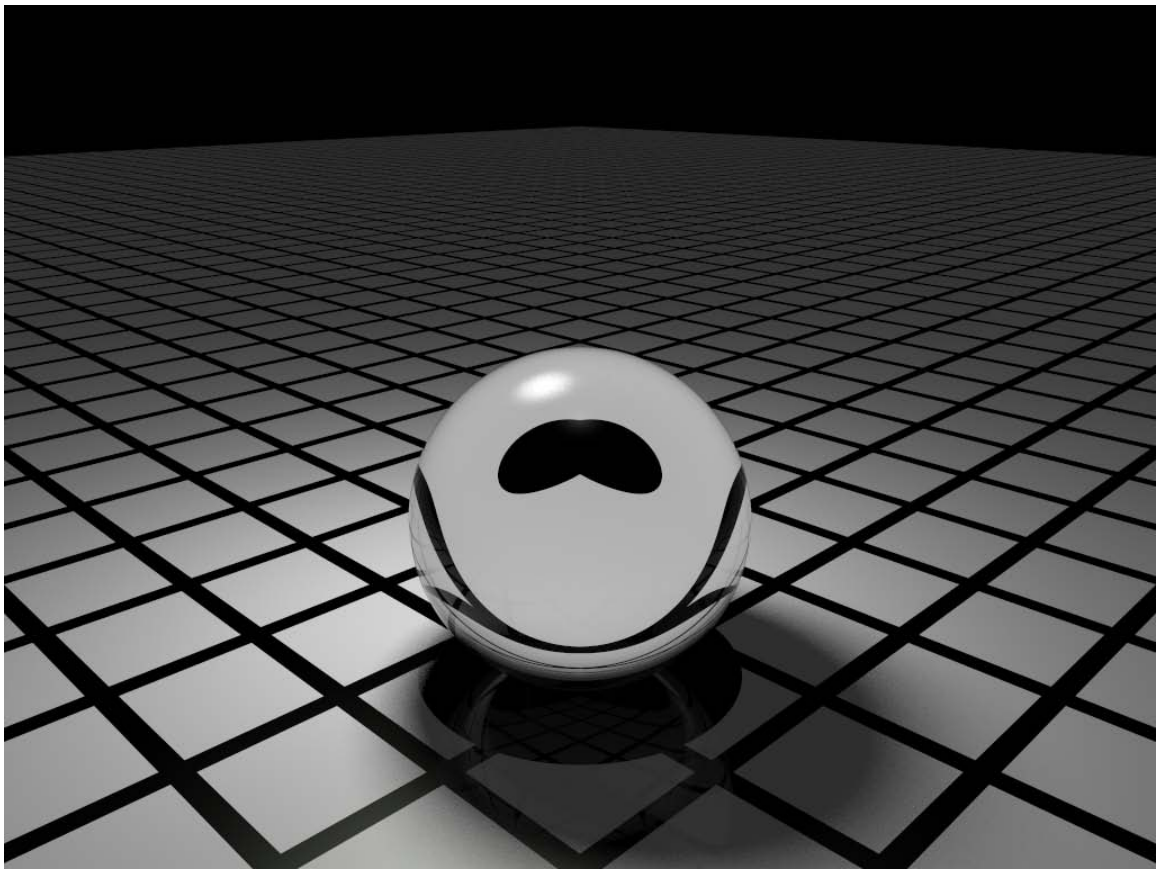
UV coordinates represent a projection of the unit space vector onto the xy-plane.

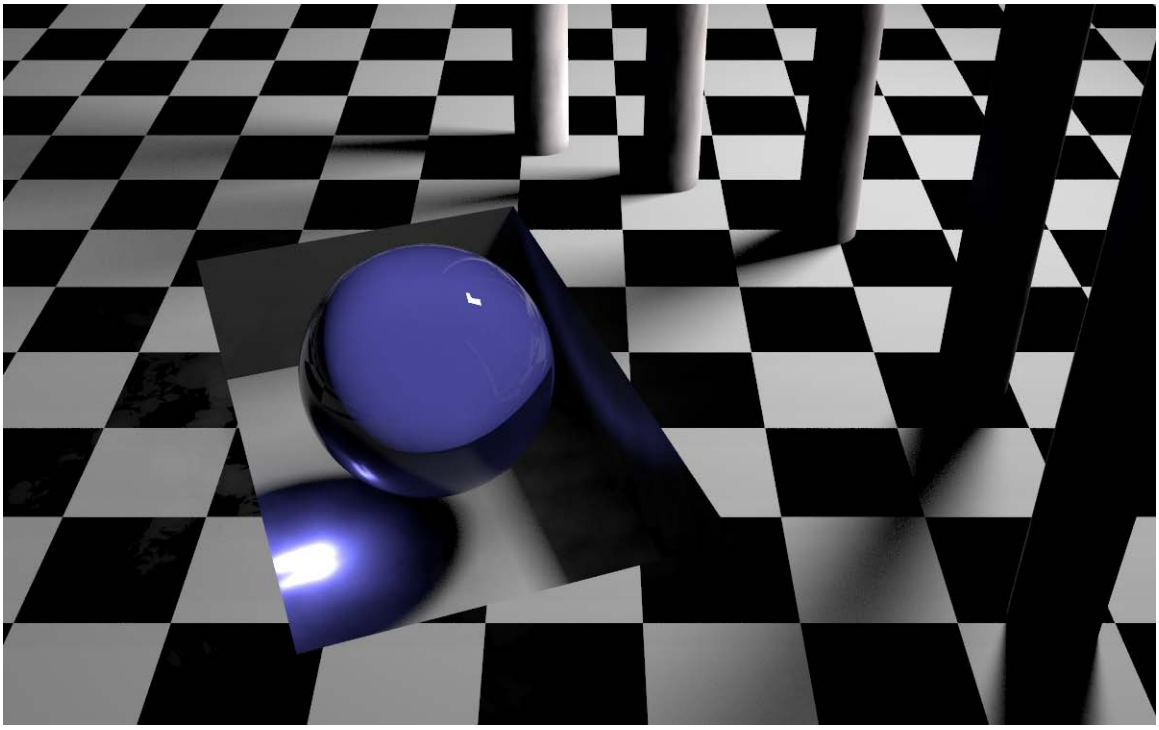
$$u = \sin\theta\cos\phi = \frac{x}{\sqrt{x^2 + y^2 + z^2}}$$
$$v = \sin\theta\sin\phi = \frac{y}{\sqrt{x^2 + y^2 + z^2}}$$

