

3D Computer Graphics

3Dc

-
Z-fighting



H. Hees

3D Computer Graphics

3Dc

-
Z-fighting

compiled by H. Hees

3D Computer Graphics

Compiled by: H. Hees

Date: 20.07.2006

BookId: uidrpqgpwhpwilvl

All articles and pictures of this book were retrieved from the Wikipedia Project (wikipedia.org) on 02.07.2006. The articles are free to use under the terms of the GNU Free Documentation License. A copy of this license is included in the section entitled "*GNU Free Documentation License*". Images in this book have diverse licenses and you can find a list of figures and the corresponding licenses in the section "*List of Figures*". The version history of all articles can be retrieved from wikipedia.org. Each Article in this book has a reference to the original article. The principal authors of articles are referenced at the end of each article unless technical difficulties did not allow for a proper determination of the principal authors.

Logo design by Jörg Pelka

Printed by InstaBook Corporation (instabook.net)

Published by pediapress.com a service offered by brainbot technologies AG , Mainz, Germany

Articles	1
3Dc	1
3D computer graphics	2
3D computer graphics software	11
3D model	14
3D projection	16
Ambient occlusion	22
Back-face culling	25
Beam tracing	26
Bilinear filtering	28
Blinn–Phong shading model	32
Bloom (shader effect)	33
Bounding volume	34
Box modeling	38
Bui Tuong Phong	39
Bump mapping	40
Carmack’s Reverse	42
Catmull–Clark subdivision surface	43
Cel-shaded animation	45
Cg programming language	52
Clipmap	56
COLLADA	56
Comparison of Direct3D and OpenGL	58
Cone tracing	67
Constructive solid geometry	67
Conversion between quaternions and Euler angles	69
Cornell Box	71
Crowd simulation	73
Cube mapping	75
Diffuse reflection	75
Digital puppetry	76
Dilution of precision (computer graphics)	79
Direct3D	79
Displacement mapping	83
Distance fog	85
Draw distance	86
Euler boolean operation	87
Flat shading	87
Forward kinematic animation	88
Fragment (computer graphics)	89
Gelato (software)	89

Geometric model	91
Geometry pipelines	92
Geometry Processing	92
GLEE	93
GLEW	94
Glide API	94
Global illumination	95
GLSL	97
GLU	101
GLUI	102
Gouraud shading	103
Graphics pipeline	105
Hidden line removal	108
Hidden surface determination	109
High dynamic range imaging	111
High dynamic range rendering	115
High Level Shader Language	128
Humanoid Animation	129
Image based lighting	130
Image plane	131
Inverse kinematic animation	131
Inverse kinematics	132
Irregular Z-buffer	133
Isosurface	135
Joint constraints	136
Lambertian reflectance	137
Lambert's cosine law	138
Level of detail (programming)	141
Low poly	143
MegaTexture	144
Mesa 3D	145
Metaballs	146
Metropolis light transport	147
Micropolygon	148
Mipmap	149
Morph target animation	151
Motion capture	152
Newell's algorithm	163
Normal mapping	164
OpenGL	168
OpenGL+ +	178

OpenGL ES	180
OpenGL Utility Toolkit	182
Open Inventor	183
OpenRT	186
Painter's algorithm	187
Parallax mapping	188
Particle system	191
Path Tracing	194
Perlin noise	196
Per-pixel lighting	197
Phong reflection model	198
Phong shading	200
Photon mapping	202
Photorealistic (Morph)	204
PLIB	205
Polygonal modeling	206
Polygon (computer graphics)	210
Polygon mesh	211
Precomputed Radiance Transfer	212
Pre-rendered	213
Procedural generation	213
Procedural texture	218
Pyramid of vision	222
Qualitative invisibility	222
Quaternions and spatial rotation	223
Radiosity	245
Ray casting	248
Ray tracing	251
Ray tracing hardware	261
Reflection mapping	262
Rendering (computer graphics)	266
Render layers	279
Retained mode	281
S3 Texture Compression	281
Scanline rendering	284
Scene graph	286
Shader	292
Shading language	298
Shadow mapping	302
Shadow volume	309
Silhouette edge	311

Solid modelling	312
Specular highlight	317
Specularity	321
Stencil buffer	321
Stencil shadow volume	322
Subdivision surface	325
Subsurface scattering	327
Surface caching	329
Surface normal	329
Synthespian	331
Tao (software)	332
Texel (graphics)	332
Texture filtering	333
Texture mapping	334
Transform and lighting	336
Unified lighting and shadowing	336
Utah teapot	338
UV mapping	342
Vertex	343
Viewing frustum	344
Volume rendering	345
Volumetric lighting	350
Voxel	350
W-buffering	353
Z-buffering	353
Z-fighting	356
GNU Free Documentation License	358
List of Figures	362
Index	365

3Dc

3Dc is a lossy data compression algorithm for normal maps invented and first implemented by ATI. It builds upon the earlier DXT5 algorithm and is an open standard. 3Dc is now implemented by both ATI and NVIDIA.

Target Application

The target application, normal mapping, is an extension of bump mapping that simulates lighting on geometric surfaces by reading surface normals from a rectilinear grid analogous to a texture map - giving simple models the impression of increased complexity.

Although processing costs are reduced, memory costs are greatly increased. Pre-existing lossy compression algorithms implemented on consumer 3d hardware lacked the precision necessary for reproducing normal maps without excessive visible artefacts, justifying the development of 3Dc.

Although 3Dc was formally introduced with the ATI x800 series cards, there is also an S3TC compatible version planned for the older R3xx series, and cards from other companies. The quality and compression will not be as good, but the visual errors will still be significantly less than offered by standard S3TC.

Algorithm

Surface normals are three dimensional vectors of unit length. Because of the length constraint only two elements of any normal need be stored. The input is therefore an array of two dimensional values.

Compression is performed in 4x4 blocks. In each block the two components of each value are compressed separately. This produces two sets of 16 numbers for compression.

The compression is achieved by finding the lowest and highest values of the 16 to be compressed and storing each of those as an 8-bit quantity. Individual elements within the 4x4 block are then stored with 3-bits each, representing their position on an 8 step linear scale from the lowest value to the highest.

Total storage is 128 bits per 4x4 block once both source components are factored in. In an uncompressed scheme with similar 8-bit precision, the source data is 32 8-bit values for the same area, occupying 256 bits. The algorithm therefore produces a 2:1 compression ratio.

The compression ratio is sometimes stated as being "up to 4:1" as it is common to use 16-bit precision for input data rather than 8-bit. This produces

compressed output that is literally 1/4 the size of the input but it is not of comparable precision.

References

- 3Dc White Paper (PDF)¹
- What is a Normal Map?²
- CREATING AND USING NORMAL MAPS³
- Creating Normal Maps⁴
- 3Dc - higher quality textures with better compression⁵

Source: <http://en.wikipedia.org/wiki/3Dc>

Principal Authors: Timharwoodx, ThomasHarte, FireFox, Mushroom, RJHall

3D computer graphics

The rewrite of this article is being devised at 3D computer graphics/Temp. Please comment or help out as necessary. Thanks!

3D computer graphics are works of graphic art that were created with the aid of digital computers and specialized 3D software. In general, the term may also refer to the process of creating such graphics, or the field of study of 3D computer graphic techniques and its related technology.

3D computer graphics are different from 2D computer graphics in that a three-dimensional representation of geometric data is stored in the computer for the purposes of performing calculations and rendering 2D images. Sometimes these images are pre-rendered, sometimes they are not.

In general, the art of 3D modeling, which prepares geometric data for 3D computer graphics is akin to sculpting or photography, while the art of 2D graphics is analogous to painting. However, 3D computer graphics relies on many of the same algorithms as 2D computer graphics.

In computer graphics software, this distinction is occasionally blurred; some 2D applications use 3D techniques to achieve certain effects such as lighting, while some primarily 3D applications make use of 2D visual techniques.

¹ <http://www.ati.com/products/radeonx800/3DcWhitePaper.pdf>

² <http://members.shaw.ca/jimht03/normal.html>

³ http://www.monitorstudios.com/bcloward/tutorials_normal_maps1.html

⁴ http://www.blender3d.org/cms/Normal_Maps.491.0.html

⁵ <http://www.neoseeker.com/Articles/Hardware/Reviews/r420preview/3.html>



Figure 1 [A 3D rendering with raytracing and ambient occlusion using Blender and Yafray

Technology

→OpenGL and →Direct3D are two popular APIs for the generation of real-time imagery. (Real-time means that image generation occurs in 'real time', or 'on the fly') Many modern graphics cards provide some degree of hardware acceleration based on these APIs, frequently enabling the display of complex 3D graphics in real-time. However, it's not necessary to employ any one of these to actually create 3D imagery.

Creation of 3D computer graphics

The process of creating 3D computer graphics can be sequentially divided into three basic phases:

- Modeling
- Scene layout setup
- Rendering

Modeling

The modeling stage could be described as shaping individual objects that are later used in the scene. There exist a number of modeling techniques, including, but not limited to the following:



Figure 2 Architectural rendering compositing of modeling and lighting finalized by rendering process

- constructive solid geometry
- NURBS modeling
- polygonal modeling
- subdivision surfaces
- implicit surfaces

Modeling processes may also include editing object surface or material properties (e.g., color, luminosity, diffuse and specular shading components — more commonly called roughness and shininess, reflection characteristics, transparency or opacity, or index of refraction), adding textures, bump-maps and other features.

Modeling **may** also include various activities related to preparing a 3D model for animation (although in a complex character model this will become a stage of its own, known as rigging). Objects may be fitted with a *skeleton*, a central framework of an object with the capability of affecting the shape or movements of that object. This aids in the process of animation, in that the movement of the skeleton will automatically affect the corresponding portions of the model. See also →Forward kinematic animation and →Inverse kinematic animation.

At the rigging stage, the model can also be given specific controls to make animation easier and more intuitive, such as facial expression controls and mouth shapes (phonemes) for lipsyncing.

Modeling can be performed by means of a dedicated program (e.g., Lightwave Modeler, Rhinoceros 3D, Moray), an application component (Shaper, Lofter in 3D Studio) or some scene description language (as in POV-Ray). In some cases, there is no strict distinction between these phases; in such cases modelling is just part of the scene creation process (this is the case, for example, with Caligari trueSpace).

Process

Scene layout setup

Scene setup involves arranging virtual objects, lights, cameras and other entities on a scene which will later be used to produce a still image or an animation. If used for animation, this phase usually makes use of a technique called "keyframing", which facilitates creation of complicated movement in the scene. With the aid of keyframing, instead of having to fix an object's position, rotation, or scaling for each frame in an animation, one needs only to set up some key frames between which states in every frame are interpolated.

Lighting is an important aspect of scene setup. As is the case in real-world scene arrangement, lighting is a significant contributing factor to the resulting aesthetic and visual quality of the finished work. As such, it can be a difficult art to master. Lighting effects can contribute greatly to the mood and emotional response effected by a scene, a fact which is well-known to photographers and theatrical lighting technicians.

Tessellation and meshes

The process of transforming representations of objects, such as the middle point coordinate of a sphere and a point on its circumference into a polygon representation of a sphere, is called tessellation. This step is used in polygon-based rendering, where objects are broken down from abstract representations ("primitives") such as spheres, cones etc, to so-called *meshes*, which are nets of interconnected triangles.

Meshes of triangles (instead of e.g. squares) are popular as they have proven to be easy to render using scanline rendering.

Polygon representations are not used in all rendering techniques, and in these cases the tessellation step is not included in the transition from abstract representation to rendered scene.

Rendering

Rendering is the final process of creating the actual 2D image or animation from the prepared scene. This can be compared to taking a photo or filming the scene after the setup is finished in real life.

Rendering for interactive media, such as games and simulations, is calculated and displayed in real time, at rates of approximately 20 to 120 frames per second. Animations for non-interactive media, such as video and film, are rendered much more slowly. Non-real time rendering enables the leveraging of limited processing power in order to obtain higher image quality. Rendering times for individual frames may vary from a few seconds to an hour or more for complex scenes. Rendered frames are stored on a hard disk, then possibly transferred to other media such as motion picture film or optical disk. These frames are then displayed sequentially at high frame rates, typically 24, 25, or 30 frames per second, to achieve the illusion of movement.

There are two different ways this is done: →Ray tracing and GPU based real-time polygonal rendering. The goals are different:

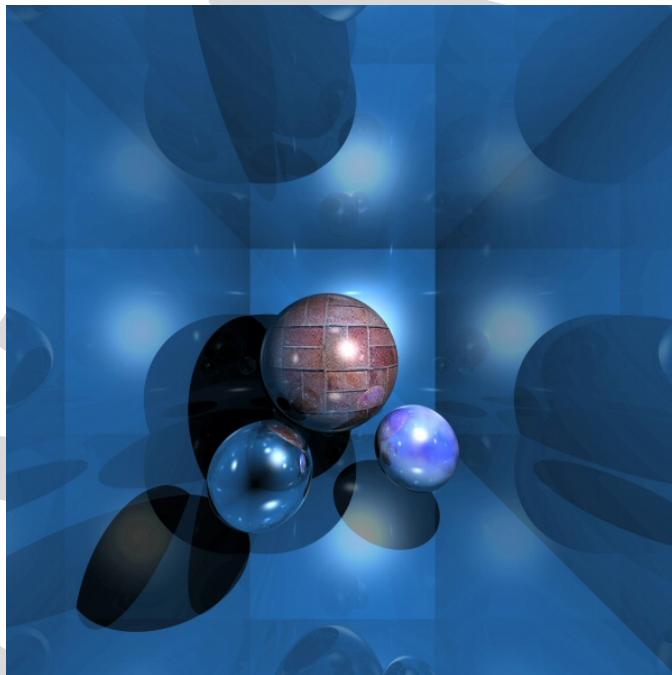


Figure 3 An example of a ray-traced image that typically takes seconds or minutes to render. The photo-realism is apparent.

In ray-tracing, the goal is photo-realism. Rendering often takes of the order of seconds or sometimes even days (for a single image/frame). This is the basic method employed in films, digital media, artistic works, etc;

In real time rendering, the goal is to show as much information as possible as the eye can process in a 30th of a second. The goal here is primarily speed and not photo-realism. In fact, here exploitations are made in the way the eye 'perceives' the world, and thus the final image presented is not necessarily that of the real-world, but one which the eye can closely associate to. This is the basic method employed in games, interactive worlds, VRML;

Photo-realistic image quality is often the desired outcome, and to this end several different, and often specialized, rendering methods have been developed. These range from the distinctly non-realistic wireframe rendering through polygon-based rendering, to more advanced techniques such as: scanline rendering, ray tracing, or radiosity.

Rendering software may simulate such visual effects as lens flares, depth of field or motion blur. These are attempts to simulate visual phenomena resulting from the optical characteristics of cameras and of the human eye. These effects can lend an element of realism to a scene, even if the effect is merely a simulated artifact of a camera.

Techniques have been developed for the purpose of simulating other naturally-occurring effects, such as the interaction of light with various forms of matter. Examples of such techniques include particle systems (which can simulate rain, smoke, or fire), volumetric sampling (to simulate fog, dust and other spatial atmospheric effects), caustics (to simulate light focusing by uneven light-refracting surfaces, such as the light ripples seen on the bottom of a swimming pool), and subsurface scattering (to simulate light reflecting inside the volumes of solid objects such as human skin).

The rendering process is computationally expensive, given the complex variety of physical processes being simulated. Computer processing power has increased rapidly over the years, allowing for a progressively higher degree of realistic rendering. Film studios that produce computer-generated animations typically make use of a render farm to generate images in a timely manner. However, falling hardware costs mean that it is entirely possible to create small amounts of 3D animation on a home computer system.

Often renderers are included in 3D software packages, but there are some rendering systems that are used as plugins to popular 3D applications. These rendering systems include Final-Render, Brazil r/s, V-Ray, mental ray, POV-Ray, and Pixar Renderman.

The output of the renderer is often used as only one small part of a completed motion-picture scene. Many layers of material may be rendered separately and integrated into the final shot using compositing software.

Projection

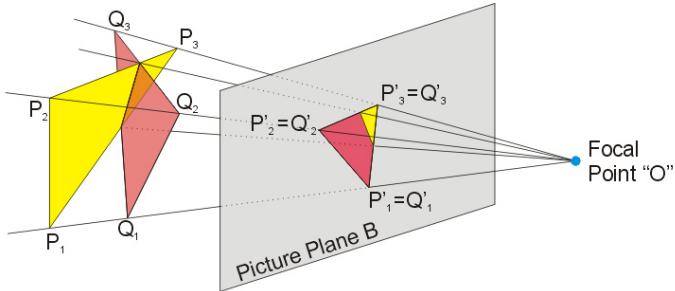


Figure 4 Perspective Projection

Since the human eye sees three dimensions, the mathematical model represented inside the computer must be transformed back so that the human eye can correlate the image to a realistic one. But the fact that the display device - namely a monitor - can display only two dimensions means that this mathematical model must be transferred to a two-dimensional image. Often this is done using projection; mostly using perspective projection. The basic idea behind the perspective projection, which unsurprisingly is the way the human eye works, is that objects that are further away are smaller in relation to those that are closer to the eye. Thus, to collapse the third dimension onto a screen, a corresponding operation is carried out to remove it - in this case, a division operation.

Orthogonal projection is used mainly in CAD or CAM applications where scientific modelling requires precise measurements and preservation of the third dimension.

Reflection and shading models

Modern 3D computer graphics rely heavily on a simplified reflection model called →*Phong reflection model* (not to be confused with →*Phong shading*). In refraction of light, an important concept is the refractive index. In most 3D programming implementations, the term for this value is "index of refraction," usually abbreviated "IOR."



Figure 5 An example of cel shading in the OGRE3D engine.

Popular reflection rendering techniques in 3D computer graphics include:

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

3D graphics APIs

3D graphics have become so popular, particularly in computer games, that specialized APIs (application programmer interfaces) have been created to ease the processes in all stages of computer graphics generation. These APIs have also proved vital to computer graphics hardware manufacturers, as they provide a way for programmers to access the hardware in an abstract way, while still taking advantage of the special hardware of this-or-that graphics card.

These APIs for 3D computer graphics are particularly popular:

- →OpenGL and the OpenGL Shading Language
- →OpenGL ES 3D API for embedded devices
- →Direct3D (a subset of DirectX)
- RenderMan
- RenderWare
- →Glide API
- TruDimension LC Glasses and 3D monitor API

There are also higher-level 3D scene-graph APIs which provide additional functionality on top of the lower-level rendering API. Such libraries under active development include:

- QSDK
- Quesa
- Java 3D
- JSR 184 (M3G)
- NVidia Scene Graph
- OpenSceneGraph
- OpenSG
- OGRE
- Irrlicht
- Hoops3D

See also

- →3D model
- 3D modeler
- →3D projection
- →Ambient occlusion
- Anaglyph image - *Anaglyphs are stereo pictures that are viewed with red-blue glasses, that allow a 3D image to be perceived as 3D by the human eye.*

- Animation
- Graphics
- History of 3D Graphics - Major Milestones/Influential people/Hardware+Software Developments
- List of 3D Artists
- Magic Eye autostereograms
- →Rendering (computer graphics)
- Panda3D
- Polarized glasses - Another method to view 3D images as 3D
- VRML
- X3D
- 3d motion controller

Source: http://en.wikipedia.org/wiki/3D_computer_graphics

Principal Authors: Frecklefoot, Dormant25, Flamurai, Wapcaplet, Pavel Vozenilek, Michael Hardy, Furykef, Blaatkoala, Mikkalai, Blueshade

3D computer graphics software

→**3D computer graphics software** is a program or collection of programs used to create 3D computer-generated imagery. There are typically many stages in the "pipeline" that studios use to create 3D objects for film and games, and this article only covers some of the software used. Note that most of the 3D packages have a very plugin-oriented architecture, and high-end plugins costing tens or hundreds of thousands of dollars are often used by studios. Larger studios usually create enormous amounts of proprietary software to run alongside these programs.

If you are just getting started out in 3D, one of the major packages is usually sufficient to begin learning. Remember that 3D animation can be very difficult, time-consuming, and unintuitive; a teacher or a book will likely be necessary. Most of the high-end packages have free versions designed for personal learning.

Major Packages

Maya (Autodesk) is currently the leading animation program for cinema; nearly every studio uses it. It is known as difficult to learn, but it is possibly the most powerful 3D package. When studios use Maya, they typically replace parts of it with proprietary software. Studios will also render using Pixar's



Figure 6 Modeling in LightWave. This interface is fairly typical of 3D packages.

Renderman, rather than the default mentalray. Autodesk, makers of 3DS Max, recently bought Maya from Alias and have rebranded it. There are still open questions about how independent Maya will be from 3DS Max.

Softimage|XSI (Avid) is often seen as head-to-head competition with Maya, and is very feature-similar. Fans of the two packages often will often argue the merits of each. XSI was once the leader in animation, but lagged as Maya surged ahead. It is now trying to reclaim the top spot.

3D Studio Max (Autodesk), often abbreviated 3DS Max, is the leading animation program in the video game industry. Experts argue that it is very good at handling low-polygon animation, but perhaps its greatest asset to the computer/video industry is its entrenched support network and its many plugins. It is also the most expensive of the high-end packages, coming at \$3500 US, compared to about \$2000 for the others. Because of its presence in the video game industry, it is also a popular hobbyist package.

Houdini (Side Effects Software) is a high-end package that is found often in studios. Its most common use is in animating special effects, rather than models.

LightWave 3D (NewTek) is a popular 3D package because of its easy-to-learn interface; many artists prefer it to the more technical Maya or 3DS Max. It has

weaker modeling and particularly animation features than some of the larger packages, but it is still used widely in film and broadcasting.

Cinema 4D (MAXON) is a lighter package than the others. Its main asset is its artist-friendliness, avoiding the complicated technical nature of the other packages. For example, a popular plugin, BodyPaint, allows artists to draw textures directly onto the surface of models.

formZ (autodesk, Inc.) is a general purpose 3D modeler. Its forte is modeling. Many of its users are architects, but also include designers from many fields including interior designers, illustrators, product designers, set designers. Its default renderer uses the LightWorks rendering engine for raytracing and radiosity. formZ has been around since 1991, available for both the Macintosh and Windows operating systems.

Other packages

Free

Blender is a free software package that mimics the larger packages. It is being developed under the GPL, after its previous owner, Not a Number Technologies, went bankrupt. Art of Illusion is another free software package developed under the GPL. Wings 3D is a BSD-licensed, minimal modeler. Anim8or is another free 3d rendering and animation package.

Not Free

MilkShape 3D is a shareware/trialware polygon 3D modelling program with extensive import/export capabilities.

Carrara (Eovia) is a 3D complete tool set package for 3D modeling, texturing animation and rendering; and Amapi and Hexagon (Eovia) are 3D packages often used for high-end abstract and organic modeling respectively. Bryce (DAZ productions) is most famous for landscapes. modo, created by developers who splintered from NewTek, is a new modeling program. Zbrush(Pixologic) is a digital sculpting tool that combines 3D/2.5D modeling, texturing and painting.

Renderers

Pixar's RenderMan is the premier renderer, used in many studios. Animation packages such as 3DS Max and Maya can pipeline to RenderMan to do all the rendering. mental ray is another popular renderer, and comes default with most of the high-end packages. POV-Ray and YafRay are two free renderers.

Related to 3D software

Swift3D is a package for transforming models in Lightwave or 3DS Max into Flash animations. Match moving software is commonly used to match live video with computer-generated video, keeping the two in sync as the camera moves. Poser is the most popular program for modeling people. After producing video, studios then edit or composite the video using programs such as Adobe Premiere or Apple Final Cut at the low end, or Autodesk Combustion or Apple Shake at the high end.

Source: http://en.wikipedia.org/wiki/3D_computer_graphics_software

Principal Authors: Bertmg, Goncalopp, ShaunMacPherson, Snarius, Skybum

3D model

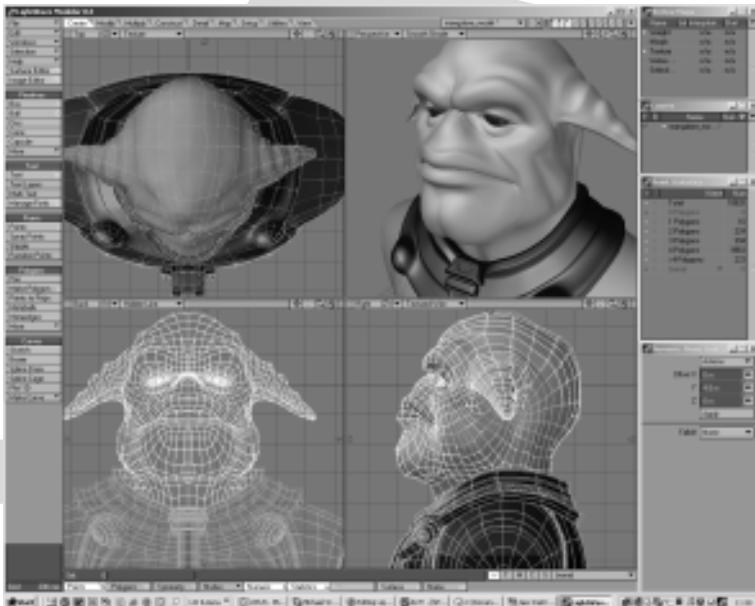


Figure 7 A 3D model of a character in the 3D modeler LightWave, shown in various manners and from different perspectives

A **3D model** is a 3D polygonal representation of an object, usually displayed with a computer or via some other video device. The object can range from a

real-world entity to fictional, from atomic to huge. Anything that can exist in the physical world can be represented as a 3D model.

3D models are most often created with special software applications called 3D modelers, but they need not be. Being a collection of data (points and other information), 3D models can be created by hand or algorithmically. Though they most often exist virtually (on a computer or a file on disk), even a description of such a model on paper can be considered a 3D model.

3D models are widely used anywhere 3D graphics are used. Actually, their use predates the widespread use of 3D graphics on personal computers. Many computer games used pre-rendered images of 3D models as sprites before computers could render them in real-time.

Today, 3D models are used in a wide variety of fields. The medical industry uses detailed models of organs. The movie industry uses them as characters and objects for animated and real-life motion pictures. The video game industry uses them as assets for computer and video games. The science sector uses them as highly detailed models of chemical compounds. The architecture industry uses them to demonstrate proposed buildings and landscapes. The engineering community uses them as designs of new devices, vehicles and structures as well as a host of other uses. In recent decades the earth science community has started to construct 3D geological models as a standard practice.

A 3D model by itself is not visual. It can be rendered as a simple wireframe at varying levels of detail, or shaded in a variety of ways. Many 3D models, however, are covered in a covering called a texture (the process of aligning the texture to coordinates on the 3D model is called texture mapping). A texture is nothing more than a graphic image, but gives the model more detail and makes it look more realistic. A 3D model of a person, for example, looks more realistic with a texture of skin and clothes, than a simple monochromatic model or wireframe of the same model.

Other effects, beyond texturing, can be done to 3D models to add to their realism. For example, the surface normals can be tweaked to effect how they are lit, certain surfaces can have bump mapping applied and any other number of 3D rendering tricks can be applied.

3D models are often animated for some uses. For example, 3D models are heavily animated for use in feature films and computer and video games. They can be animated from within the 3D modeler that created them or externally. Often extra data is added to the model to make it easier to animate. For example, some 3D models of humans and animals have entire bone systems so they will look realistic when they move and can be manipulated via joints and bones.

Source: http://en.wikipedia.org/wiki/3D_model

Principal Authors: Frecklefoot, Furrykef, HolgerK, Equendil, ZeroOne

3D projection

A **3D projection** is a mathematical transformation used to project three dimensional points onto a two dimensional plane. Often this is done to simulate the relationship of the camera to subject. 3D projection is often the first step in the process of representing three dimensional shapes two dimensionally in computer graphics, a process known as rendering.

The following algorithm was a standard on early computer simulations and videogames, and it is still in use with heavy modifications for each particular case. This article describes the simple, general case.

Data necessary for projection

Data about the objects to render is usually stored as a collection of points, linked together in triangles. Each point is a series of three numbers, representing its X,Y,Z coordinates from an origin relative to the object they belong to. Each triangle is a series of three points or three indexes to points. In addition, the object has three coordinates X,Y,Z and some kind of rotation, for example, three angles *alpha*, *beta* and *gamma*, describing its position and orientation relative to a "world" reference frame.

Last comes the observer (the term *camera* is the one commonly used). The camera has a second set of three X,Y,Z coordinates and three *alpha*, *beta* and *gamma* angles, describing the observer's position and the direction along which it is pointing.

All this data is usually stored in floating point, even if many programs convert it to integers at various points in the algorithm, to speed up the calculations.

- Warning: The author has used the \times symbol to denote multiplication of matrices, e.g. ' $A \times B$ ' to mean 'A times B'. This is confusing since in this field this symbol is instead used to indicate the cross product of vectors, i.e. ' $A \times B$ ' usually means 'A cross B'. The two operators are different. In matrix algebra A times B is usually written as ' $A \cdot B$ '. I have NOT edited the rest of the document to reflect this.

First step: world transform

The first step is to transform the points coordinates taking into account the position and orientation of the object they belong to. This is done using a set of four matrices: (The Matrix we use is column major i.e. $v' = \text{Matrix}^*v$ the same in OpenGL but different in Directx)

$$\begin{bmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

— object translation

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos \alpha & -\sin \alpha & 0 \\ 0 & \sin \alpha & \cos \alpha & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

— rotation about the x-axis

$$\begin{bmatrix} \cos \beta & 0 & \sin \beta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin \beta & 0 & \cos \beta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

— rotation about the y-axis

$$\begin{bmatrix} \cos \gamma & -\sin \gamma & 0 & 0 \\ \sin \gamma & \cos \gamma & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

— rotation about the z-axis.

The four matrices are multiplied together, and the result is the *world transform matrix*: a matrix that, if a point's coordinates were multiplied by it, would result in the point's coordinates being expressed in the "world" reference frame.

Note that, unlike multiplication between numbers, the order used to multiply the matrices is significant: changing the order will change the results too. When dealing with the three rotation matrices, a fixed order is good for the necessity of the moment that must be chosen. The object should be rotated before it is translated, since otherwise the position of the object in the world would get rotated around the centre of the world, wherever that happens to be.

World transform = Translation \times Rotation

To complete the transform in the most general way possible, another matrix called the scaling matrix is used to scale the model along the axes. This matrix is multiplied to the four given above to yield the complete world transform.

The form of this matrix is:

$$\begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

— where s_x , s_y , and s_z are the scaling factors along the three co-ordinate axes.

Since it is usually convenient to scale the model in its own model space or co-ordinate system, scaling should be the first transformation applied. The final transform thus becomes:

World transform = Translation \times Rotation \times Scaling

(as in some computer graphics book or some computer graphic programming API such as Directx, it use mactrics with translation vectors in the bottom row, in this scheme, the order of matrices would be reversed.)

$$\begin{bmatrix} s_x \cos \gamma \cos \beta & -s_y \sin \gamma \cos \beta & s_z \sin \beta & x \\ s_x \cos \gamma \sin \beta \sin \alpha + s_x \sin \gamma \cos \alpha & s_y \cos \gamma \cos \alpha - s_y \sin \gamma \sin \beta \sin \alpha & -s_z \cos \beta \sin \alpha & y \\ s_x \sin \gamma \sin \alpha - s_x \cos \gamma \sin \beta \cos \alpha & s_y \sin \gamma \sin \beta \cos \alpha + s_y \sin \alpha \cos \gamma & s_z \cos \beta \cos \alpha & z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

— final result of Translation \times $x \times y \times z \times$ Scaling.

Second step: camera transform

The second step is virtually identical to the first one, except for the fact that it uses the six coordinates of the observer instead of the object, and the inverses of the matrixes should be used, and they should be multiplied in the opposite order. (Note that $(A \times B)^{-1} = B^{-1} \times A^{-1}$.) The resulting matrix can transform coordinates from the world reference frame to the observer's one.

The camera looks in its z direction, the x direction is typically left, and the y direction is typically up.

$$\begin{bmatrix} 1 & 0 & 0 & -x \\ 0 & 1 & 0 & -y \\ 0 & 0 & 1 & -z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

— inverse object translation (the inverse of a translation is a translation in the opposite direction).

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos \alpha & \sin \alpha & 0 \\ 0 & -\sin \alpha & \cos \alpha & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

— inverse rotation about the x-axis (the inverse of a rotation is a rotation in the opposite direction. Note that $\sin(-x) = -\sin(x)$, and $\cos(-x) = \cos(x)$).

$$\begin{bmatrix} \cos \beta & 0 & -\sin \beta & 0 \\ 0 & 1 & 0 & 0 \\ \sin \beta & 0 & \cos \beta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

— inverse rotation about the y-axis.

$$\begin{bmatrix} \cos \gamma & \sin \gamma & 0 & 0 \\ -\sin \gamma & \cos \gamma & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

— inverse rotation about the z-axis.

The two matrices obtained from the first two steps can be multiplied together to get a matrix capable of transforming a point's coordinates from the object's reference frame to the observer's reference frame.

Camera transform = inverse rotation \times inverse translation

Transform so far = camera transform \times world transform.

Third step: perspective transform

The resulting coordinates would be already good for an isometric projection or something similar, but realistic rendering requires an additional step to correctly simulate the perspective distortion. Indeed, this simulated perspective is the main aid for the viewer to judge distances in the simulated view.

A perspective distortion can be generated using the following 4×4 matrix:

$$\begin{bmatrix} 1/\tan\mu & 0 & 0 & 0 \\ 0 & 1/\tan\nu & 0 & 0 \\ 0 & 0 & \frac{B+F}{B-F} & \frac{-2BF}{B-F} \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

where μ is the angle between a line pointing out of the camera in z direction and the plane through the camera and the right-hand edge of the screen, and ν is the angle between the same line and the plane through the camera and the top edge of the screen. This projection should look correct, if you are looking with one eye; your actual physical eye is located on the line through the centre of the screen normal to the screen, and μ and ν are physically measured assuming your eye is the camera. On typical computer screens as of 2003, $\tan\mu$ is probably about $1\frac{1}{3}$ times $\tan\nu$, and $\tan\mu$ might be about 1 to 5, depending on how far from the screen you are.

F is a positive number representing the distance of the observer from the front clipping plane, which is the closest any object can be to the camera. B is a positive number representing the distance to the back clipping plane, the farthest away any object can be. If objects can be at an unlimited distance from the camera, B can be infinite, in which case $(B + F)/(B - F) = 1$ and $-2BF/(B - F) = -2F$.

If you are not using a Z-buffer and all objects are in front of the camera, you can just use 0 instead of $(B + F)/(B - F)$ and $-2BF/(B - F)$. (Or anything you want.)

All the calculated matrices can be multiplied together to get a final transformation matrix. One can multiply each of the points (represented as a vector of three coordinates) by this matrix, and directly obtain the screen coordinate at which the point must be drawn. The vector must be extended to four dimensions using homogeneous coordinates:

$$\begin{bmatrix} x' \\ y' \\ z' \\ \omega' \end{bmatrix} = [\text{Perspective transform}] \times [\text{Camera transform}] \times [\text{World transform}] \times \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}.$$

Note that in computer graphics libraries, such as →OpenGL, you should give the matrices in the *opposite* order as they should be applied, that is, first the

perspective transform, then the camera transform, then the object transform, as the graphics library applies the transformations in the *opposite* order than you give the transformations in! This is useful, since the world transform typically changes more often than the camera transform, and the camera transform changes more often than the perspective transform. One can, for example, pop the world transform off a stack of transforms and multiply a new world transform on, without having to do anything with the camera transform and perspective transform.

Remember that $\{x'/\omega', y'/\omega'\}$ is the final coordinates, where $\{-1, -1\}$ is typically the bottom left corner of the screen, $\{1, 1\}$ is the top right corner of the screen, $\{1, -1\}$ is the bottom right corner of the screen and $\{-1, 1\}$ is the top left corner of the screen.

If the resulting image may turn out upside down, swap the top and bottom.

If using a Z-buffer, a z'/ω' value of -1 corresponds to the front of the Z-buffer, and a value of 1 corresponds to the back of the Z-buffer. If the front clipping plane is too close, a finite precision Z-buffer will be more inaccurate. The same applies to the back clipping plane, but to a significantly lesser degree; a Z-buffer works correctly with the back clipping plane at an infinite distance, but not with the front clipping plane at 0 distance.

Objects should only be drawn where $-1 \leq z'/\omega' \leq 1$. If it is less than -1, the object is in front of the front clipping plane. If it is more than 1, the object is behind the back clipping plane. To draw a simple single-colour triangle, $\{x'/\omega', y'/\omega'\}$ for the three corners contains sufficient information. To draw a textured triangle, where one of the corners of the triangle is behind the camera, all the coordinates $\{x', y', z', \omega'\}$ for all three points are needed, otherwise the texture would not have the correct perspective, and the point behind the camera would not appear in the correct location. In fact, the projection of a triangle where a point is behind the camera is not technically a triangle, since the area is infinite and two of the angles sum to more than 180° , the third angle being effectively negative. (Typical modern graphics libraries use all four coordinates, and can correctly draw "triangles" with some points behind the camera.) Also, if a point is on the plane through the camera normal to the camera direction, ω' is 0, and $\{x'/\omega', y'/\omega'\}$ is meaningless.