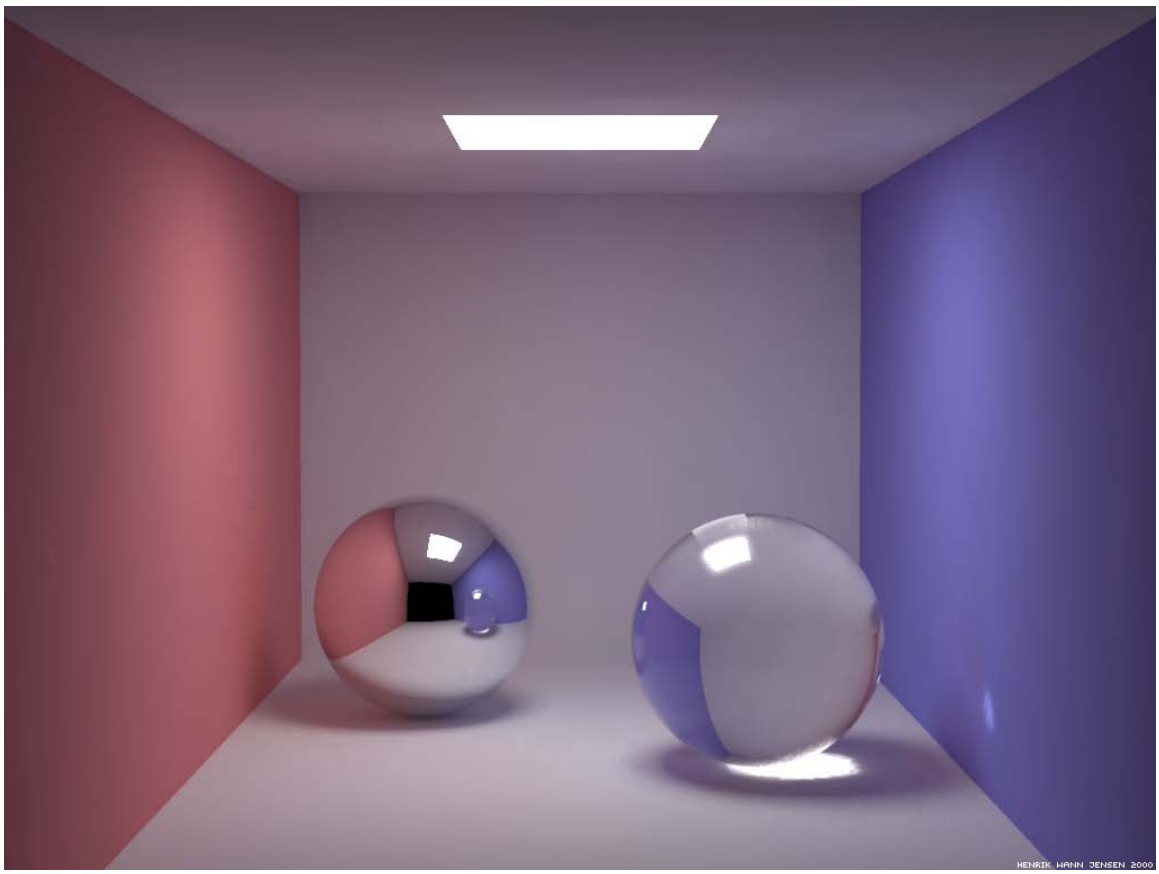
Sphere mapping

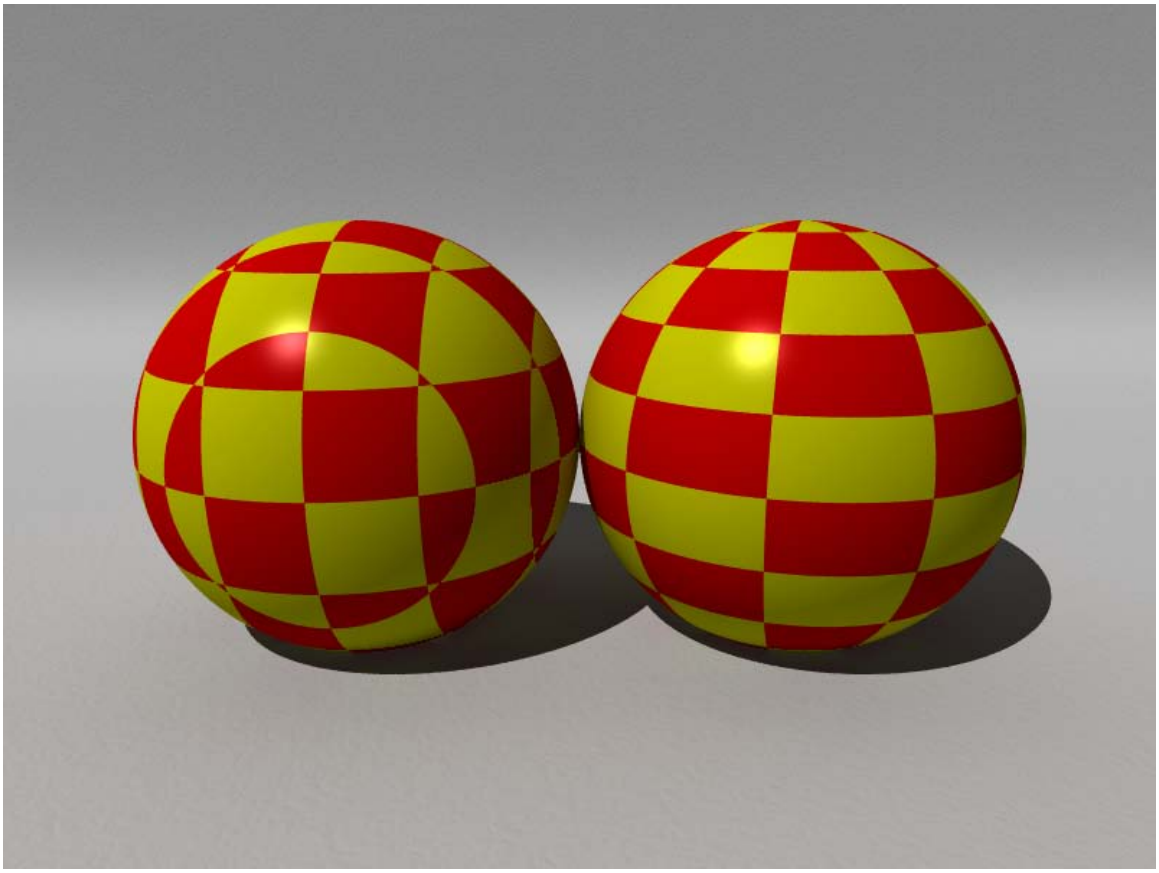


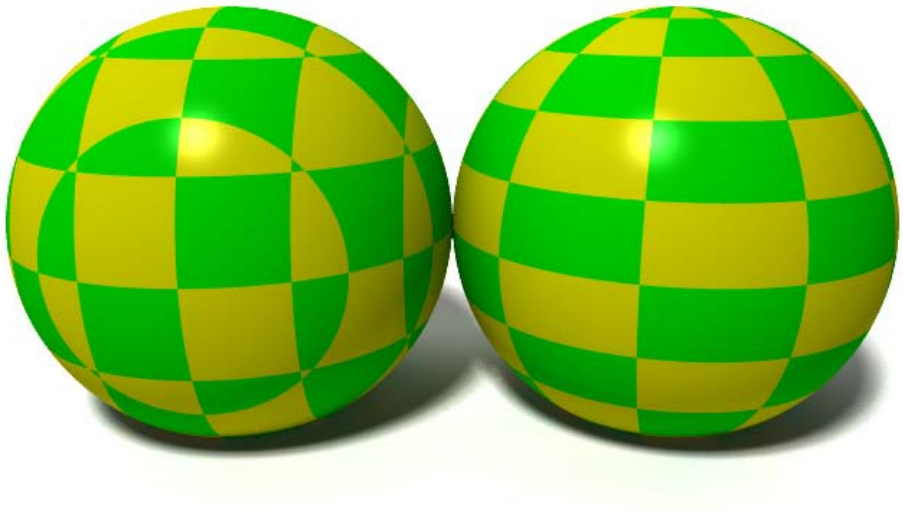







HENRIK WANN JENSEN 2000





In computer graphics, **sphere mapping** (or **spherical environment mapping**) is a type of reflection mapping that approximates reflective surfaces by considering the environment to be an infinitely far-away spherical wall. This environment is stored as a texture depicting what a mirrored sphere would look like if it were placed into the environment, using an orthographic projection (as opposed to one with perspective). This texture contains reflective data for the entire environment, except for the spot directly behind the sphere.