Simple version

$$X_{2D} = X_{3D} - \frac{DX}{Z_{3D} + \text{Eye distance}} \times X_{3D}$$

$$Y_{2D} = Y_{3D} - \frac{DY}{Z_{3D} + \text{Eye distance}} \times Y_{3D}$$

3D projection

where $\frac{DX}{DY}$ is the distance between the eye and the 3D point in the X/Y axis, a large positive Z is towards the horizon and 0 is screen.

See also

- Computer graphics
- →3D computer graphics
- Graphics card
- →Transform and lighting
- →Texture mapping
- Perspective distortion

Source: http://en.wikipedia.org/wiki/3D_projection

Principal Authors: Michael Hardy, Cyp, Alfio, Oleg Alexandrov, Bobbygao

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. 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.

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 industry, ambient occlusion is often referred to as "sky light."



Figure 8 ambient occlusion



Figure 9 diffuse only

The occlusion A_p at a point p on a surface with normal N can be computed by integrating the visibility function over the hemisphere Ω with respect to projected solid angle:

$$A_p = \frac{1}{\pi} \int_{\Omega} V_{p,\omega}(N \cdot \omega) d\omega$$

where $V_{p,\omega}$ is the visibility function at p , defined to be zero if p is occluded in the direction ω and one otherwise. 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 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 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 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.,

External links

- Depth Map based Ambient Occlusion⁶
- Ambient Occlusion⁷

⁶ http://www.andrew-whitehurst.net/amb_occlude.html



Figure 10 combined ambient and diffuse



Figure 11 ambient occlusion

- Assorted notes about ambient occlusion⁸
- Ambient Occlusion Fields⁹ — real-time ambient occlusion using cube maps
- Fast Precomputed Ambient Occlusion for Proximity Shadows¹⁰ real-time ambient occlusion using volume textures
- Dynamic Ambient Occlusion and Indirect Lighting¹¹ a real time self ambient occlusion method from Nvidia's GPU Gems 2 book
- ShadeVis¹² an open source tool for computing ambient occlusion

Source: http://en.wikipedia.org/wiki/Ambient_occlusion

Principal Authors: Mrtheplague, ALoopingIcon, SimonP, Gaius Cornelius, Jotapeh

⁷ <http://www-viz.tamu.edu/students/bmoyer/617/ambocc/>

⁸ <http://www.cs.unc.edu/~coombe/research/ao/>

⁹ <http://www.tml.hut.fi/~janne/aofields/>

¹⁰ <http://www.inria.fr/rrrt/rr-5779.html>

¹¹ http://download.nvidia.com/developer/GPU_Gems_2/GPU_Gems2_ch14.pdf

¹² <http://vcg.sourceforge.net/tiki-index.php?page=ShadeVis>



Figure 12 diffuse only



Figure 13 combined ambient and diffuse

Back-face culling

In computer graphics, **back-face culling** determines whether a polygon of a graphical object, within the field of view of the camera, is visible. It is a step in the graphical pipeline that uses the test

$$Ax + By + Cz + D < 0$$

to test if the polygon is visible or not. If this test is true, the normal vector of the polygon is pointed away from the camera, meaning that the polygon is facing away and does not need to be drawn.

The process makes rendering objects quicker and more efficient by reducing the number of polygons for the program to draw. For example, in a city street scene, there is generally no need to draw the polygons on the sides of the buildings facing away from the camera; they are completely occluded by the sides facing the camera.

A related technique is clipping, which determines whether polygons are within the camera's field of view at all.

It is important to note that this technique only works with single-sided polygons, which are only visible from one side. Double-sided polygons are rendered from both sides, and thus have no back-face to cull.



Figure 14 ambient occlusion



Figure 15 diffuse only

Further reading

- Geometry Culling in 3D Engines¹³, by Pietari Laurila

Source: http://en.wikipedia.org/wiki/Back-face_culling

Principal Authors: Eric119, Charles Matthews, Mdd4696, Deepomega, Canderson7

Beam tracing

Beam tracing is a derivative of the ray tracing algorithm that replaces rays, which have no thickness, with beams. Beams are shaped like unbounded pyramids, with (possibly complex) polygonal cross sections. Beam tracing was first proposed by Paul Heckbert and Pat Hanrahan.

In beam tracing, a pyramidal beam is initially cast through the entire viewing frustum. This initial viewing beam is intersected with each polygon in the environment, from nearest to farthest. Each polygon that intersects with the beam must be visible, and is removed from the shape of the beam and added to a render queue. When a beam intersects with a reflective or refractive polygon, a new beam is created in a similar fashion to ray-tracing.

A variant of beam tracing casts a pyramidal beam through each pixel of the image plane. This is then split up into sub-beams based on its intersection with

¹³ <http://www.gamedev.net/reference/articles/article1212.asp>



Figure 16 combined ambient and diffuse

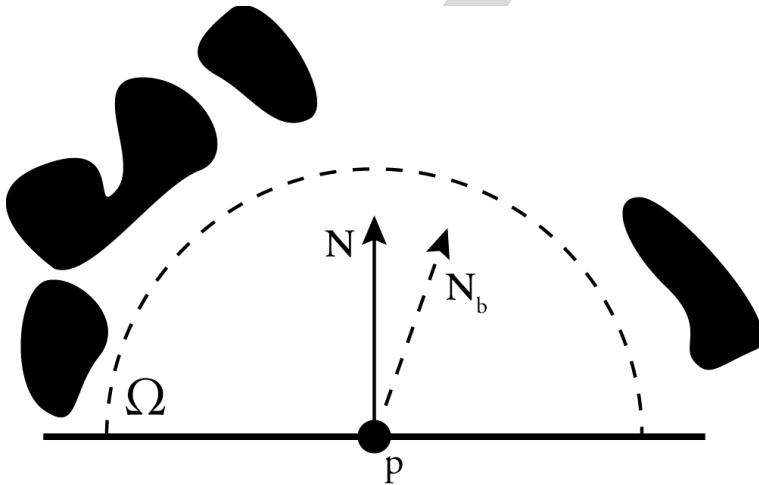


Figure 17

scene geometry. Reflection and transmission (refraction) rays are also replaced by beams. This sort of implementation is rarely used, as the geometric processes involved are much more complex and therefore expensive than simply casting more rays through the pixel.

Beam tracing solves certain problems related to sampling and aliasing, which can plague conventional ray tracing. However, the additional computational complexity that beams create has made them unpopular. In recent years, increases in computer speed have made Monte Carlo algorithms like distributed ray tracing much more viable than beam tracing.

However, beam tracing has had a renaissance in the field of acoustic modelling, in which beams are used as an efficient way to track deep reflections from sound source to receiver (or vice-versa), a field where ray tracing is notoriously prone to sampling errors.

Beam tracing is related in concept to cone tracing.

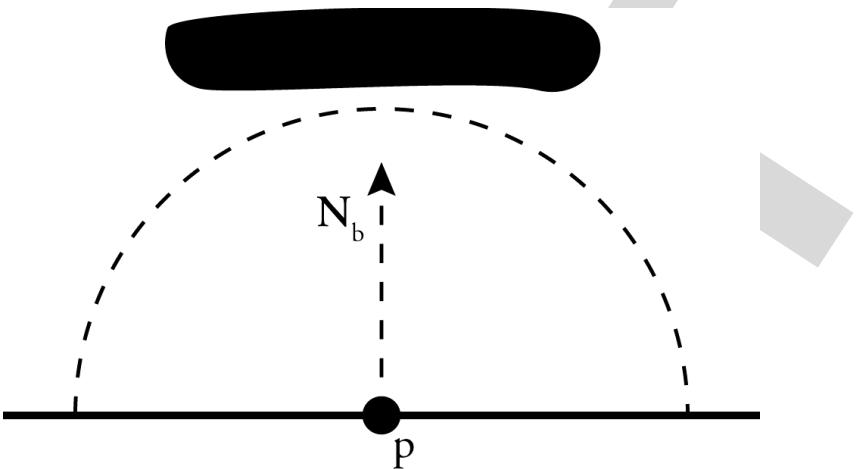


Figure 18

Source: http://en.wikipedia.org/wiki/Beam_tracing

Principal Authors: Kibibu, Reedbeta, RJJR, Hetar, RJHall

Bilinear filtering

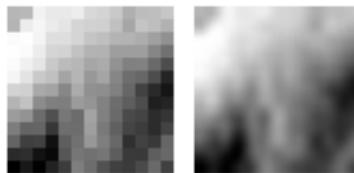


Figure 19 A zoomed small portion of a bitmap of a cat, using nearest neighbor filtering (left) and bicubic filtering (right). Bicubic filtering is similar to bilinear filtering but with a different interpolation function.

Bilinear filtering is a texture mapping 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

Bilinear filtering

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. 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.

$$y_a = y_0 + \frac{y_1 - y_0}{u_1 - u_0} (u - u_0)$$

$$y_b = y_2 + \frac{y_3 - y_2}{u_3 - u_2} (u - u_2)$$

$$y = y_b + \frac{y_b - y_a}{v_2 - v_0} (v - v_0)$$

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, mipmaping 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, intermingled with the numbers from the centers of boxes from a 3-texel-wide 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 you end 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 is probably not that great an idea, but it might work if the texture is designed to be laid over a solid background or be transparent.
- Repeat the edge texels out to infinity. This works well if the texture is designed to not be repeated.

See also

- Trilinear filtering
- Anisotropic filtering

Source: http://en.wikipedia.org/wiki/Bilinear_filtering

Principal Authors: Vorn, Msikma, Skulvar, Rgaddipa, HarrisX

Blinn–Phong shading model

The **Blinn–Phong shading model** (also called **Blinn–Phong reflection model** or **modified Phong reflection model**) is a modification to the →Phong reflection model, trading visual precision for computing efficiency.

In Phong shading, one must continually recalculate the angle $R \cdot V$ between a viewer (V) and the beam from a light-source (L) reflected (R) on a surface.

If we instead calculate a *halfway vector* between the viewer and light-source vectors,

$$H = \frac{L+V}{|L+V|}$$

we can replace $R \cdot V$ with $N \cdot H$.

This dot product represents the cosine of an angle that is half of the angle represented by Phong's dot product, if V , L , N and R all lie in the same plane. The angle between N and H is therefore sometimes called the *halfway angle*.

The halfway angle is smaller than the angle we want in Phong's model, but considering that Phong is using $(R \cdot V)^\alpha$, we can set an exponent $\alpha/2$ so that $(N \cdot H)^{\alpha/2}$ is closer to the former expression.

Blinn–Phong is the default shading model used in →OpenGL, and is carried out on each vertex as it passes down the graphics pipeline; pixel values between vertices are interpolated by →Gouraud shading by default, rather than the more expensive →Phong shading.

See also

- →Phong reflection model for Phong's corresponding model
- →Phong shading
- →Gouraud shading

Source: http://en.wikipedia.org/wiki/Blinn%E2%80%93Phong_shading_model

Bloom (shader effect)



Figure 20 Doom 3 mini-mod showing the bloom shader effect (screenshot by Motoxpro @ doom3world)

Bloom (sometimes referred to as **Light bloom**) is a computer graphics shader effect used by computer games.

In high dynamic range rendering (HDR) applications, it is used when the necessary brightness exceeds the contrast ratio that a computer monitor can display. A bloom works by radiating, or *blooming*, the high intensity light around the very bright object, so that the bright light appears to be coming from around the object, not only from the object itself. This gives the illusion that the object is brighter than it really is. It can be used without HDR rendering to represent a surface that is being illuminated with an extremely bright light source, and as a result, is often mistaken for a full implementation of HDR Rendering.

Bloom became very popular after *Tron 2.0* and is used in many games and modifications. The first game to use blooming was *Deus Ex: Invisible War*.

See also

- →High dynamic range rendering
- Tone mapping

Source: http://en.wikipedia.org/wiki/Bloom_%28shader_effect%29

Bounding volume



Figure 21 A bounding box for a three dimensional model

In computer graphics and computational geometry, a **bounding volume** for a set of objects is a closed volume that completely contains the union of the objects in the set. Bounding volumes are used to improve the efficiency of geometrical operations by using simple volumes to contain more complex objects. Normally, simpler volumes have simpler ways to test for overlap.

Bounding volume

A bounding volume for a set of objects is also a bounding volume for the single object consisting of their union, and the other way around. Therefore it is possible to confine the description to the case of a single object, which is assumed to be non-empty and bounded (finite).

Uses of bounding volumes

Bounding volumes are most often used to accelerate certain kinds of tests.

In ray tracing, bounding volumes are used in ray-intersection tests, and in many rendering algorithms, it is used for viewing frustum tests. If the ray or viewing frustum does not intersect the bounding volume, it cannot intersect the object contained in the volume. These intersection tests produce a list of objects that must be displayed. Here, displayed means rendered or rasterized.

In collision detection, when two bounding volumes do not intersect, then the contained objects cannot collide, either.

Testing against a bounding volume is typically much faster than testing against the object itself, because of the bounding volume's simpler geometry. This is because an 'object' is typically composed of polygons or data structures that are reduced to polygonal approximations. In either case, it is computationally wasteful to test each polygon against the view volume if the object is not visible. (Onscreen objects must be 'clipped' to the screen, regardless of whether their surfaces are actually visible.)

To obtain bounding volumes of complex objects, a common way is to break the objects/scene down using a scene graph or more specifically bounding volume hierarchies like e.g. OBB trees. The basic idea behind this is to organize a scene in a tree-like structure where the root comprises the whole scene and each leaf contains a smaller subpart.

Common types of bounding volume

The choice of the type of bounding volume for a given application is determined by a variety of factors: the computational cost of computing a bounding volume for an object, the cost of updating it in applications in which the objects can move or change shape or size, the cost of determining intersections, and the desired precision of the intersection test. It is common to use several types in conjunction, such as a cheap one for a quick but rough test in conjunction with a more precise but also more expensive type.

The types treated here all give convex bounding volumes. If the object being bounded is known to be convex, this is not a restriction. If non-convex bounding volumes are required, an approach is to represent them as a union of a

number of convex bounding volumes. Unfortunately, intersection tests become quickly more expensive as the bounding boxes become more sophisticated.

A **bounding sphere** is a sphere containing the object. In 2-D graphics, this is a circle. Bounding spheres are represented by centre and radius. They are very quick to test for collision with each other: two spheres intersect when the distance between their centres does not exceed the sum of their radii. This makes bounding spheres appropriate for objects that can move in any number of dimensions.

A **bounding cylinder** is a cylinder containing the object. In most applications the axis of the cylinder is aligned with the vertical direction of the scene. Cylinders are appropriate for 3-D objects that can only rotate about a vertical axis but not about other axes, and are constrained to move by horizontal translation only, orthogonal to the vertical axis. Two vertical-axis-aligned cylinders intersect when, simultaneously, their projections on the vertical axis intersect – which are two line segments – as well their projections on the horizontal plane – two circular disks. Both are easy to test. In video games, bounding cylinders are often used as bounding volumes for people standing upright.

A **bounding box** is a cuboid, or in 2-D a rectangle, containing the object. In dynamical simulation, bounding boxes are preferred to other shapes of bounding volume such as bounding spheres or cylinders for objects that are roughly cuboid in shape when the intersection test needs to be fairly accurate. The benefit is obvious, for example, for objects that rest upon other, such as a car resting on the ground: a bounding sphere would show the car as possibly intersecting with the ground, which then would need to be rejected by a more expensive test of the actual model of the car; a bounding box immediately shows the car as not intersecting with the ground, saving the more expensive test.

In many applications the bounding box is aligned with the axes of the coordinate system, and it is then known as an **axis-aligned bounding box (AABB)**. To distinguish the general case from an AABB, an arbitrary bounding box is sometimes called an **oriented bounding box (OBB)**. AABBs are much simpler to test for intersection than OBBs, but have the disadvantage that they cannot be rotated.

A **discrete oriented polytope (DOP)** generalizes the AABB. A DOP is a convex polytope containing the object (in 2-D a polygon; in 3-D a polyhedron), constructed by taking a number of suitably oriented planes at infinity and moving them until they collide with the object. The DOP is then the convex polytope resulting from intersection of the half-spaces bounded by the planes. Popular choices for constructing DOPs in 3-D graphics include the **axis-aligned bounding box**, made from 6 axis-aligned planes, and the **beveled bounding box**,

made from 10 (if beveled only on vertical edges, say) 18 (if beveled on all edges), or 26 planes (if beveled on all edges and corners). A DOP constructed from k planes is called a k -DOP; the actual number of faces can be less than k , since some can become degenerate, shrunk to an edge or a vertex.

A **convex hull** is the smallest convex volume containing the object. If the object is the union of a finite set of points, its convex hull is a polytope, and in fact the smallest possible containing polytope.

Basic Intersection Checks

For some types of bounding volume (OBB and Convex Polyhedra), an effective check is that of the Separating Axis Test. The idea here is that, if there exists an axis by which the objects do not overlap, then the objects do not intersect. Usually the axes checked are those of the basic axes for the volumes (the unit axes in the case of an AABB, or the 3 base axes from each OBB in the case of OBBs). Often, this is followed by also checking the cross-products of the previous axes (one axis from each object).

In the case of an AABB, this test becomes a simple set of overlap tests in terms of the unit axes. For an AABB defined by M, N against one defined by O, P they do not intersect if $(M_x > P_x)$ or $(O_x > N_x)$ or $(M_y > P_y)$ or $(O_y > N_y)$ or $(M_z > P_z)$ or $(O_z > N_z)$.

An AABB can also be projected along an axis, for example, if it has edges of length L and is centered at C , and is being projected along the axis N :

$$r = 0.5L_x|N_x| + 0.5L_y|N_y| + 0.5L_z|N_z|, \text{ and } b = C^*N \text{ or } b = C_xN_x + C_yN_y + C_zN_z, \text{ and } m = b - r, n = b + r$$

where m and n are the minimum and maximum extents.

An OBB is similar in this respect, but is slightly more complicated. For an OBB with L and C as above, and with I, J , and K as the OBB's base axes, then:

$$r = 0.5L_x|N^*I| + 0.5L_y|N^*J| + 0.5L_z|N^*K|$$

$$m = C^*N - r \text{ and } n = C^*N + r$$

For the ranges m, n and o, p it can be said that they do not intersect if $m > p$ or $o > n$. Thus, by projecting the ranges of 2 OBBs along the I, J , and K axes of each OBB, and checking for non-intersection, it is possible to detect non-intersection. By additionally checking along the cross products of these axes ($I_0 \times I_1, I_0 \times J_1, \dots$) one can be more certain that intersection is impossible.

This concept of determining non-intersection via use of axis projection also extends to Convex Polyhedra, however with the normals of each polyhedral face being used instead of the base axes, and with the extents being based on

the minimum and maximum dot products of each vertex against the axes. Note that this description assumes the checks are being done in world space.

External link

- Illustration of several DOPs for the same model, from [epicgames.com](http://udn.epicgames.com/Two/CollisionTutorial/kdop_sizes.jpg)¹⁴

Source: http://en.wikipedia.org/wiki/Bounding_volume

Principal Authors: Gdr, Lambiam, Jaredwf, Tosha, Frank Shearar

Box modeling

Box modeling is a technique in 3D modeling where the model is created by modifying primitive shapes in a way to create a "rough draft" of the final model. This is in contrast with the edge modeling method, where the modeler edits individual vertices. While a primary function of box modeling involves extruding and scaling the flat planes that make up a model, called *faces*, another, more prominent feature of this art style gives it a second, less rudimentary name of subdivision modeling.

Subdivision

Subdivision modeling is derived from the idea that as a work is progressed, should the artist want to make his work appear less sharp, or "blocky", each face would be divided up into smaller, more detailed faces (usually into sets of four). However, more experienced box modelers manage to create their model without subdividing the faces of the model. Basically, box modeling is broken down into the very basic concept of polygonal management.

Quads

Quadrilateral faces, or quads, are the fundamental entity in box modeling. Obviously, if one were to start off from a cube, the artist would have six quad faces to work with before extrusion. While most applications for three-dimensional art provide abilities for faces up to any size, results are often more predictable and consistent by working with quads. This is so because of the fact if you were to draw an X connecting the corner vertices of a quad, the surface normal is nearly always the same. (We say *nearly* under the logic that should a

¹⁴ http://udn.epicgames.com/Two/CollisionTutorial/kdop_sizes.jpg

quad be something other than a perfect parallelogram, such as a rhombus, or a trapezoid, the surface normal would not be the same.)

Advantages and disadvantages

Box modeling is a modeling method that is quick and easy to learn. It is also appreciably faster than placing each point individually. However, it is difficult to add high amounts of detail to models created using this technique without practice.

Box modeling in web design

Box modeling is also a technique used in web design to lay out the various elements of a web page before the design is implemented in HTML or XHTML.

Source: http://en.wikipedia.org/wiki/Box_modeling

Principal Authors: Greatpoo, Furrykef, Scott5114, Jonburney, Derbeth

Bui Tuong Phong

Bui Tuong Phong (1942–1975) was a Vietnamese-born computer graphics researcher and pioneer. His works are most often indexed under his family name, **Bui Tuong** (from the Vietnamese **Bùi Tuòng**), which comes before his given name by Vietnamese name convention. But his inventions are remembered under his given name **Phong**.

Dr. Bui Tuong was the inventor of the →Phong reflection model and the →Phong shading interpolation method, techniques widely used in computer graphics. Bui Tuong published the description of the algorithms in his 1973 PhD dissertation¹⁵ and a 1975 paper¹⁶. He received his Ph.D. from the University of Utah in 1973.

References

Source: http://en.wikipedia.org/wiki/Bui_Tuong_Phong

Principal Authors: Dicklyon, T-tus, Hathawayc, Calvin08, Mikkalai

¹⁵ Bui Tuong Phong, *Illumination of Computer-Generated Images*, Department of Computer Science, University of Utah, UTEC-CSs-73-129, July 1973.

¹⁶ Bui Tuong Phong, "Illumination for Computer Generated Images," Comm. ACM, Vol 18(6):311-317, June 1975.

Bump mapping

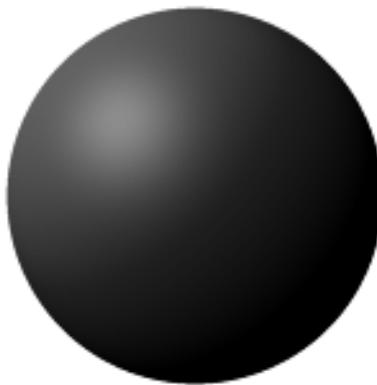


Figure 22 A sphere without bump mapping.

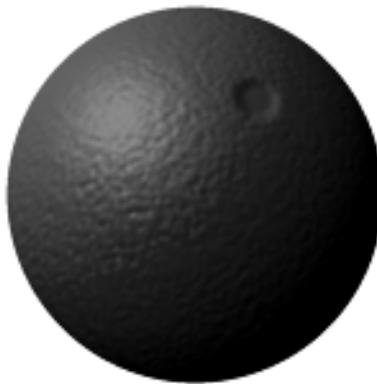


Figure 23 This sphere is geometrically the same as the first, but has a bump map applied. This changes how it reacts to shading, giving it the appearance of a bumpy texture resembling that of an orange.

Bump mapping is a computer graphics technique where at each pixel, a perturbation to the surface normal of the object being rendered is looked up in a heightmap and applied before the illumination calculation is done (see, for instance, →Phong shading). The result is a richer, more detailed surface representation that more closely resembles the details inherent in the natural world.

The difference between displacement mapping and bump mapping is evident in the example images; in bump mapping, the normal alone is perturbed, not the geometry itself. This leads to artifacts in the silhouette of the object (the sphere still has a circular silhouette).

Fake bump mapping

Programmers of 3D graphics sometimes use computationally cheaper **fake bump mapping** techniques in order to simulate bump mapping. One such method uses texel index alteration instead of altering surface normals, often used for '2D' bump mapping. As of GeForce 2 class card this technique is implemented in graphics accelerator hardware.

Full-screen 2D fake bump mapping, which could be easily implemented with a very simple and fast rendering loop, was a very common visual effect in the demos of the 1990's.

References

- Blinn, James F. "Simulation of Wrinkled Surfaces", Computer Graphics, Vol. 12 (3), pp. 286-292 SIGGRAPH-ACM (August 1978)

Links

- <http://www.jawed.com/bump/> Real-time bump mapping demo in Java (includes source code)

See also

- →Normal mapping
- →Parallax mapping
- →Displacement mapping

Source: http://en.wikipedia.org/wiki/Bump_mapping

Principal Authors: Viznut, Loisel, Brion VIBBER, Engwar, Kimiko, Xezbeth, Madoka, Mrwojo, Branko, Jumbuck

Carmack's Reverse

Carmack's Reverse is a →3D computer graphics technique for stencil shadow volumes that solves the problem of when the viewer's "eye" enters the shadow volume by tracing backwards from some point at infinity to the eye of the camera. It is named for one of its inventors, game programmer John Carmack.

According to a widely reported forum posting, Sim Dietrich may have presented the technique to the public at a developer's forum in 1999. William Bilodeau and Michael Songy filed a US patent application for the technique the same year. U.S. Patent 6384822¹⁷ entitled "Method for rendering shadows using a shadow volume and a stencil buffer" issued in 2002.

Bilodeau and Songy assigned their patent ownership rights to Creative Labs. Creative Labs, in turn, granted Id Software a license to use the invention free of charge in exchange for future support of EAX technology.

Carmack independently invented the algorithm in 2000 during the development of *Doom 3*. He is generally given credit for it since he, unlike Bilodeau and Songy, advertised his discovery to the larger public.

References

- *Creative Pressures id Software With Patents*¹⁸. Slashdot. (July 28, 2004). Retrieved on 2006-05-16.
- *Creative patents Carmack's reverse*¹⁹. The Tech Report. (July 29, 2004). Retrieved on 2006-05-16.
- *Creative gives background to Doom III shadow story*²⁰. The Inquirer. (July 29, 2004). Retrieved on 2006-05-16.

See also

- List of software patents

Source: http://en.wikipedia.org/wiki/Carmack%27s_Reverse

Principal Authors: Frecklefoot, AlistairMcMillan, RL, Fredrik, CesarB, Andymadigan

¹⁷ <http://patft.uspto.gov/netacgi/nph-Parser?patentnumber=6384822>

¹⁸ <http://games.slashdot.org/games/04/07/28/1529222.shtml>

¹⁹ <http://techreport.com/onearticle.x/7113>

²⁰ <http://www.theinquirer.net/?article=17525>

Catmull-Clark subdivision surface

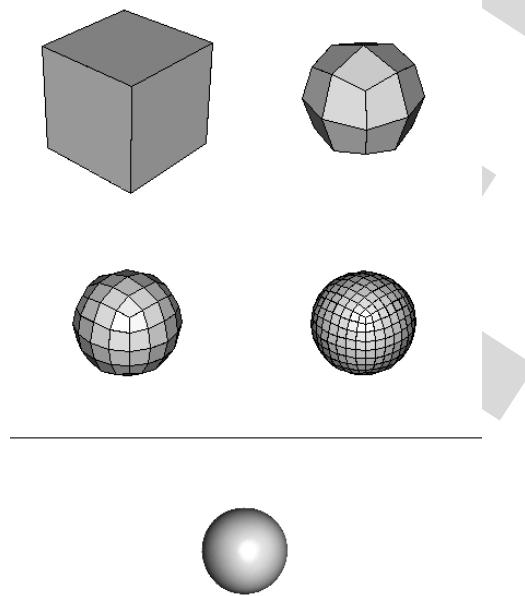


Figure 24 First three steps of Catmull-Clark subdivision of a cube with subdivision surface below

The **Catmull-Clark** algorithm is used in subdivision surface modeling to create smooth surfaces. It was devised by Edwin Catmull (of Pixar) and Jim Clark.

Procedure

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

- For each face, add a **face point**
 - Set each face point to be the *average of all original points for the respective face*.
 - 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 edge, add an **edge point**.

- Set each edge point to be the *average of all neighbouring face points and original points.*
- For each original point P , take the average F of all n face points for faces touching P , and take the average E of all n edge points for edges touching P . **Move each original point to the point** $\frac{F+2E+(n-3)P}{n}$.

The new mesh will consist only of quadrilaterals, which won't in general be flat. 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 C_1 at extraordinary vertices and C_2 everywhere else.

Software using Catmull-Clark subdivision surfaces

- Art of Illusion
- Blender
- Wings 3D

External links

- Recursively generated B-spline surfaces on arbitrary topological surfaces²¹, by E. Catmull and J. Clark (Computer-Aided Design²² 10(6):350-355, November 1978).
- Catmull-Clark Subdivision Surfaces²³

Source: http://en.wikipedia.org/wiki/Catmull-Clark_subdivision_surface

Principal Authors: Orderud, Ati3414, Rasmus Faber, Cyp, Mystaker1

²¹ <http://www.idi.ntnu.no/~fredrior/files/Recursively%20generated%20B-spline%20surfaces%20on%20arbitrary%20topological%20surfaces.PDF>

²² <http://www.sciencedirect.com/science/journal/00104485>

²³ <http://symbolcraft.com/graphics/subdivision/>

Cel-shaded animation

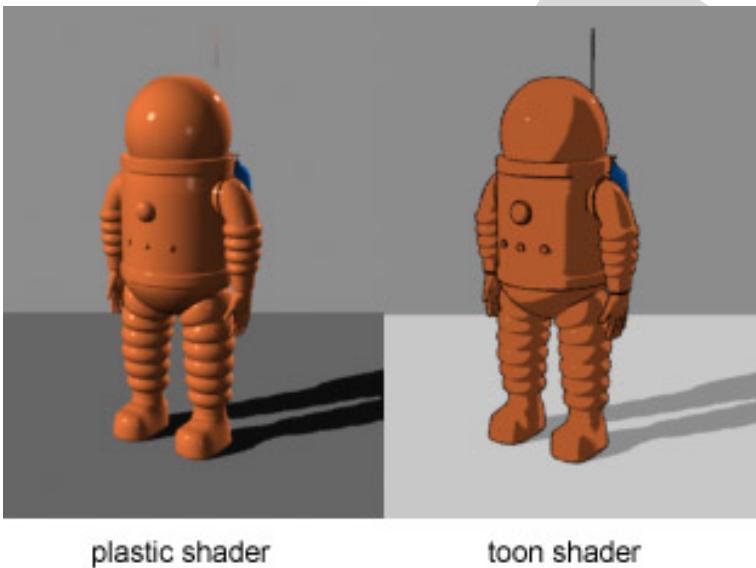


Figure 25 Object with a basic cel-shader (AKA "toon shader") and border detection.

Cel-shaded animation (also called "**cel-shading**", "**cell-shading**", or "**toon shading**") is a type of non-photorealistic rendering designed to make computer graphics appear to be hand-drawn. Cel-shading is often used to mimic the style of a comic book or cartoon. It is a quite recent addition to computer graphics, most commonly turning up in console video games. Though the end result of cel-shading has a very simplistic feel like that of hand-drawn animation, the process is complex. The name comes from the clear sheets of acetate, called **cels**, that are painted on for use in traditional 2D animation, such as Disney classics.

Process

Hand-drawn animation

Main article: Traditional animation

In **hand-drawn animation** or **traditional animation**, from the mid-19th century onwards, artists would start by creating pencil drawings; these would then

Cel-shaded animation

be transferred onto celluloid, made of cellulose nitrate, either by using xerography (a photocopying technique) and paint, or ink and paint. Later, by the mid-20th century, celluloid was made using cellulose acetate instead of cellulose nitrate due to the latter burning easily and suffering from spontaneous decomposition, though the process remained the same.

Digital animation

The cel-shading process starts with a typical 3D model. The difference occurs when a cel-shaded object is drawn on-screen. The rendering engine only selects a few shades of each colour for the object, producing a flat look. This is not the same as using only a few shades of texture for an object, as lighting and other environmental factors would come into play and ruin the effect. Therefore, cel-shading is often implemented as an additional rendering pass after all other rendering operations are completed.

In order to draw black ink lines outlining an object's contours, the back-face culling is inverted to draw back-faced triangles with black-coloured vertices. The vertices must be drawn multiple times with a slight change in translation to make the lines "thick." This produces a black-shaded silhouette. The back-face culling is then set back to normal to draw the shading and optional textures of the object. Finally, the image is composited via →Z-buffering, as the back-faces always lie deeper in the scene than the front-faces. The result is that the object is drawn with a black outline, and even contours that reside inside the object's surface in screen space.

History

Video games

The first 3D video game to feature true real-time cel shading was *Jet Set Radio* for the Sega Dreamcast in 2000. An earlier game, *Fear Effect* for the Sony PlayStation, was released in the same year and was noted for its use of dramatic textures to give an anime appearance to its characters, but lacked outlines and dynamic light-sourcing. Games before *Fear Effect* have used textures in a similar fashion, but not as starkly apparent or stylized as the game. *Wacky Races*, released on Dreamcast a few months before *Jet Set Radio* with an outline effect some mistake for cel shading, but it in-fact, still used traditional shading techniques.

In the years following *Jet Set Radio*, numerous other cel-shaded games were introduced during a minor fad involving cel-shaded graphics, yet only a few would fully match or surpass its mainstream appeal. The next games with cel-shading to capture the industry's attention in some form were 2002's *Jet Set*



Figure 26 *The Legend of Zelda: The Wind Waker* is a well-known cel-shaded game

Radio Future and *Sly Cooper and the Thievius Raccoonus*. Over time, more cel-shaded titles such as *Dark Cloud 2*, *Cel Damage*, *Klonoa 2*, the *Viewtiful Joe* series, and *XIII* were released with positive feedback, though none were considered blockbusters in terms of sales figures. Originally the only cel-shaded games to receive both positive ratings and sales after *Sly Cooper and the Thievius Raccoonus* were *The Legend of Zelda: The Wind Waker* and *Sly 2: Band of Thieves*.

NOTE: Some dispute whether games like *Klonoa 2* and *Sly Cooper* are cel-shaded. Like *Wacky Races*, they use the black outline effects associated with cel-shading, but they feature smoothly shaded surfaces with gentle lighting gradients as opposed to the harsh contrasts associated with the cel-shading technique.

Originally, Sega's *The House of the Dead 3* for the Xbox was cel-shaded. Early in HotD3's development Sega released screenshots of the then current cel-shaded graphics to the gaming community. Shortly after those initial screenshots were released, Sega announced that they were dropping the cel-shaded graphics in favour of conventional graphic techniques. There are several suspected reasons for Sega's change of heart, the most popular and most likely is that the screenshots met much negative response from gamers who disliked the cel-shaded graphical style. Many gamers claimed the cel-shading was used

purely as a gimmick in an attempt to sell more games. HotD3 was a bloody, gory and very violent light gun game which featured zombies and other mutated and deformed creatures. Many felt the cel-shaded look clashed greatly with the game's themes and content.

The use of cel-shading in video games has slowed somewhat since its inception, but the technique continues to be employed in the modern era. Recent and upcoming examples include *Dragon Quest VIII*, *Killer 7*, *Metal Gear Acid 2*, and *Ōkami*.

Examples of digital cel-shading

Main article: List of cel-shaded video games

Some of the more prominent films that have featured cel-shaded graphics are:

- *Appleseed*
- *The Animatrix* short films
- *Beyblade: The Movie - Fierce Battle*
- Various sequences from *Futurama*, a US TV series
- Parts of *Ghost in the Shell: Stand Alone Complex*, a Japanese TV series
 - The *Tachikoma Days* sequences at the end of the episodes are entirely CG using cel-shading.
- *Ghost in the Shell: S.A.C. 2nd GIG* and *Innocence: Ghost in the Shell*
- *Hot Wheels Highway 35 World Race*
- *Lego Exo-Force*
- *Rescue Heroes: The Movie*
- Various sequences from *Sonic X*, a Japanese TV series
- *Star Wars: Clone Wars*
- *Stuart Little 3: Call of the Wild*
- *The Iron Giant*
- *Tom and Jerry Blast Off to Mars*

Some of the more prominent games that have featured cel-shaded graphics are:

- *Auto Modellista*
- *Bomberman Generation*
- *Bomberman Jetters*
- *Cel Damage*
- *Crazyracing Kartrider*
- *Dark Cloud 2*
- *Dragon Ball Z: Budokai 2, 3, and Tenkaichi*

- *Dragon Quest VIII*
- *Gunggrave* series
- *Harvest Moon: Save the Homeland*
- *Jet Set Radio* series
- *Killer 7*
- *Klonoa 2*
- *The Legend of Zelda: Phantom Hourglass*
- *The Legend of Zelda: The Wind Waker*
- *Mega Man X7*
- *Mega Man X Command Mission*
- *Metal Gear Acid 2*
- *Monster Rancher 3*
- *Robotech: Battlecry*
- *Runaway: A Road Adventure*
- *Shin Megami Tensei III: Nocturne*
- *Shin Megami Tensei: Digital Devil Saga and Digital Devil Saga 2*
- *Sly Cooper* series
- *Ultimate Spider-Man*
- *Tales of Symphonia*
- *Tony Hawk's American Sk8land*
- *Viewtiful Joe* series
- *Wild Arms 3*
- *XIII*
- *X-Men Legends II: Rise of Apocalypse*
- *Zone of the Enders: The 2nd Runner*

In addition, some TV series and commercials also use cel-shading effects:

- *Atomic Betty*
- *Canada's Worst Driver*
- *Class of the Titans*
- *Daily Planet* (2005-2006 season)
- *Delilah and Julius*
- *D.I.C.E.*
- *Dragon Booster*
- *Duck Dodgers*
- *Family Guy*
- *Fairly OddParents*
- *Funky Cops*
- *Futurama* (mainly in scene-setting shots)
- *G.I. Joe: Sigma 6*
- *Gundam Seed*



Figure 27 Spider-Man squatting.