To use this data, the surface normal of the object, view direction from the object to the camera, and/or reflected direction from the object to the environment is used to calculate a texture coordinate to look up in the aforementioned texture map. The result appears like the environment is reflected in the surface of the object that is being rendered.

Usage example

In the simplest case for generating texture coordinates, suppose:

- The map has been created as above, looking at the sphere along the z-axis.
- The texture coordinate of the center of the map is (0,0), and the sphere's image has radius 1.
- We are rendering an image in the same exact situation as the sphere, but the sphere has been replaced with a reflective object.
- The image being created is orthographic, or the viewer is infinitely far away, so that the view direction does not change as one moves across the image.

At texture coordinate (x,y), note that the depicted location on the sphere is (x,y,z) (where z is $\sqrt{1 - x^2 - y^2}$), and the normal at that location is also < x,y,z > . However, we are given the reverse task (a normal for which we need to produce a texture map coordinate). So the texture coordinate corresponding to normal < x,y,z > is (x,y).

UVW mapping

UVW mapping is a mathematical technique for coordinate mapping. In computer graphics, it is most commonly a $\mathbb{R}^2$to $\mathbb{R}^3$map, suitable for converting a 2D image (a texture) to a three dimensional object of a given topology. "UVW", like the standard Cartesian coordinate system, has three dimensions; the third dimension allows texture maps to wrap in complex ways onto irregular surfaces. Each point in a UVW map corresponds to a point on the surface of the object. The graphic designer or programmer generates the specific mathematical function to implement the map, so that points on the texture are assigned to (XYZ) points on the target surface. Generally speaking, the more orderly the unwrapped polygons are, the easier it is for the texture artist to paint features onto the texture. Once the texture is finished, all that has to be done is to wrap the UVW map back onto the object, projecting the texture in a way that is far more flexible and advanced, preventing graphic artifacts that accompany more simplistic texture mappings such as planar projection. For this reason, UVW mapping is commonly used to texture map non-platonic solids, non-geometric primitives, and other irregularly-shaped objects, such as characters and furniture.

Chapter 16

3D Projection and Texture Filtering

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 affect 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 point in 3D space that is to be projected.
- $\mathbf{c}_{x,y,z}$- the location of the camera.
- $\theta_{x,y,z}$- The rotation of the camera. When $\mathbf{c}_{x,y,z} = \langle 0, 0, 0 \rangle$, and $\theta_{x,y,z} = \langle 0, 0, 0 \rangle$, the 3D vector <1,2,0> is projected to the 2D vector <1,2>.
- $\mathbf{e}_{x,y,z}$- the viewer's position relative to the display surface.

Which results in:

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

First, we define a point $\mathbf{d}_{x,y,z}$ as a translation of point $\mathbf{a}$ into a coordinate system defined by $\mathbf{c}$. This is achieved by subtracting $\mathbf{c}$ from $\mathbf{a}$ and then applying a vector rotation matrix using $-\theta$ 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):

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

This representation correspond 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 ($\theta_{x,y,z} = \langle 0, 0, 0 \rangle$), 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

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

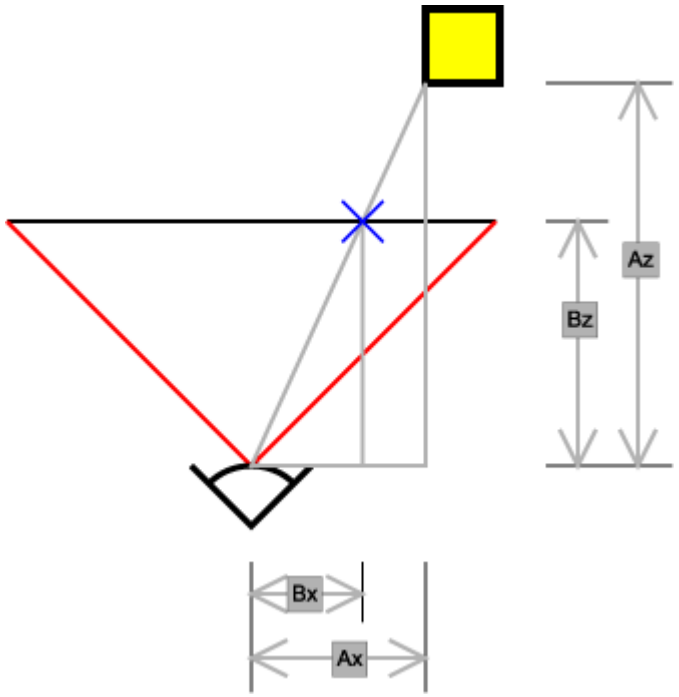
in conjunction with an argument using similar triangles, leads to division by the homogeneous coordinate, giving

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

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)

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)

Texture filtering

In computer graphics, **texture filtering** is the method used to determine the texture color for a texture mapped pixel, using the colors of nearby texels (pixels of the texture). In short, it blends the texture pixels together by breaking them up into tinier pixels. Another term for texture filtering is called texture smoothing. There are many methods of texture filtering, which make different trade-offs between computational complexity and image quality. Since texture filtering is an attempt to find a value at some point given a set of discrete samples at nearby points, it is a form of interpolation.

Application

Texture filtering is a cheap approximation to anti-aliasing. Soft textures, especially height maps, look better when interpolated.

The need for filtering

During the texture mapping process, a 'texture lookup' takes place to find out where on the texture each pixel center falls. Since the textured surface may be at an arbitrary distance and orientation relative to the viewer, one pixel does not usually correspond directly to one texel. Some form of filtering has to be applied to determine the best color for the pixel. Insufficient or incorrect filtering will show up in the image as artifacts (errors in the image), such as 'blockiness', jaggies, or shimmering.

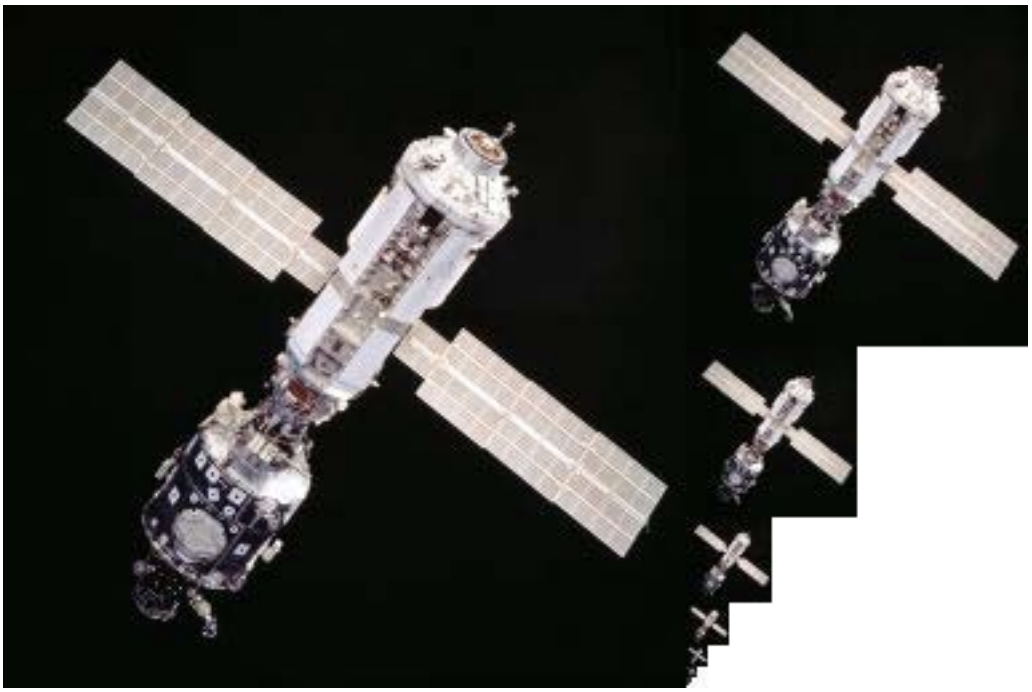
There can be different types of correspondence between a pixel and the texel/texels it represents on the screen. These depend on the position of the textured surface relative to the viewer, and different forms of filtering are needed in each case. Given a square texture mapped on to a square surface in the world, at some viewing distance the size of one screen pixel is exactly the same as one texel. Closer than that, the texels are larger than screen pixels, and need to be scaled up appropriately - a process known as *texture magnification*. Farther away, each texel is smaller than a pixel, and so one pixel covers multiple texels. In this case an appropriate color has to be picked based on the covered texels, via *texture minification*. Graphics APIs such as OpenGL allow the programmer to set different choices for minification and magnification filters.

Note that even in the case where the pixels and texels are exactly the same size, one pixel will not necessarily match up exactly to one texel - it may be misaligned, and cover parts of up to four neighboring texels. Hence some form of filtering is still required.

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.

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. 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.

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.

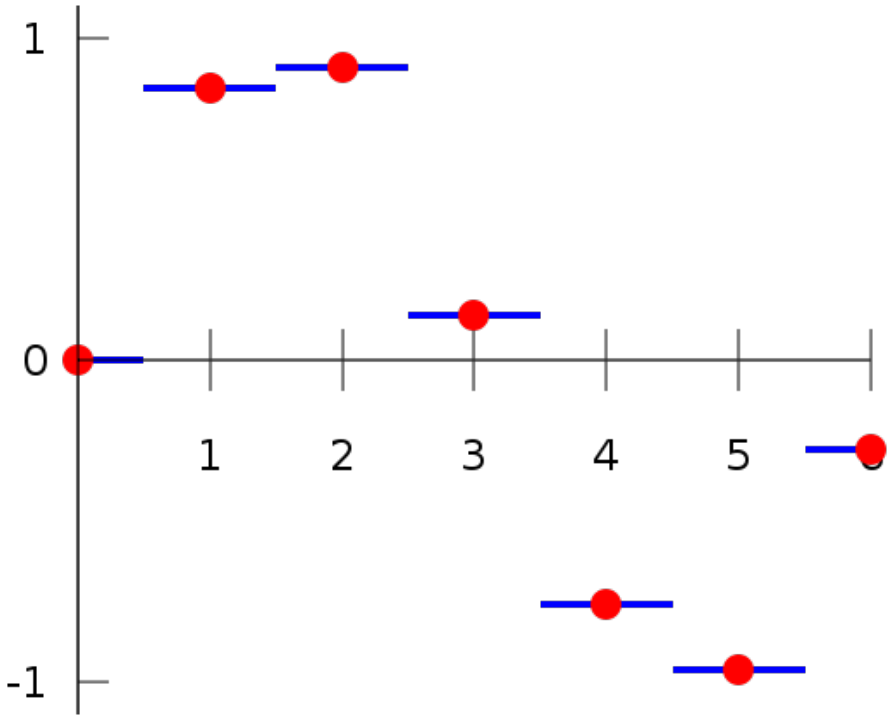
Origin

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.

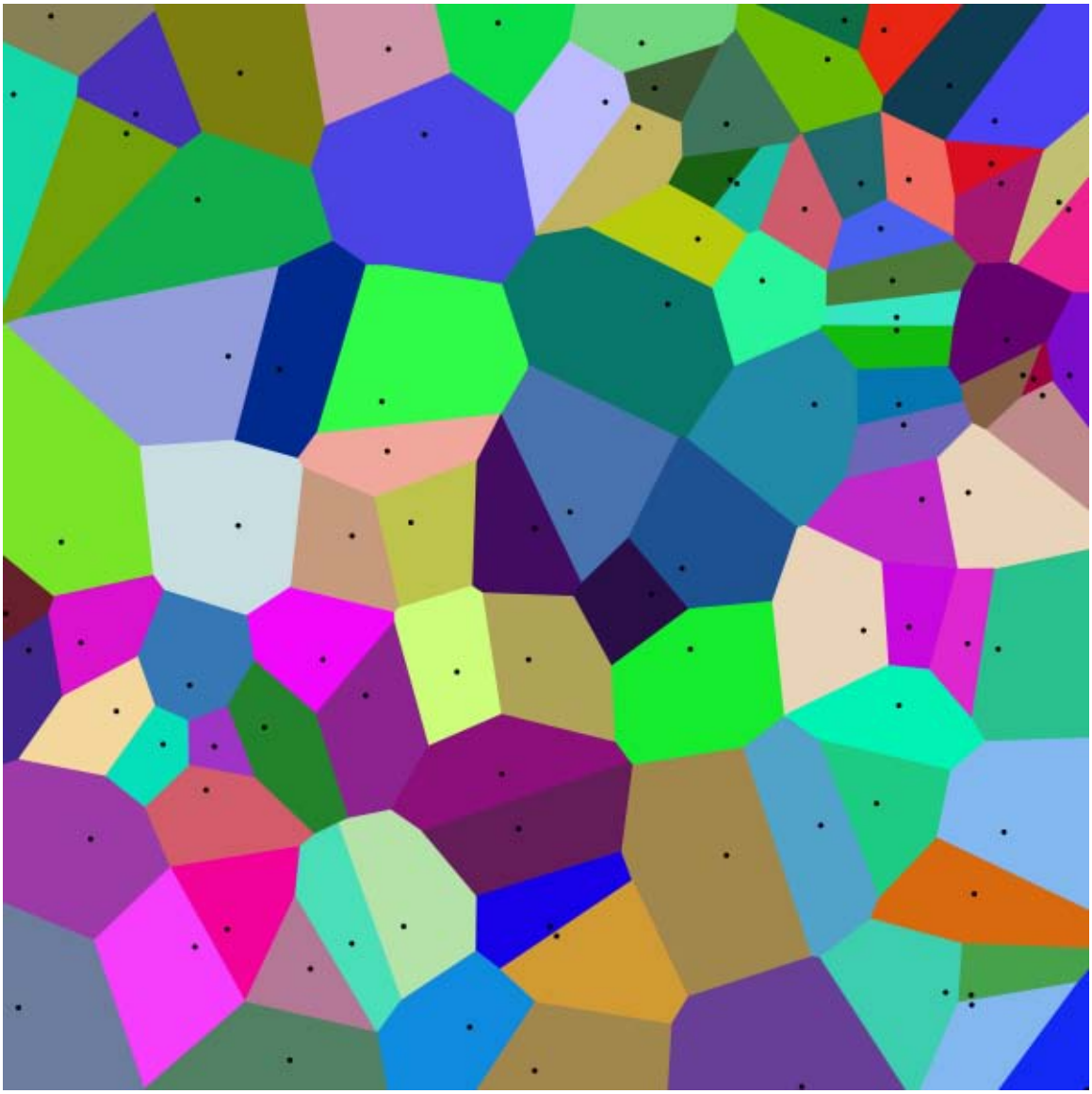
Filtering methods

This section lists the most common texture filtering methods, in increasing order of computational cost and image quality.