- *He-Man and the Masters of the Universe*
- *Initial D: 4th Stage*
- *Invader Zim*
- *Kirby: Right Back at Ya!*
- *MegaMan NT Warrior*
- *Monster House*
- *Ōban Star-Racers*
- *ReBoot episode My Two Bobs*
- *The Simpsons*
- *Sonic X*
- *Skyland*
- *Spider-Man*
- *Superior Defender Gundam Force*
- *The Littlest Robo*
- *Transformers Cybertron*
- *Transformers Energon*
- *Winx Club*
- *Zoids*

Cel-shaded animation

- *Lego Exo-Force* (commercial)
- *McCain's Zwak Punch* (commercial)
- *Sola/Nero/Vena/Olera/Zonte* sparkling wine (commercial)

See also

- List of cel-shaded video games

Other types of animation

- Traditional animation
- Special effects animation
- Character animation
- Computer animation and →3D computer graphics
- Skeletal animation
- See Animation#Styles and techniques of animation for more info

References

- CelShading.com²⁴. More information on 3D cel-shading including an image gallery.
- Celshader.com FAQ²⁵. Retrieved August 2, 2005.
- IGN: Jet Grind Radio Review²⁶. Retrieved August 4, 2005.
- GameDev.net - Cel-Shading²⁷. Retrieved August 5, 2005.

Source: http://en.wikipedia.org/wiki/Cel-shaded_animation

Principal Authors: Joshfist, Jacob Poon, Andrevan, GrumpyTroll, Jacoplane, CyberSkull, MIT Trekkie, Sherool

²⁴ <http://www.celshading.com/>

²⁵ <http://www.celshader.com/FAQ.html>

²⁶ <http://dreamcast.ign.com/articles/163/163512p1.html>

²⁷ <http://www.gamedev.net/reference/articles/article1438.asp>

Cg programming language

Cg or C for Graphics is a High level shader language created by NVIDIA for programming vertex and pixel shaders.

Cg is based on the C programming language and although they share the same syntax, some features of C were modified and new data types were added to make Cg more suitable for programming graphics processing units.

Background

As a result of technical advancements in graphics cards, some areas of 3D graphics programming have become quite complex. To simplify the process, new features were added to graphics cards, including the ability to modify their rendering pipelines using vertex and pixel shaders.

In the beginning, vertex and pixel shaders were programmed at a very low level with only the assembly language of the graphics processing unit. Although using the assembly language gave the programmer complete control over code and flexibility, it was fairly hard to use. A portable, higher level language for programming the GPU was needed, so Cg was created to overcome these problems and make shader development easier.

Some of the benefits of using Cg over assembly are:

- High level code is easier to learn, program, read, and understand than assembly code.
- Cg code is portable to a wide range of hardware and platforms, unlike assembly code, which usually depends on hardware and the platforms it's written for.
- The Cg compiler can optimize code and do lower level tasks automatically, which are hard to do and error prone in assembly.

Details

Data types

Cg has six basic data types, some of them are the same as in C, others are especially added for GPU programming, these types are:

- float - a 32bit floating point number
- half - a 16bit floating point number
- int - a 32bit integer
- fixed - a 12bit fixed point number

- bool - a boolean variable
- sampler* - represents a texture object

Cg also features vector and matrix data types that are based on the basic data types, such as float3, float4x4, such data types are quite common when dealing with 3D graphics programming, Cg also has struct and array data types, which work in a similar way to C equivalents.

Operators

Cg supports a wide range of operators, including the common arithmetic operators from C, the equivalent arithmetic operators for vector and matrix data types, and the common logical operators.

Functions and control structures

Cg shares the basic control structures with C, like if/else, while, and for. It also has a similar way of defining functions.

The standard Cg library

As in C, Cg features a set of functions for common tasks in GPU programming. Some of the functions have equivalents in C, like the mathematical functions abs and sin, while others are specialized in GPU programming tasks, like the texture mapping functions tex1D and tex2D.

The Cg runtime library

Cg programs are merely vertex and pixel shaders, and they need supporting programs that handle the rest of the rendering process, Cg can be used with two APIs, →OpenGL or DirectX, each has its own set of Cg functions to communicate with the Cg program, like setting the current Cg shader, passing parameters, and such tasks.

A sample Cg vertex shader

```
// input vertex
struct VertIn {
    float4 pos    : POSITION;
    float4 color  : COLOR0;
};

// output vertex
struct VertOut {
    float4 pos    : POSITION;
    float4 color  : COLOR0;
```

```

};

// vertex shader main entry
VertOut main(VertIn IN, uniform float4x4 modelViewProj) {
    VertOut OUT;
    OUT.pos      = mul(modelViewProj, IN.pos); // calculate output
    OUT.coords   = IN.coords; // copy input coords to output
    OUT.color     = IN.color; // copy input color to output
    OUT.color.z  = 1.0f; // blue component of color = 1.0f
    return OUT;
}

```

Applications and games that use Cg

- *Far Cry*
- PlayStation 3 compatibility
- *Crystal Space*
- *OGRE*
- Virtools Dev

Further reading

- Randima Fernando, Mark J. Kilgard, *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics*, Addison-Wesley Professional, ISBN 0-32119-496-9
- Randima Fernando, *GPU Gems: Programming Techniques, Tips, and Tricks for Real-Time Graphics*, Addison-Wesley Professional, ISBN 0-32122-832-4
- William R. Mark, R. Steven Glanville, Kurt Akeley, Mark J. Kilgard, *Cg: A System for Programming Graphics Hardware in a C-like Language*²⁸, Proceedings of SIGGRAPH 2003²⁹.
- Mark J. Kilgard, *Cg in Two Pages*³⁰, 2003.

See also

- Computer programming
- Computer graphics
- Vertex and pixel shaders
- High level shader language

²⁸ <http://www.cs.utexas.edu/~billmark/papers/Cg/>

²⁹ <http://www.cs.brown.edu/~tor/sig2003.html>

³⁰ <http://xxx.lanl.gov/ftp/cs/papers/0302/0302013.pdf>

- OpenGL shading language
- Shader Model
- →OpenGL
- DirectX

External links

- Cg in Two Pages³¹
- NVIDIA³²
- Cg home page³³
- OpenGL home page³⁴
- DirectX home page³⁵
- CgShaders.org³⁶
- NeHe Cg vertex shader tutorial³⁷
- Far Cry³⁸
- A glimpse at Cg Shader Toolkit³⁹
- Virtools⁴⁰

Source: http://en.wikipedia.org/wiki/Cg_programming_language

Principal Authors: Ayman, Optim, Orderud, T-tus, RJHall

³¹ <http://xxx.lanl.gov/ftp/cs/papers/0302/0302013.pdf>

³² <http://www.nvidia.com/>

³³ http://developer.nvidia.com/page/cg_main.html

³⁴ <http://www.opengl.org>

³⁵ <http://www.microsoft.com/windows/directx/default.aspx>

³⁶ <http://www.cgshaders.org/>

³⁷ <http://nehe.gamedev.net/data/lessons/lesson.asp?lesson=47>

³⁸ <http://www.farcry-thegame.com/>

³⁹ <http://deathfall.com/feature.php?op=showcontent&id=12>

⁴⁰ <http://www.virtools.com>

Clipmap

Clipmapping is a method of clipping a mipmap to a subset of data pertinent to the geometry being displayed. This is useful for loading as little data as possible when memory is limited, such as on a graphics card.

External Links

- SGI paper from 2002⁴¹
- SGI paper from 1996⁴²
- Description from SGI's developer library⁴³

Source: <http://en.wikipedia.org/wiki/Clipmap>

COLLADA

COLLADA is a **COLLAborative Design Activity** for establishing an interchange file format for interactive 3D applications.

COLLADA defines a standard XML schema for data interchange.

Originally established by Sony Computer Entertainment as the official format for PlayStation 3 and PlayStation Portable development, COLLADA continues to evolve through the efforts of the Khronos contributors. Dozens of commercial game studios and game engines have adopted the standard.

COLLADA version 1.4, released in January 2006, supports features such as character skinning and morph targets, rigid body dynamics and shader effects for multiple shading languages including the →Cg programming language, GLSL and HLSL.

Solutions exist to transport data from one Digital Content Creation (DCC) tool to another. Supported DCCs include Maya (using ColladaMaya), 3D Studio Max (using ColladaMax), Softimage XSI and Blender. Game engines, such as Unreal engine and the open-source C4 Engine, have also adopted this format.

⁴¹ <http://www.cs.virginia.edu/~gfx/Courses/2002/BigData/papers/Texturing/Clipmap.pdf>

⁴² http://www.sgi.com/products/software/performer/presentations/clipmap_intro.pdf

⁴³ http://techpubs.sgi.com/library/tpl/cgi-bin/getdoc.cgi?coll=0650&db=bks&srch=&fname=/SGI_Developer/Perf_PG/sgi_html/ch15.html

Google Earth (release 4) has adopted COLLADA (1.4) as their native format for describing the objects populating the earth. Users can simply drag and drop a COLLADA (.dae) file on top of the virtual earth. Alternatively, Google sketchup can also be used to create .kmz files, a zip file containing a KML file, a COLLADA (.dae) file, and all the texture images.

Two open-source utility libraries are available to simplify the import and export of COLLADA documents: the COLLADA DOM and the FCollada library. The COLLADA DOM is generated at compile-time from the COLLADA schema. It provides a low-level interface that eliminates the need for hand-written parsing routines, but is limited to reading and writing only one version of COLLADA, making it difficult to upgrade as new versions are released. In contrast, Feeling Software's FCollada provides a higher-level interface and can import all versions of COLLADA. FCollada is used in ColladaMaya, ColladaMax and several commercial game engines.

See also

- X3D
- List of vector graphics markup languages

External links

- Official homepage⁴⁴
- Forum⁴⁵
- COLLADA book⁴⁶
- COLLADA tools including: ColladaMaya, ColladaMax, FCollada and the Feeling Viewer⁴⁷
- COLLADA DOM, COLLADA RT, and COLLADA FX libraries are on sourceforge⁴⁸
- Bullet Physics Library - First open source rigid body dynamics simulator with COLLADA Physics import⁴⁹

Source: <http://en.wikipedia.org/wiki/COLLADA>

Principal Authors: Racklever, Karada, Agentsoo, Who, Khalid hassani, N00body

⁴⁴ <http://www.khronos.org/collada/>

⁴⁵ <http://collada.org/>

⁴⁶ <http://www.akpeters.com/product.asp?ProdCode=2876>

⁴⁷ <http://www.feelingsoftware.com>

⁴⁸ <http://sourceforge.net/projects/collada-dom>

⁴⁹ <http://bullet.sourceforge.net>

Comparison of Direct3D and OpenGL

In computer software, →**Direct3D** is a proprietary API designed by Microsoft Corporation that provides a standardized API for hardware 3D acceleration on the Windows platform. Direct3D is implemented, like OpenGL on the same platform, in the display driver.

→**OpenGL** is an open standard Application Programming Interface that provides a number of functions for the rendering of 2D and 3D graphics. An implementation is available on most modern operating systems.

Following is a comparison of the two APIs, structured around various considerations mostly relevant to game development.

Portability

For the most part, Direct3D is only implemented on Microsoft's Windows family of operating systems, including the embedded versions used in the Xbox family of video game consoles. Several partially functional ports of the Direct3D API have been implemented by Wine, a project to port common Windows APIs to Linux, but this work is difficult due to the dependency of DirectX as a whole on many other components of the Windows operating systems. Microsoft once started development on a port to Apple Computer's Mac OS, but later abandoned this project.

OpenGL, on the other hand, has implementations available across a wide variety of platforms including Microsoft Windows, Linux, UNIX-based systems, Mac OS X and game consoles by Nintendo and Sony, such as the PlayStation 3 and Wii. With the exception of Windows, all operating systems that allow for hardware-accelerated 3D graphics have chosen OpenGL as the primary API. Even more operating systems have OpenGL software renderers.

Microsoft's supplied OpenGL driver in Windows (including Vista) provides no hardware acceleration or direct support for extensions. Microsoft's stance on this is that eliminating OpenGL inconveniences few and allows them to support more hardware in the same testing time. Other platforms supply very few drivers for any hardware, OpenGL or otherwise.

Windows thus requires the correct drivers from the GPU's manufacturer or vendor for OpenGL hardware support, as well as the full performance in its Direct3D support. These manufacturer-supplied drivers nearly all include OpenGL support through the **ICD** (Installable Client Driver).

In terms of portability, Direct3D is a much more limiting choice; however, this "lock-in" will only be a problem for some applications. Those aiming at the

desktop computer gaming market should consider that its non-Windows segment is still relatively small. On the other hand, while Microsoft's segment of the console market is growing, it is very small, especially compared to that of Sony's, and predicted by analysts to remain so. While no current or upcoming console uses exactly standard graphics APIs, the Microsoft Xbox 360 uses a variant of Direct3D while the Sony PlayStation 3 uses a variant of →OpenGL ES.

Ease of use

In its earliest incarnation, Direct3D was known to be rather clumsy to use — to perform a state change, for instance, it required a number of complex operations. For example, to enable alpha blending, one had to create a so-called **execute buffer**, lock it, fill it with the correct opcodes (along with a structure header telling how many opcodes the buffer contained and a special "exit" opcode), unlock it, and finally send it to the driver for execution. In contrast, OpenGL only requires a single "glEnable(GL_BLEND);" call. The Direct3D model frustrated many programmers. The most famous complaint was probably made by high-profile game developer John Carmack in the .plan file in which he urged Microsoft to abandon Direct3D in favor of OpenGL.

Version 5 (the second version, but it was released with DirectX 5) abandoned execute buffers and improved the API significantly, but it was still cumbersome and didn't provide much in terms of texture management or vertex array management. Direct3D 7 provided texture management, and Direct3D 8, among other things, provided vertex array management. Currently, Direct3D's API is a well-designed API. Direct3D and OpenGL still follow different paradigms, though.

Direct3D is built upon Microsoft's COM. The use of this framework means that C++ code is a little unusual. Functions for acquiring values don't return the value in the return argument for the function because all COM functions return an "HRESULT" that tells whether the function executed correctly or not. The plus side of using COM is that you can use the same API in any COM-aware language. These include Visual Basic and Visual Basic Script, among others.

OpenGL is a specification based on the C programming language. It is built around the concept of a finite state machine, though more recent OpenGL versions have transformed it into much more of an object based system. Though the specification is built on C, it can be implemented in other languages as well.

In general, Direct3D is designed to be a 3D hardware interface. The featureset of D3D is derived from the featureset of what hardware provides. OpenGL, on the other hand, is designed to be a 3D rendering system that may be hardware

accelerated. As such, its featureset is derived from that which users find useful. These two API's are fundamentally designed under two separate modes of thought. The fact that the two APIs have become so similar in functionality shows how well hardware is converging into user functionality.

Even so, there are functional differences in how the two APIs work. Direct3D expects the application to manage hardware resources; OpenGL makes the implementation do it. This makes it much easier for the user in terms of writing a valid application, but it leaves the user more susceptible to implementation bugs that the user may be unable to fix. At the same time, because OpenGL hides hardware details (including whether hardware is even being used), the user is unable to query the status of various hardware resources. So the user must trust that the implementation is using hardware resources "best".

Until recently, another functional difference between the APIs was the way they handled rendering to textures: the Direct3D method (`SetRenderTarget()`) is convenient, while previous versions of OpenGL required the manipulation of P-buffers (pixel buffers). This was cumbersome and risky: if the programmer's codepath was different from that anticipated by the driver manufacturer, the code would have fallen back to software rendering, causing a substantial performance drop. According to a Gamasutra article⁵⁰ (registration required), the aforementioned John Carmack considered switching from OpenGL to Direct3D because of the contrived use of P-buffers. However, widespread support for the "frame buffer objects" extension, which provides an OpenGL equivalent of the Direct3D method, has successfully addressed this shortcoming.

Outside of a few minor functional differences, typically with regard to rendering to textures (the "framebuffer objects" extension did not cover everything, but the ARB is working to address this), the two APIs provide nearly the same level of functionality.

Performance

Shortly after the establishment of both Direct3D and OpenGL as viable graphics libraries, Microsoft and SGI engaged in what has been called the "API Wars". Much of the argument revolved around which API offered superior performance. This question was relevant due to the very high cost of graphics accelerators during this time, which meant the consumer market was using software renderers implemented by Microsoft for both Direct3D and OpenGL.

Microsoft had marketed Direct3D as faster based on in-house performance comparisons of these two software libraries. The performance deficit was blamed on the rigorous specification and conformance required of OpenGL.

⁵⁰ http://www.gamasutra.com/features/20040830/mcguire_02.shtml

This perception was changed at the 1996 SIGGRAPH (Special Interest Group on Computer Graphics) conference. At that time, SGI challenged Microsoft with their own optimized Windows software implementation of OpenGL called CosmoGL which in various demos matched or exceeded the performance of Direct3D. For SGI, this was a critical milestone as it showed that OpenGL's poor software rendering performance was due to Microsoft's inferior implementation, and not to design flaws in OpenGL itself.

Direct3D 9 and below have a particular disadvantage with regard to performance. Drawing a vertex array in Direct3D requires that the CPU switch to kernel-mode and call the graphics driver immediately. In OpenGL, because an OpenGL driver has portions that run in user-mode, can perform marshalling activities to limit the number of kernel-mode switches and batch numerous calls in one kernel-mode switch. In effect, the number of vertex array drawing calls in a D3D application is limited to the speed of the CPU, as switching to kernel-mode is a fairly slow and CPU intensive operation.

Direct3D 10 allows portions of drivers to run in user-mode, thus allowing D3D10 applications to overcome this performance limitation.

Outside of this, Direct3D and OpenGL applications have no significant performance differences.

Structure

OpenGL, originally designed for then-powerful SGI workstations, includes a number of features that are only useful for workstation applications. The API as a whole contains about 250 calls, but only a subset of perhaps 100 are useful for game development. However, no official gaming-specific subset was ever defined. MiniGL, released by 3Dfx as a stopgap measure to support glQuake, might have served as a starting point, but additional features like stencil were soon adopted by games, and support for the entire OpenGL standard continued. Today, workstations and consumer machines use the same architectures and operating systems, and so modern incarnations of the OpenGL standard still include these features, although only special workstation-class video cards accelerate them.

OpenGL is designed as a feature rich API regardless of hardware support. The specification often drives the implementation of hardware acceleration for these features. For example, when the GeForce 256 graphics card came out in 1999, games like Quake III Arena could already benefit from its acceleration of Transform & Lighting, because the API was designed to provide this feature. Meanwhile, Direct3D developers had to wait for the next version of Direct3D to be released and rewrite their games to use the new API before they could take advantage of hardware-based Transform and Lighting.

The advantage of OpenGL's inclusive, extensible approach is limited in practice, however, by the market dominance Direct3D has achieved. In recent years, games have rarely implemented features until Direct3D has supported them, and graphics cards vendors have been reluctant to implement features that current or upcoming versions of Direct3D will not support.

Extensions

The OpenGL **extension** mechanism is probably the most heavily disputed difference between the two APIs. OpenGL includes a mechanism where any driver can advertise its own extensions to the API, thus introducing new functionality such as blend modes, new ways of transferring data to the GPU, or different texture wrapping parameters. This allows new functionality to be exposed quickly, but can lead to confusion if different vendors implement similar extensions with different APIs. Many of these extensions are periodically standardized by the OpenGL Architecture Review Board, and some are made a core part of future OpenGL revisions.

On the other hand, Direct3D is specified by one vendor (Microsoft) only, leading to a more consistent API, but denying access to vendor-specific features. nVidia's UltraShadow⁵¹ technology, for instance, is not available in the stock Direct3D APIs at the time of writing. It should be noted that Direct3D does support texture format extensions (via so-called **FourCC's**). These were once little-known and rarely used, but are now used for DXT texture compression.

When graphics cards added support for pixel shaders (known on OpenGL as "fragment programs"), Direct3D provided a single "Pixel Shader 1.1" (PS1.1) standard which the GeForce 3 and up, and Radeon 8500 and up, claimed compatibility with. Under OpenGL the same functionality was accessed through a variety of custom extensions.

In theory, the Microsoft approach allows a single code path to support both brands of card, whereas under OpenGL the programmer had to write two separate systems. In reality, though, because of the limits on pixel processing of those early cards, Pixel Shader 1.1 was nothing more than a pseudo-assembly language version of the nVidia-specific OpenGL extensions. For the most part, the only cards that claimed PS 1.1 functionality were nVidia cards, and that is because they were built for it natively. When the Radeon 8500 was released, Microsoft released an update to Direct3D that included Pixel Shader 1.4, which was nothing more than a pseudo-assembly language version of the ATI-specific OpenGL extensions. The only cards that claimed PS 1.4 support were ATI cards because they were designed with the precise hardware necessary to make that

⁵¹ http://www.nvidia.com/object/feature_ultrashadow.html

functionality happen. In terms of early Pixel shaders, D3D's attempt at a single code path fared no better than the OpenGL mechanism.

Fortunately, this situation only existed for a short time under both APIs. Second-generation pixel shading cards were much more similar in functionality, with each architecture evolving towards the same kind of pixel processing conclusion. As such, Pixel Shader 2.0 allowed a unified code path under Direct3D. Around the same time OpenGL introduced its own ARB-approved vertex and pixel shader extensions (`GL_ARB_vertex_program` and `GL_ARB_fragment_program`), and both sets of cards supported this standard as well.

The Windows Vista Issue

An issue has arisen between Direct3D and OpenGL recently (mid-to-late 2005), with regard to support for Microsoft's new operating system, Windows Vista. The Vista OS has a feature called Desktop Compositing (which is similar in concept to Mac OS X's Quartz Compositor or Compiz on Linux based systems) that allows individual windows to blend into the windows beneath them as well as a number of other cross-window effects. The new OS's rendering system that allows this to function operates entirely through Direct3D; as such, every Windows application is automatically a Direct3D application.

The ICD driver model used in prior versions of Windows will still function however. There is one important caveat: activating an ICD will cause Vista to deactivate Desktop Compositing.

Note that this functionality only truly matters for applications running in windowed mode. Full-screen applications, the way most people run games, will not be affected. Desktop compositing wouldn't affect them regardless, and the user won't notice that windows that the user can't see have had desktop compositing deactivated. However, the problem remains for running windowed OpenGL ICDs.

Microsoft has also updated their own implementation of OpenGL. In pre-Vista OSs, if you didn't select a device context from an ICD driver, you were given Microsoft's OpenGL implementation. This was a software OpenGL implementation which was frozen at version 1.1. In Vista, Microsoft's implementation has become a wrapper around Direct3D, supporting GL versions through 1.3. Activating this driver does not turn off desktop compositing, since it is just a D3D wrapper.

The potential problem with the wrapper is that, at the time of writing this, it is known to only support OpenGL 1.4. Since 1.4 was released, GL versions have progressed to 2.0. The features introduced into the core of 2.0 have given OpenGL significant functionality improvements, particularly as it relates to

shaders. Before 2.0, developers had to rely on widely-supported extensions for shaders; only 2.0 and beyond have shader functionality built in. Additionally, Microsoft's OpenGL 1.4 implementation will not support extensions, so users will not be able to use shaders on that implementation.

According to IHVs, the problem with ICD's turning off desktop compositing can be solved in the ICDs themselves. However, in order to do so, the IHVs will need to be provided with information about the inner workings of the desktop compositing subsystem of Vista. As of yet, it is unknown whether Microsoft has supplied IHVs with the necessary information or not. Speculation may suggest that, since Vista is in beta, the necessary information may actually change between now and release, so the release of such information should wait until the release of Vista comes closer.

Update, as of March 15, 2006:

It would appear that this issue has been resolved. According to a Microsoft Blog⁵², there are 2 OpenGL paths under Vista. An application can use the default implementation, frozen at OpenGL version 1.4. An application can use an ICD as well, which comes in two flavors: legacy and Vista-compatible. A legacy ICD functions as specified above: the activation of one will turn off the desktop compositor. A Vista-compatible ICD, made by IHVs using a new internal API path provided by Microsoft, will be completely compatible with the desktop compositor. Given the statements made by the two primary OpenGL ICD vendors (ATi and nVidia), it would be reasonable to expect both to provide full Vista-compatible ICDs for Windows Vista.

Users

OpenGL has always seen more use in the professional graphics market than DirectX (Microsoft even acknowledges OpenGL's advantage in this field), while DirectX is used mostly for computer games.

At one point many professional graphics cards only supported OpenGL, however, nowadays all the major professional card manufacturers (Nvidia, ATI Technologies and Matrox) support both OpenGL and Direct3D on Microsoft Windows.

The reasons for OpenGL's advantage in the professional market is partly historical. Many professional graphics applications (for example, Softimage|3D, Alias PowerAnimator) were originally written in IRIS GL for high-end SGI workstations, then ported to OpenGL. Even long after SGI no longer dominated the market, many professional graphics cards only supported OpenGL.

⁵² <http://blogs.msdn.com/kamvedbrat/archive/2006/02/22/537624.aspx>

The many extra features of OpenGL that were previously mentioned as not useful for game development are also a factor in OpenGL's professional market advantage, because many of them are useful in professional applications.

The other reason for OpenGL's advantage there is marketing and design. DirectX is a set of APIs that were not marketed towards professional graphics applications. Indeed, they were not even designed with those applications in mind. DirectX was an API designed for low-level, high-performance hardware access for the purpose of game development. OpenGL is a much more general purpose 3D API, so it provides features that aren't necessarily exclusive towards any particular kind of user.

The principal reason for Direct3D's dominance in the gaming industry is historical. In the earliest days of hardware-accelerated 3D graphics, 3dfx was the dominant force, and their →Glide API was used by far more games than D3D or OpenGL. Glide was much lower-level than D3D or OpenGL, and thus its performance was greater than either. Performance is the most important facet for game developers, so the less easy to use Glide API was preferred over the other two. This helped catapult 3DFx into the forefront of 3D hardware in those days.

As hardware got faster, however, the performance advantages of Glide began to be outweighed by the ease of use. Also, because Glide was restricted to 3dfx hardware, and 3dfx was not being as smart about hardware design as its main competitor nVidia, a hardware neutral API was needed. The very earliest versions of Direct3D (part of DirectX version 3) was not the simplest API to use. The next Direct3D version (in DirectX 5) was much more lucid. As interest in making Glide only games or games with multiple renderers dropped, there was a choice to make: OpenGL or Direct3D 5.

Making games that use OpenGL while using the non-Direct3D portion of the DirectX API is no more difficult than making a game using all of the DirectX API. The decision to use Direct3D over OpenGL was made from simple pragmatism: in those days, OpenGL implementations were difficult to work with. Writing an OpenGL implementation requires implementing every feature of OpenGL, even if the hardware doesn't support it. If the hardware can't do it, you have to write a software rasterizer that can handle that feature.

Different GL implementations would, when activating some feature, spontaneously go into a slow software renderer. Because OpenGL has no mechanism for telling the user whether or not a feature, or combination of features, will kick the renderer into software mode, users of OpenGL had to carefully test everything that they did on every piece of hardware that they were going to support.

Adding to that is the fact that an OpenGL implementation is a complex piece of code. It is much more than a simple graphics driver that is just a low-level interface to hardware registers. It needs to keep track of a great deal of state information, and that requires a lot of code. In the early days of OpenGL implementations, the implementations themselves were quite buggy. Indeed, a perfectly functioning game could break when downloading a new graphics driver; this is a complication that many game developers didn't want to have to deal with. This feature is very useful if performance is not a primary goal, such as in professional graphics applications or off-line renderers; it guarantees the existence of functionality. However, in a game situation, where a loss of performance can destroy the feeling of the game, it is more desirable to know that the functionality doesn't exist and to simply avoid using it.

Direct3D didn't have these problems. A Direct3D driver is (or, was in those days) just a low-level interface to hardware registers. And D3D has a query mechanism that tells the application whether or not a particular feature is available in hardware. So game developers chose to use it because it did what they needed. While IHVs did resolve the bug issue to a significant degree, the issue of hardware specificity was never addressed. Even so, the need for it has decreased as more and more OpenGL specified functionality becomes implemented in hardware. Later versions of OpenGL would rarely add functionality that wasn't actually widely available in hardware. As such, the issue has, for the most part, become a non-issue.

At this point, the Windows Vista issue aside, the reason for using one over the other is typically inertia. It is what they have used in the past, so it is what they use now.

See also

- Fahrenheit graphics API

External links

- Direct3D vs. OpenGL: Which API to Use When, Where, and Why⁵³

Source: http://en.wikipedia.org/wiki/Comparison_of_Direct3D_and_OpenGL

Principal Authors: Sesse, Korval, Racklever, Warrens, Rufous, LesmanaZimmer, Imroy, Joe Jarvis, Gargaj, Smccandlish

⁵³ <http://www.gamedev.net/reference/articles/article1775.asp>

Cone tracing

Cone tracing is a derivative of the ray tracing algorithm that replaces rays, which have no thickness, with cones. Cone tracing is related to beam tracing, but uses circular rather than polygonal cross sections.

Cone tracing solves certain problems related to sampling and aliasing, which can plague conventional ray tracing. However, cones tracing creates a host of problems of its own. For example, just intersecting a cone with scene geometry leads to an enormous variety of possible results. For this reason, cone tracing has always been unpopular. In recent years, increases in computer speed have made Monte Carlo algorithms like distributed ray tracing much more viable than cone tracing, which has been all but forgotten.

Source: http://en.wikipedia.org/wiki/Cone_tracing

Principal Authors: RJHall, RJJR, CesarB, Reedbeta

Constructive solid geometry



Figure 28 The light cycles from the movie *Tron* were constructed using Constructive Solid Geometry

Constructive solid geometry (CSG) is a technique used in solid modeling. CSG is often, but not always, a procedural modeling technique used in →3D

Constructive solid geometry

computer graphics and CAD. Constructive solid geometry allows a modeler to create a complex surface or object by using Boolean operators to combine objects. Often CSG presents a model or surface that appears visually complex, but is actually little more than cleverly combined or decombined objects. (In some cases, constructive solid geometry is performed on polygonal meshes, and may or may not be procedural and/or parametric.)

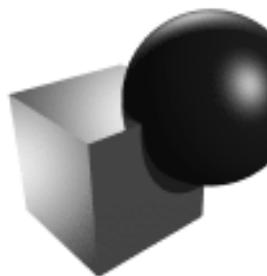
The simplest solid objects used for the representation are called **primitives**. Typically they are the objects of simple shape: cuboids, cylinders, prisms, pyramids, spheres, cones. The set of allowable primitives is limited by each software package. Some software packages allow CSG on curved objects while other packages do not.

It is said that an object is **constructed** from primitives by means of allowable **operations**, which are typically Boolean operations on sets: union, intersection and difference.

Operations

In modeling packages, basic geometric objects such as the cube or 'box', sphere or ellipse, torus, and a number of other shapes that can be described using a mathematical formula, are commonly known as *primitives*. These objects can typically be described by a procedure which accepts some number of parameters; for example, a sphere may be described by the coordinates of its center point, along with a radius value. These primitives can be combined into compound objects using operations like these:

Boolean union



The merger of two objects into one.

Boolean difference



The subtraction of one object from another.

Boolean intersection



The portion common to both objects.

Table 1 Operations in constructive solid geometry

Applications of CSG

Constructive solid geometry has a number of practical uses. It is used in cases where simple geometric objects are desired, or where mathematical accuracy is important. The Unreal engine uses this system, as do *Hammer* (the mapping engine for the Source engine) and *Quake*. (*Hammer* actually started out as an editor for *Quake*.) BRL-CAD is a solid modeling CAD package that is fundamentally based on CSG modeling techniques. CSG is popular because a modeler can use a set of relatively simple objects to create very complicated geometry. When CSG is procedural or parametric, the user can revise their complex geometry by changing the position of objects or by changing the Boolean operation used to combine those objects.

Source: http://en.wikipedia.org/wiki/Constructive_solid_geometry

Principal Authors: Oleg Alexandrov, Mikkalai, Michael Hardy, Operativem, Merovingian

Conversion between quaternions and Euler angles

Spatial rotations in three dimensions can be parametrized using both Euler angles and unit quaternions. This article explains how to convert between the two representations. Actually this simple use of "quaternions" was first presented by Euler some seventy years earlier than Hamilton to solve the problem of "magic squares." For this reason the dynamics community commonly refers to quaternions in this application as "Euler parameters".

A unit quaternion can be described as:

$$\mathbf{q} = [q_0 \quad q_1 \quad q_2 \quad q_3]^T$$

$$|\mathbf{q}|^2 = q_0^2 + q_1^2 + q_2^2 + q_3^2 = 1$$

$$q_0 = \cos(\alpha/2)$$

$$q_1 = \sin(\alpha/2) \cos(\beta_x)$$

$$q_2 = \sin(\alpha/2) \cos(\beta_y)$$

$$\mathbf{q}_3 = \sin(\alpha/2) \cos(\beta_z)$$

where α is a simple rotation angle and $\beta_x, \beta_y, \beta_z$, are the "direction cosines" locating the axis of rotation (Euler's Theorem).



Figure 29

Similarly for Euler angles, we use (in terms of flight dynamics):

- Roll - ϕ : rotation about the X-axis
- Pitch - θ : rotation about the Y-axis
- Yaw - ψ : rotation about the Z-axis

where the X-axis points forward, Y-axis to the right and Z-axis downward and in the example to follow the rotation occurs in the order yaw, pitch, roll (about body-fixed axes).

Rotation matrices

The orthogonal matrix corresponding to a rotation by the unit quaternion \mathbf{q} is given by

$$\begin{bmatrix} 1 - 2(q_2^2 + q_3^2) & 2(q_1q_2 - q_0q_3) & 2(q_0q_2 + q_1q_3) \\ 2(q_1q_2 + q_0q_3) & 1 - 2(q_1^2 + q_3^2) & 2(q_2q_3 - q_0q_1) \\ 2(q_1q_3 - q_0q_2) & 2(q_0q_1 + q_2q_3) & 1 - 2(q_1^2 + q_2^2) \end{bmatrix}$$

The orthogonal matrix corresponding to a rotation with Euler angles ϕ, θ, ψ , is given by

$$\begin{bmatrix} \cos \theta \cos \psi & -\cos \phi \sin \psi + \sin \phi \sin \theta \cos \psi & \sin \phi \sin \psi + \cos \phi \sin \theta \cos \psi \\ \cos \theta \sin \psi & \cos \phi \cos \psi + \sin \phi \sin \theta \sin \psi & -\sin \phi \cos \psi + \cos \phi \sin \theta \sin \psi \\ -\sin \theta & \sin \phi \cos \theta & \cos \phi \cos \theta \end{bmatrix}$$

Conversion

By comparing the terms in the two matrices, we get

$$\mathbf{q} = \begin{bmatrix} \cos(\phi/2) \cos(\theta/2) \cos(\psi/2) + \sin(\phi/2) \sin(\theta/2) \sin(\psi/2) \\ \sin(\phi/2) \cos(\theta/2) \cos(\psi/2) - \cos(\phi/2) \sin(\theta/2) \sin(\psi/2) \\ \cos(\phi/2) \sin(\theta/2) \cos(\psi/2) + \sin(\phi/2) \cos(\theta/2) \sin(\psi/2) \\ \cos(\phi/2) \cos(\theta/2) \sin(\psi/2) - \sin(\phi/2) \sin(\theta/2) \cos(\psi/2) \end{bmatrix}$$

For Euler angles we get:

$$\begin{bmatrix} \phi \\ \theta \\ \psi \end{bmatrix} = \begin{bmatrix} \arctan \frac{2(q_0 q_1 + q_2 q_3)}{1 - 2(q_1^2 + q_2^2)} \\ \arcsin(2(q_0 q_2 - q_3 q_1)) \\ \arctan \frac{2(q_0 q_3 + q_1 q_2)}{1 - 2(q_2^2 + q_3^2)} \end{bmatrix}$$

Singularities

One must be aware of singularities in the Euler angle parametrization when the pitch approaches $\pm 90^\circ$ (north/south pole). These cases must be handled specially.

Source: http://en.wikipedia.org/wiki/Conversion_between_quaternions_and_Euler_angles

Principal Authors: Orderud, Patrick, Icairns, Oleg Alexandrov, Woohookitty

Cornell Box

The **Cornell Box** is a test aimed at determining the accuracy of rendering software by comparing the rendered scene with an actual photograph of the same scene. It was created by the Cornell University Program of Computer Graphics for a paper published in 1984.