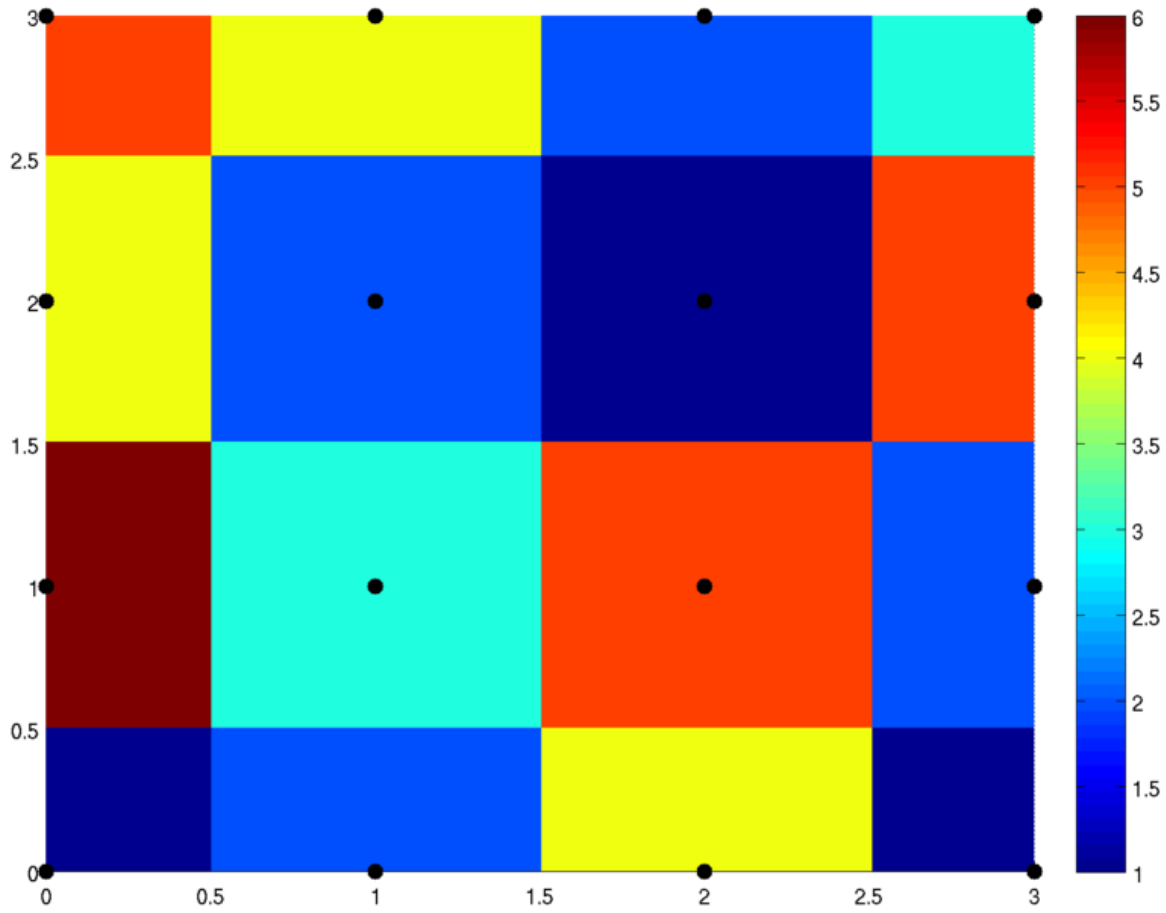
1. Nearest-neighbor interpolation



Nearest neighbor interpolation (blue lines) in one dimension on a (uniform) dataset (red points)



Example of nearest neighbor interpolation of a random set of points (black dots) in 2D. Each coloured cell indicates the area in which all the points have the black point in the cell as their nearest black point.



Nearest neighbor interpolation on a uniform 2D grid (black points)

Nearest-neighbor interpolation (also known as **proximal interpolation** or **point sampling** in some contexts) is a simple method of multivariate interpolation in 1 or more dimensions. Interpolation is the problem of approximating the value for a non-given point in some space, when given some values of points *around* that point. The nearest neighbor algorithm simply selects the value of the nearest point, and does not consider the values of other neighboring points at all, yielding a piecewise-constant interpolant. The algorithm is very simple to implement, and is commonly used (usually along with mipmapping) in real-time 3D rendering to select color values for a textured surface.

Connection to Voronoi diagram

For a given set of points in space, a Voronoi diagram is a decomposition of space into cells, one for each given point, so that anywhere in space, the closest given point is inside the cell. This is equivalent to nearest neighbour interpolation, by assigning the function value at the given point to all the points inside the cell. The figures on the right side show by colour the shape of the cells.

2. Bilinear filtering

A zoomed small portion of a bitmap, using nearest-neighbor filtering *(left)*, bilinear filtering *(center)*, and bicubic filtering *(right)*.

Bilinear filtering is a texture filtering method used to smooth textures when displayed larger or smaller than they actually are.

Most of the time, when drawing a textured shape on the screen, the texture is not displayed exactly as it is stored, without any distortion. Because of this, most pixels will end up needing to use a point on the texture that's 'between' texels, assuming the texels are points (as opposed to, say, squares) in the middle (or on the upper left corner, or anywhere else; it doesn't matter, as long as it's consistent) of their respective 'cells'. Bilinear filtering uses these points to perform bilinear interpolation between the four texels nearest to the point that the pixel represents (in the middle or upper left of the pixel, usually).

The formula

In these equations, u_k and v_k are the texture coordinates and y_k is the color value at point k. Values without a subscript refer to the pixel point; values with subscripts 0, 1, 2, and 3 refer to the texel points, starting at the top left, reading right then down, that immediately surround the pixel point. So y_0 is the color of the texel at texture coordinate (u_0, v_0). These are linear interpolation equations. We'd start with the bilinear equation, but since this is a special case with some elegant results, it is easier to start from linear interpolation.

$$y_a = y_0 + \frac{y_1 - y_0}{u_1 - u_0}(u - u_0)$$
$$y_b = y_2 + \frac{y_3 - y_2}{u_3 - u_2}(u - u_2)$$
$$y = y_a + \frac{y_b - y_a}{v_2 - v_0}(v - v_0)$$

Assuming that the texture is a square bitmap,

$$v_1 = v_0$$
$$v_2 = v_3$$
$$u_1 = u_3$$
$$u_2 = u_0$$

$$v_3 - v_0 = u_3 - u_0 = w$$

Are all true. Further, define

$$U = \frac{u - u_0}{w}$$
$$V = \frac{v - v_0}{w}$$

With these we can simplify the interpolation equations:

$$y_a = y_0 + (y_1 - y_0)U$$
$$y_b = y_2 + (y_3 - y_2)U$$
$$y = y_a + (y_b - y_a)V$$

And combine them:

$$y = y_0 + (y_1 - y_0)U + (y_2 - y_0)V + (y_3 - y_2 - y_1 + y_0)UV$$

Or, alternatively:

$$y = y_0(1 - U)(1 - V) + y_1U(1 - V) + y_2(1 - U)V + y_3UV$$

Which is rather convenient. However, if the image is merely scaled (and not rotated, sheared, put into perspective, or any other manipulation), it can be considerably faster to use the separate equations and store y_b (and sometimes y_a, if we are increasing the scale) for use in subsequent rows.