A physical model of the box is created and photographed with a CCD camera. The exact settings are then measured from the scene: emission spectrum of the light source, reflectance spectra of all the surfaces, exact position and size of all objects, walls, light source and camera.

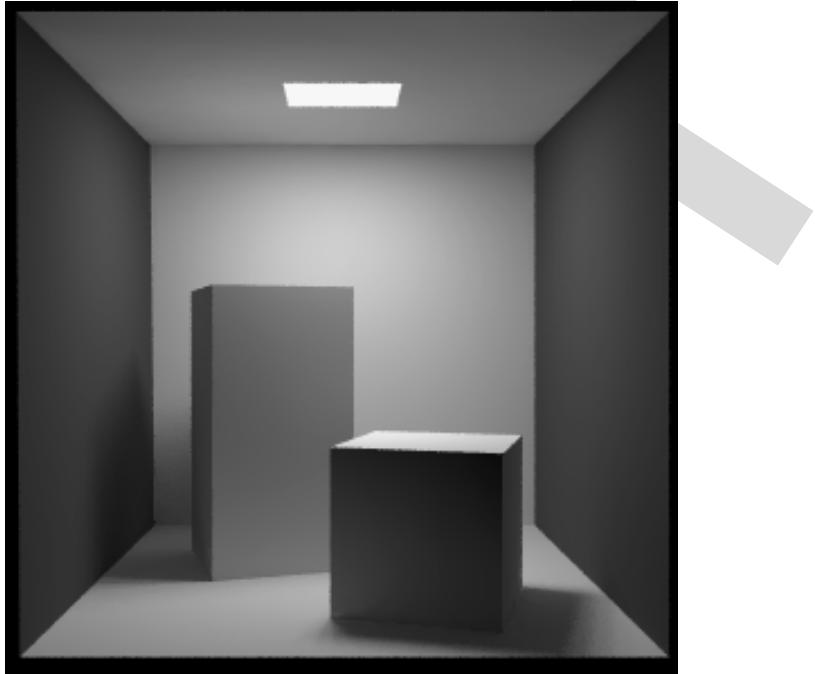


Figure 30 Standard Cornell Box rendered with POV-Ray

The same scene is then reproduced in the renderer, and the output file is compared with the photography.

The basic environment consists of:

- One light source in the center of a white ceiling
- A green right wall
- A red left wall
- A white back wall
- A white floor

Objects are often placed inside the box. The first objects placed inside the environment were two white boxes. Another common version first used to test photon mapping includes two spheres: one with a perfect mirror surface and one made of glass.

The physical properties of the box are designed to show *diffuse interreflection*. For example, some light should reflect off the red and green walls and bounce onto the white walls, so parts of the white walls should appear slightly red or green.

Today, the Cornell Box is often used to show off renderers in a similar way as the Stanford Bunny, the →Utah teapot, and Lenna: computer scientists often use the scene just for its visual properties without comparing it to test data from a physical model.

See also

- →Utah teapot
- Lenna
- Stanford Bunny
- Suzanne

External links

- The Cornell Box website⁵⁴

Source: http://en.wikipedia.org/wiki/Cornell_Box

Principal Authors: SeeSchloss, T-tus, John Fader, 1983, Reedbeta

Crowd simulation

Crowd simulation is the process of simulating the movement of a large number of objects or characters, now often appearing in →3D computer graphics for film.

The need for crowd simulation arises when a scene calls for more characters than can be practically animated using conventional systems, such as skeletons/bones.

Animators typically create a library of motions, either for the entire character or for individual body parts. To simplify processing, these animations are sometimes *baked* as morphs. Alternatively, the motions can be generated *procedurally* - i.e. choreographed automatically by software.

The actual movement and interactions of the crowd is typically done in one of two ways:

- *Particle Motion*: The characters are attached to point particles, which are then animated by simulating wind, gravity, attractions, and collisions. The

⁵⁴ <http://www.graphics.cornell.edu/online/box/>

particle method is usually inexpensive to implement, and can be done in most 3D software packages. However, the method is not very realistic because it is difficult to direct individual entities when necessary, and because motion is generally limited to a flat surface.

- **Crowd AI:** The entities - also called agents - are given artificial intelligence, which guides the entities based on one or more of sight, hearing, basic emotion, energy level, aggressiveness level, etc.. The entities are given goals and then interact with each other as members of a real crowd would. They are often programmed to respond to changes in environment, enabling them to climb hills, jump over holes, scale ladders, etc. This system is much more realistic than particle motion, but is very expensive to program and implement.

The most notable examples of AI simulation can be seen in New Line Cinema's *The Lord of the Rings* films, where AI armies of many thousands battle each other. The crowd simulation was done using Weta Digital's MASSIVE software. **Crowd simulation** can also refer to simulations based on group dynamics and crowd psychology, often in public safety planning. In this case, the focus is just the behavior of the crowd, and not the visual realism of the simulation.

See also

- →3D computer graphics
- Artificial intelligence
- Emergent behavior
- MASSIVE (animation)
- Multi-agent system
- →Particle system

External links

- NetLogo⁵⁵, a free software for multi-agent modeling, simulation, and the like.
- MASSIVE⁵⁶, the software used in *The Lord of the Rings* films.

Source: http://en.wikipedia.org/wiki/Crowd_simulation

Principal Authors: Aniboy2000, The Anome, Pmkpmk, Kuru, JonHarder

⁵⁵ <http://ccl.northwestern.edu/netlogo>

⁵⁶ <http://massivesoftware.com>

Cube mapping

In computer graphics, **cube mapping** is a technique that takes a three dimensional texture coordinate and returns a texel from a given cube map (very similar to a skybox's texture's format). This texture coordinate is a vector that specifies which way to look from the center of the cube mapped cube to get the desired texel.

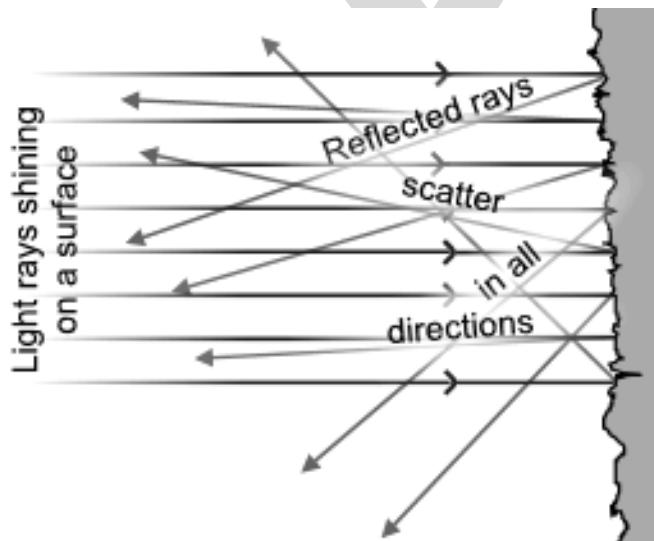
See also:

- Skybox (video games)
- Cube mapped reflection

Source: http://en.wikipedia.org/wiki/Cube_mapping

Principal Authors: TopherTG

Diffuse reflection



Diffuse reflection is the reflection of light from an uneven or granular surface such that an incident ray is seemingly reflected at a number of angles. It is the complement to specular reflection. If a surface is completely nonspecular,

Diffuse reflection

the reflected light will be evenly spread over the hemisphere surrounding the surface ($2 \times \pi$ steradians).

The most familiar example of the distinction between specular and diffuse reflection would be matte and glossy paints as used in home painting. Matte paints have a higher proportion of diffuse reflection, while gloss paints have a greater part of specular reflection.

Diffuse interreflection is a process whereby light reflected from an object strikes other objects in the surrounding area, illuminating them. Diffuse interreflection specifically describes light reflected from objects which are not shiny or specular. In real life terms what this means is that light is reflected off non-shiny surfaces such as the ground, walls, or fabric, to reach areas not directly in view of a light source. If the diffuse surface is colored, the reflected light is also colored, resulting in similar coloration of surrounding objects.

In →3D computer graphics, diffuse interreflection is an important component of global illumination. There are a number of ways to model diffuse interreflection when rendering a scene. →Radiosity and photon mapping are two commonly used methods.

See also

- Optics
- Reflectivity
- Specular reflection
- →Lambertian reflectance

Source: http://en.wikipedia.org/wiki/Diffuse_reflection

Principal Authors: JohnOwens, Srleffler, Patrick, Francs2000, Flamurai

Digital puppetry

Digital puppetry is the manipulation and performance of digitally animated 2D or 3D figures and objects in a virtual environment that are rendered in real-time by computers. It is most commonly used in film and television production, but has also been utilized in interactive theme park attractions and live theatre.

The exact definition of what is and is not digital puppetry is subject to debate within the puppetry and computer graphics communities, but it is generally agreed that digital puppetry differs from conventional computer animation in

that it involves performing characters in real time, rather than animating them frame by frame.

Digital puppetry is closely associated with motion capture technologies and 3D animation. It is sometimes referred to as Performance Animation. Machinima is a form of digital puppetry, although most machinima creators do not identify themselves as puppeteers.

History & Usage

Digital puppetry was pioneered by the late Jim Henson, creator of The Muppets. The character Waldo C. Graphic in the Muppet television series *The Jim Henson Hour* is widely regarded to have been the first example of a digitally animated figure being performed and rendered in real-time on television and grew out of experiments Henson conducted in 1987 with a computer generated version of Kermit the Frog for InnerTube, which was the unaired television pilot for Jim Henson Hour.

Another early digital puppet was Mike Normal, which was developed by computer graphics company deGraf/Wahrman and was first demonstrated at the 1988 SIGGRAPH convention. The system developed by deGraf/Wahrman to perform Mike Normal was later used to create a representation of the villain Cain in the motion picture *RoboCop 2*, which is believed to be the first example of digital puppetry being used to create a character in a full-length motion picture.

In 1994, the BBC introduced a live digital puppet cat called Ratz, in the TV show *Live & Kicking*. He became the first real-time rendered digital puppet to appear on live TV. He also co-presented Children's BBC, and was eventually given his own show, *RatzRun*.

A more recent example of digital puppetry from 2003 is "Bugs Live", a digital puppet of Bugs Bunny created by Phillip Reay for Warner Brothers Pictures. The puppet was created using hand drawn frames of animation that were puppeteered by Bruce Lanoil and David Barclay. The Bugs Live puppet was used to create nearly 900 minutes of live, fully interactive interviews of 2D animated Bugs character about his role in then recent movie *Looney Tunes: Back In Action* in English and Spanish. Bugs Live also appeared at the 2004 SIGGRAPH Digital Puppetry Special Session with the Muppet puppet Gonzo.

In 2004 Walt Disney Imagineering used digital puppetry techniques to create the Turtle Talk with Crush attractions at the Walt Disney World and Disney's California Adventure theme parks. In the attraction, a hidden puppeteer performs and voices a digital puppet of Crush, the laid-back sea turtle from *Finding Nemo*, on a large rear-projection screen. To the audience Crush appears to be

swimming inside an aquarium and engages in unscripted, real-time conversations with theme park guests.

Types of Digital Puppetry

Waldo Puppetry - A digital puppet is controlled onscreen by a puppeteer who uses a telemetric input device connected to the computer. The X-Y-Z axis movement of the input device causes the digital puppet to move correspondingly. A keyboard is sometimes used in place of a telemetric input device.

Motion Capture Puppetry (Mocap Puppetry) - An object (puppet) or human body is used as a physical representation of a digital puppet and manipulated by a puppeteer. The movements of the object or body are matched correspondingly by the digital puppet in real-time.

Machinima - A production technique that can be used to perform digital puppets. Machinima as a production technique concerns the rendering of computer-generated imagery (CGI) using low-end 3D engines in video games. Players act out scenes in real-time using characters and settings within a game and the resulting footage is recorded and later edited into a finished film.

See also

- →Motion capture
- Machinima
- Puppets
- Puppeteer

External links

- Machin-X⁵⁷ - Discussion of theories, tools and applications of digital puppetry as well as news from the digital puppetry community.
- Machinima.com⁵⁸ - Large web portal for machinima.

Source: http://en.wikipedia.org/wiki/Digital_puppetry

⁵⁷ <http://machin-x.blogspot.com>

⁵⁸ <http://www.machinima.com>

Dilution of precision (computer graphics)

Dilution of precision is an algorithmic trick used to handle difficult problems in hidden line removal, caused when horizontal and vertical edges lay on top of each other due to numerical instability. Numerically, the severity escalates when a CAD model is viewed along the principal axii or when a geometric form is viewed end-on. The trick is to alter the view vector by a small amount, thereby hiding the flaws. Unfortunately, this mathematical modification introduces new issues of its own, namely that the exact nature of the original problem has been destroyed, and visible artifacts of this kluge will continue to haunt the algorithm. One such issue is that edges that were well defined and hidden will now be problematic. Another common issue is that the bottom edges on circles viewed end-on will often become visible and propagate their visibility thought the problem.

External link

Source: http://en.wikipedia.org/wiki/Dilution_of_precision_%28computer_graphics%29

Principal Authors: Wheger, David Levy, RJHall, Whitepaw

Direct3D

Direct3D is part of Microsoft's DirectX API. Direct3D is only available for Microsoft's various Windows operating systems (Windows 95 and above) and is the base for the graphics API on the Xbox and Xbox 360 console systems. Direct3D is used to render three dimensional graphics in applications where performance is important, such as games. Direct3D also allows applications to run fullscreen instead of embedded in a window, though they can still run in a window if programmed for that feature. Direct3D uses hardware acceleration if it is available on the graphic board.

Direct3D is a 3D API. That is, it contains many commands for 3D rendering, but contains few commands for rendering 2D graphics.⁵⁹ Microsoft strives to continually update Direct3D to support the latest technology available on 3D graphics cards. Direct3D offers full vertex software emulation but no pixel

⁵⁹ Microsoft DirectX SDK Readme (December 2005)(<http://msdn.microsoft.com/directx/sdk/readmepage/default.aspx>)

software emulation for features not available in hardware. For example, if a program programmed using Direct3D requires pixel shaders and the graphics card on the user's computer does not support that feature, Direct3D will not emulate it. The program will most likely exit with an error message. The API does define a *Reference Rasterizer* (or REF device), which emulates a generic graphics card, although it's too slow to be used in any application to emulate pixel shaders and is usually ignored.

Direct3D's main competitor is →OpenGL. There are numerous features and issues that proponents for either API disagree over, see comparison of Direct3D and OpenGL for a summary.

Version history

In 1992, Servan Keondjian started a company named RenderMorphics, which developed a 3D graphics API named Reality Lab, which was used in medical imaging and CAD software. Two versions of this API were released. Microsoft bought RenderMorphics in February 1995, bringing Keondjian on board to implement a 3D graphics engine for Windows 95. This resulted in the first version of Direct3D.

In DirectX version 7.0, the .dds texture format was introduced.

As of DirectX version 8.0, Direct3D was rolled up into a package called DirectX Graphics. DirectX Graphics was meant to be a combination of DirectDraw and Direct3D, but in reality was just Direct3D with a few DirectDraw features added. Most users still refer to DirectX Graphics as Direct3D.

Direct3D was not considered to be user friendly, but as of DirectX version 8.1, many usability problems were resolved. Direct3D (DX8) contained many very powerful 3D graphics features, such as vertex shaders, pixel shaders, fog, bump mapping and texture mapping.

DirectX version 9.0 added a new version of the →High Level Shader Language, support for high dynamic range lighting, multiple render targets, and vertex buffer indexing.

Direct3D 10

Microsoft is currently working on a large update to Direct3D API. Originally called Windows Graphics Foundation, DirectX 10, and later DirectX Next, but currently referred to as Direct3D 10, it will appear as part of Windows Vista. Version 10 will represent a departure from the driver model of DirectX Graphics 9.0, with the addition of a scheduler and memory virtualization system. Direct3D 10 will forego the current DirectX practice of using "capability bits" to indicate which features are active on the current hardware. Instead, Direct3D

10 will define a minimum standard of hardware capabilities which must be supported for a display system to be "Direct3D 10 compatible". According to Microsoft, Direct3D 10 will be able to display graphics up to 8 times more quickly than DirectX Graphics 9.0c. In addition, Direct3D 10 will incorporate Microsoft Shader Model 4.0.

Vista will ship with Windows Graphics Foundation (WGF) 1.0. WGF 2.0 will ship later (after Vista). Beneath the WGF is either legacy (XP) drivers with limited functionality or Windows Display Driver Model (WDDM) drivers in one of two flavors: Basic (for existing hardware) and Advanced (for post-Vista hardware). What's currently referred to as DX10 is really WGF 2.0 atop Advanced WDDM drivers running on post-Vista hardware. WGF 1.0 (atop either flavor of WDDM), as shipping with Vista, is referred to by MS as DX9.L – it's most definitely not DX10. It certainly does have some benefits – managed resources for "unlimited" memory (limited by virtual memory), better gamma control and text rendering, some performance improvements, etc – but it's a relatively minor upgrade, and it's not going to drive radical new development in either games or GPUs.

DX10 functionality will require WGF 2.0, and will require the Advanced WDDM, which in turn will require new graphics hardware, even beyond unified shaders and an updated shader model: it includes things like additional pipeline stages (geometry shaders), hardware virtualization (to allow multiple threads/processes time-sliced use of the GPU), demand-paged virtual graphics memory, offload of various operations from the CPU to the GPU, and more. No GPU is going to be available with the full WGF 2.0 featureset in 2006.

Note also that games that take advantage of WGF 1.0 will run successfully on Vista atop existing hardware. However, once they make the transition to WGF 2.0 it's unclear if they'll be able to degrade gracefully to 1.0, meaning that the first "must have" game using WGF 2.0 is going to require not just Vista but a new video card: while there will be XP drivers for "DX10" hardware XP won't support WGF 2.0, and older cards running Vista will at most support WGF 1.0 and Basic WDDM drivers. So the first WGF 2.0 games will be looking at a fairly small market with the requisite hardware base, and thus there may be a bit of delay until such titles are released (fortunately the new hardware should offer enough of an improvement on existing titles that we won't see a chicken-and-egg situation develop).

New Features

- Fixed pipeline⁶⁰ is being done away with in favor of a fully programmable pipeline (often referred to as a unified pipeline architecture), which can be programmed to emulate the same.
- Paging of graphics memory, to allow data to be loaded to Video RAM when needed and move it out when not needed. This enables usage of the system memory to hold graphics data, such as textures, thereby allowing use of more and higher resolution textures in games.
- There is no limit on the number of objects which can be rendered, provided enough resources are available.⁶¹
- Virtualization of the graphics hardware, to allow multiple threads/processes to use it, in turns.
- New state object to enable the GPU to change states efficiently.
- Shader Model 4.0, enhances the programmability of the graphics pipeline. It adds instructions for integer and bitwise calculations.
- Geometry shaders, which work on individual triangles which form a mesh.
- Texture arrays enable swapping of textures in GPU without CPU intervention.
- Resource View enables pre-caching of resources, thereby reducing latency.
- Predicated Rendering allows drawing calls to be ignored based on some other conditions. This enables rapid occlusion culling, which prevents objects from being rendered if it is not visible or too far to be visible.

Related tools

DirectX comes with D3DX, a library of tools designed to perform common mathematical calculations and several more complicated tasks, such as compiling or assembling shaders used for 3D graphic programming. It also includes several classes that simplify the use of 3D-models and, for example, particle systems. D3DX is provided as a dynamic link library (DLL).

DXUT (also called the sample framework) is a layer built on top of the Direct3D API. The framework is designed to help the programmer spend less time with mundane tasks, such as creating a window, creating a device, processing Windows messages and handling device events.

⁶⁰ CNet News(http://news.com.com/An+inside+look+at+Windows+Vista+-+page+4/2100-1043_3-6051736-4.html?tag=st.num)

⁶¹ SDK March 2006(<http://www.ati.com/developer/radeonSDK.html?tag=Radeon>)

References

Source: <http://en.wikipedia.org/wiki/Direct3D>

Principal Authors: Frecklefoot, Soumyasch, Warrens, Wrendelgeth, Algumacoisaqq

Displacement mapping

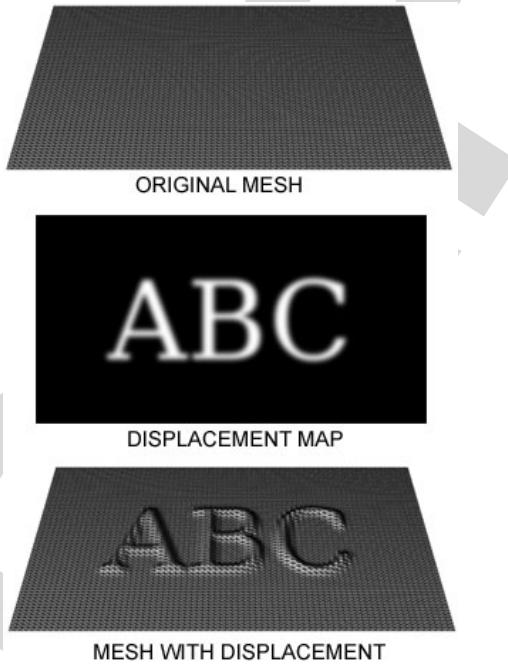


Figure 31 Displacement mapping

Displacement mapping is an alternative technique in contrast to bump mapping, normal mapping, and parallax mapping, using a heightmap to cause an effect where the actual geometric position of points over the textured surface are *displaced* along the surface normal according to the values stored into the texture.

For years, displacement mapping was a peculiarity of high-end rendering systems like RenderMan, while realtime Application Programming Interfaces, like

→OpenGL and DirectX, lacked this possibility. One of the reasons for this absence is that the original implementation of displacement mapping required an adaptive tessellation of the surface in order to obtain micropolygons whose size matched the size of a pixel on the screen.

With the newest generation of graphics hardware, displacement mapping can be interpreted as a kind of vertex-texture mapping, where the values of the texture map do not alter the pixel color, but instead change the position of the vertex. Unlike bump mapping and normal mapping, displacement mapping can in this way produce a genuine *rough* surface. It is currently implemented only in a few desktop graphics adapters, and it has to be used in conjunction with adaptive tessellation techniques (that increases the number of rendered polygons according current viewing settings) to produce highly detailed meshes, and to give a more 3D feel and a greater sense of depth and detail to textures to which displacement mapping is applied.

Further reading

- Photoshop CS Tutorial Displacement Mapping Effect⁶²
- Relief Texture Mapping⁶³ website
- *Real-Time Relief Mapping on Arbitrary Polygonal Surfaces*⁶⁴ paper
- *Relief Mapping of Non-Height-Field Surface Details*⁶⁵ paper

See also

- →Bump mapping
- →Normal mapping
- →Parallax mapping
- Demo effects
- Heightmap

Source: http://en.wikipedia.org/wiki/Displacement_mapping

Principal Authors: Tommstein, ALoopingIcon, Furrykef, Engwar, T-tus

⁶² <http://www.psdesignzone.com/photoshop-tutorials/displacement-mapping-in-photoshop.html>

⁶³ <http://www.inf.ufrgs.br/%7Eoliveira/RTM.html>

⁶⁴ http://www.inf.ufrgs.br/%7Eoliveira/pubs_files/Policarpo_Oliveira_Comba_RTRM_I3D_2005.pdf

⁶⁵ http://www.inf.ufrgs.br/%7Eoliveira/pubs_files/Policarpo_Oliveira_RTM_multilayer_I3D2006.pdf

Distance fog

Distance fog is a technique used in →3D computer graphics to enhance the perception of distance.

Because many of the shapes in graphical environments are relatively simple, and complex shadows are difficult to render, many graphics engines employ a "fog" gradient, so that objects further from the camera are progressively more obscured by haze. This technique works because of light scattering, which causes more distant objects to appear hazier to the eye, especially in outdoor environments.



Figure 32 Example of fog

Fogging is another use of distance fog in mid to late nineties games, where processing power was not enough to render far viewing distances, and clipping was employed. However, the effect could be very distracting, and by applying a medium-ranged fog, the clipped polygons would fade in more realistically from the haze, even though the effect may have been considered unrealistic in some cases. Many early Nintendo 64 games used this effect — most (in)famously in *Turok: Dinosaur Hunter* and *Superman 64*.

See also

- Computer graphics
- Virtual reality

Source: http://en.wikipedia.org/wiki/Distance_fog

Principal Authors: Graft, SimonP, Reedbeta, WolfenSilva, RJHall

Draw distance

Draw distance is a computer graphics term, defined as the distance in a 3 dimensional scene that is still drawn by the rendering engine. Polygons that lie behind the draw distance won't be drawn to the screen.

As the draw distance increases more polygons need to be drawn onto the screen which requires more computing power. This means the graphic quality and realism of the scene will increase as draw distance increases, but the overall performance (frames per second) will decrease. Many games and applications will allow users to manually set the draw distance to balance performance and visuals.

Older games had far shorter draw distances, most noticeable in vast, open scenes. Racing arcade games were particularly infamous, as the open highways and roads often led to "pop-up graphics" - an effect where distant objects suddenly appear without warning as the camera gets closer to it. This is a hallmark of poor draw distance, and still plagues large, open-ended games like the *Grand Theft Auto* series.

A common trick used in games to disguise a short draw distance is to obscure the area with a distance fog. Alternative methods have been developed to sidestep the problem altogether using level of detail manipulation. *Black & White* was one of the earlier games to use adaptive level of detail to decrease the number of polygons in objects as they moved away from the camera, allowing it to have a massive draw distance while maintaining detail in close-up views.

Grand Theft Auto 3 made particular use of fogging, however, this made the game less playable when driving or flying at high speed, as object would pop-up out of the fog and cause you to crash into them.

Source: http://en.wikipedia.org/wiki/Draw_distance

Principal Authors: Goncalopp, 25, Deepomega, Oswax, Xhin

Euler boolean operation

In constructive solid geometry, a **Euler boolean operation** is a series of modifications to solid modelling which preserves the Euler characteristic in the boundary representation at every stage. One or more of these Euler boolean operations is stored in a change state, so as to only represent models which are physically realizable.

Failing to maintain the Euler characteristic would result in geometric and topological entities often depicted by M. C. Escher. Esher's geometry artwork comes close to preserving the Euler characteristic (usually a problem with just the hole count).

Source: http://en.wikipedia.org/wiki/Euler_boolean_operation

Principal Authors: Wheger, Charles Matthews, Gaius Cornelius, Zzyzx11, RHall

Flat shading

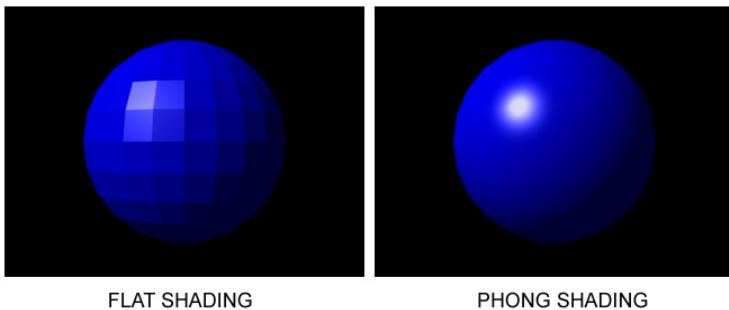


Figure 33 Flat shading interpolation example

Flat shading is a lighting technique used in →3D computer graphics. It shades each polygon of an object based on the angle between the polygon's surface normal and the direction of the light source, their respective colors and the intensity of the light source. It is usually used for high speed rendering where more advanced shading techniques are too computationally expensive.

The disadvantage of flat shading is that it gives low-polygon models a faceted look. Sometimes this look can be advantageous though, such as in modeling

boxy objects. Artists sometimes use flat shading to look at the polygons of a solid model they are creating. More advanced and realistic lighting and shading techniques include →Gouraud shading and →Phong shading.

See also: computer graphics

Source: http://en.wikipedia.org/wiki/Flat_shading

Principal Authors: RJFJR, Mrwojo, AndreasB, Poccil, Whitepaw, D3

Forward kinematic animation

Forward kinematic animation is a method in →3D computer graphics for animating models.

The essential concept of forward kinematic animation is that the positions of particular parts of the model at a specified time are calculated from the position and orientation of the object, together with any information on the joints of an articulated model. So for example if the object to be animated is an arm with the shoulder remaining at a fixed location, the location of the tip of the thumb would be calculated from the angles of the shoulder, elbow, wrist, thumb and knuckle joints. Three of these joints (the shoulder, wrist and the base of the thumb) have more than one degree of freedom, all of which must be taken into account. If the model were an entire human figure, then the location of the shoulder would also have to be calculated from other properties of the model.

Forward kinematic animation can be distinguished from inverse kinematic animation by this means of calculation - in inverse kinematics the orientation of articulated parts is calculated from the desired position of certain points on the model. It is also distinguished from other animation systems by the fact that the motion of the model is defined directly by the animator - no account is taken of any physical laws that might be in effect on the model, such as gravity or collision with other models.

Source: http://en.wikipedia.org/wiki/Forward_kinematic_animation

Principal Authors: Onebyone, Jiang, TimBentley, Charles Matthews, Bryan Derksen

Fragment (computer graphics)

A **fragment** is a computer graphics term for all of the data necessary needed to generate a pixel in the frame buffer. This may include

- raster position
- depth
- interpolated attributes (color, texture coordinates , etc.)
- stencil
- alpha
- window ID
- etc.

It can be thought of as the data needed to shade the pixel, plus the data needed to test whether the fragment survives to become a pixel (depth, alpha, stencil, scissor, window ID, etc.)

See also

- →Graphics pipeline

Source: http://en.wikipedia.org/wiki/Fragment_%28computer_graphics%29

Gelato (software)

Gelato	
Maintainer:	NVIDIA Corporation
Latest release:	2.0R4 / May 2006
OS:	Windows XP, Linux (Red Hat or SUSE)
Use:	→3D computer graphics
License:	Proprietary
Website:	NVIDIA's Gelato site ⁶⁶

Gelato is hardware-accelerated non-real-time renderer created by graphics card manufacturer NVIDIA originally for use with its Quadro FX GPU, although a Quadro class GPU is not a requirement any longer as it now also supports GeForce cards. It was designed to produce film-quality images. Gelato uses a

shading language very similar to the RenderMan shading language. After BM-RT resolved their issues with Pixar, NVIDIA quickly acquires the brain behind the moon and not long after, released Gelato in April 2004.

With the release of Gelato 2.0, in push to popularize GPU accelerated rendering (as opposed to the traditional CPU rendering), NVIDIA released a free version of Gelato for users of personal computers with locked out advanced features for production rendering that your average joe render monkeys would rarely use.

Gelato Pro

at \$1500 per render node, Gelato pro is relatively cheaper than other renderer's such as Pixar's Renderman Pro Server.

Extra Gelato Pro features are:

- NVIDIA® Sorbetto™ interactive relighting technology
- DSO shadeops
- Network parallel rendering
- Multithreading
- Native 64-bit support
- Maintenance and support from NVIDIA's High-Quality-Rendering Team

External links

- Official website⁶⁷
- A Gelato review⁶⁸

Source: http://en.wikipedia.org/wiki/Gelato_%28software%29

Principal Authors: Flamurai, T-tus, -Ril-, Marudubshinki, Asparagus

⁶⁷ <http://film.nvidia.com/page/gelato.html>

⁶⁸ <http://deathfall.com/feature.php?op=showcontent&id=36>

Geometric model

A **geometric model** describes the shape of a physical or mathematical object by means of geometric concepts. **Geometric model(I)ing** is the construction or use of geometric models. Geometric models are used in computer graphics, computer-aided design and manufacturing, and many applied fields such as medical image processing.

Geometric models can be built for objects of any dimension in any geometric space. Both 2D and 3D geometric models are extensively used in computer graphics. 2D models are important in computer typography and technical drawing. 3D models are central to computer-aided design and manufacturing, and many applied technical fields such as geology and medical image processing.

Geometric models are usually distinguished from procedural and object-oriented models, which define the shape implicitly by an algorithm. They are also contrasted with digital images and volumetric models; and with implicit mathematical models such as the zero set of an arbitrary polynomial. However, the distinction is often blurred: for instance, geometric shapes can be represented by objects; a digital image can be interpreted as a collection of colored squares; and geometric shapes such as circles are defined by implicit mathematical equations. Also, the modeling of fractal objects often requires a combination of geometric and procedural techniques.

See also

- Computational geometry

Source: http://en.wikipedia.org/wiki/Geometric_model

Principal Authors: Oleg Alexandrov, Mdd, Jorge Stolfi, Kbdank71, Patrick

Geometry pipelines

Geometry Pipelines, also called Geometry Engines (GE) are the first stage in a classical Graphics Pipeline, such as the Reality Engine. They do the transformation from 3D coordinates used to specify the geometry to a unified coordinate system used by the Raster Manager (RM) to rasterize the geometry into frame-buffer pixels. The Display Generator (DG) scans these pixels into a video signal understood by a monitor. In →OpenGL, this transformation is defined by the Modelview Matrix and the Projection Matrix. Typically, the modelview matrix defines the transformation of the incoming vertices into world coordinates, a coordinate system used for all vertices. The projection matrix defines how this 3-dimensional coordinate space is projected to the Viewport. In addition to this transformation, the GEs compute the vertex colors based on the light settings, may perform texture coordinate generation as well as clipping of the geometry. The Geforce graphics cards from nVidia introduced these functionalities for the first time in the consumer market, labelled as hardware-based Transform and Lighting (T&L).

See also

- Computer graphics
- James H. Clark
- Silicon Graphics, Inc.

Source: http://en.wikipedia.org/wiki/Geometry_pipelines

Principal Authors: Jpbowen, Bumm13, JoJan, Joyous!

Geometry Processing

Geometry Processing is a fast-growing area of research that uses concepts from applied mathematics, computer science, and engineering to design efficient algorithms for the acquisition, reconstruction, analysis, manipulation, simulation and transmission of complex 3D models. Applications of geometry processing algorithms already cover a wide range of areas from multimedia, entertainment, and classical computer-aided design, to biomedical computing, reverse engineering, and scientific computing.

See also

- Computer-aided design (CAD)

External links

- " Digital Geometry Processing⁶⁹", by Peter Schroder and Wim Sweldens
- Symposium on Geometry Processing⁷⁰
- Multi-Res Modeling Group⁷¹, Caltech
- Computer Graphics & Multimedia Department⁷², RWTH-Aachen University

Source: http://en.wikipedia.org/wiki/Geometry_Processing

Principal Authors: Betamod, RJHall

GLEE

The OpenGL Easy Extension library (GLEE) automatically links →OpenGL extensions and core functions at initialisation time. This saves programmers the effort of manually linking every required extension, and effectively brings the OpenGL library up to date.

GLEe is compatible with Windows, Linux and FreeBSD platforms. It is also likely to be compatible with other unix-like systems which use X windows.

External links

Source: <http://en.wikipedia.org/wiki/GLEE>

⁶⁹ <http://www.multires.caltech.edu/pubs/DGPCourse/DGP.pdf>

⁷⁰ <http://www.geometryprocessing.org/>

⁷¹ <http://www.multires.caltech.edu/>

⁷² <http://www-i8.informatik.rwth-aachen.de/index.html>

GLEW

The **OpenGL Extension Wrangler Library (GLEW)** is a cross-platform C/C++ library that helps in querying and loading →OpenGL extensions. GLEW provides efficient run-time mechanisms for determining which OpenGL extensions are supported on the target platform. All OpenGL extensions are exposed in a single header file, which is machine-generated from the official extension list. GLEW is available for a variety of operating systems, including Windows, Linux, Mac OS X, FreeBSD, Irix, and Solaris.

External links

GLEW English surname meaning wise. Believed to be from the wiseman of the tribe.

Source: <http://en.wikipedia.org/wiki/GLEW>

Glide API

Glide was a proprietary 3D graphics API developed by 3dfx used on their Voodoo graphics cards. It was dedicated to gaming performance, supporting geometry and texture mapping primarily, in data formats identical to those used internally in their cards. The Voodoo cards were the first to offer performance to really make 3D games work well, and Glide became fairly widely used as a result. The introduction of DirectX and full OpenGL implementations from other vendors eventually doomed Glide, and 3dfx along with it.

Glide is based on the basic geometry and "world view" of →OpenGL. OpenGL is a very large library with about 250 calls in the API, many of which are of limited use. Glide was an effort to select those features that were truly useful for gaming, leaving the rest out. The result was an API that was small enough to be implemented entirely in hardware. That hardware, of course, was 3dfx's own Voodoo cards. This led to several odd limitations in Glide – for instance, it only supported 16-bit color.

The combination of the Voodoo's raw performance and Glide's easy-to-use API resulted in Voodoo cards generally dominating the gaming market from the mid to late 1990s. The name Glide was chosen to be indicative of the GL underpinnings, while being different enough to avoid trademark problems. 3dfx also supported a low-level MiniGL driver, making their cards particularly popular for players of the various Quake-derived games. MiniGL was essentially a

"different Glide" with a wider selection of OpenGL calls and lacking the dedication to a single hardware platform. Due to the Voodoo's "GL-like" hardware, MiniGL on Voodoo was very "thin" and ran almost as well as Glide.

As new cards entered the market 3dfx managed to hold the performance crown for a short time, based largely on the tight integration between Glide and their hardware. This allowed them to be somewhat lax in hardware terms, which was important as the small gamer-only market 3dfx sold into wasn't large enough to support a large development effort. It was not long before offerings from nVidia and ATI Technologies were able to outperform the latest Voodoo cards using standard APIs. 3dfx responded by releasing Glide as an open source API, but it was too late. By late 1999 they announced that almost all games had moved to →Direct3D, and to a lesser extent, OpenGL.

Today old Glide supporting games can often be run on modern graphics cards with help of a Glide wrapper. Numerous such computer programs exist.

External links

- GLIDE programming manual⁷³
- Glide Wrappers⁷⁴ at VoodooFiles

Source: http://en.wikipedia.org/wiki/Glide_API

Principal Authors: Maury Markowitz, Lproven, Doug Bell, The Anome, GreatWhiteNortherner

Global illumination

Global illumination algorithms used in →3D computer graphics are those which, when determining the light falling on a surface, take into account not only the light which has taken a path directly from a light source (*direct illumination*), but also light which has undergone reflection from other surfaces in the world (*indirect illumination*).

Images rendered using global illumination algorithms are more photorealistic than images rendered using local illumination algorithms. However, they are also much slower and more computationally expensive. A common approach is to compute the global illumination of a scene and store that information with

⁷³ <http://www.gamers.org/dEngine/xf3D/glide/glidepgm.htm>

⁷⁴ http://www.voodoofiles.com/type.asp?cat_id=14

the geometry (ie. 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.

→Radiosity, ray tracing, beam tracing, cone tracing, →Path Tracing, metropolis light transport and photon mapping are examples of algorithms used in global illumination, some of which may be used together.

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 too which makes them more accurate algorithms to solve the lighting equation and provide a more realistic globally 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, global illumination is sometimes approximated by an "ambient" term in the lighting equation.

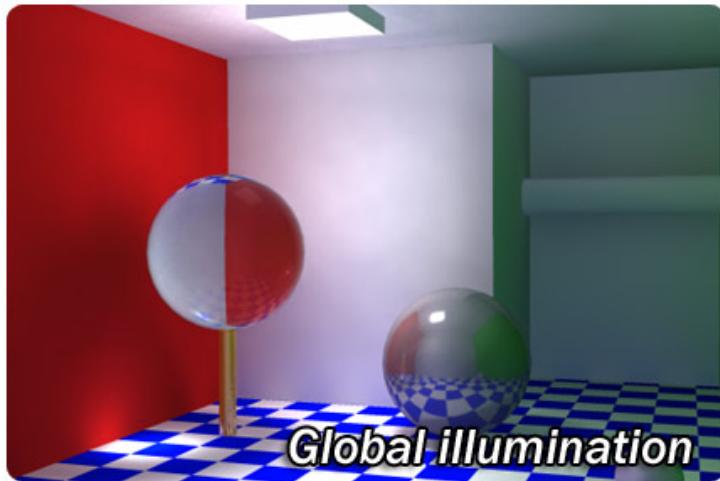


Figure 34 An example of a global illumination rendering, demonstrating how surface materials are reflected in other surfaces.

See also

- Povray free rendering software featuring global illumination
- YafRay free rendering software featuring global illumination
- Radiance - highly accurate ray-tracing software system for UNIX computers. Free to non-commercial users. It has been open-sourced.

External links

- PBRT⁷⁵ - Literate programming, has a great accompanying book.
- Vraywiki⁷⁶ - Vraywiki website.
- SplutterFish⁷⁷ - developers of Brazil, a rendering system based on global illumination. The site has an extensive gallery of contributed images
- Perceptuum⁷⁸ - good source for photon mapping, and other global illumination techniques.
- Mental Images⁷⁹ - makers of the MentalRay renderer. The renderer is used in packages such as Softimage XSI, Maya and 3D Studio Max

Source: http://en.wikipedia.org/wiki/Global_illumination

Principal Authors: Dormant25, Reedbeta, Paranoid, Heron, Nohat, Jsnow, Peter bertok, Arru, RJHall, Jose Ramos

GLSL

GLSL - OpenGL Shading Language also known as **GLslang** is a high level shading language based on the C programming language. It was created by the OpenGL ARB for programming graphics processing units directly, thus giving developers control of the graphics pipeline at the lowest level.

Background

With the recent advancements in graphics cards, new features have been added to allow for increased flexibility in the rendering pipeline at the vertex and fragment level. Programmability at this level is achieved with the use of fragment and vertex shaders.

Originally, this functionality was achieved through the use of shaders written in assembly language. Assembly language is non-intuitive and rather complex for developers to use. The OpenGL ARB created the OpenGL Shading Language to provide a more intuitive method for programming the graphics processing unit while maintaining the open standards advantage that has driven OpenGL throughout its history.

⁷⁵ <http://pbrt.org/>

⁷⁶ <http://www.vraywiki.com/>

⁷⁷ <http://www.splutterfish.com/>

⁷⁸ <http://www.hxa7241.org/perceptuum/perceptuum.html>

⁷⁹ <http://www.mentalimages.com/>

Originally introduced as an extension to OpenGL 1.5, the OpenGL ARB formally included GLSL into the →OpenGL 2.0 core. →OpenGL 2.0 is the first major revision to →OpenGL since the creation of →OpenGL 1.0 in 1992.

Some benefits of using GLSL are:

- Cross platform compatibility on multiple operating systems, including Windows and Linux.
- The ability to write shaders that can be used on any hardware vendor's graphics card that supports the OpenGL Shading Language.
- Each hardware vendor includes the GLSL compiler in their driver, thus allowing each vendor to create code optimized for their particular graphics card's architecture.

Details

Data types

The OpenGL Shading Language Specification defines 22 basic data types, some are the same as used in the C programming language, while others are specific to graphics processing.

- void – used for functions that do not return a value
- bool – conditional type, values may be either true or false
- int – a signed integer
- float – a floating point number
- vec2 – a 2 component floating point vector
- vec3 – a 3 component floating point vector
- vec4 – a 4 component floating point vector
- bvec2 – a 2 component Boolean vector
- bvec3 – a 3 component Boolean vector
- bvec4 – a 4 component Boolean vector
- ivec2 – a 2 component vector of integers
- ivec3 – a 3 component vector of integers
- ivec4 – a 4 component vector of integers
- mat2 – a 2X2 matrix of floating point numbers
- mat3 – a 3X3 matrix of floating point numbers
- mat4 – a 4X4 matrix of floating point numbers
- sampler1D – a handle for accessing a texture with 1 dimension
- sampler2D – a handle for accessing a texture with 2 dimensions
- sampler3D – a handle for accessing a texture with 3 dimensions
- samplerCube – a handle for accessing cube mapped textures

- sampler1Dshadow – a handle for accessing a depth texture in one dimension
- sampler2Dshadow – a handle for accessing a depth texture in two dimensions

Operators

The OpenGL Shading Language provides many operators familiar to those with a background in using the C programming language. This gives shader developers flexibility when writing shaders. GLSL contains the operators in C and C++, with the exception of bitwise operators and pointers...

Functions and control structures

Similar to the C programming language, GLSL supports loops and branching, including if, else, if/else, for, do-while, break, continue, etc.

User defined functions are supported, and a wide variety of commonly used functions are provided built-in as well. This allows the graphics card manufacturer the ability to optimize these built in functions at the hardware level if they are inclined to do so. Many of these functions are similar to those found in the C programming language such as exp() and abs() while others are specific to graphics programming such as smoothstep() and texture2D().

Compilation and Execution

GLSL shaders are not stand-alone applications; they require an application that utilizes the →OpenGL API. C, C++, C#, Delphi and Java all support the →OpenGL API and have support for the OpenGL Shading Language.

GLSL shaders themselves are simply a set of strings that are passed to the hardware vendor's driver for compilation from within an application using the OpenGL API's entry points. Shaders can be created on the fly from within an application or read in as text files, but must be sent to the driver in the form of a string.

A sample trivial GLSL Vertex Shader

```
void main(void)
{
    gl_Position = ftransform();
}
```

A sample trivial GLSL Fragment Shader

```
void main(void)
{
    gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);
}
```

References

- Rost, Randi. *OpenGL Shading Language*. 1st ed. Pearson Education, Inc, 2004. ISBN 0321197895
- Kessenich, John, & Baldwin, David, & Rost, Randi. *The OpenGL Shading Language*. Version 1.10.59. 3Dlabs, Inc. Ltd. <http://developer.3dlabs.com/documents/index.htm>

See also

- Shader Languages
- Computer programming
- Computer graphics
- →OpenGL
- Shaders

External links

- The Official OpenGL Web Site⁸⁰
- GLSL Resources and Documentation⁸¹
- Tutorials and Examples from Lighthouse3D⁸²
- Tutorials and Examples from Nehe⁸³
- A GLSL Development Environment⁸⁴
- RenderMonkey Shader Development Environment⁸⁵

Source: <http://en.wikipedia.org/wiki/GLSL>

Principal Authors: Csl77, Zemyla, Rufous, Enochlau, Fritz Saalfeld

⁸⁰ <http://www.opengl.org>

⁸¹ <http://developer.3dlabs.com/>

⁸² <http://www.lighthouse3d.com/opengl/>

⁸³ <http://nehe.gamedev.net>

⁸⁴ <http://www.typhoonlabs.com>

⁸⁵ <http://www.ati.com/developer/rendermonkey/>

GLU

GLU is the →OpenGL Utility Library.

It consists of a number of functions that use the base OpenGL library to provide higher-level drawing routines from the more primitive routines that OpenGL provides. It is usually distributed with the base OpenGL package.

Among these features are mapping between screen- and world-coordinates, generation of texture mipmaps, drawing of quadric surfaces, NURBS, tessellation of polygonal primitives, interpretation of OpenGL error codes, an extended range of transformation routines for setting up viewing volumes and simple positioning of the camera, generally in more human-friendly terms than the routines presented by OpenGL. It also provides additional primitives for use in OpenGL applications, including spheres, cylinders and disks.

GLU functions can be easily recognized by looking at them because they all have glu as a prefix. An example function is gluOrtho2D() which defines a two dimensional orthographic projection matrix.

Specifications for GLU are available at the OpenGL specification page⁸⁶

See also

- GLUT
- →GLUI

Source: <http://en.wikipedia.org/wiki/GLU>

Principal Authors: LesmanaZimmer, Elie De Brauwer, Tipiac, Segv11, Joseph Myers

⁸⁶ <http://www.opengl.org/documentation/specs/>

GLUI

GLUI is a GLUT-based C++ user interface library which provides controls such as buttons, checkboxes, radio buttons, and spinners to OpenGL applications. It is window- and operating-system independent, relying on GLUT to handle all system-dependent issues, such as window and mouse management.

It lacks the features of a more full-fledged GUI toolkit such as QT, wxWindows, or FLTK, but it has a very small footprint and is extremely easy to use. A research or demonstration program using only GLUT can be modified in just a couple of hours, by someone new to the toolkit, to have a useful control panel. Weaknesses include the lack of a file chooser (perhaps the most frustrating omission) and mediocre responsiveness.

It was first written by Paul Rademacher to help him with his academic work. It is now maintained by Nigel Stewart.

See Also

- →GLU
- GLUT

External links

- Main Site (download)⁸⁷
- GLUI SourceForge Project⁸⁸

Source: <http://en.wikipedia.org/wiki/GLUI>

Principal Authors: Nigosh, Inike, Niteowlneils, Mark Foskey, LesmanaZimmer

⁸⁷ <http://glui.sourceforge.net/>

⁸⁸ <http://sourceforge.net/projects/glui/>

Gouraud shading

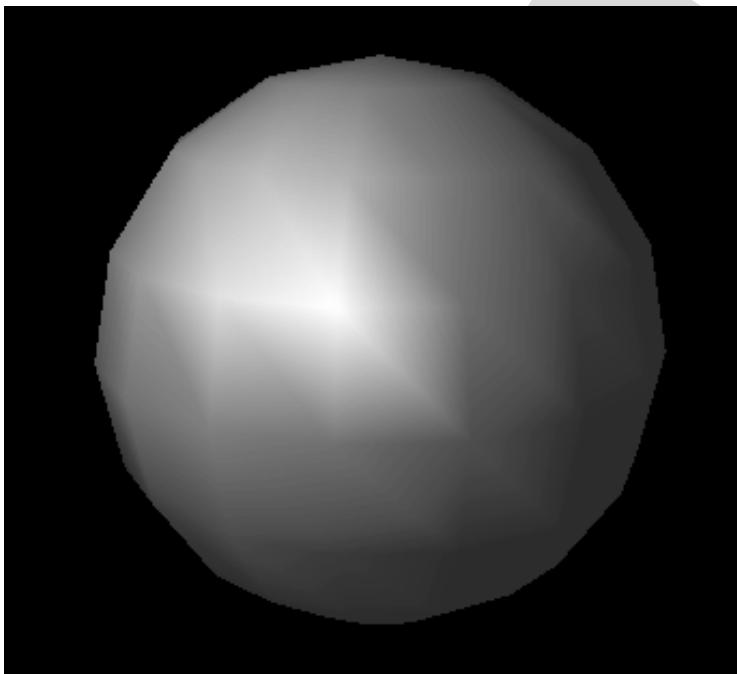


Figure 35 Gouraud shaded sphere - note the inaccuracies towards the edges of the polygons.

Gouraud shading is a method used in computer graphics to simulate the differing effects of light and color across the surface of an object. In practice, Gouraud shading is used to achieve smooth lighting on low-polygon surfaces without the heavy computational requirements of calculating lighting for each pixel. The technique was first presented by Henri Gouraud in 1971.

The basic principle behind the method is to calculate the surface normals at the vertices of polygons in a 3D computer model. These normals are then averaged for all the polygons that meet at each point. Lighting computations are then performed to produce color intensities at vertices. The lighting calculation used by Gouraud was based on the Lambertian diffuse lighting model.

These color values are then interpolated along the edges of the polygons. To complete the shading, the image is filled by lines drawn across the image that interpolate between the previously calculated edge intensities.

Gouraud shading

Gouraud shading is much less processor intensive than →Phong shading, but does not calculate all desirable lighting effects as accurately. For instance, the white shiny spot on the surface of an apple (called a specular highlight) is highly dependent on the normal within that spot. If a model's vertices are not within that spot, their colors are blended across it, making it disappear altogether. This problem is made more obvious when the light source is relocated, moving the highlight across a vertex. Using Gouraud shading, the specular highlight will appear mysteriously and grow in intensity as the light moves toward a position of reflection from the observer across the vertex. The desired result would be to see the highlight move smoothly rather than fade out and in between vertices.

Despite the drawbacks, Gouraud shading is much superior to flat shading which requires significantly less processing than Gouraud, but gives low-polygon models a sharp, faceted look.

Original publications

- H. Gouraud, "Continuous shading of curved surfaces," *IEEE Transactions on Computers*, 20(6):623–628, 1971.
- H. Gouraud, *Computer Display of Curved Surfaces*, Doctoral Thesis, University of Utah, USA, 1971.
- H. Gouraud, *Continuous shading of curved surfaces*. In Rosalee Wolfe (editor), *Seminal Graphics: Pioneering efforts that shaped the field*⁸⁹, ACM Press, 1998. ISBN 1-58113-052-X.

See also

- →Blinn–Phong shading model

Source: http://en.wikipedia.org/wiki/Gouraud_shading

Principal Authors: Jpbowen, Blueshade, Kocio, Michael Hardy, Mrwojo, Jaxl, Zundark, Poccil, The Anome

⁸⁹ <http://www.siggraph.org/publications/seminal-graphics.html>

Graphics pipeline

In 3D computer graphics, the terms **graphics pipeline** or **rendering pipeline** most commonly refer to the current state of the art method of rasterization-based rendering as supported by commodity graphics hardware. The graphics pipeline typically accepts some representation of a 3D scene as an input and results in a 2D raster image as output.

Stages of the graphics pipeline:

- Modeling transformation
- Lighting
- Viewing transformation
- Projection transformation
- Clipping
- Scan conversion or rasterization
- Texturing or shading
- Display

Modeling transformation

In this stage the 3D geometry provided as input is established in what is known as 3D world-space—a conceptual orientation and arrangement in 3D space. This could include transformations on the local object-space of geometric primitives such as translation and rotation.

Lighting

Geometry in the complete 3D scene is lit according to the defined locations of light sources and reflectance and other surface properties. Current hardware implementations of the graphics pipeline compute lighting only at the vertices of the polygons being rendered. The lighting values between vertices are then interpolated during rasterization. Per-pixel lighting can be done on modern graphics hardware as a post-rasterization process by means of a fragment shader program.

Viewing transformation

Objects are transformed from 3D world-space coordinates into a 3D coordinate system based on the position and orientation of a virtual camera. This results in the original 3D scene as seen from the camera's point of view, defined in what it called eye-space or camera-space.

Projection transformation

In this stage of the graphics pipeline, geometry is transformed from the eye-space of the rendering camera into 2D image-space, mapping the 3D scene onto a plane as seen from the virtual camera.

Clipping

For more details on this topic, see Clipping (computer graphics).

Geometric primitives that now fall outside of the viewing frustum will not be visible and are discarded at this stage. Clipping is not necessary to achieve a correct image output, but it accelerates the rendering process by eliminating the unneeded rasterization and post-processing on primitives that will not appear anyway.

Scan conversion or rasterization

Rasterization is the process by which the 2D image-space representation of the scene is converted into raster format and the correct resulting pixel values are determined.

Texturing or shading

At this stage of the pipeline individual fragments (or pre-pixels) are assigned a color based on values interpolated from the vertices during rasterization or from a texture in memory.

Display

The final colored pixels can then be displayed on a computer monitor or other display.

The Graphics Pipeline in Hardware

The rendering pipeline is mapped onto current graphics acceleration hardware such that the input to the graphics card (GPU) is in the form of vertices. These vertices then undergo transformation and per-vertex lighting. At this point in modern GPU pipelines a custom vertex shader program can be used to manipulate the 3D vertices prior to rasterization. Once transformed and lit, the vertices undergo clipping and rasterization resulting in fragments. A second custom shader program can then be run on each fragment before the final pixel values are output to the frame buffer for display.

The graphics pipeline is well suited to the rendering process because it allows the GPU to function as a stream processor since all vertices and fragments

can be thought of as independent. This allows all stages of the pipeline to be used simultaneously for different vertices or fragments as they work their way through the pipe. In addition to pipelining vertices and fragments, their independence allows graphics processors to use parallel processing units to process multiple vertices or fragments in a single stage of the pipeline at the same time.

References

1. ↑ Graphics pipeline. (n.d.). Computer Desktop Encyclopedia. Retrieved December 13, 2005, from Answers.com Web site: ⁹⁰
2. ↑ Raster Graphics and Color⁹¹ 2004 by Greg Humphreys at the University of Virginia

See also

- Rendering
- Graphics Processing Unit
- Stream Processor
- →Shader
- NVIDIA
- ATI

External links

- MIT OpenCourseWare Computer Graphics, Fall 2003⁹²
- Answers.com overview of the graphics pipeline⁹³
- OpenGL⁹⁴
- DirectX⁹⁵

Source: http://en.wikipedia.org/wiki/Graphics_pipeline

Principal Authors: Flamurai, Seasage, Sfingram, Piotrus, Hellisp

⁹⁰ <http://www.answers.com/topic/graphics-pipeline>

⁹¹ <http://www.cs.virginia.edu/~gfx/Courses/2004/Intro.Fall.04/handouts/01-raster.pdf>

⁹² <http://ocw.mit.edu/OcwWeb/Electrical-Engineering-and-Computer-Science/6-837Fall2003/CourseHome/index.htm>

⁹³ <http://www.answers.com/topic/graphics-pipeline>

⁹⁴ <http://www.opengl.org/about/overview.html>

⁹⁵ <http://www.microsoft.com/windows/directx/default.aspx>

Hidden line removal

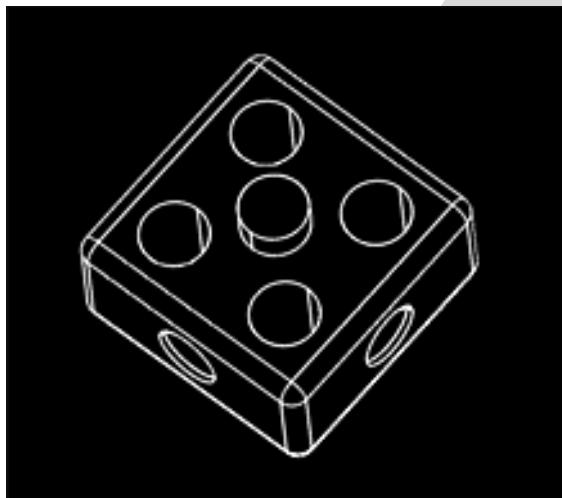


Figure 36 Line removal technique in action

Hidden line removal is an extension of wireframe rendering where lines (or segments of lines) covered by surfaces are not drawn.

This is not the same as hidden face removal since this involves depth and occlusion while the other involves normals.

A commonly used algorithm to implement it is Arthur Appel's algorithm (developed at IBM in the late 1960's). This algorithm works by propagating the visibility from a segment with a known visibility to a segment whose visibility is yet to be determined. Certain pathological cases exist in making this algorithm difficult to implement, those cases are (i) vertices on edges and (ii) edges on vertices and (iii) edges on edges. This algorithm is unstable because an error in visibility will be propagated to subsequent nodes (although there are ways to compensate for this problem). James Blinn published a paper on this problem.

External links

- Patrick-Gilles Maillot's Thesis⁹⁶ an extension of the Bresenham line drawing algorithm to perform 3D hidden lines removal; also published in MICAD

⁹⁶ <http://www.chez.com/pmaillot>

'87 proceedings on CAD/CAM and Computer Graphics, page 591 - ISBN 2-86601-084-1.

- Vector Hidden Line Removal⁹⁷ An article by Walter Heger with a further description (of the pathological cases) and more citations.

Source: http://en.wikipedia.org/wiki/Hidden_line_removal

Principal Authors: MrMambo, Grutness, Qz, RJHall, Oleg Alexandrov

Hidden surface determination

In →3D computer graphics, **hidden surface determination** is the process used to determine which surfaces and parts of surfaces are not visible from a certain viewpoint. A hidden surface determination algorithm is a solution to the **visibility problem**, which was one of the first major problems in the field of 3D computer graphics. The process of hidden surface determination is sometimes called **hiding**, and such an algorithm is sometimes called a **hider**. The analogue for line rendering is hidden line removal.

Hidden surface determination is necessary to render an image correctly, as parts of the image that are not visible should not be drawn. It also speeds up rendering since objects that aren't visible can be removed from the graphics pipeline.

There are many techniques for hidden surface determination, and the core differences between most rendering algorithms is how they handle this problem. There are also different stages of hidden surface determination. These stages include:

Backface culling

Since meshes are hollow shells, not solid objects, the back side of some faces, or polygons, in the mesh will never face the camera. Typically, there is no reason to draw such faces. This is responsible for the effect often seen in computer and video games in which, if the camera happens to be inside a mesh, rather than seeing the "inside" surfaces of the mesh, it disappears completely (all faces are seen from behind, and are culled).

Viewing frustum culling

⁹⁷ <http://wheger.tripod.com/vhl/vhl.htm>

The viewing frustum is a geometric representation of the volume visible to the virtual camera. Naturally, objects outside this volume will not be visible in the final image, so they are discarded. Often, objects lie on the boundary of the viewing frustum. These objects are cut into pieces along this boundary in a process called clipping, and the pieces that lie outside the frustum are discarded.

Occlusion culling

Occlusion culling is the process of determining which portions of objects are hidden by other objects from a given viewpoint. This is one of the fundamental problems in computer graphics, and many different occlusion culling algorithms have been developed. The simplest is painter's algorithm, in which polygons are sorted, then drawn back to front. The most common in real-time computer graphics is z-buffering, in which the depth value at each pixel is stored as each polygon is rendered. The pixel is only overwritten if the depth value of the current point is less than the depth value stored in the z-buffer. Both of these methods operate on polygon meshes.

Contribution culling

Often, objects are so far away that they do not contribute significantly to the final image. These objects are thrown away if their screen projection is too small.

Though hidden surface determination is most often used to determine what is visible in the final image, it is also has other applications, such as determining which parts of objects are in shadow.

Visible surface determination

Sometimes the majority of surfaces is invisible, so why ever touch them? This approach is opposite to **hidden surface removal** and used in:

- Ray tracer : →Ray tracing, which can also operate on parametric geometry, attempts to model the path of light rays into a viewpoint by tracing rays from the viewpoint into the scene. The first object the ray intersects is rendered, as it naturally is the object visible to the camera. Additional data structures used to solve this sorting problem include bsp trees and octrees.
- Portal rendering
- some Heightfield renderers work this way. Google Earth certainly does not process the whole earth for every frame.

Source: http://en.wikipedia.org/wiki/Hidden_surface_determination

Principal Authors: Flamurai, Fredrik, B4hand, Connelly, Arnero

High dynamic range imaging



Figure 37 Tone Mapped HDRI example showing stained glass windows in south alcove of Old Saint Paul's, Wellington, New Zealand.

In computer graphics and cinematography, **high dynamic range imaging** (HDRI for short) is a set of techniques that allow a far greater dynamic range of exposures than normal digital imaging techniques. The intention is to accurately represent the wide range of intensity levels found in real scenes, ranging from direct sunlight to the deepest shadows.

This provides the opportunity to shoot a scene and have total control of the final imaging from the beginning to the end of the photography project. An example would be that it provides the possibility to re-expose. One can capture as wide a range of information as possible on location and choose what is wanted later.

Gregory Ward is widely considered to be the founder of the file format for high dynamic range imaging. The use of high dynamic range imaging in computer graphics has been pioneered by Paul Debevec. He is considered to be the first

person to create computer graphic images using HDRI maps to realistically light and animate CG objects.

When preparing for display, a high dynamic range image is often tone mapped and combined with several full screen effects.

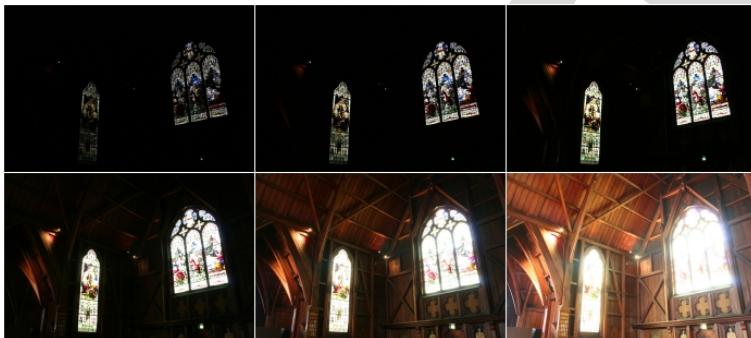


Figure 38 The six individual exposures used to create the previous HDRI. Exposure times from top left are: 1/40sec, 1/10sec, 1/2sec, 1sec, 6sec, 25sec

Difference between high dynamic range and traditional digital images

Information stored in high dynamic range (HDR) images usually corresponds to the physical values of luminance or radiance that can be observed in the real world. This is different from traditional digital images, which represent colors that should appear on a monitor or a paper print. Therefore HDR image formats are often called scene-referred, in contrast to traditional digital images, which are device-referred or output-referred. Furthermore, traditional images are usually encoded for the human visual system (maximizing the visual information stored in the fixed number of bits), which is usually called gamma encoding or gamma correction. The values stored for HDR images are linear, which means that they represent relative or absolute values of radiance or luminance (gamma 1.0).

HDR images require a higher number of bits per color channel than traditional images, both because of the linear encoding and because they need to represent values from 10^{-4} to 10^8 (the range of visible luminance values) or more. 16-bit ("half precision") or 32-bit floating point numbers are often used to represent HDR pixels. However, when the appropriate transfer function is used, HDR pixels for some applications can be represented with as few as 10-12 bits for luminance and 8 bits for chrominance without introducing any visible quantization artifacts ^{98 99}.

Example HDR Images

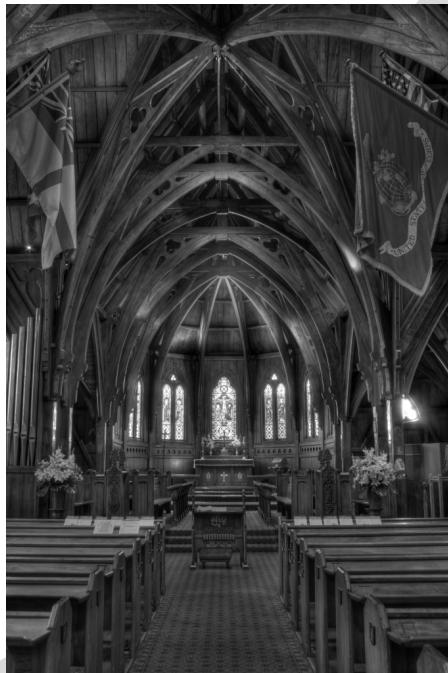


Figure 39 Nave in Old Saint Paul's, Wellington, New Zealand. Eight exposures ranging from 1/20th of a second to 30 seconds

See also

- →High dynamic range rendering (rendering virtual scenes using high dynamic range lighting calculation, notably in computer games)
- OpenEXR
- Radiance file format
- Logluv TIFF file format
- CinePaint image editor
- Pixel image editor
- Panoscan

⁹⁸ http://www.anywhere.com/gward/hdrenc/hdr_encodings.html

⁹⁹ <http://www.mpi-sb.mpg.de/resources/hdrvvideo/>

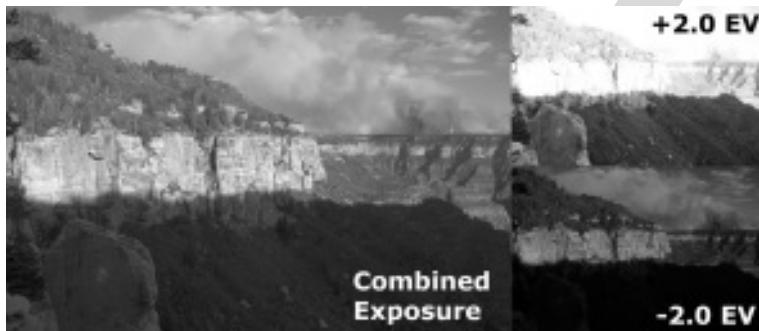


Figure 40 This scene of the Grand Canyon from the North Rim is an example of how two different exposures (4 stops difference) can be combined and the dynamic range subsequently compressed to make the image viewable on a standard display

External links

- Photomatix¹⁰⁰ HDRI Composition and Tone Mapping software
- Real Time High Dynamic Range Image Based Lighting Demo¹⁰¹
- High Dynamic Range Image Encodings¹⁰² by Greg Ward, Anyhere Software
- High Dynamic Range images under Linux¹⁰³ by Nathan Willis
- Hyperfocal Design¹⁰⁴ Commercial HDRIs, tutorials, software reviews, news and
- HDR Image and Video Processing from acquisition to display¹⁰⁵
- AHDRIA¹⁰⁶ Software for capturing HDRI with standard digital cameras.
- practical description of fully automatic Gradient Domain High Dynamic Range Compression¹⁰⁷
- HDRShop¹⁰⁸ High Dynamic Range Imaging Processing and Manipulation Software Resources
- HDRIE¹⁰⁹ (High Dynamic Range Image Editor) - an open-source project inspired by HDRShop.
- HDRI rendering in Renderman¹¹⁰
- PFSTools¹¹¹ - open-source package for creating and editing HDR images

¹⁰⁰ <http://www.hdrsoft.com/>

¹⁰¹ <http://www.daionet.gr.jp/~masa/rthdrlib/>

¹⁰² http://www.anyhere.com/gward/hdrenc/hdr_encodings.html

¹⁰³ <http://www.linux.com/article.pl?sid=05/12/06/2115258>

¹⁰⁴ <http://www.hyperfocaldesign.com>

¹⁰⁵ <http://www.mpi-inf.mpg.de/resources/hdr/>

¹⁰⁶ http://www.cs.uh.edu/~somalley/hdri_images.html

¹⁰⁷ <http://www.gregdowning.com/HDRI/stitched/>

¹⁰⁸ <http://www.hdrshop.com>

¹⁰⁹ http://www.acm.uiuc.edu/siggraph/eoh_projects/eoh2002.html

¹¹⁰ <http://www.rendermania.com/HDR/>

Photography:

- High Dynamic Range (HDR) in Photography¹¹² - Implementation in Photoshop CS2

HDR displays:

- <http://www.brightsidetech.com/> High Dynamic Range Displays
- Technical information on HDR displays¹¹³

Studio rendering:

- <http://www.hdri-studio.com> Commercial HDR maps derived from studio lighting setups
- <http://www.sachform.com> Commercial HDR panoramas and viewer.
- PixelBox Academy HDRI Tutorial¹¹⁴ HDRI in PRMan using Image Based Illumination

On Rendering, but need descriptions, should be deleted otherwise:

Source: http://en.wikipedia.org/wiki/High_dynamic_range_imaging

Principal Authors: Imroy, Tnikkel, Rafm, Deanpemberton, Diliff, Noclip, Toytoy

High dynamic range rendering

High dynamic range rendering (HDRR or HDR Rendering) or less commonly, **high dynamic range lighting (HDR Lighting)**, is the rendering of 3D computer graphics scenes by using lighting calculations done in a high dynamic range. Specifically it refers to the new lighting model used to illuminate 3D worlds. Video games and computer generated movies greatly benefit from this as it creates far more realistic scenes than with conventional lighting models.

¹¹¹ <http://www.mpi-inf.mpg.de/resources/pfstools/>

¹¹² <http://www.cambridgeincolour.com/tutorials/high-dynamic-range.htm>

¹¹³ <http://www.cs.ubc.ca/~heidrich/Projects/HDRDisplay>

¹¹⁴ <http://pbacademy.com.sapo.pt/tutorials/renderman/hdri.htm>

History

The use of high dynamic range imaging (HDRI) in computer graphics was introduced by Greg Ward in 1985 with his Radiance rendering and *lighting simulation* software which created the first file format to retain a high dynamic range image. HDRI languished for many years with limited use as computing power and storage, as well as capture methods, had to be developed for HDRI to be put in to practical use.

In 1997 Paul Debevec presented *Recovering high dynamic range radiance maps from photographs* at SIGGRAPH and the following year presented *Rendering synthetic objects into real scenes*. These two papers laid the framework for creating HDR *light probes* of a location and then using this probe to light a rendered scene.

In gaming applications, after E³ 2003, Valve Software released a demo movie of their Source Engine rendering a cityscape in a high dynamic range. The term wouldn't be brought up again until E³ 2004 where it gained much more attention when Valve Software announced Half-Life 2: Lost Coast and Epic Megagames showcased Unreal Engine 3.

Features and limits



Figure 41 FarCry with HDR.

High dynamic range rendering



Figure 42 FarCry without HDR.

Preservation of detail in large contrast differences

The primary feature of HDRR is that both dark and bright areas of a scene can be accurately represented. Without HDR (sometimes called low dynamic range, or LDR, in comparison), dark areas are 'clipped' to black and the bright areas are clipped to white (represented by the hardware as a floating point value of 0.0 and 1.0, respectively).

Graphics processor company NVIDIA summarizes one of HDRR's features in three points:

- Bright things can be really bright
- Dark things can be really dark
- And details can be seen in both

The images on the right are from Ubisoft's FarCry, demonstrating a new patch that enables Shader Model 3.0 effects. The image on top shows a scene rendered with a high dynamic range, while the image on the bottom is not. Notice that the beams of light in the top image are vibrant and that there's more color to them. Also the walls where these beams illuminate appear brighter than the image on the bottom. Slightly more details can be seen in the HDRR render, despite that the darker areas of the scene (namely the dark area between

High dynamic range rendering

the two beams) have about the same amount of visibility in both pictures. In regards to color vibrance, the scene without HDR appears dull.

Accurate reflection of light



Figure 43 From Half-Life 2: Lost Coast, a comparison image between the same scene rendered with HDRR and without HDRR.

Without HDRR, the sun and most lights are clipped to 100% (1.0 in the frame-buffer). When this light is reflected the result must then be less than or equal to 1.0, since the reflected value is calculated by multiplying the original value by the surface's reflectiveness, usually in the range [0, 1]. This makes the lights appear dull. Using HDR the light produced by the sun and other lights can be represented with appropriately high values, exceeding the 1.0 clamping limit in the frame buffer, with the sun possibly being stored as 60000. When the light from them is reflected it will remain relatively high (even for very poor reflectors), which will be clipped to white or properly tonemapped when rendered. Looking at the example picture above, from Valve's Half-Life 2: Lost Coast, you can see these reflection differences upon the water and the sand.