Sample code

This code assumes that the texture is square (an extremely common occurrence), that no mipmapping comes into play, and that there is only one channel of data (not so common. Nearly all textures are in color so they have red, green, and blue channels, and many have an alpha transparency channel, so we must make three or four calculations of y, one for each channel).

```
 double getBilinearFilteredPixelColor(Texture tex, double u, double v)
{
   u *= tex.size;
   v *= tex.size;
   int x = floor(u);
   int y = floor(v);
   double u_ratio = u - x;
   double v_ratio = v - y;
   double u_opposite = 1 - u_ratio;
   double v_opposite = 1 - v_ratio;
```

```
   double result = (tex[x][y]   * u_opposite  + tex[x+1][y]   *
u_ratio) * v_opposite +
                   (tex[x][y+1] * u_opposite  + tex[x+1][y+1] *
u_ratio) * v_ratio;
   return result;
 }
```

Limitations

Bilinear filtering is rather accurate until the scaling of the texture gets below half or above double the original size of the texture - that is, if the texture was 256 pixels in each direction, scaling it to below 128 or above 512 pixels can make the texture look bad, because of missing pixels or too much smoothness. Often, mipmapping is used to provide a scaled-down version of the texture for better performance; however, the transition between two differently-sized mipmaps on a texture in perspective using bilinear filtering can be very abrupt. Trilinear filtering, though somewhat more complex, can make this transition smooth throughout.

For a quick demonstration of how a texel can be missing from a filtered texture, here's a list of numbers representing the centers of boxes from an 8-texel-wide texture (in red and black), intermingled with the numbers from the centers of boxes from a 3-texel-wide down-sampled texture (in blue). The red numbers represent texels that would not be used in calculating the 3-texel texture at all.

0.0625, 0.1667, 0.1875, 0.3125, 0.4375, 0.5000, 0.5625, 0.6875, 0.8125, 0.8333, 0.9375

Special cases

Textures aren't infinite, in general, and sometimes one ends up with a pixel coordinate that lies outside the grid of texel coordinates. There are a few ways to handle this:

- Wrap the texture, so that the last texel in a row also comes right before the first, and the last texel in a column also comes right above the first. This works best when the texture is being tiled.
- Make the area outside the texture all one color. This may be of use for a texture designed to be laid over a solid background or to be transparent.
- Repeat the edge texels out to infinity. This works best if the texture is not designed to be repeated.

3. Trilinear filtering

Trilinear filtering is an extension of the bilinear texture filtering method, which also performs linear interpolation between mipmaps.

Bilinear filtering has several weaknesses that make it an unattractive choice in many cases: using it on a full-detail texture when scaling to a very small size causes accuracy problems from missed texels, and compensating for this by using multiple mipmaps throughout the polygon leads to abrupt changes in blurriness, which is most pronounced in polygons that are steeply angled relative to the camera.

To solve this problem, trilinear filtering interpolates between the results of bilinear filtering on the two mipmaps nearest to the detail required for the polygon at the pixel. If the pixel would take up 1/100 of the texture in one direction, trilinear filtering would interpolate between the result of filtering the 128*128 mipmap as y_1 with x_1 as 128, and the result of filtering on the 64*64 mipmap as y_2 with x_2 as 64, and then interpolate to x = 100.

The first step in this process is of course to determine how big in terms of the texture the pixel in question is. There are a few ways to do this, and the ones mentioned here are not necessarily representative of all of them.

- Use the distance along the texture between the current pixel and the pixel to its right (or left, or above, or below) as the size of the pixel.
- Use the smallest (or biggest, or average) of the various sizes determined by using the above method.
- Determine the uv-values of the corners of the pixel, use those to calculate the area of the pixel, and figure out how many pixels of exactly the same size would take up the whole texture.

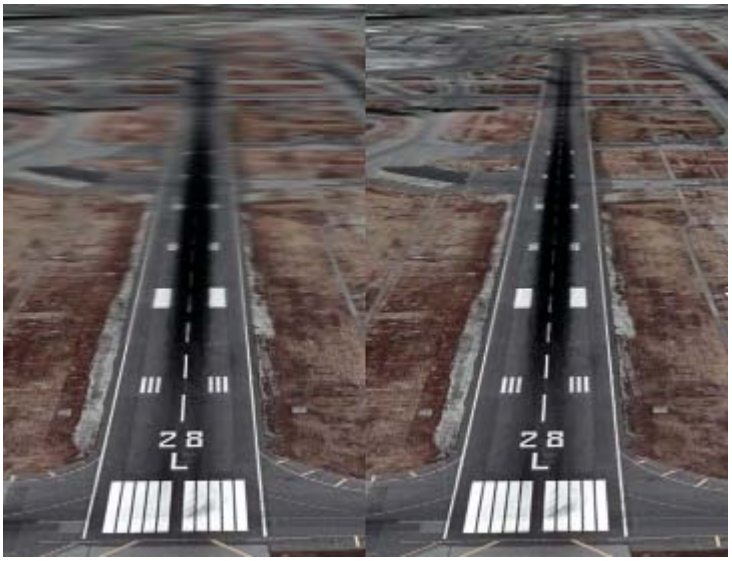
Once this is done the rest becomes easy: perform bilinear filtering on the two mipmaps with pixel sizes that are immediately larger and smaller than the calculated size of the pixel, and then interpolate between them as normal.

Since it uses both larger and smaller mipmaps, trilinear filtering cannot be used in places where the pixel is smaller than a texel on the original texture, because mipmaps larger than the original texture are not defined. Fortunately bilinear filtering still works, and can be used in these situations without worrying too much about abruptness because bilinear and trilinear filtering provide the same result when the pixel size is exactly the same as the size of a texel on the appropriate mipmap.

Trilinear filtering still has weaknesses, because the pixel is still assumed to take up a square area on the texture. In particular, when a texture is at a steep angle compared to the camera, detail can be lost because the pixel actually takes up a narrow but long trapezoid: in the narrow direction, the pixel is getting information from more texels than it actually covers (so details are smeared), and in the long direction the pixel is getting

information from fewer texels than it actually covers (so details fall between pixels). To alleviate this, anisotropic ("direction dependent") filtering can be used.

4. Anisotropic filtering



An illustration of texture filtering methods showing trilinear MIP map texture on the left and 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 it 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 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 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.

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. A display can easily contain over a million pixels, and the desired frame rate tends to be as high as 30–60 frames per second or more, so the texture memory bandwidth can get very high (tens to hundreds of gigabytes per second) very quickly. Fortunately, several factors mitigate in favor of better performance. The probes themselves share cached texture samples, both inter- and intra-pixel. Even with 16-tap anisotropic filtering, not all 16 taps are always needed, because only distant highly oblique pixel fill tends to be highly anisotropic, and such fill tends to cover small regions of the screen, and finally magnification texture filters require no anisotropic filtering.

Chapter 17

Image Resolution

Image resolution describes the detail an image holds. The term applies to digital images, film images, and other types of images. Higher resolution means more image detail.

Image resolution can be measured in various ways. Basically, resolution quantifies how close lines can be to each other and still be visibly *resolved*. Resolution units can be tied to physical sizes (e.g. lines per mm, lines per inch), to the overall size of a picture (lines per picture height, also known simply as lines, or TV lines), or to angular subtenant. Line pairs are often used instead of lines; a line pair comprises a dark line and an adjacent light line. A *Line* (or *TV line*, TVL) is either a dark line or a light line. A resolution of 10 lines per millimeter means 5 dark lines alternating with 5 light lines, or 5 line pairs per millimeter (5 LP/mm). Photographic lens and film resolution are most often quoted in line pairs per millimeter.

Resolution of digital images

The resolution of digital images can be described in many different ways.