An example

Imagine a dark room lit with moonlight, with a desk against the wall, a computer monitor on that desk, and a poster on the wall directly behind the monitor. If the scene was rendered in 3D using either 8-bit, 16-bit, or 32-bit lighting precision, there would be very little difference due to the low dynamic range inherent in the scene. Now if the monitor is displaying this web page, the dynamic range would be high. Human eyes could depict the detail on both the poster and monitor simultaneously better than say a digital camera could, because the digital camera can only "see" in a 256:1 contrast ratio of perception (assuming a linear response curve, which is the case for most CCD-based technology), whereas human eyes can perceive a much higher contrast ratio. This

High dynamic range rendering

would mean that the digital camera would have to capture the scene with a bit of bias between brightening the scene or darkening the scene by adjusting its exposure. If the scene was brightened, the text upon the monitor would be washed out in order for the poster to be seen. If the scene was darkened, the detail on the poster would be lost in favor upon capturing the text displayed on the monitor.

If this scene was rendered with HDR rendering, the detail on both the monitor and the poster would be preserved, without placing bias on brightening or darkening the scene.

Limitations

The human eye supports its high dynamic range in part through better brightness resolution than what can be done with 8 bits, but it also needs the iris to move its range of sensitivity from nearly pitch-black to a white wall in direct sunlight at the same time and computer displays can't do either.

A natural scene exposed in the sunlight can display a contrast of about 50,000:1. Negative black and white film can capture a dynamic range of about 4096:1 (12 stops) maximum, while color slide film can typically capture a dynamic range of 64:1 (6 stops). Printing has the same problems as displaying on LDR monitors as color paper only has about 64:1 (6 stops).

On average, most computer monitors have a specified contrast ratio between 500:1 and 1000:1, sometimes 1500:1. Current plasma displays are specified at a 10,000:1 contrast ratio (most are 50% lower). However, the contrast of commercial displays is measured as the ratio of a full white screen to a full black screen in a completely dark room. The simultaneous contrast of real content under normal viewing conditions is significantly lower.

One of the few monitors that can display in true HDR is the BrightSide Technologies¹¹⁵ HDR monitor, which has a simultaneous contrast ratio of around 200,000:1 for a brightness of 3000 cd.m-2, measured on a checkerboard image. In fact this higher contrast is equivalent to a ANSI9 contrast of 60,000:1, or about 60 times higher than the one of a TFT screen (about 1000:1). The brightness is 10 times higher than the one of the most CRT or TFT. But such display should only be useful if it needs to operate in a pitch-black room and in two seconds under bright lightning, and the eye should be able to see a full dynamic range on the display in both situations.

This means that HDR rendering systems have to map the full dynamic range to what the eye would see in the rendered situation. This tone mapping is done relative to what the virtual scene camera sees, combined with several full

¹¹⁵ <http://www.brightsidetech.com/>

screen effects, e.g. to simulate dust in the air which is lit by direct sunlight in a dark cavern.

There is currently two graphical effects used to combat these limitations, tone mapping and light blooming, which are often used together.

However, due to the stressing nature of HDRR, it's not recommended that full screen anti-aliasing (FSAA) be enabled while HDRR is enabled at the same time on current hardware, as this will cause a severe performance drop. Many games that support HDRR will only allow FSAA or HDRR. However some games which use a simpler HDR rendering model will allow FSAA at the same time.

Tone mapping



Figure 44 A screenshot of *Day of Defeat: Source*. While looking into a dark interior from the outside, it appears dark.

The limitations of display devices (whose gamuts are very limited compared to a gamut which supports HDR colors) prevent colors in HDR from being displayed properly. Tone mapping attempts to solve the problem of displaying HDR colors on a LDR display device by mapping colors in a HDR image to LDR. Many different tone mapping operators exist, and vary wildly in complexity



Figure 45 Inside, the interior is brighter.



Figure 46 Looking outside, the exterior appears brighter than before.

High dynamic range rendering



Figure 47 But the exterior's brightness normalizes as one goes outside.

and visual results. Some tone mapping algorithms are also time dependant, as seen in the above images, in an attempt to approximate a human eye's iris regulation of light entering the eye.

Light Bloom

Main article: →Bloom (shader effect)

Light blooming "spreads" out a light source. For example, a bright light in the background will appear to bleed over onto objects in the foreground. This effect is achieved by multiplying the image of the screen (lighten lighter areas and darken darker areas,) blurring the image, and drawing it over top of the original image. If there's a light source that is "brighter" than what the monitor can show, light blooming helps to create an illusion that makes the object appear brighter than it is, but at the cost of softening the scene. A common misconception is that a game that uses a bloom filter uses HDR. This is untrue, however, as blooming is often used alongside HDR, but is not a component of HDR.

The reason blooming is used alongside HDR is that when a saturated LDR image is blurred, the result is equivalent to blurring an image with lots of white

areas, simply because values above 1.0 aren't preserved, so bright areas blur to gradients in the range of 1.0 -> 0.0. The result is simply a linear white to black gradient, regardless of the original color of the bright part. HDR, on the other hand, preserves colors above 1.0, so the blur result in a gradient between the saturated value that slowly becomes less and less saturated. A bright blue light may appear white at its center, but with HDR it retains the blue color in a halo around it when the image is blurred.

Applications in computer entertainment

Currently HDRR has been prevalent in games. Though these games are mostly for the PC, it is also possible to render scenes in a high dynamic range in Microsoft's Xbox 360, Sony's PlayStation 3, and Nintendo's Wii. It has also been simulated on the PlayStation 2, GameCube, and Xbox. In desktop publishing and gaming, colour values are often processed several times over. As this includes multiplication and division it is useful to have the extended accuracy and range of 16 bit integer or 16 bit floating point format. This is useful irrespective of the abovementioned limitations in some hardware.

The development of HDRR into real time rendering mostly came from Microsoft's DirectX API.

DirectX's role in HDRR development

Complex shader effects began its days back with the release of Shader Model 1.0 with DirectX 8. Shader Model 1.0 illuminated 3D worlds with what is now called standard lighting. However, standard lighting had two problems:

1. Lighting precision was confined to 8 bit integers, which limited the contrast ratio to 256:1. Using the HVS color model, the value (V), or brightness of a color has a range of 0 - 255. This means the brightest white (a value of 255) is only 256 times brighter than the darkest black (a value of 0).
2. Lighting calculations were integer based, which didn't offer much accuracy because the real world is not confined to whole numbers ¹¹⁶.

Before HDRR was fully developed and implemented, games would fake the illusion of HDR by using light blooming and sometimes using an option called "Enhanced Contrast Settings" (Need For Speed Underground 2 had this as an option, while Metal Gear Solid 3 had it on at all times).

On December 24, 2002, Microsoft released a new version of DirectX. DirectX 9.0 introduced Shader Model 2.0 which offered one of the necessary components to enable rendering of high dynamic range rendering, lighting precision

¹¹⁶ http://download.nvidia.com/developer/presentations/2004/6800_Leagues/6800_Leagues_HDR.pdf

was not limited to just 8-bits. Although 8-bits was minimum in applications, programmers could choose up to a maximum of 24-bits for lighting precision. However, all calculations were still integer based. One of the first graphics cards to take advantage of DirectX 9.0 was ATI's Radeon 9700, though the effect wasn't programmed into games for years to come. On August 23, 2003, Microsoft updated DirectX to DirectX 9.0b, which enabled Pixel Shader 2.x (Extended) profile for ATI's Radeon X series and NVIDIA's GeForce FX series of graphics processing units.

On August 9, 2004, Microsoft updated DirectX once more to DirectX 9.0c. This also exposed the Shader Model 3.0 profile for high level shader language (HLSL). Shader Model 3.0's lighting precision, according to Dr. Sim Dietrich Jr., has a minimum of 32-bits as opposed to 2.0's 8-bit minimum. Also all lighting precision calculations are now floating-point based. NVIDIA states that contrast ratios using Shader Model 3.0 can be as high as 65535:1 using 32-bit lighting precision. At first, HDRR was only capable on video cards capable of Shader Model 3.0 effects, but software developers soon added compatibility for Shader Model 2.0. The difference between an HDRR scene rendered in Shader Model 3.0 or 2.0 is negligible at this point. As a side note, when referred as Shader Model 3.0 HDR, the HDRR is really done by FP16 blending. FP16 blending is not part of Shader Model 3.0, but supported only by cards also capable of Shader Model 3.0, excluding the GeForce 6200 models as they lack the capabilities. FP16 blending is used as a faster way to render HDR in video games.

It is unsure what HDR upgrades will be available from Shader Model 4.0 (to be released with DirectX 10).

Graphics cards which support HDRR

This is a list of Graphics cards that may or can support HDRR. It is implied that because the minimum requirement for HDR rendering is Shader Model 2.0 (or in this case DirectX 9), any graphics card that supports Shader Model 2.0 can do HDR rendering. However, HDRR may greatly impact the performance of the software using it, refer to your software's recommended specifications in order to run with acceptable performance.

Graphics cards designed for games

	Shader Model 2 Compliant (Includes versions 2.0, 2.0a and 2.0b)
From ATI	R300 series: 9500, 9500 Pro, 9550, 9550 SE, 9600, 9600 SE, 9600 AIW, 9600 Pro, 9600 XT, 9650, 9700, 9700 AIW, 9700 Pro, 9800, 9800 SE, 9800 AIW, 9800 Pro, 9800XT, X300, X300 SE, X550, X600 AIW, X600 Pro, X600 XT R420 series: X700, X700 Pro, X700 XT, X800, X800SE, X800 GT, X800 GTO, X800 Pro, X800 AIW, X800 XL, X800 XT, X800 XTPE, X850 Pro, X850 XT, X850 XTPE

From NVIDIA	GeForce FX (includes PCX versions): 5100, 5200, 5200 SE/XT, 5200 Ultra, 5300, 5500, 5600, 5600 SE/XT, 5600 Ultra, 5700, 5700 VE, 5700 LE, 5700 Ultra, 5750, 5800, 5800 Ultra, 5900 5900 ZT, 5900 SE/XT, 5900 Ultra, 5950, 5950 Ultra
From Intel	GMA: 900, 950
From S3 Graphics	Delta Chrome: S4, S4 Pro, S8, S8 Nitro, F1, F1 Pole Gamma Chrome: S18 Pro, S18 Ultra, S25, S27
From SiS	Xabre: Xabre II
From XGI	Volari: V3 XT, V5, V5, V8, V8 Ultra, Duo V5 Ultra, Duo V8 Ultra, 8300, 8600, 8600 XT
	Shader Model 3.0 Compliant
From ATI	R520 series: X1300 HyperMemory Edition, X1300, X1300 Pro, X1600 Pro, X1600 XT, X1800 GTO, X1800 XL AIW, X1800 XL, X1800 XT, X1900 AIW, X1900 GT, X1900 XT, X1900 XTX
From NVIDIA	GeForce 6: 6100, 6150, 6200, 6200 TC, 6500, 6600, 6600 LE, 6600 DDR2, 6600 GT, 6610 XL, 6700 XL, 6800, 6800 LE, 6800 XT, 6800 GS, 6800 GTO, 6800 GT, 6800 Ultra, 6800 Ultra Extreme GeForce 7: 7300 GS, 7600 GS, 7600 GT, 7800 GS, 7800 GT, 7800 GTX, 7800 GTX 512MB, 7900 GT, 7900 GTX, 7900 GX2, 7950 GX2

Graphics cards designed for workstations

	Shader Model 2 Compliant (Includes versions 2.0, 2.0a and 2.0b)
From ATI	FireGL: Z1-128, T2-128, X1-128, X2-256, X2-256t, V3100, V3200, X3-256, V5000, V5100, V7100
From NVIDIA	Quadro FX: 330, 500, 600, 700, 1000, 1100, 1300, 2000, 3000
	Shader Model 3.0 Compliant
From ATI	FireGL: V7300, V7350
From NVIDIA	Quadro FX: 350, 540, 550, 560, 1400, 1500, 3400, 3450, 3500, 4000, 4400, 4500, 4500SDI, 4500 X2, 5500
From 3Dlabs	Wildcat Realizm: 100, 200, 500, 800

Games which include HDR rendering

It will be only a matter of time before HDRR becomes a standard for future games, but until then, here's a list of games that support it.

Shader Model 3.0 HDR	Shader Model 2.0 HDR	Limited HDR
----------------------	----------------------	-------------

<ul style="list-style-type: none"> • <i>Act of War: High Treason</i> • <i>Age of Empires III</i> • <i>Bet on Soldier</i> • <i>Bioshock</i> • <i>Brothers In Arms: Hell's Highway</i> • <i>Crysis</i> • <i>Duke Nukem Forever</i> • <i>Far Cry</i> <ul style="list-style-type: none"> • (with patch 1.3 installed) • <i>Far Cry Instincts: Predator</i> <p>Romani in Spatiu</p> <ul style="list-style-type: none"> • <i>Gears of War</i> • <i>Huxley</i> • <i>Juiced</i> <ul style="list-style-type: none"> • (PC version only) • <i>Kameo: Elements of Power</i> • <i>The Lord of the Rings: The Battle for Middle-earth</i> • <i>The Lord of the Rings: The Battle for Middle-earth II</i> • <i>Lineage II</i> <ul style="list-style-type: none"> • (with Chronicle 4 updated) • <i>Project Gotham Racing 3</i> • <i>Project Offset</i> • <i>Perfect Dark Zero</i> • <i>Serious Sam II</i> • <i>Stranglehold</i> • <i>The Elder Scrolls IV: Oblivion</i> • <i>TimeShift</i> • <i>Tom Clancy's Ghost Recon: Advanced Warfighter</i> • <i>Tom Clancy's Splinter Cell: Chaos Theory</i> <ul style="list-style-type: none"> • (PC version only) • <i>Tom Clancy's Splinter Cell: Double Agent</i> <ul style="list-style-type: none"> • (PC and Xbox 360 version only) • <i>Tomb Raider: Legend</i> <ul style="list-style-type: none"> • (PC and Xbox 360 version only) • <i>Unreal Tournament 2007</i> • <i>Vanguard: Saga of Heroes</i> 	<ul style="list-style-type: none"> • <i>Act of War: High Treason</i> • <i>Bioshock</i> • <i>Brothers In Arms: Hell's Highway</i> • <i>Brothers in Arms: Earned in Blood</i> • <i>Brothers in Arms: Road to Hill 30</i> • <i>Call of Duty 2</i> • <i>Counter-Strike: Source</i> <ul style="list-style-type: none"> • (Limited to certain maps) • <i>Crysis</i> • <i>Day of Defeat: Source</i> • <i>Far Cry</i> <ul style="list-style-type: none"> • (SM 2.0 HDR patch soon) • <i>Gears of War</i> • <i>Half-Life 2: Episode One</i> • <i>Half-Life 2: Lost Coast</i> • <i>Huxley</i> • <i>Red Orchestra: Ostfront 41-45</i> • <i>Stranglehold</i> • <i>Tom Clancy's Splinter Cell: Chaos Theory</i> <ul style="list-style-type: none"> • (PC version only with patch 1.4 installed) • <i>Tom Clancy's Splinter Cell: Double Agent</i> <ul style="list-style-type: none"> • (PC version only) • <i>Unreal Tournament 2007</i> • <i>Vanguard: Saga of Heroes</i> 	<ul style="list-style-type: none"> • <i>Deus Ex: Invisible War</i> • <i>Need for Speed: Underground 2</i> • <i>Shadow of the Colossus</i> • <i>Operation Flashpoint: Elite Pariah</i> • <i>Metal Gear Solid 3: Snake Eater</i> • <i>Spartan: Total Warrior</i> • <i>SWAT 4</i> • <i>SWAT 4: The Stetchkov Syndicate</i> • <i>Thief: Deadly Shadows</i> • <i>Tribes Vengeance</i> • <i>Unreal Championship 2</i> • <i>WarPath</i>
--	---	--

Links

External links and sources

- NVIDIA's HDRR technical summary¹¹⁷ (PDF)
- Microsoft's technical brief on SM3.0 in comparison with SM2.0¹¹⁸
- High Dynamic Range Rendering in Photoshop CS2¹¹⁹ - Applications to digital photography
- Horror Blog¹²⁰
- Tom's Hardware: New Graphic Card Features of 2006¹²¹
- The Contrast Ratio Number Game¹²²
- List of GPU's compiled by Chris Hare¹²³
- techPowerUp! GPU Database¹²⁴

Software developer websites

- Source Engine Licensing¹²⁵ (see Source Engine for more)
- Half-Life 2¹²⁶
- Unreal Engine 3¹²⁷
- Emergent's Gamebryo¹²⁸
- CryEngine¹²⁹
- FarCry¹³⁰

Real-time HDR rendering

- Real time HDR demo for PCs¹³¹
- 3D Engine with HDR previewing¹³²
- Unigine with HDR Rendering Demos¹³³

¹¹⁷ http://download.nvidia.com/developer/presentations/2004/6800_Leagues/6800_Leagues_HDR.pdf

¹¹⁸ http://www.microsoft.com/whdc/winhec/partners/shadermodel30_NVIDIA.mspx

¹¹⁹ <http://www.cambridgeincolour.com/tutorials/high-dynamic-range.htm>

¹²⁰ <http://www.codinghorror.com/blog/archives/000324.html>

¹²¹ http://www.tomshardware.com/2006/01/13/new_3d_graphics_card_features_in_2006/

¹²² <http://www.practical-home-theater-guide.com/contrast-ratio.html>

¹²³ <http://users.erols.com/charle/video.htm>

¹²⁴ <http://www.techpowerup.com/gpudb/>

¹²⁵ <http://www.valvesoftware.com/sourcelicense/>

¹²⁶ <http://www.half-life2.com/>

¹²⁷ <http://www.unrealtechnology.com/html/technology/ue30.shtml>

¹²⁸ <http://www.emergent.net>

¹²⁹ <http://www.crytek.com>

¹³⁰ <http://www.farcry-thegame.com>

¹³¹ <http://www.daionet.gr.jp/~masa/rthdribl/>

¹³² <http://powerrender.com/>

¹³³ <http://unigine.com/>

References

- ↑ Paul E. Debevec and Jitendra Malik (1997). "Recovering High Dynamic Range Radiance Maps from Photographs"¹³⁴. SIGGRAPH.
- ↑ Paul E. Debevec (1998). "Rendering Synthetic Objects into Real Scenes: Bridging Traditional and Image-Based Graphics with Global Illumination and High Dynamic Range Photography"¹³⁵. SIGGRAPH.

Source: http://en.wikipedia.org/wiki/High_dynamic_range_rendering

Principal Authors: XenoL-Type, Unico master 15, Tnikkel, NoSoftwarePatents, Coldpower27

High Level Shader Language

The **High Level Shader Language** (HLSL) is a shader language developed by Microsoft for use with →Direct3D, and is very similar to Cg.

HLSL allows expensive graphical computations to be done on the graphics card, thus freeing up the CPU for other purposes.

Shader Model Comparison

	PS_2_0	PS_2_a	PS_2_b	PS_3_0
Dependent texture limit	4	No Limit	4	No Limit
Texture instruction limit	32	Unlimited	Unlimited	Unlimited
Position register	No	No	No	Yes
Instruction slots	32 + 64	512	512	≥ 512
Executed instructions	32 + 64	512	512	≥ 65535
Interpolated registers	2 + 8	2 + 8	2 + 8	10
Instruction predication	No	Yes	No	Yes
Index input registers	No	No	No	Yes
Temp registers	12	22	32	32
Constant registers	32	32	32	224
Arbitrary swizzling	No	Yes	No	Yes
Gradient instructions	No	Yes	No	Yes
Loop count register	No	No	No	Yes
Face register (2-sided lighting)	No	No	No	Yes
Dynamic flow control	No	No	No	24

¹³⁴ <http://www.debevec.org/Research/HDR/debevec-siggraph97.ps.gz>

¹³⁵ <http://www.debevec.org/Research/IBL/debevec-siggraph98.pdf>

- **PS_2_0** = DirectX 9.0 original **Shader Model 2** specification.
- **PS_2_a** = NVIDIA GeForce FX-optimized model, DirectX 9.0a.
- **PS_2_b** = ATI Radeon X700, X800, X850 shader model, DirectX 9.0b.
- **PS_3_0** = **Shader Model 3**.

External links

- **HLSL Introduction**¹³⁶
- Shader Model Comparison at Beyond3D¹³⁷

Source: http://en.wikipedia.org/wiki/High_Level_Shader_Language

Principal Authors: Swaaye, Nabla, TheSock, Unixxx, Soumyasch

Humanoid Animation

Humanoid Animation (H-Anim) is an approved ISO standard for humanoid modeling and animation. H-Anim defines a specification for defining interchangeable human figures so that those characters can be used across a variety of 3D games and simulation environments.

External links

- Humanoid Animation Working Group¹³⁸
- Web3D Consortium¹³⁹
- X3D Specification¹⁴⁰
- ISO/IEC 19774:2006 Humanoid Animation (H-Anim)¹⁴¹

Source: http://en.wikipedia.org/wiki/Humanoid_Animation

Principal Authors: Mditto, X42bn6, Perfecto

¹³⁶ <http://www.neatware.com/lbstudio/web/hlsl.html>

¹³⁷ http://www.beyond3d.com/reviews/ati/r420_x800/index.php?p=8

¹³⁸ <http://www.h-anim.org>

¹³⁹ <http://www.web3d.org>

¹⁴⁰ <http://www.web3d.org/x3d/specifications>

¹⁴¹ <http://www.iso.org/iso/en/CatalogueDetailPage.CatalogueDetail?CSNUMBER=33912>

Image based lighting

Image based lighting is a global illumination method used in 3D photo realistic rendering in which an image is used to light the scene in conjunction with traditional light models. Image based lighting, or IBL, is generally HDR, although LDR can be used, and provides a larger range of light sampling than can be obtained through traditional light models. LDR, low dynamic range, works similar to HDR, but offers a much smaller range of lighting information.

Almost every modern renderer offers some type of image based lighting, however the terminology used in their application might be slightly different.

Image based lighting is also starting to show up in game technology since most game consoles and computers have such an immense amount of processing power today.

References

- Tutorial¹⁴²

See also

- →Ambient occlusion
- Final gathering
- →Global illumination
- →High dynamic range imaging
- Light transport theory

Source: http://en.wikipedia.org/wiki/Image_based_lighting

¹⁴² <http://www.debevec.org/CGAIBL2/ibl-tutorial-cga2002.pdf>

Image plane

In →3D computer graphics, the **image plane** is that plane in the world which is identified with the plane of the monitor. If one makes the analogy of taking a photograph to rendering a 3D image, the surface of the film is the image plane. In this case, the viewing transformation is a projection that maps the world onto the image plane. A rectangular region of this plane, called the **viewing window** or **viewport**, maps to the monitor. This establishes the mapping between pixels on the monitor and points (or rather, rays) in the 3D world.

Source: http://en.wikipedia.org/wiki/Image_plane

Principal Authors: TheParanoidOne, RJHall, CesarB, Reedbeta

Inverse kinematic animation

Inverse kinematic animation (IKA) refers to a process utilized in 3D computer graphic animation, to calculate the required articulation of a series of limbs or joints, such that the end of the limb ends up in a particular location. In contrast to forward kinematic animation, where each movement for each component must be planned, only the starting and ending locations of the limb are necessary.

For example, when one wants to reach for a door handle, their brain must make the necessary calculations to position his limbs and torso such that the hand locates near the door. The main objective is to move the hand but the many complex articulations of several joints must occur to get the hand to the desired location. Similarly with many technological applications, inverse kinematic mathematical calculations must be performed to articulate limbs in the correct ways to meet desired goals.

One example where inverse kinematic calculations are often essential is robotics, where an operator wants to position a tool using a robot arm but certainly doesn't want to manipulate each robot joint individually. Other applications include computer animation where animators may want to operate a computer generated character, but find it impossibly difficult to animate individual joints. The solution is to model the virtual joints of the puppet and allow the animator to move the hands feet and torso, and the computer automatically generates the required limb positions to accomplish this using inverse kinematics.

Key to the successful implementation of inverse kinematics is animation within constraints: computer characters' limbs must behave within reasonable anthropomorphic limits. Similarly, robotic devices have physical constraints such as the environment they operate in, the limitations of the articulations their joints are capable of, and the finite physical loads and speeds at which they are able to operate.

See also: →Inverse kinematics

Source: http://en.wikipedia.org/wiki/Inverse_kinematic_animation

Principal Authors: Stevertigo, FrenchIsAwesome, Diberri, Stefan, Bryan Derksen

Inverse kinematics

Inverse kinematics is the process of determining the parameters of a jointed flexible object in order to achieve a desired pose. For example, with a 3D model of a human body, what are the required wrist and elbow angles to move the hand from a resting position to a waving position? This question is vital in robotics, where manipulator arms are commanded in terms of joint angles. Inverse kinematics are also relevant to game programming and →3D modeling, though its importance there has decreased with the rise of use of large libraries of motion capture data.

An articulated figure consists of a set of rigid segments connected with joints. Varying angles of the joints yields an indefinite number of configurations. The solution to the forward kinematics problem, given these angles, is the pose of the figure. The more difficult solution to the *inverse kinematics problem* is to find the joint angles given the desired configuration of the figure (i.e., end-effector). In the general case there is no analytic solution for the inverse kinematics problem. However, inverse kinematics may be solved via nonlinear programming techniques. Certain special kinematic chains—those with a spherical wrist—permit kinematic decoupling. This treats the end-effector's orientation and position independently and permits an efficient closed-form solution.

For animators, the inverse kinematics problem is of great importance. These artists find it far simpler to express spatial appearance rather than joint angles. Applications of inverse kinematic algorithms include interactive manipulation, animation control and collision avoidance.

See also: →Inverse kinematic animation

External links

- Inverse Kinematics algorithms¹⁴³
- Robot Inverse Kinematics¹⁴⁴
- HowStuffWorks.com article *How do the characters in video games move so fluidly?*¹⁴⁵ with an explanation of inverse kinematics
- 3D Theory Kinematics¹⁴⁶
- Protein Inverse Kinematics¹⁴⁷

Source: http://en.wikipedia.org/wiki/Inverse_kinematics

Principal Authors: Frecklefoot, GTubio, K.Nevelsteen, Charles Matthews, Dvavasour

Irregular Z-buffer

The **irregular Z-buffer** is an algorithm designed to solve the visibility problem in real-time 3-d computer graphics. It is related to the classical Z-buffer in that it maintains a depth value for each image sample and uses these to determine which geometric elements of a scene are visible. The key difference, however, between the classical Z-buffer and the irregular Z-buffer is that the latter allows arbitrary placement of image samples in the image plane, whereas the former requires samples to be arranged in a regular grid (See Figure 1).

These depth samples are explicitly stored in a two-dimensional spatial data structure. During rasterization, triangles are projected onto the image plane as usual, and the data structure is queried to determine which samples overlap each projected triangle. Finally, for each overlapping sample, the standard Z-compare and (conditional) frame buffer update are performed.

¹⁴³ http://freespace.virgin.net/hugo.elias/models/m_ik2.htm

¹⁴⁴ <http://www.learnaboutrobots.com/inverseKinematics.htm>

¹⁴⁵ <http://entertainment.howstuffworks.com/question538.htm>

¹⁴⁶ <http://www.euclideanspace.com/physics/kinematics/joints/index.htm>

¹⁴⁷ <http://cnx.org/content/m11613/latest/>

Implementation

The classical rasterization algorithm projects each polygon onto the image plane, and determines which sample points from a regularly-spaced set lie inside the projected polygon. Since the locations of these samples (i.e. pixels) are implicit, this determination can be made by testing the edges against the implicit grid of sample points. If, however the locations of the sample points are irregularly spaced and cannot be computed from a formula, then this approach does not work. The irregular Z-buffer solves this problem by storing sample locations explicitly in a two-dimensional spatial data structure, and later querying this structure to determine which samples lie within a projected triangle. This latter step is referred to as "irregular rasterization".

Although the particular data structure used may vary from implementation to implementation, the two studied approaches are the kd-tree, and a grid of linked lists. A balanced kd-tree implementation has the advantage that it guarantees $O(\log(N))$ access. It's chief disadvantage is that parallel construction of the kd-tree may be difficult, and traversal requires expensive branch instructions. The grid of lists has the advantage that it can be implemented more effectively on GPU hardware, which is designed primarily for the classical Z-buffer.

Applications

The irregular Z-buffer can be used for any application which requires visibility calculations at arbitrary locations in the image plane. It has been shown to be particularly adept at shadow mapping, an image space algorithm for rendering hard shadows. In addition to shadow rendering, potential applications include adaptive anti-aliasing, jittered sampling, and environment mapping.

External links

- The Irregular Z-Buffer And Its Application to Shadow Mapping¹⁴⁸ (pdf warning)
- Alias-Free Shadow Maps¹⁴⁹ (pdf warning)

Source: http://en.wikipedia.org/wiki/Irregular_Z-buffer

Principal Authors: Fooberman, DabMachine

¹⁴⁸ <http://www.cs.utexas.edu/ftp/pub/techreports/tr04-09.pdf>

¹⁴⁹ http://www.tml.hut.fi/~timo/publications/aila2004egsr_paper.pdf

Isosurface

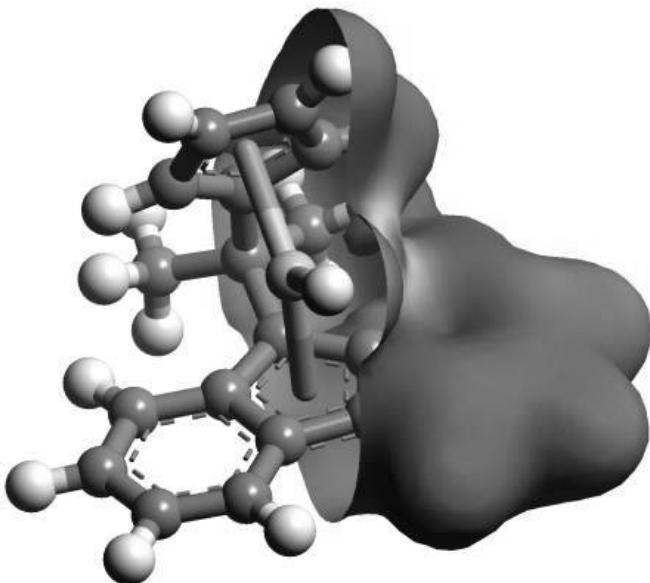


Figure 48 Zirconocene with an isosurface showing areas of the molecule susceptible to electrophilic attack. Image courtesy of Accelrys (<http://www.accelrys.com>)

An **isosurface** is a three-dimensional analog of an isocontour. It is a surface that represents points of a constant value (e.g. pressure, temperature, velocity, density) within a volume of space; in other words, it is a level set of a continuous function whose domain is 3-space.

Isosurfaces are normally displayed using computer graphics, and are used as data visualization methods in computational fluid dynamics (CFD), allowing engineers to study features of a fluid flow (gas or liquid) around objects, such as aircraft wings. An isosurface may represent an individual shockwave in supersonic flight, or several isosurfaces may be generated showing a sequence of pressure values in the air flowing around a wing. Isosurfaces tend to be a popular form of visualization for volume datasets since they can be rendered by a simple polygonal model, which can be drawn on the screen very quickly.

In medical imaging, isosurfaces may be used to represent regions of a particular density in a three-dimensional CT scan, allowing the visualization of internal organs, bones, or other structures.

Numerous other disciplines that are interested in three-dimensional data often use isosurfaces to obtain information about pharmacology, chemistry, geo-physics and meteorology.

A popular method of constructing an isosurface from a data volume is the marching cubes algorithm.

Source: <http://en.wikipedia.org/wiki/Isosurface>

Principal Authors: StoaBringer, The demiruge, Michael Hardy, RedWolf, Taw

Joint constraints

Joint constraints are rotational constraints on the joints of an artificial bone system. They are used in an inverse kinematics chain, for such things as 3D animation or robotics. Joint constraints can be implemented in a number of ways, but the most common method is to limit rotation about the X, Y and Z axis independently. An elbow, for instance, could be represented by limiting rotation on Y and Z axis to 0 degrees of freedom, and constraining the X-axis rotation to 130 degrees of freedom.

To simulate joint constraints more accurately, dot-products can be used with an independent axis to repulse the child bones orientation from the unreachable axis. Limiting the orientation of the child bone to a border of vectors tangent to the surface of the joint, repulsing the child bone away from the border, can also be useful in the precise restriction of shoulder movement.

Source: http://en.wikipedia.org/wiki/Joint_constraints

Principal Authors: Rofthorax, Dreadlocke, Salmar, Ravedave, Banana04131

Lambertian reflectance

If a surface exhibits **Lambertian reflectance**, light falling on it is scattered such that the apparent brightness of the surface to an observer is the same regardless of the observer's angle of view. More technically, the surface luminance is the same regardless of angle of view. For example, unfinished wood exhibits roughly Lambertian reflectance, but wood finished with a glossy coat of polyurethane does not (depending on the viewing angle, specular highlights may appear at different locations on the surface). Not all rough surfaces are perfect Lambertian reflectors, but this is often a good approximation when the characteristics of the surface are unknown.

In computer graphics, Lambertian reflection is often used as a model for diffuse reflection, and is calculated by taking the dot product of the surface's normalized normal vector \mathbf{N} and a normalized vector \mathbf{L} pointing from the surface to the light source. This number is then multiplied by the color of the surface and the intensity of the light hitting the surface:

$$I_D = \mathbf{L} \cdot \mathbf{N} * C * I_L,$$

where I_D is the intensity of the diffusely reflected light (surface brightness), C is the color and I_L is the intensity of the incoming light. Because

$$\mathbf{L} \cdot \mathbf{N} = |\mathbf{N}| |\mathbf{L}| \cos \alpha,$$

where α is the angle between the direction of the two vectors, the intensity will be the highest if the normal vector points in the same direction as the light vector ($\cos(0) = 1$, the surface will be perpendicular to the direction of the light), and the lowest if the normal vector is perpendicular to the light vector ($\cos(\pi) = 0$, the surface runs parallel with the direction of the light).

Lambertian reflection is typically accompanied by specular reflection, where the surface luminance is highest when the observer's angle is the same as the angle of the light source. This is simulated in computer graphics with →Phong shading.

Spectralon is a material which is designed to exhibit almost perfect Lambertian reflectance.

See also

- →Lambert's cosine law
- Specular reflection
- →Diffuse reflection

Source: http://en.wikipedia.org/wiki/Lambertian_reflectance

Principal Authors: Srleffler, KYN, Pedrose, PAR, Pflatau

Lambert's cosine law

Lambert's cosine law says that the total radiant power observed from a "Lambertian" surface is directly proportional to the cosine of the angle θ between the observer's line of sight and the surface normal. The law is also known as the **cosine emission law** or **Lambert's emission law**. It is named after Johann Heinrich Lambert, from his *Photometria*, published in 1760.

An important consequence of Lambert's cosine law is that when an area element on the surface is viewed from any angle, it has the same radiance. This means, for example, that to the human eye it has the same apparent brightness (or luminance). It has the same radiance because although the emitted power from an area element is reduced by the cosine of the emission angle, the observed size (solid angle) of the area element is also reduced by that same amount, so that while the area element appears smaller, its radiance is the same. For example, in the visible spectrum, the Sun is almost a Lambertian radiator, and as a result the brightness of the Sun is almost the same everywhere on an image of the solar disk. Also, a black body is a perfect Lambertian radiator.

Lambertian reflectors

When an area element is radiating as a result of being illuminated by an external source, the irradiance (energy or photons/time/area) landing on that area element will be proportional to the cosine of the angle between the illuminating source and the normal. A Lambertian reflector will then reflect this light according to the same cosine law as a Lambertian emitter. This means that although the radiance of the surface depends on the angle from the normal to the illuminating source, it will not depend on the angle from the normal to the observer. For example, if the moon were a Lambertian reflector, one would expect to see its reflected brightness appreciably diminish towards the terminator

due to the increased angle at which sunlight hit the surface. The fact that it does not diminish illustrates that the moon is not a Lambertian reflector, and in fact tends to reflect more light into the oblique angles than would a Lambertian reflector.

Details of equal brightness effect

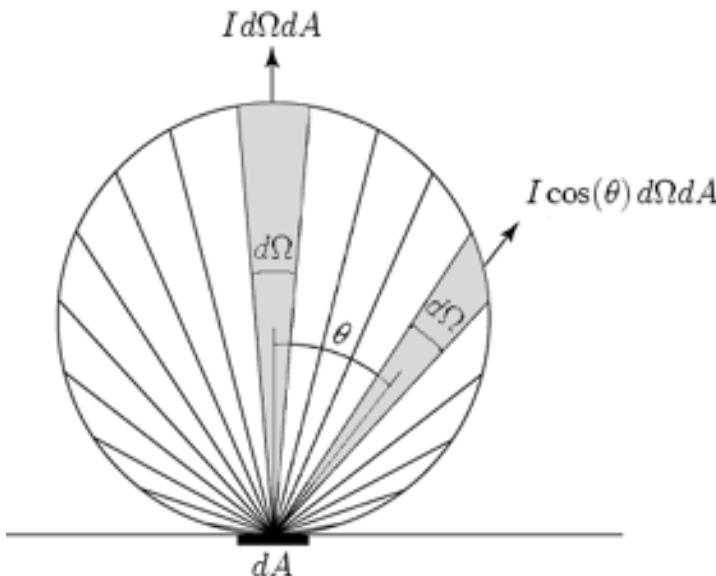


Figure 57 Figure 1: Emission rate (photons/s) in a normal and off-normal direction. The number of photons/sec directed into any wedge is proportional to the area of the wedge.