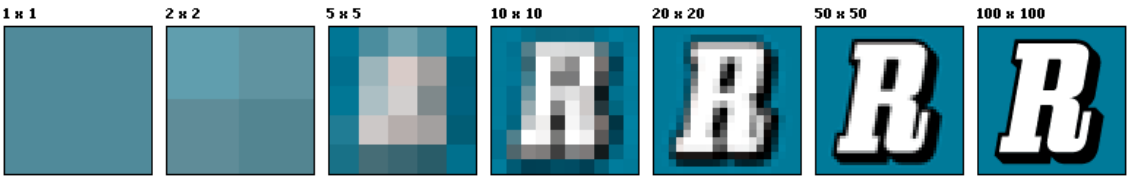
Pixel resolution

The term *resolution* is often used for a pixel count in digital imaging, even though American, Japanese, and international standards specify that it should not be so used, at least in the digital camera field. An image of N pixels high by M pixels wide can have any resolution less than N lines per picture height, or N TV lines. But when the pixel counts are referred to as resolution, the convention is to describe the *pixel resolution* with the set of two positive integer numbers, where the first number is the number of pixel columns (width) and the second is the number of pixel rows (height), for example as *640 by 480*. Another popular convention is to cite resolution as the total number of pixels in the image, typically given as number of megapixels, which can be calculated by multiplying pixel columns by pixel rows and dividing by one million. Other conventions include describing pixels per length unit or pixels per area unit, such as pixels per inch or per square inch. None of these *pixel resolutions* are true resolutions, but they are widely referred to as such; they serve as upper bounds on image resolution.

According to the same standards, the number of *effective pixels* that an image sensor or digital camera has is the count of elementary pixel sensors that contribute to the final

image, as opposed to the number of *total pixels*, which includes unused or light-shielded pixels around the edges.

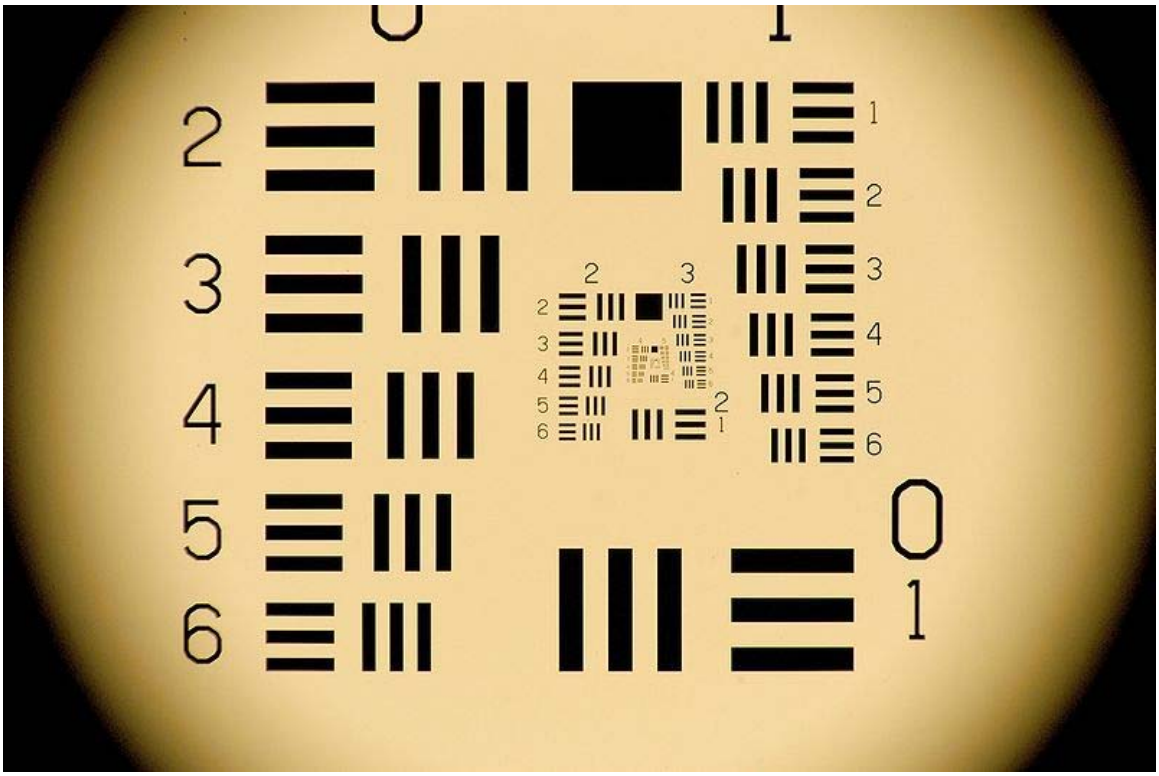
Below is an illustration of how the same image might appear at different pixel resolutions, if the pixels were poorly rendered as sharp squares (normally, a smooth image reconstruction from pixels would be preferred, but for illustration of pixels, the sharp squares make the point better).



An image that is 2048 pixels in width and 1536 pixels in height has a total of 2048×1536 = 3,145,728 pixels or 3.1 megapixels. One could refer to it as 2048 by 1536 or a 3.1-megapixel image.

Unfortunately, the count of pixels isn't a real measure of the resolution of digital camera images, because color image sensors are typically set up to alternate color filter types over the light sensitive individual pixel sensors. Digital images ultimately require a red, green, and blue value for each pixel to be displayed or printed, but one individual pixel in the image sensor will only supply one of those three pieces of information. The image has to be interpolated or demosaiced to produce all three colors for each output pixel.

Spatial resolution



The 1951 USAF resolution test target is a classic test target used to determine spatial resolution of imaging sensors and imaging systems.

The measure of how closely lines can be resolved in an image is called spatial resolution, and it depends on properties of the system creating the image, not just the pixel resolution in pixels per inch (ppi). For practical purposes the clarity of the image is decided by its spatial resolution, not the number of pixels in an image. In effect, spatial resolution refers to the number of *independent* pixel values per unit length.

The spatial resolution of computer monitors is generally 72 to 100 lines per inch, corresponding to pixel resolutions of 72 to 100 ppi. With scanners, *optical resolution* is sometimes used to distinguish spatial resolution from the number of pixels per inch.

In geographic information systems (GISs), spatial resolution is measured by the ground sample distance (GSD) of an image, the pixel spacing on the Earth's surface.

In astronomy one often measures spatial resolution in data points per arcsecond subtended at the point of observation, since the physical distance between objects in the image depends on their distance away and this varies widely with the object of interest. On the other hand, in electron microscopy, line or fringe resolution refers to the minimum separation detectable between adjacent parallel lines (e.g. between planes of atoms), while *point resolution* instead refers to the minimum separation between adjacent points that can be both detected *and interpreted* e.g. as adjacent columns of atoms, for instance.

The former often helps one detect periodicity in specimens, while the latter (although more difficult to achieve) is key to visualizing how individual atoms interact.

In Stereoscopic 3D images, spatial resolution could be defined as the spatial information recorded or captured by two viewpoints of a stereo camera (left and right camera). The effects of spatial resolution on overall perceived resolution of an image on a person's mind are yet not fully documented. It could be argued that such "spatial resolution" could add an image that then would not depend solely on pixel count or Dots per inch alone, when classifying and interpreting overall resolution of an given photographic image or video frame.

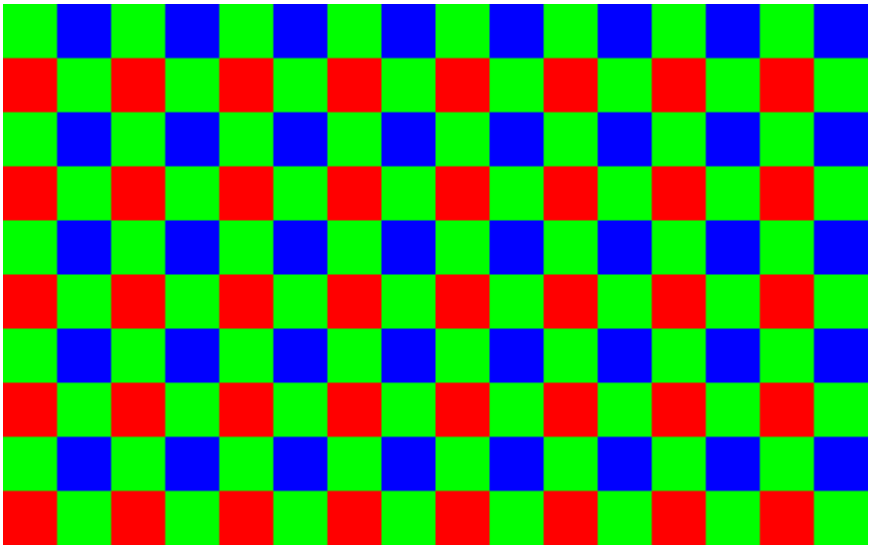
Spectral resolution

Color images distinguish light of different spectra. Multi-spectral images resolve even finer differences of spectrum or wavelength than is needed to reproduce color. That is, they can have higher spectral resolution. that is the strength of each band that is created (Lihongeni mulama: 2008)

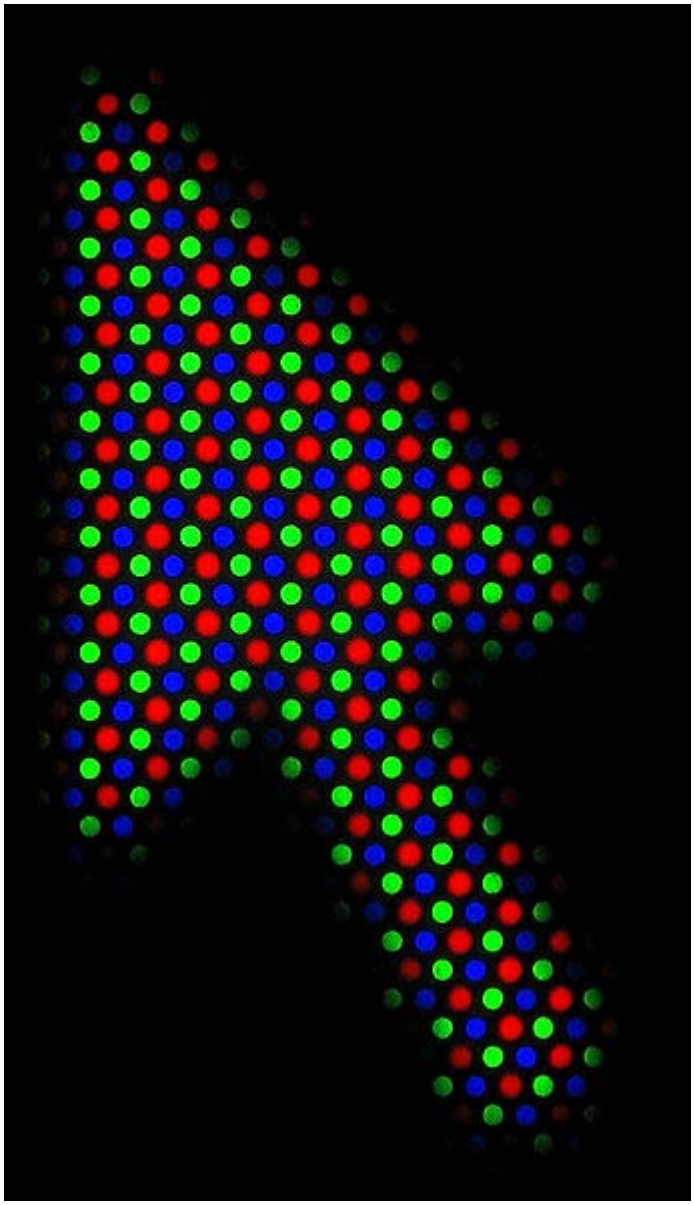
Temporal resolution

Movie cameras and high-speed cameras can resolve events at different points in time. The time resolution used for movies is usually 15 to 30 frames per second (frames/s), while high-speed cameras may resolve 100 to 1000 frames/s, or even more.

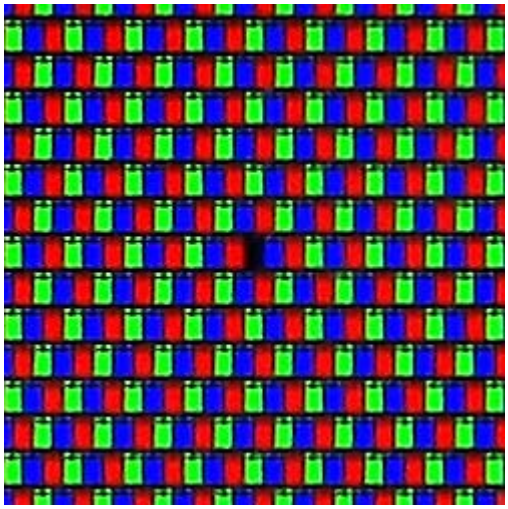
Many cameras and displays offset the color components relative to each other or mix up temporal with spatial resolution:



digital camera (Bayer color filter array)



CRT (shadow mask)



LCD (Triangular pixel geometry)

Radiometric resolution

Radiometric resolution determines how finely a system can represent or distinguish differences of intensity, and is usually expressed as a number of levels or a number of bits, for example 8 bits or 256 levels which is typical of computer image files. The higher the radiometric resolution, the better subtle differences of intensity or reflectivity can be represented, at least in theory. In practice, the effective radiometric resolution is typically limited by the noise level, rather than by the number of bits of representation.

Resolution in various media

This is a list of traditional, analog horizontal resolutions for various media. The list only includes popular formats, not rare formats, and all values are approximate (rounded to the nearest 10), since the actual quality can vary machine-to-machine or tape-to-tape. For ease-of-comparison, all values are for the NTSC system. (For PAL systems, replace 480 with 570.)

- Analog and early digital
  - 350×240 : Video CD
  - 300×480 : Umatic, Betamax, VHS, Video8
  - 350×480 : Super Betamax, Betacam (pro)
  - 420×480 : LaserDisc, Super VHS, Hi8
  - 500×480 : Analog broadcast
  - 670×480 : Enhanced Definition Betamax

- Digital
  - 720×480 : D-VHS, DVD, miniDV, Digital8, Digital Betacam (pro)
  - 720×480 : Widescreen DVD (anamorphic)
  - 1280×720 : D-VHS, HD DVD, Blu-ray, HDV (miniDV)
  - 1440×1080 : HDV (miniDV)

    - 1920×1080 : HDV (miniDV), AVCHD, HD DVD, Blu-ray, HDCAM SR (pro)
    - 2048×1080 : 2K Digital Cinema
    - 4096×2160 : 4K Digital Cinema
    - 7680×4320 : UHDTV
    - Sequences from newer films are scanned at 2,000, 4,000, or even 8,000 columns, called 2K, 4K, and 8K, for quality visual-effects editing on computers.

- Film
    - 35 mm film is scanned for release on DVD at 1080 or 2000 lines as of 2005.
    - 35 mm original camera negative motion picture film can resolve up to 6,000 lines.
    - 35 mm projection positive motion picture film has about 2,000 lines which results from the analog printing from the camera negative of an interpositive, and possibly an internegative, then a projection positive.
    - IMAX, including IMAX HD and OMNIMAX: approximately 10,000×7000 (7000 lines) resolution.