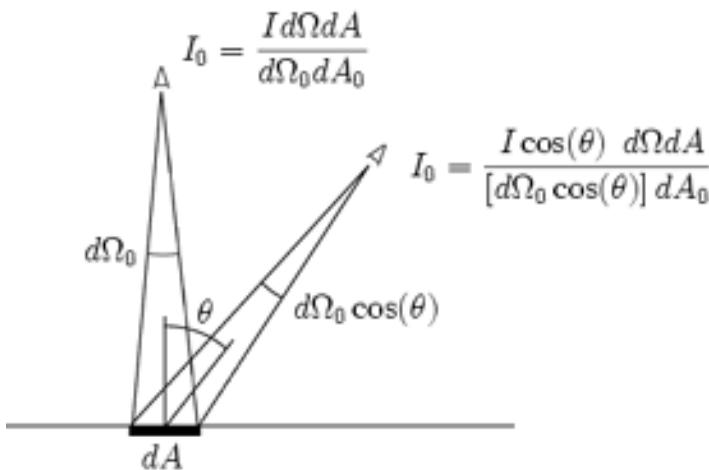


Figure 58 Figure 2: Observed intensity (photons/(s·cm²·sr)) for a normal and off-normal observer; dA_0 is the area of the observing aperture and $d\Omega$ is the solid angle subtended by the aperture from the viewpoint of the emitting area element.

This situation for a Lambertian reflector is illustrated in Figures 1 and 2. For conceptual clarity we will think in terms of photons rather than energy or luminous energy. The wedges in the circle each represent an equal angle $d\Omega$ and the number of photons per second emitted into each wedge is proportional to the area of the wedge.

It can be seen that the height of each wedge is the diameter of the circle times $\cos(\theta)$. It can also be seen that the maximum rate of photon emission per unit solid angle is along the normal and diminishes to zero for $\theta = 90^\circ$. In mathematical terms, the radiance along the normal is I photons/(s·cm²·sr) and the number of photons per second emitted into the vertical wedge is $I d\Omega dA$. The number of photons per second emitted into the wedge at angle θ is $I \cos(\theta) d\Omega dA$.

Figure 2 represents what an observer sees. The observer directly above the area element will be seeing the scene through an aperture of area dA_0 and the area element dA will subtend a (solid) angle of $d\Omega_0$. We can assume without loss of generality that the aperture happens to subtend solid angle $d\Omega$ when "viewed" from the emitting area element. This normal observer will then be recording $I d\Omega dA$ photons per second and so will be measuring a radiance of

$$I_0 = \frac{Id\Omega dA}{d\Omega_0 dA_0} \text{ photons/(s·cm}^2\cdot\text{sr}\text{)}.$$

The observer at angle θ to the normal will be seeing the scene through the same aperture of area dA_0 and the area element dA will subtend a (solid) angle of $d\Omega_0 \cos(\theta)$. This observer will be recording $I \cos(\theta) d\Omega dA$ photons per second, and so will be measuring a radiance of

$$I_0 = \frac{I \cos(\theta) d\Omega dA}{d\Omega_0 \cos(\theta) dA_0} = \frac{I d\Omega dA}{d\Omega_0 dA_0} \text{ photons/(s}\cdot\text{cm}^2\cdot\text{sr}),$$

which is the same as the normal observer.

Source: http://en.wikipedia.org/wiki/Lambert%27s_cosine_law

Principal Authors: PAR, Dbenbenn, Srleffler, Oleg Alexandrov, Michael Hardy

Level of detail (programming)

In computer programming and computer graphics, **level of detail** involves decreasing the detail of a model or object as it moves away from the viewer. Level of detail is used in computer and video games because it increases the efficiency of rendering by decreasing the polygon count — a desirable effect. The reduced visual quality of the model or object is, ideally, unnoticed due to the distance of the object. This, of course, depends on the individual system or game.

Black & White was one of the first games to use level of detail effects in its rendering. Each map consisted of a single, large island, sometimes populated with hundreds of villagers. As the player zoomed the camera out, progressively lower-detailed models were substituted for the villagers, buildings, and even the island itself. While not perfect, the technique allowed for a huge island to have detail in close-ups, but still be visible without slow-down when zoomed out. *Messiah* used a similar technique and was released before *Black & White*.

150

Halo 2 used level of detail adjustments to allow for much greater detail in cutscenes and maps, but at the cost of a "detail pop-in" effect when a level or scene was first loaded. Models would appear to have little or no detail at first, with more detail suddenly appearing over time as the more detailed models and textures loaded.

In order to make the detail decrease less obvious, fog is commonly used in computer and video games to reduce the visibility of details on the object,

150 <http://pc.ign.com/articles/123/123117p1.html>

simulating the haze that falls over distant objects in reality. When the object is out of the range of visibility, it won't be rendered anymore.

One of the most common methods used in automatic level of detail construction is based on the edge collapse transformation of a 3D mesh. Each pair of vertices in the 3D mesh are given a weighted error metric based on curvature or other criteria. The pair with the least given error is then merged/collapsed into one vertex. This is repeated until the desired triangle count is reached.

Other more advanced methods include the creation of lists containing precalculated edge collapse and vertex split values. Such lists can then be used for realtime calculation of level of detail objects. This method is often referred to as progressive meshes.

More advanced level of detail runtime systems use algorithms that are able to merge several 3D objects and simplify the merged objects in order to enhance reduction. Such algorithms are referred to as hierarchical level of detail algorithms.

External links

- *Level of Detail for 3D Graphics*¹⁵¹, by D. Luebke et. al.; one of the most complete books on level of detail for computer graphics
- Phd dissertation on hierarchical levels of details (PDF)¹⁵²
- Donya research¹⁵³; research and development of level of detail software company

Source: http://en.wikipedia.org/wiki/Level_of_detail_%28programming%29

Principal Authors: ZS, Drat, Jtalledo, Deepomega, GreatWhiteNortherner

¹⁵¹ <http://lodbook.com/>

¹⁵² <http://www.cs.unc.edu/~geom/HLOD/Dissertation/Dissertation.pdf>

¹⁵³ <http://www.donya.se>

Low poly

→ Low poly is a term used by videogame creators, players and journalists alike to describe a three-dimensional computer generated character or object that appears to be lacking in polygons.

In computer technology, polygons are what designers typically use to create any three-dimensional object that is output to the screen. The polygons are usually triangles. The more triangles that are used to create an object, the more detailed it will appear, but the more computing power it will take to render the object. Because of this, designers of videogames often have to be creative or cut corners with their polygon budget (the number of polygons that can be rendered per frame in a scene). This leads to objects that are often described as being low poly.

Objects that are said to be low poly often appear blocky (square heads) and lacking in detail (no individual fingers). Objects that are supposed to be circular or spherical are most obviously low poly, as the number of triangles needed to make a circle is high relative to other shapes, due to the curved line that is a circle.

The low poly issue is mostly confined to videogames and other software that the user manipulates in real time, e.g. the parts of a game that are playable. Low poly and polygon budgets are not an issue in, for example, computer-generated imagery effects like Gollum from the Lord of the Rings films or the entirety of Pixar animated films because they are created on large networks of computers called render farms. Each frame takes about an hour to create, despite the enormous computer power involved. This is why FMV sequences in videogames look so much better than the games themselves.

Low poly is a phrase which is used relative to the time it is released. As computing power inevitably increases, the number of polygons that can be used increases as well. For example, Super Mario 64 would be considered low poly today, but was considered a stunning achievement when it was released in 1996.

See also

- →Bump mapping
- →Normal mapping
- Sprites
- Nurbs

Source: http://en.wikipedia.org/wiki/Low_poly

Principal Authors: RJHall, David Levy, Plasmatics, Praetor alpha, Josh Parris

MegaTexture

MegaTexture refers to a texture mapping technique used in Splash Damage's upcoming game, *Enemy Territory: Quake Wars*. It was developed by id Software technical director John Carmack. MegaTexture is a design to eliminate repeating textures over an environment. The original version of the *Doom 3* engine was criticized over its perceived inability to handle landscapes and large outdoor areas. The MegaTexture technology addresses this issue by introducing a means to create expansive outdoor scenes. By painting a single massive texture (About 32000x32000 pixels, or 1 gigapixel) covering the entire polygon map and highly detailed terrain, the desired effects can be achieved. The MegaTexture can also store physical information about the terrain such as the amount of traction in certain areas or indicate what sound effect should be played when walking over specific terrain types on the map. i.e. walking on rock will sound different from walking on grass. It is expected that this will result in a considerably more detailed scene than the majority of existing technologies, using tiled textures, allow.

It has been suggested that MegaTexture technique is a modification of Clip Mapping.

See also

- →Texture mapping
- id Software
- *Enemy Territory: Quake Wars*
- *Doom 3*

External links

- id Software¹⁵⁴
- Splash Damage¹⁵⁵

¹⁵⁴ <http://www.idsoftware.com>

¹⁵⁵ <http://www.splashdamage.com>

- John Carmack on MegaTexture¹⁵⁶
- Arnout van Meer on Quake Wars¹⁵⁷

Source: <http://en.wikipedia.org/wiki/MegaTexture>

Principal Authors: Mojohompo, R. Koot, Skrapion, Vahid83, Amren

Mesa 3D

Mesa 3D is an open source graphics library, initially developed by Brian Paul in August 1993, that provides a generic →OpenGL implementation for rendering three-dimensional graphics on multiple platforms. Though Mesa is not an officially licensed OpenGL implementation, the structure, syntax and semantics of the API is that of OpenGL.

Advantages

- In its current form, Mesa 3D is available and can be compiled on virtually all modern platforms.
- Though not an official OpenGL implementation for licensing reasons, the Mesa 3D authors have worked to keep the API in line with the most current OpenGL standards and conformance tests, as set forth by the OpenGL ARB.
- Mesa 3D is distributed under the MIT License.
- Whilst Mesa 3D supports several hardware graphics accelerators, it may also be compiled as a software-only renderer. Since it is also Open Sourced, it is possible to use it to study the internal workings of an →OpenGL-compatible renderer.
- It is sometimes possible to find subtle bugs in OpenGL applications by linking against Mesa 3D and using a conventional debugger to track problems into the lower level library.

¹⁵⁶ <http://www.gamerwithin.com/?view=article&article=1319&cat=2>

¹⁵⁷ <http://www.beyond3d.com/interviews/etqw/>

External links

- Mesa 3D Homepage¹⁵⁸ - Information and documentation for the latest version of Mesa 3D.

Source: http://en.wikipedia.org/wiki/Mesa_3D

Principal Authors: Imroy, SteveBaker, Agentsoo, Caiyu, Aegis Maelstrom

Metaballs

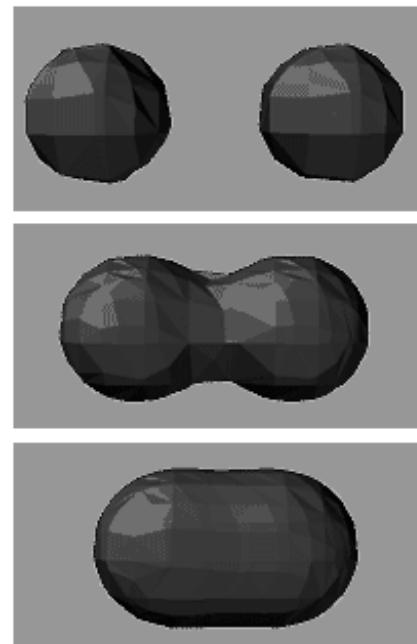


Figure 59 Two metaballs

Metaballs, in computer graphics terms, are organic-looking n-dimensional objects. The technique for rendering metaballs was invented by Jim Blinn in the early 1980s.

¹⁵⁸ <http://www.mesa3d.org/>

Each metaball is defined as a function in n-dimensions (ie. for three dimensions, $f(x, y, z)$; three-dimensional metaballs tend to be most common). A thresholding value is also chosen, to define a solid volume. Then,

$$\sum_{i=0}^n \text{metaball}_i(x, y, z) \leq \text{threshold}$$

represents whether the volume enclosed by the surface defined by n metaballs is filled at (x, y, z) or not.

A typical function chosen for metaballs is $f(x, y, z) = 1/((x - x_0)^2 + (y - y_0)^2 + (z - z_0)^2)$, where (x_0, y_0, z_0) is the center of the metaball. However, due to the divide, it is computationally expensive. For this reason, approximate polynomial functions are typically used (examples?).

There are a number of ways to render the metaballs to the screen. The two most common are brute force raycasting and the marching cubes algorithm.

2D metaballs used to be a very common demo effect in the 1990s. The effect is also available as an XScreensaver module.

Further reading

- Blinn, James F. "A Generalization of Algebraic Surface Drawing." *ACM Transactions on Graphics* 1(3), July 1982, pp. 235–256.

Source: <http://en.wikipedia.org/wiki/Metaballs>

Principal Authors: Iron Wallaby, Viznut, T-tus, Kibibu, Felsir

Metropolis light transport

This SIGGRAPH 1997 paper by Eric Veach and Leonidas J. Guibas describes an application of a variant of the Monte Carlo method called the Metropolis-Hastings algorithm to the rendering equation for generating images from detailed physical descriptions of three dimensional scenes.

The procedure constructs paths from the eye to a light source using bi-directional path tracing, then constructs slight modifications to the path. Some careful statistical calculation (the Metropolis algorithm) is used to compute the appropriate distribution of brightness over the image. This procedure has the advantage, relative to bidirectional path tracing, that once a path has been found from light to eye, the algorithm can then explore nearby paths; thus

difficult-to-find light paths can be explored more thoroughly with the same number of simulated photons.

In short, the algorithm generates a path and stores the paths 'nodes' in a list. It can then modify the path by adding extra nodes and creating a new light path. While creating this new path, the algorithm decides how many new 'nodes' to add and whether or not these new nodes will actually create a new path.

External link

- Metropolis project at Stanford¹⁵⁹

Source: http://en.wikipedia.org/wiki/Metropolis_light_transport

Principal Authors: Pfortuny, Loisel, Aarchiba, The Anome, Levork

Micropolygon

In →3D computer graphics, a **micropolygon** is a polygon that is at least as small as the size of a pixel in the output image. The concept of micropolygons was developed to be used by the Reyes algorithm. At rendering time, the geometric primitive is tessellated into a rectangular grid of tiny four-sided polygons or micropolygons. A shader later assigns colors to the vertices of these faces. Commonly the size of these micropolygons is the same as the area of a pixel. Using micropolygons allows the renderer to create a highly detailed image.

Shaders that operate on micropolygons can process an entire grid at once in SIMD fashion. This often leads to faster shader execution, and allows shaders to compute spatial derivatives (e.g. for texture filtering) by comparing values at neighboring micropolygon vertices.

A renderer that uses micropolygons can support displacement mapping simply by perturbing micropolygon vertices during shading.

Further reading

- Steve Upstill: *The RenderMan Companion: A Programmer's Guide to Realistic Computer Graphics*, Addison-Wesley, ISBN 0-201-50868-0
- Anthony A. Apodaca, Larry Gritz: *Advanced RenderMan: Creating CGI for Motion Pictures*, Morgan Kaufmann Publishers, ISBN 1-55860-618-1

¹⁵⁹ <http://graphics.stanford.edu/papers/metro/>

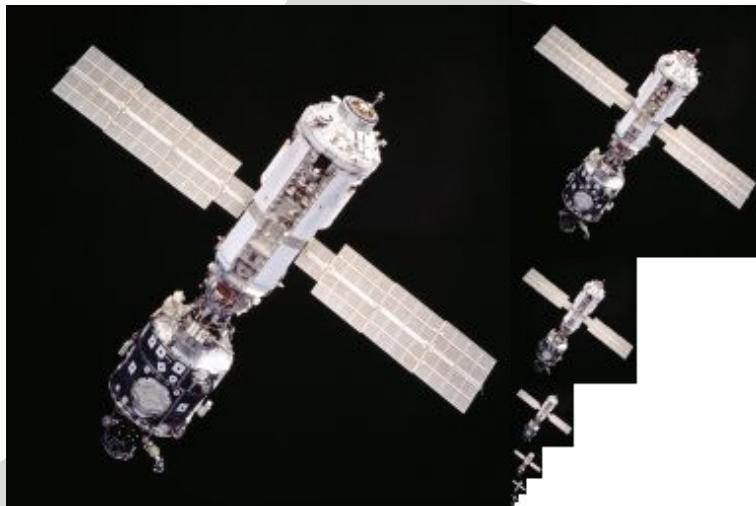
Source: <http://en.wikipedia.org/wiki/Micropolygon>

Principal Authors: Flamurai, RJHall, Dmaas, T-tus

Mipmap

In →3D computer graphics texture mapping, **MIP maps** (also **mipmaps**) are pre-calculated, optimized collections of bitmap images that accompany a main texture, intended to increase rendering speed and reduce 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".

How it works



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) 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 (textures are typically square and must have side lengths equal to a power of 2), then the associated mipmap set may contain a series of 8 images, each one-fourth the size 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 an interpolation of the 64×64 and the 32×32 mipmaps 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 $1/4 + 1/16 + 1/256 + \dots$ converges to $1/3$. (This assumes compression is not being used.) This is a major advantage to this selection of resolutions. However, 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.

To reduce the memory requirement, and simultaneously give more resolutions to work with, summed-area tables were conceived. Given a texture (t_{jk}), we can build a summed area table (s_{jk}) as follows. The summed area table has the same number of entries as there are texels in the texture map. Then, define

$$s_{mn} := \sum_{1 \leq j \leq m, 1 \leq k \leq n} t_{jk}$$

Then, the average of the texels in the rectangle $(a_1, b_1] \times (a_2, b_2]$ is given by

$$\frac{s_{a_2 b_2} - s_{a_1 b_2} - s_{a_2 b_1} + s_{a_1 b_1}}{(a_2 - a_1)(b_2 - b_1)}$$

However, this approach tends to exhibit poor cache behavior. Also, a summed area table needs to have wider types to store the partial sums s_{jk} than the word size used to store t_{jk} . 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.

See also

- Anti-aliasing

Source: <http://en.wikipedia.org/wiki/Mipmap>

Principal Authors: MIT Trekkie, RJHall, Spoon!, Mat-C, Tarquin, Grendelkhan

Morph target animation

Morph target animation (or **per-vertex animation**) is a method of 3D computer animation that is sometimes used in alternative to skeletal animation. Morph target animation is stored as a series of vertex positions. In each keyframe of the animation, the vertices are moved to a different position.

Depending on the renderer, the vertices will move along paths to fill in the blank time between the keyframes or the renderer will simply switch between the different positions, creating a somewhat jerky look. The former is used more commonly.

There are advantages to using morph target animation over skeletal animation. The artist has more control over the movements because he or she can define the individual positions of the vertices within a keyframe, rather than being

constrained by skeletons. This can be useful for animation cloth, skin, and facial expressions because it can be difficult to conform those things to the bones that are required for skeletal animation.

However, there are also disadvantages. Vertex animation is usually a lot more time-consuming than skeletal animation because every vertex position would have to be calculated. (3D models in modern computer and video games often contain something to the order of 4,000-9,000 vertices.) Also, in methods of rendering where vertices move from position to position during in-between frames, a distortion is created that doesn't happen when using skeletal animation. This is described by critics of the technique as looking "shaky." However, there are some who like this slightly distorted look.

Not all morph target animation has to be done by actually editing vertex positions. It is also possible to take vertex positions found in skeletal animation and then use those rendered as morph target animation.

Sometimes, animation done in one 3D application suite will need to be taken into another for rendering. To avoid issues in export, animation will often be converted from whatever format it was in to morph target animation. This is sometimes necessary because things such as bones and special effects are not programmed using consistent systems among different 3D application suites.

See also

- Skeletal animation
- Computer and video games
- →3D computer graphics

Source: http://en.wikipedia.org/wiki/Morph_target_animation

Motion capture

Motion capture, or **mocap**, is a technique of digitally recording movements for entertainment, sports and medical applications.

It started as an analysis tool in biomechanics research, but has grown increasingly important as a source of motion data for computer animation as well as education, training and sports and recently for both cinema and video games. A performer wears a set of one type of marker at each joint: acoustic, inertial, LED, magnetic or reflective markers, or combinations, to identify the motion



Figure 60 A dancer wearing a suit used in an optical motion capture system

of the joints of the body. Sensors track the position or angles of the markers, optimally at least two times the rate of the desired motion. The motion capture computer program records the positions, angles, velocities, accelerations and impulses, providing an accurate digital representation of the motion. This can reduce the costs of animation, which otherwise requires the animator to draw each frame, or with more sophisticated software, key frames which are interpolated by the software. *Motion capture* saves time and creates more natural movements than manual animation, but is limited to motions that are anatomically possible. Some applications might require additional *impossible* movements like animated super hero martial arts.

Optical systems triangulate the 3D position of a marker with a number of cameras with high precision (sub-millimeter resolution or better). These systems produce data with 3 degrees of freedom for each marker, and rotational information must be inferred from the relative orientation of three or more markers; for instance shoulder, elbow and wrist markers providing the angle of the elbow. A related technique match moving can derive 3D camera movement from a single 2D image sequence without the use of photogrammetry, but is often ambiguous below centimeter resolution, due to the inability to distinguish pose

and scale characteristics from a single vantage point. One might extrapolate that future technology might include full-frame imaging from many camera angles to record the exact position of every part of the actor's body, clothing, and hair for the entire duration of the session, resulting in a higher resolution of detail than is possible today. A newer technique discussed below uses higher resolution linear detectors to derive the one dimensional positions, requiring more sensors and more computations, but providing higher resolutions (sub millimeter down to 10 micrometres time averaged) and speeds than possible using area arrays^{160 161 162}.

Passive optical systems use reflective markers and triangulate each marker from its relative location on a 2D map with multiple cameras calibrated to provide overlapping projections combined to calculate 3D positions. Data can be cleaned up with the aid of kinematic constraints and predictive gap filling algorithms. Passive systems typically use sensors such as Micron's¹⁶³ where the sensor captures an image of the scene, reduces it to bright spots and finds the centroid. These 1.3 megapixel sensors can run at frame rates up to 60,000,000 pixels per second divided by the resolution, so at 1.3 megapixels they can operate at 500 frames per second. The 4 megapixel sensor costs about \$1,000 and can run at 640,000,000 pixels per second divided by the applied resolution. By decreasing the resolution down to 640 x 480, they can run at 2,000 frames per second, but then trade off spatial resolution for temporal resolution. At full resolution they run about 166 frames per second, with about 200 LED strobes synchronized to the CMOS sensor. The ease of combining a hundred dollars worth of LEDs to a \$1,000 sensor has made these system very popular. Professional vendors have sophisticated software to reduce problems from marker swapping since all markers appear identical. These systems are popular for entertainment, biomechanics, engineering, and virtual reality applications; tracking a large number of markers and expanding the capture area with the addition of more cameras. Unlike active marker systems and magnetic systems, passive systems do not require the user to wear wires or electronic equipment. Passive markers are usually spheres or hemispheres made of plastic or foam 25 to 3mm in diameter with special retroreflective tape. This type of system is the dominate favorite among entertainment and biomechanics groups currently due to its ability to capture large numbers of markers at frame rates as high as 2000fps and high 3D accuracy. Active marker system run into a disadvantage when marker counts climb as they perform "frame slicing", providing

¹⁶⁰ <http://www.ndigital.com>

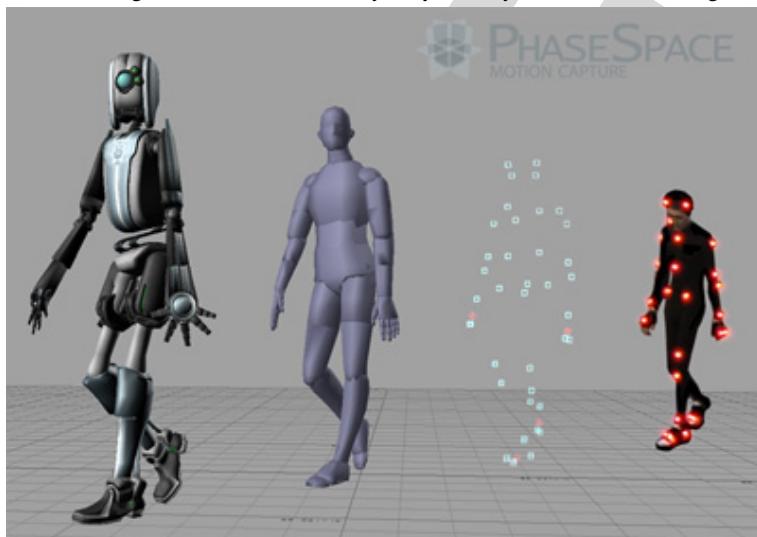
¹⁶¹ <http://www.phasespace.com/>

¹⁶² <http://www.vicon.com/>

¹⁶³ <http://www.micron.com/products/imaging/products/MT9M413.html>

a frame for each active marker, or by using coded pulsing which can lead to mis-identification. Manufacturers of this type of system include Vicon-Peak¹⁶⁴, Motion Analysis¹⁶⁵ and BTS¹⁶⁶.

Active marker systems have an advantage over passive in that there is no doubt about which marker is which. In general, the overall update rate drops as the marker count increases; 5000 frames per second divided by 100 markers would provide updates of 50 hertz. As a result, these systems are popular in the biomechanics market. Two such active marker systems are Optotrak by Northern Digital¹⁶⁷ and the Visualeyez system by PhoeniX Technologies Inc.¹⁶⁸.



Active marker systems such as PhaseSpace¹⁶⁹ modulate the active output of the LED to differentiate each marker, allowing several markers to be on at the same time, to provide marker IDs in real time eliminating marker swapping and providing much cleaner data than older technologies. Active markers require wires to the LEDs, allowing motion capture outdoors in direct sunlight, while providing the higher resolution of 3,600 x 3,600 or 12 megapixel resolution while capturing up to 480 frames per second. The advantage of using active markers is intelligent processing allows higher speed and higher resolution of optical systems at a lower price. This higher accuracy and resolution requires

¹⁶⁴ <http://www.viconpeak.com/>

¹⁶⁵ <http://www.motionanalysis.com>

¹⁶⁶ <http://www.bts.it>

¹⁶⁷ <http://www.ndigital.com>

¹⁶⁸ <http://www.ptiphoenix.com/>

¹⁶⁹ <http://www.phasespace.com/>

more processing than older passive technologies, but the additional processing is done at the camera to improve resolution via a subpixel or centroid processing, providing both high resolution and high speed. By using newer processing and technology, these motion capture systems are about 1/3 the cost of older systems.

Magnetic systems, calculate position and orientation by the relative magnetic flux of three orthogonal coils on both the transmitter and each receiver. The relative intensity of the voltage or current of the three coils allows these systems to calculate both range and orientation by meticulously mapping the tracking volume. Since the sensor output is 6DOF, useful results can be obtained with two-thirds the number of markers required in optical systems; one on upper arm and one on lower arm for elbow position and angle. The markers are not occluded by nonmetallic objects but are susceptible to magnetic and electrical interference from metal objects in the environment, like rebar (steel reinforcing bars in concrete) or wiring, which affect the magnetic field, and electrical sources such as monitors, lights, cables and computers. The sensor response is nonlinear, especially toward edges of the capture area. The wiring from the sensors tends to preclude extreme performance movements. The capture volumes for magnetic systems are dramatically smaller than they are for optical systems. With the magnetic systems, there is a distinction between "AC" and "DC" systems: one uses square pulses, the other uses sine wave pulses. Two magnetic systems are Ascension technology¹⁷⁰ and Polhemus¹⁷¹.

A motion capture session records only the movements of the actor, not his visual appearance. These movements are recorded as *animation data* which are mapped to a 3D model (human, giant robot, etc.) created by a computer artist, to move the model the same way. This is comparable to the older technique of rotoscope where the visual appearance of the motion of an actor was filmed, then the film used as a guide for the frame by frame motion of a hand-drawn animated character.

Inertial systems use devices such as accelerometers or gyroscopes to measure positions and angles. They are often used in conjunction with other systems to provide updates and global reference, since they only measure relative changes, not absolute position.

RF (radio frequency) positioning systems are becoming more viable as higher frequency RF devices allow greater precision than older RF technologies. The speed of light is 30 centimeters per nanosecond (billionth of a second), so a 10 gigahertz (billion cycles per second) RF signal enables an accuracy of about

¹⁷⁰ <http://www.ascension-tech.com/>

¹⁷¹ <http://www.polhemus.com/>

3 centimeters. By measuring amplitude to a quarter wavelength, it is possible to improve the resolution down to about 8 mm. To achieve the resolution of optical systems, frequencies of 50 gigahertz or higher are needed, which are almost as line of sight and as easy to block as optical systems. Multipath and reradiation of the signal are likely to cause additional problems, but these technologies will be ideal for tracking larger volumes with reasonable accuracy, since the required resolution at 100 meter distances isn't likely to be as high.



Mechanical motion capture systems directly track body joint angles and are often referred to as exo-skeleton motion capture systems, due to the way the sensors are attached to the body. A performer attaches the skeletal-like structure to their body and as they move so do the articulated mechanical parts, measuring the performer's relative motion. Mechanical motion capture systems are real-time, relatively low-cost, free-of-occlusion, and wireless (untethered) systems that have unlimited capture volume. Typically, they are rigid structures of jointed, straight metal or plastic rods linked together with potentiometers that articulate at the joints of the body. However, a newer and more flexible take on exo-skeleton motion capture systems is the ShapeWrap II¹⁷² system by Measurand Inc.¹⁷³ ShapeWrap II offers a mocap system based on ShapeTapes¹⁷⁴ that flex. By conforming to limbs instead of following rigid paths, ShapeWrap II moves with the body to capture fine details of shape on a wide variety of body types.

¹⁷² <http://www.measurand.com/products/ShapeWrap.html>

¹⁷³ <http://www.measurand.com/>

¹⁷⁴ <http://www.measurand.com/products/ShapeTape.html>

The procedure

In the motion capture session, the movements of one or more actors are sampled many times per second. High resolution optical motion capture systems can be used to sample body, facial and finger movement at the same time.

If desired, a camera can pan, tilt, or dolly around the stage while the actor is performing and the motion capture system can capture the camera and props as well. This allows the computer generated characters, images and sets, to have the same perspective as the video images from the camera. A computer processes the data and displays the movements of the actor, as inferred from the 3D position of each marker.

After processing, the software exports animation data, which computer animators can associate with a 3D model and then manipulate using normal computer animation software such as Maya or 3D Studio Max. If the actor's performance was good and the software processing was accurate, this manipulation is limited to placing the actor in the scene that the animator has created and controlling the 3D model's interaction with objects.

Advantages

Mocap offers several advantages over traditional computer animation of a 3D model:

- Mocap can take far fewer man-hours of work to animate a character. One actor working for a day (and then technical staff working for many days afterwards to clean up the mocap data) can create a great deal of animation that would have taken months for traditional animators.
- Mocap can capture secondary animation that traditional animators might not have had the skill, vision, or time to create. For example, a slight movement of the hip by the actor might cause his head to twist slightly. This nuance might be understood by a traditional animator but be too time consuming and difficult to accurately represent, but it is captured accurately by mocap, which is why mocap animation often seems shockingly realistic compared with hand animated models. Incidentally, one of the hallmarks of rotoscope in traditional animation is just such secondary "business."
- Mocap can accurately capture difficult-to-model physical movement. For example, if the mocap actor does a backflip while holding nunchaku by the chain, both sticks of the nunchucks will be captured by the cameras moving in a realistic fashion. A traditional animator might not be able to physically simulate the movement of the sticks adequately due to other motions by the actor. Secondary motion such as the ripple of a body as an actor is punched

or is punching requires both higher speed and higher resolution as well as more markers.

Disadvantages

On the negative side, mocap data requires special programs and time to manipulate once captured and processed, and if the data is wrong, it is often easier to throw it away and reshoot the scene rather than trying to manipulate the data. Many systems allow real time viewing of the data to decide if the take needs to be redone.

Another important point is that while it is common and comparatively easy to mocap a human actor in order to animate a biped model, applying motion capture to animals like horses can be difficult.

Motion capture equipment costs tens of thousands of dollars for the digital video cameras, lights, software, and staff to run a mocap studio, and this technology investment can become obsolete every few years as better software and techniques are invented. Some large movie studios and video game publishers have established their own dedicated mocap studios, but most mocap work is contracted to individual companies that specialize in mocap.

Applications

Video games use motion capture for football, baseball and basketball players or the combat moves of a martial artist.

Movies use motion capture for CG effects, in some cases replacing traditional cell animation, and for completely computer-generated creatures, such as Gollum, Jar-Jar Binks, and King Kong, in live-action movies.

Virtual Reality and Augmented Reality require real time input of the user's position and interaction with their environment, requiring more precision and speed than older motion capture systems could provide. Noise and errors from low resolution or low speed systems, and overly smoothed and filtered data with long latency contribute to "simulator sickness" where the lag and mismatch between visual and vestibular cues and computer generated images caused nausea and discomfort.

High speed - high resolution active marker systems can provide smooth data at low latency, allowing real time visualization in virtual and augmented reality systems. The remaining challenge that is almost possible with powerful graphic cards is mapping the images correctly to the real perspectives to prevent image mismatch.

Motion capture technology is frequently used in digital puppetry systems to aid in the performance of computer generated characters in real-time.

Related techniques

Facial motion capture is utilized to record the complex movements in a human face, especially while speaking with emotion. This is generally performed with an optical setup using multiple cameras arranged in a hemisphere at close range, with small markers glued or taped to the actor's face.

Performance capture is a further development of these techniques, where both body motions and facial movements are recorded. This technique was used in making of *The Polar Express*, where all actors were animated this way.

An alternative approach was developed by a Russian company VirtuSphere¹⁷⁵, where the actor is given an unlimited walking area through the use of a rotating sphere, similar to a



, which contains internal sensors recording the angular movements, removing the need for external cameras and other equipment. Even though this technology could potentially lead to much lower costs for mocap, the basic sphere is only capable of recording a single continuous direction. Additional sensors worn on the person would be needed to record anything more.

¹⁷⁵ <http://www.virtusphere.net/>

See also

- Animation
- Rotoscope
- Match moving, also known as "motion tracking"

External links

- Perspective Studios¹⁷⁶ - Motion Capture Studio located in New York.
- LUKOtronic¹⁷⁷ - Portable optical motion capture systems.
- Motion Analysis¹⁷⁸ - Maker of optical motion capture systems.
- Vicon Peak¹⁷⁹ - Maker of optical motion capture systems.
- Audiomotion Studios¹⁸⁰ - Award Winning Motion Capture Service Provider Based in the UK.
- VirtualCinematography.org¹⁸¹ - several papers on Universal Capture use in Matrix films
- Motion Reality Inc.¹⁸² - A company specializing in motion capture systems and applications
- Measurand Inc.¹⁸³ - A company specializing in motion capture systems and applications
- Motion Reality Golf¹⁸⁴ - A golf training application using motion capture technology
- Optical Motion Capture¹⁸⁵ - Active Marker LED based real time motion tracking hardware and software for VR, AR, Telerobotics, medical and entertainment applications.
- Giant Studios Inc.¹⁸⁶ - A company specializing in motion capture systems and software (Polar Express, Chronicles of Narnia)
- motionVR Corporation¹⁸⁷ - A company that produces software for creating semi-3D models of real life locations.

¹⁷⁶ <http://www.perspectivestudios.com/>

¹⁷⁷ <http://www.LUKOtronic.com/>

¹⁷⁸ <http://www.motionanalysis.com/>

¹⁷⁹ <http://www.vicon.com/>

¹⁸⁰ <http://www.audiomotion.com/>

¹⁸¹ <http://www.virtualcinematography.org/>

¹⁸² <http://www.motionrealityinc.com/>

¹⁸³ <http://www.measurand.com/>

¹⁸⁴ <http://www.motionrealtygolf.com/>

¹⁸⁵ <http://www.PhaseSpace.com/>

¹⁸⁶ <http://www.giantstudios.com/>

¹⁸⁷ <http://www.motionVR.com>

- VirtuSphere Inc.¹⁸⁸ - A company designed an alternative way of motion capture.
- Mocap.ca¹⁸⁹ - Motion Capture resources and services including software and motion capture data.
- House of Moves¹⁹⁰ - Motion Capture Facility based in Los Angeles

Motion capture hardware

- LUKOtronic - Motion Capture Systems¹⁹¹ - Commercially available motion capture systems and tools
- The ShapeWrap II & ShapeHand Motion Capture Systems¹⁹² - Commercially available motion capture systems and tools
- PhaseSpace Optical Motion Capture¹⁹³ - Active Marker LED based real time motion tracking hardware and software for VR, AR, Telerobotics, medical and entertainment applications.
- PhoeniX Technologies Inc. - Real-time Motion Capture¹⁹⁴ - Features the Vi-sualeyez real-time, active-optical motion capture system. Hardware and software for diverse applications: biomechanics research, virtual environments, animation, game development, tele-robotics, and more.

Source: http://en.wikipedia.org/wiki/Motion_capture

Principal Authors: Tmcsherry, Myf, Tempshill, Plowboyifestyle, Paranoid, Lindosland, RobertM52, Sergeyey, Michael Snow

¹⁸⁸ <http://www.virtusphere.net/>

¹⁸⁹ <http://www.mocap.ca/>

¹⁹⁰ <http://www.moves.com/>

¹⁹¹ <http://www.LUKOtronic.com/>

¹⁹² <http://www.measurand.com/>

¹⁹³ <http://www.PhaseSpace.com/>

¹⁹⁴ <http://www.ptiphoenix.com/>

Newell's algorithm

Newell's Algorithm is a →3D computer graphics procedure for elimination of polygon cycles in the depth sorting required in hidden surface removal. It was proposed in 1972 by M. E. Newell, R. Newell and T. Sancha.

In the depth sorting phase of hidden surface removal, if two polygons have no overlapping **extents** or extreme minimum and maximum values in the x,y, and z directions, then they can be easily sorted. If two polygons, Q and P do have overlapping extents in the Z direction then it is possible that cutting is necessary.

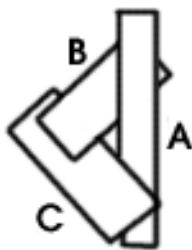


Figure 61 Cyclic polygons must be eliminated to correctly sort them by depth

In that case Newell's algorithm tests the following :

1. Test for Z overlap; implied in the selection of the face Q from the sort list
2. The extreme coordinate values in X of the two faces do not overlap(minimax test in X)
3. The extreme coordinate values in Y of the two faces do not overlap (minimax test in Y)
4. All vertices of P lie deeper than the plane of Q
5. All vertices of Q lie closer to the viewpoint than the plane of P
6. The rasterisation of P and Q do not overlap

Note that the tests are given in order of increasing computational difficulty.

Note also that the polygons must be planar.

If the tests are all false, then the polygons must be split. Splitting is accomplished by selecting one polygon and cutting it along the line of intersection with the other polygon. The above tests are again performed and the algorithm continues until all polygons pass the above tests.

References

- Ivan E. Sutherland, Robert F. Sproull, and Robert A. Schumacker, "A Characterization of Ten Hidden-Surface Algorithms", Computing Surveys, Vol 6, No 1, March 1974
- Newell, M. E., Newell R. G., and Sancha, T. L, "A New Approach to the Shaded Picture Problem", Proc ACM National Conf. 1972

See also

- →Painter's algorithm

Source: http://en.wikipedia.org/wiki/Newell%27s_algorithm

Normal mapping

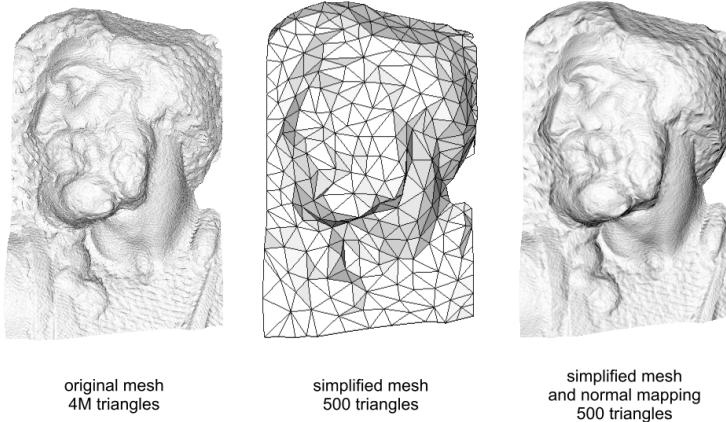


Figure 62 Normal mapping used to re-detail simplified meshes.

In →3D computer graphics, **normal mapping** is an application of the technique known as bump mapping. Normal mapping is sometimes referred to as "Dot3 bump mapping". While bump mapping perturbs the existing normal (the way the surface is facing) of a model, normal mapping replaces the normal entirely. Like bump mapping, it is used to add details to shading without using more polygons. But where a bump map is usually calculated based on a single-channel (interpreted as grayscale) image, the source for the normals in normal

mapping is usually a multichannel image (that is, channels for "red", "green" and "blue" as opposed to just a single color) derived from a set of more detailed versions of the objects.

Normal mapping is usually found in two varieties: object-space and tangent-space normal mapping. They differ in coordinate systems in which the normals are measured and stored.

One of the most interesting uses of this technique is to greatly enhance the appearance of a low poly model exploiting a normal map coming from a high resolution model. While this idea of taking geometric details from a high resolution model had been 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, its application to more common triangle meshes came later. In 1998 two papers were presented with the idea of transferring details as normal maps from high to low poly meshes: "Appearance Preserving Simplification", by Cohen et al. SIGGRAPH 1998, and "A general method for recovering attribute values on simplified meshes" by Cignoni et al. IEEE Visualization '98. The former presented a particular constrained simplification algorithm that during the simplification process tracks how the lost details should be mapped over the simplified mesh. The latter presented a simpler approach that decouples the high and low polygonal mesh and allows the recreation of the lost details in a way that is not dependent on how the low model was created. This latter approach (with some minor variations) is still the one used by most of the 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. Many other lighting models also involve some sort of dot product with the normal vector. Imagine a polygonal model of a sphere - you can only approximate the shape of the surface. By using an RGB bitmap textured across the model, more detailed normal vector information can be encoded. Each color channel in the bitmap (red, green and blue) 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.

Normal mapping in computer entertainment

Interactive normal map rendering was originally only possible on PixelFlow, a parallel graphics 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 frame buffer operations or on low end PC hardware with some tricks using palettes textures. However, with the increasing processing power and sophistication of home PCs and gaming consoles, normal mapping has spread to the public consciousness through its use in several high-profile games, including: *Far Cry* (Crytek), *Deus Ex: Invisible War* (Eidos Interactive), *Thief: Deadly Shadows* (Eidos Interactive), *The Chronicles of Riddick: Escape from Butcher Bay* (Vivendi Universal), *Halo 2* (Microsoft), and *Doom 3* (id Software), *Half-Life 2* (Valve Software), *Call of Duty 2* (Activision), and *Tom Clancy's Splinter Cell: Chaos Theory* (Ubisoft). It is also used extensively in the upcoming third version of the Unreal engine (Epic Games). Normal mapping's increasing popularity amongst video-game designers is due to its combination of excellent graphical quality and decreased processing requirements versus other methods of producing similar effects. This decreased processing requirement translates into better performance and is made possible by distance-indexed detail scaling, a technique which decreases the detail of the normal map of a given texture (cf. mipmapping). Basically, this means that more distant surfaces require less complex lighting simulation. This in turn cuts the processing burden, while maintaining virtually the same level of detail as close-up textures.

Currently, normal mapping has been utilized successfully and extensively on both the PC and gaming consoles. Initially, Microsoft's Xbox was the only home game console to fully support this effect, whereas other consoles use a software-only implementation as they don't support it directly on hardware. Next generation consoles such as the Xbox360 rely heavily on normal mapping, and even use parallax mapping.

See also

- →Bump mapping
- →Parallax mapping
- →Displacement mapping
- Linear algebra

External links

- GIMP normalmap plugin¹⁹⁵
- Photoshop normalmap plugin¹⁹⁶
- normal mapping tutorials¹⁹⁷, Ben Cloward
- Free xNormal normal mapper tool¹⁹⁸, Santiago Orgaz
- Maya normal mapping plugin¹⁹⁹, Olivier Renouard
- Normal Mapping with palettes textures²⁰⁰ using old OpenGL extensions.
- Normal Mapping without hardware assistance²⁰¹, Lux aeterna luceat eis Amiga demo from Ephidrena
- ZMapper²⁰², Pixologic

Bibliography

- Fitting Smooth Surfaces to Dense Polygon Meshes²⁰³, Krishnamurthy and Levoy, SIGGRAPH 1996
- (PDF) Appearance-Preserving Simplification²⁰⁴, Cohen et. al, SIGGRAPH 1998
- (PDF) A general method for recovering attribute values on simplified meshes²⁰⁵, Cignoni et al, IEEE Visualization 1998
- (PDF) Realistic, Hardware-accelerated Shading and Lighting²⁰⁶, Heidrich and Seidel, SIGGRAPH 1999

Source: http://en.wikipedia.org/wiki/Normal_mapping

Principal Authors: Xmnemonic, Tommstein, AlistairMcMillan, CobbSalad, ALooPingIcon, Andrew pmk

¹⁹⁵ <http://nifelheim.dyndns.org/%7ecocidius/normalmap/>
¹⁹⁶ http://developer.nvidia.com/object/photoshop_dds_plugins.html
¹⁹⁷ http://www.monitorstudios.com/bcloward/tutorials_normal_maps1.html
¹⁹⁸ <http://www.santyesprogramadorynografista.net/projects.aspx>
¹⁹⁹ http://www.drone.org/tutorials/displacement_maps.html
²⁰⁰ <http://vcg.isti.cnr.it/activities/geometrygraphics/bumpmapping.html>
²⁰¹ <http://ada.untergrund.net/showdemo.php?demoid=534&pv=1#Comments>
²⁰² <http://206.145.80.239/zbc/showthread.php?t=031281>
²⁰³ <http://www-graphics.stanford.edu/papers/surfacefitting/>
²⁰⁴ <http://www.cs.unc.edu/~geom/APS/APS.pdf>
²⁰⁵ <http://vcg.isti.cnr.it/publications/papers/rocchini.pdf>
²⁰⁶ <http://www.cs.ubc.ca/~heidrich/Papers/Siggraph.99.pdf>

OpenGL

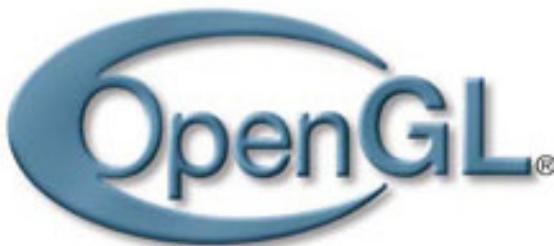


Figure 63 OpenGL official logo

OpenGL (Open Graphics Library) is a standard specification defining a cross-language cross-platform API for writing applications that produce →3D computer graphics (and 2D computer graphics as well). The interface consists of over 250 different function calls which can be used to draw complex three-dimensional scenes from simple primitives. OpenGL was developed by Silicon Graphics and is popular in the video games industry where it competes with →Direct3D on Microsoft Windows platforms (see Direct3D vs. OpenGL). OpenGL is widely used in CAD, virtual reality, scientific visualization, information visualization, flight simulation and video game development.

Specification

At its most basic level, OpenGL is a specification, meaning it is simply a document that describes a set of functions and the precise behaviours that they must perform. From this specification, hardware vendors create implementations — libraries of functions created to match the functions stated in the OpenGL specification, making use of hardware acceleration where possible. Hardware vendors have to meet specific tests to be able to qualify their implementation as an OpenGL implementation.

Efficient vendor-supplied implementations of OpenGL (making use of graphics acceleration hardware to a greater or lesser extent) exist for Mac OS, Windows, Linux, many Unix platforms and PlayStation 3. Various software implementations exist, bringing OpenGL to a variety of platforms that do not have vendor support. Notably, the open source library Mesa is a fully software-based

graphics API which is code-compatible with OpenGL. However to avoid licensing costs associated with formally calling itself an OpenGL implementation, it claims merely to be a "very similar" API.

The OpenGL specification is currently overseen by the OpenGL Architecture Review Board (ARB), which was formed in 1992. The ARB consists of a set of companies with a vested interest in creating a consistent and widely available API. Voting members of the ARB as of April 2006 include 3D hardware manufacturers SGI, 3Dlabs, ATI Technologies, NVIDIA and Intel, and computer manufacturers IBM, Apple Computer, Dell, and Sun Microsystems. Microsoft, one of the founding members, left in March 2003. Aside from these corporations, each year many other companies are invited to be part of the OpenGL ARB for one-year terms. With so many companies involved with such a diverse set of interests, OpenGL has become very much a general-purpose API with a wide range of capabilities.

According to current plans, control of OpenGL will pass to the Khronos Group by the end of 2006. This is being done in order to improve the marketing of OpenGL, and to remove barriers between the development of OpenGL and →OpenGL ES.

Kurt Akeley and Mark Segal authored the original OpenGL specification. Chris Frazier edited version 1.1. Jon Leech edited versions 1.2 through the present version 2.0.

Design

OpenGL serves two main purposes:

- To hide the complexities of interfacing with different 3D accelerators, by presenting the programmer with a single, uniform API.
- To hide the differing capabilities of hardware platforms, by requiring that all implementations support the full OpenGL feature set (using software emulation if necessary).

OpenGL's basic operation is to accept primitives such as points, lines and polygons, and convert them into pixels. This is done by a graphics pipeline known as the OpenGL state machine²⁰⁷. Most OpenGL commands either issue primitives to the graphics pipeline, or configure how the pipeline processes these primitives. Prior to the introduction of OpenGL 2.0, each stage of the pipeline performed a fixed function and was configurable only within tight limits but in OpenGL 2.0 several stages are fully programmable using →GLSL.

²⁰⁷ <http://www.opengl.org/documentation/specs/version1.1/state.pdf>

OpenGL is a low-level, procedural API, requiring the programmer to dictate the exact steps required to render a scene. This contrasts with descriptive (aka scene graph or retained mode) APIs, where a programmer only needs to describe a scene and can let the library manage the details of rendering it. OpenGL's low-level design requires programmers to have a good knowledge of the graphics pipeline, but also gives a certain amount of freedom to implement novel rendering algorithms.

OpenGL has historically been influential on the development of 3D accelerators, promoting a base level of functionality that is now common in consumer-level hardware:

- Rasterised points, lines and polygons as basic primitives
- A transform and lighting pipeline
- →Z-buffering
- →Texture mapping
- Alpha blending

Many modern 3D accelerators provide functionality far above this baseline, but these new features are generally enhancements of this basic pipeline rather than radical reinventions of it.

Example

We first clear the color buffer, in order to start with a blank canvas:

```
glClear( GL_COLOR_BUFFER_BIT );
```

We now set the modelview matrix, which controls the position of the camera relative to the primitives we render. We move it backwards 3 units along the Z axis, which leaves it pointing towards the origin:

```
glMatrixMode( GL_MODELVIEW );           /* Subsequent matrix com-
mands will affect the modelview matrix */
glLoadIdentity();                      /* Initialise the modelview
to identity */
glTranslatef( 0, 0, -3 );             /* Translate the modelview 3
units along the Z axis */
```

The projection matrix governs the perspective effect applied to primitives, and is controlled in a similar way to the modelview matrix:

```

glMatrixMode( GL_PROJECTION );      /* Subsequent matrix com-
mands will affect the projection matrix */
glLoadIdentity();                  /* Initialise the projection
matrix to identity */
glFrustum( -1, 1, -1, 1, 1, 1000 ); /* Apply a perspective-
projection matrix */

```

Finally, we issue a polygon - a green square oriented in the XY plane:

```

glBegin( GL_POLYGON );           /* Begin issuing a polygon
*/
	glColor3f( 0, 1, 0 );           /* Set the current color to
green */
 glVertex3f( -1, -1, 0 );        /* Issue a vertex */
 glVertex3f( -1, 1, 0 );         /* Issue a vertex */
 glVertex3f( 1, 1, 0 );          /* Issue a vertex */
 glVertex3f( 1, -1, 0 );         /* Issue a vertex */
 glEnd();                      /* Finish issuing the poly-
gon */

```

Documentation

Part of the popularity of OpenGL is the excellence of its official documentation. The OpenGL ARB released a series of manuals along with the specification which have been updated to track changes in the API. These are almost universally known by the colors of their covers:

- The Red Book - The OpenGL Programmer's guide. ISBN 0321335732
- : A readable tutorial and reference book - this is a 'must have' book for OpenGL programmers.
- The Blue Book - The OpenGL Reference manual. ISBN 032117383X
- : Essentially a hard-copy printout of the man pages for OpenGL.
- : Includes a poster-sized fold-out diagram showing the structure of an idealised OpenGL implementation.
- The Green Book - Programming OpenGL for the X Window System. ISBN 0201483599
- : A book about X11 interfacing and GLUT.
- The Alpha Book (which actually has a white cover) - OpenGL Programming for Windows 95 and Windows NT. ISBN 0201407094
- : A book about interfacing OpenGL to Microsoft Windows.

Then, for OpenGL 2.0 and beyond:

- The Orange Book - The OpenGL Shading Language. ISBN 0321334892
- : A readable tutorial and reference book for GLSL.

Extensions

The OpenGL standard allows individual vendors to provide additional functionality through *extensions* as new technology is created. Extensions may introduce new functions and new constants, and may relax or remove restrictions on existing OpenGL functions. Each vendor has an alphabetic abbreviation that is used in naming their new functions and constants. For example, NVIDIA's abbreviation (NV) is used in defining their proprietary function `glCombinerParameterfvNV()` and their constant `GL_NORMAL_MAP_NV`. It may happen that more than one vendor agrees to implement the same extended functionality. In that case, the abbreviation *EXT* is used. It may further happen that the Architecture Review Board "blesses" the extension. It then becomes known as a *standard extension*, and the abbreviation *ARB* is used. The first ARB extension was `GL_ARB_multitexture`, introduced in version 1.2.1. Following the official extension promotion path, multitexturing is no longer an optionally implemented ARB extension, but has been a part of the OpenGL core API since version 1.3.

Before using an extension a program must first determine its availability, and then obtain pointers to any new functions the extension defines. The mechanism for doing this is platform-specific and libraries such as →GLEW and →GLEE exist to simplify the process.

Specifications for nearly all extensions can be found at the official extension registry²⁰⁸.

Associated utility libraries

Several libraries are built on top of or beside OpenGL to provide features not available in OpenGL itself. Libraries such as →GLU can always be found with OpenGL implementations, and others such as GLUT and SDL have grown over time and provide rudimentary cross platform windowing and mouse functionality and if unavailable can easily be downloaded and added to a development environment. Simple graphical user interface functionality can be found in libraries like →GLUI or FLTK. Still others libraries like AUX are deprecated libraries that have been superseded by functionality commonly available in more popular libraries, but code still exists out there particularly in simple tutorials. Other libraries have been created to provide OpenGL application developers a simple means of managing OpenGL extensions and versioning, examples of

²⁰⁸ <http://oss.sgi.com/projects/ogl-sample/registry/>

these libraries include →GLEW "The OpenGL Extension Wrangler Library" and →GLEE "The OpenGL Easy Extension library".

In addition to the aforementioned simple libraries other higher level object oriented scene graph retained mode libraries exist such as →PLIB, OpenScene-Graph, and OpenGL Performer, these are available as cross platform Open Source or proprietary programming interfaces written on top of OpenGL and systems libraries to enable the creation of real-time visual simulation applications.

→Mesa 3D ⁽²⁰⁹⁾ is an Open Sourced implementation of OpenGL. It supports pure software rendering as well as providing hardware acceleration for several 3D graphics cards under Linux. As of February 2 2006 it implements the 1.5 standard, and provides some of its own extensions for some platforms.

Bindings

In order to emphasize its multi-language and multi-platform characteristics, various bindings and ports have been developed for OpenGL in many languages. Most notably, the Java 3D library can rely on OpenGL for its hardware acceleration. Direct bindings are also available like the Lightweight Java Game Library²¹⁰ which has a direct binding of OpenGL for Java and other game related components. Very recently, Sun has released beta versions of the JOGL system, which provides direct bindings to C OpenGL commands, unlike Java 3D which does not provide such low level support. The OpenGL official page ²¹¹ lists various bindings for Java, Fortran 90, Perl, Pike, Python, Ada, and Visual Basic. Bindings are also available for C++ and C#, see ²¹².

Higher level functionality

OpenGL was designed to be graphic output-only: it provides only rendering functions. The core API has no concept of windowing systems, audio, printing to the screen, keyboard/mouse or other input devices. While this seems restrictive at first, it allows the code that does the rendering to be completely independent of the operating system it is running on, allowing cross-platform development. However some integration with the native windowing system is required to allow clean interaction with the host system. This is performed through the following add-on APIs:

²⁰⁹ <http://www.mesa3d.org>

²¹⁰ <http://lwjgl.org/>

²¹¹ <http://www.opengl.org/>

²¹² <http://www.exocortex.org/3dengine/>

- GLX - X11 (including network transparency)
- WGL - Microsoft Windows

Additionally the GLUT and SDL libraries provide functionality for basic windowing using OpenGL, in a portable manner. Mac OS X has three APIs to get OpenGL support: AGL for Carbon, NSOpenGL for Cocoa and CGL for lower-level access.

History

Today the digital generation of animated scenes in three dimensions is a regular fixture in everyday life. Scientists utilize computer graphics to analyze simulations of every possibility. Engineers and architects design virtual models using computer graphics. Movies employ computer graphics to create stunning special effects or entire animated films. And over the past few years, computer games have brought computer graphics technology to regular consumers, using graphics to bring their players into worlds that could never exist.

Bringing digital graphics technology to such widespread use was not without its challenges. Fifteen years ago, developing software that could function with a wide range of graphics hardware and all of their different interfaces was time consuming. Each team of programmers developed interfaces separately, and there was consequently much duplicated code. This was hindering the growing industry of computer graphics.

Silicon Graphics Incorporated specializes in the creation of specialized graphics hardware and software. By the early 1990's, SGI had become the world leader in 3D graphics, and its programming API, IrisGL, had become a defacto industry standard, over-shadowing the open-standards-based PHIGS. There were several reasons for the market superiority: SGI usually had the best and fastest hardware; the IrisGL programming interface (API) was elegant, easy-to-use, and, importantly, supported immediate-mode rendering. By contrast, PHIGS was clunky, hard to use, and was several generations behind IrisGL in function and capability, primarily due to the dysfunctional PHIGS standardization process. None-the-less, competing vendors, including Sun, Hewlett-Packard and IBM were able to bring to market credible 3D hardware, supported by proprietary extensions to PHIGS. By the early 90's, 3D graphics hardware technology was fairly well understood by a large number of competitors and was no longer a discriminating factor in computer systems purchases. Thus, rather than prolonging a contentious and dangerous fight between IrisGL and PHIGS, SGI sought to turn a defacto standard into a true open standard.

The IrisGL API itself wasn't suitable for opening (although it had been previously licensed to IBM and others), in part because it had accumulated cruft over

the years. For example, it included a windowing, keyboard and mouse API, in part because it was developed before the X11 Window System versus Sun's NeWS battle had resolved. Thus, the API to be opened needed to be cleaned up. In addition, IrisGL had a large software vendor (ISV) portfolio; the change to the OpenGL API would keep ISV's locked onto SGI (and IBM) hardware for a few years while market support for OpenGL matured. Meanwhile, SGI would continue to try to maintain a vendor lock by pushing the higher-level and proprietary Iris Inventor and Iris Performer programming API's.

The result is known as **OpenGL**. OpenGL standardised access to hardware, and pushed the development responsibility of hardware interface programs, sometimes called device drivers, to hardware manufacturers and delegated windowing functions to the underlying operating system. With so many different kinds of graphic hardware, getting them all to speak the same language in this way had a remarkable impact by giving software developers a higher level platform for 3D-software development.

In 1992, SGI led the creation of the OpenGL architectural review board (OpenGL ARB), the group of companies that would maintain and expand the OpenGL specification for years to come. OpenGL evolved from (and is very similar in style to) SGI's earlier 3D interface, *IrisGL*. One of the restrictions of IrisGL was that it only provided access to features supported by the underlying hardware. If the graphics hardware did not support a feature, then the application could not use it. OpenGL overcame this problem by providing support in software for features unsupported by hardware, allowing applications to use advanced graphics on relatively low-powered systems.

In 1994 SGI played with the idea of releasing something called "→OpenGL++" which included elements such as a scene-graph API (presumably based around their Performer technology). The specification was circulated among a few interested parties - but never turned into a product.

When →Direct3D was released in 1995, Microsoft, SGI, and Hewlett-Packard initiated the Fahrenheit project, which was a joint effort with the goal of unifying the OpenGL and Direct3D interfaces - and again, adding a scene-graph API. It initially showed some promise of bringing order to the world of interactive 3D computer graphics APIs, but on account of financial constraints at SGI and general lack of industry support it was abandoned. The engineers involved at SGI held a beach party in celebration - complete with bonfires on which they burned piles of Fahrenheit documentation.

OpenGL 2.0

OpenGL 2.0 was conceived by 3Dlabs to address concerns that OpenGL was stagnating and lacked a strong direction. 3Dlabs proposed a number of major additions to the standard, the most significant of which was →GLSL (the OpenGL Shading Language, also *slang*). This would enable the programmer to replace the OpenGL fixed-function vertex and fragment pipelines with shaders written in a C-like language, massively expanding the range of graphical effects possible. GLSL was notable for making relatively few concessions to the limitations of the hardware then available; this harkened back to the earlier tradition of OpenGL setting an ambitious, forward-looking target for 3D accelerators rather than merely tracking the state of currently available hardware. The final OpenGL 2.0 specification ²¹³ includes support for GLSL, but omits many of the other features originally proposed which were deferred to later versions of OpenGL (although many are now available as extensions).

Sample renderings

See also

- →GLSL - OpenGL's shading language
- Cg - nVidia's shading language that works with OpenGL
- Java OpenGL - Java bindings for OpenGL
- →OpenGL ES - OpenGL for embedded systems
- OpenAL - The Open Audio Library - designed to work well with OpenGL.
- OpenGL ES - Another audio library.
- →Graphics pipeline

OpenGL support libraries

- GLUT - The OpenGL utility toolkit.
- →GLU - Some additional functions for OpenGL programs.

Other graphics API's

- →Mesa 3D - An OpenSourced implementation of OpenGL.
- →Direct3D and →Comparison of Direct3D and OpenGL - A competitor to OpenGL.

²¹³ <http://www.opengl.org/documentation/specs/version2.0/glspec20.pdf>

- Light Weight Java Game Library
- VirtualGL

Further reading

- Richard S. Wright Jr. and Benjamin Lipchak: *OpenGL Superbible*, Third Edition, Sams Publishing, 2005, ISBN 0-67232-601-9
- Astle, Dave and Hawkins, Kevin: *Beginning OpenGL Game Programming*, Course Technology PTR, ISBN 1-59200-369-9
- Fosner, Ron: *OpenGL Programming for Windows 95 and Windows NT*, Addison Wesley, ISBN 0-20140-709-4
- Kilgard, Mark: *OpenGL for the X Window System*, Addison-Wesley, ISBN 0-20148-359-9
- Lengyel, Eric: *The OpenGL Extensions Guide*, Charles River Media, ISBN 1-58450-294-0
- OpenGL Architecture Review Board, et al: *OpenGL Reference Manual: The Official Reference Document to OpenGL*, Version 1.4, Addison-Wesley, ISBN 0-32117-383-X
- OpenGL Architecture Review Board, et al: *OpenGL Programming Guide: The Official Guide to Learning OpenGL*, Version 2, Fifth Edition, Addison-Wesley, ISBN 0-32133-573-2
- Rost, Randi J.: *OpenGL Shading Language*, Addison-Wesley, ISBN 0-32119-789-5

External links

- Official website²¹⁴
- Official OpenGL wiki²¹⁵
- SGI's OpenGL website²¹⁶
- Red Book²¹⁷ - Online copy of an early edition of The OpenGL Programming Guide.
- Blue Book²¹⁸ - Online copy of an early edition of The OpenGL Reference Manual.
- MSDN²¹⁹ - Microsoft's OpenGL reference

²¹⁴ <http://www.opengl.org/>

²¹⁵ <http://www.opengl.org/wiki>

²¹⁶ <http://www.sgi.com/products/software/opengl/>

²¹⁷ http://www.opengl.org/documentation/red_book/

²¹⁸ http://www.opengl.org/documentation/blue_book/

²¹⁹ http://msdn.microsoft.com/library/default.asp?url=/library/en-us/opengl/apxb4_82lh.asp

- OpenGL 2.0 specification²²⁰ (PDF)
- OpenGL tutorials, software, etc²²¹ at the Open Directory Project

Source: <http://en.wikipedia.org/wiki/OpenGL>

Principal Authors: SteveBaker, Owaldo, Klassobanieras, Haakon, Imroy, Robertbowering, Stoat-Bringer, BSTRhino, Fresheneesz, Flamurai

OpenGL++

OpenGL++ was intended to be a powerful layer above the →OpenGL 3D graphics system written in C++ that supported object-oriented data structures. The project started as the result of a partnership between SGI, IBM and Intel (and later Digital Equipment Corporation as well) to provide a higher level API than the "bare metal" support of OpenGL. Work on OpenGL++ ended when SGI decided to partner with Microsoft instead, leading to the Fahrenheit project, which also died.

Development

OpenGL++ (OGL++) was intended to offer a selection of routines and standardized data structures to dramatically simplify writing "real" programs using OpenGL. Instead of the programmer having to keep track of the objects in the 3D world and make sure they were culled properly, OpenGL++ would include its own scene graph system and handle many of the basic manipulation duties for the programmer. In addition, OGL++ included a system for modifying the scene graph on the fly, re-arranging it for added performance.

Much of OGL++ was a combination of ideas from earlier SGI projects in the same vein, namely →Open Inventor which offered ease-of-use, and OpenGL Performer which was written separately from Inventor to deliver a system that optimized scene graphs for increased performance and exploited scalable architectures. It was later intended that a new design could get the best of both worlds while forming the underlying framework for several projects including CAD, image processing, visual simulation, scientific visualization and user interfaces or 3D manipulators allowing them to interoperate, thereby offering both rapid development and high performance.

²²⁰ <http://www.opengl.org/documentation/specs/version2.0/glspec20.pdf>

²²¹ <http://dmoz.org/Computers/Programming/Graphics/Libraries/OpenGL/>

SGI had already almost completed one effort to merge the functionality of scene graphsCosmo 3D, Cosmo 3D was in fact the spinoff from an earlier collaboration with Sun which was supposed to produce a scene graph for Java in conjunction with SGI's new scene graph, Sun and SGI went their separate ways with Java3D and Cosmo3D. When SGI announced the OGL++ effort, they halted development of Cosmo3D when it had just reached a beta release. By then a CAD/"Large Model Visualization" layer of functionality called OpenGL Optimizer had already been implemented on Cosmo3D and then released as a product. Other "front end" packages like, Cosmo Code, a VRML authoring tool, were produced by a different division and use of the Cosmo name was merely part of a broader marketing strategy, it ran on OpenGL. OGL++ was intended to be a cleaned up and more flexible version of Cosmo3D, most of the Cosmo3D team started work on OGL++ and a lot of the effort was aimed at a specification and implementation that could deliver on the promise of a truly powerful yet generic scene graph.

At the end

In the end, there is little to show for any of these efforts. Partnerships with Sun Microsystems, Intel and IBM and Microsoft all led to nothing as SGI jumped from project to project. In retrospect, SGI reacted badly to a rapidly changing environment. An internal desire to create a new improved generic but extensible scene graph was constantly sidetracked by a belief that SGI couldn't go it alone. Partnerships were formed and later abandoned due to irreconcilable differences or simply as priorities and internal pressures shifted. OGL++ was the most nascent of these efforts and although it was the option that rapidly gained the strongest interest the power of the idea forced an unholy alliance between Microsoft and SGI in the form of Fahrenheit, SGI joining because of its long held belief that it couldn't go it alone and Microsoft because it wanted to avert the possibility of a truly open 3D scene graph. Ancillary issues like powerful CAD APIs running on Cosmo3D complicated the picture. In the final analysis the new unified scene graph concept was bounced from project to project, and eventually died in 2000 when Fahrenheit was killed.

Today, no such standardized scene graph exists, and SGI has all but exited the API world. SGI has released the earlier Open Inventor code into open source, but the source to OGL++ was never completed to any satisfactory degree. No specification exists and as with OpenGL the spec and idea behind such an open platform would have been what lent it its lasting value, not a single implementation of a scene graph idea.

Source: <http://en.wikipedia.org/wiki/OpenGL%2B%2B>

Principal Authors: Maury Markowitz, Tim Starling, Michael Hardy, Heron, Engwar, Zondor, Jwestbrook, Nikola Smolenski

OpenGL ES

OpenGL ES (OpenGL for Embedded Systems) is a subset of the →OpenGL 3D graphics API designed for embedded devices such as mobile phones, PDAs, and video game consoles. It is defined and promoted by the Khronos Group, graphics hardware and software industry consortium interested in open APIs for graphics and multimedia.

In creating OpenGL ES 1.0 much functionality has been stripped from the original OpenGL API and a little bit added, two of the more significant differences between OpenGL ES and OpenGL are the removal of the `glBegin`–`glEnd` calling semantics for primitive rendering (in favor of vertex arrays) and the introduction of fixed-point data types for vertex coordinates and attributes to better support the computational abilities of embedded processors, which often lack an FPU. Many other areas of functionality have been removed in version 1.0 to produce a lightweight interface: for example, quad and polygon primitive rendering, texgen, line and polygon stipple, polygon mode, antialiased polygon rendering (with alpha border fragments, not multisample), ARB_Image class pixel operation functionality, bitmaps, 3D texture, drawing to the frontbuffer, accumulation buffer, copy pixels, evaluators, selection, feedback, display lists, push and pop state attributes, two-sided lighting, and user defined clip planes.

Several versions of the OpenGL ES specification now exist. OpenGL ES 1.0 is drawn up against the OpenGL 1.3 specification, OpenGL ES 1.1 is defined relative to the OpenGL 1.5 specification and OpenGL ES 2.0 is defined relative to the OpenGL 2.0 specification. Version 1.0 and 1.1 both have *common* and *common lite* profiles, the difference being that the common lite profile only supports fixed-point in lieu of floating point data type support, whereas common supports both.

OpenGL ES 1.1 adds to the OpenGL ES 1.0 functionality by introducing additional features such as mandatory support for multitexture, better multitexture support (with combiners and dot product texture operations), automatic mipmap generation, vertex buffer objects, state queries, user clip planes, and greater control over point rendering.

The common profile for OpenGL ES 2.0, publicly released in August 2005, completely eliminates all fixed-function API support in favor of an entirely programmable model, so features like the specification of surface normals in the

API for use in a lighting calculation are eliminated in favor of abstract variables, the use of which is defined in a shader written by the graphics programmer.

OpenGL ES also defines an additional safety-critical profile that is intended to be testable and demonstrably robust subset for safety-critical embedded applications such as glass cockpit avionics displays.

OpenGL ES has been chosen as the official graphics API used for PlayStation 3 gaming platform development and as the official 3D graphics API in Symbian OS.

Further reading

- Astle, Dave and David Durnil: *OpenGL ES Game Development*, Course Technology PTR, ISBN 1-592-00370-2

External links

- Official website²²²
- Open Source Software Implementation of OpenGL ES 1.0²²³
- ALT Software's Commercial Implementation of OpenGL ES²²⁴
- A Swedish wiki dedicated to OpenGL/OpenGL ES and the development round it.²²⁵

Source: http://en.wikipedia.org/wiki/OpenGL_ES

Principal Authors: Flamurai, Gaius Cornelius, Nova77, Cmdrjameson, Orderud

²²² <http://www.khronos.org/opengles/>

²²³ <http://ogl-es.sourceforge.net/>

²²⁴ <http://www.altsoftware.com/products/opengl/>

²²⁵ <http://www.opengl.se>

OpenGL Utility Toolkit

This article is about the OpenGL toolkit. GLUT can also stand for glucose transporter.

The →**OpenGL Utility Toolkit (GLUT)** is a library of utilities for OpenGL programs, which primarily perform system-level I/O with the host operating system. Functions performed include window definition, window control, and monitoring of keyboard and mouse input. Routines for drawing a number of geometric primitives (both in solid and wireframe mode) are also provided, including cubes, spheres, and the →Utah teapot. GLUT even has some limited support for creating pop-up windows.

GLUT was written by Mark J. Kilgard, author of *OpenGL Programming for the X Window System* and *The CG Tutorial: The Definitive Guide to Programmable Real-Time Graphics* while he was working for Silicon Graphics Inc.

The two aims of GLUT are to allow the creation of rather portable code between operating systems (GLUT is cross-platform) and to make learning OpenGL easier. Getting started with OpenGL programming while using GLUT often takes only a few lines of code and requires no knowledge of operating system-specific windowing APIs.

All GLUT functions start with the glut prefix (for example, glutPostRedisplay rerenders the current screen).

freeglut and its spin-off, OpenGLUT, are free software alternatives to GLUT. Freeglut attempts to be a fairly exact clone, OpenGLUT adds a number of new features to the API. Both have the advantage of licensing that permits users to modify and redistribute the library.

See also

- →GLU
- →GLUI

External links

- GLUT documentation²²⁶
- OpenGLUT²²⁷
- FreeGLUT²²⁸

²²⁶ <http://www.opengl.org/documentation/specs/glut/spec3/spec3.html>

²²⁷ <http://openglut.sourceforge.net/>

Source: http://en.wikipedia.org/wiki/OpenGL.Utility_Toolkit

Principal Authors: SteveBaker, Orderud, Elie De Brauwer, MarSch, Bogdangiusca

Open Inventor

Open Inventor, originally **IRIS Inventor**, is a C++ object oriented “retained mode” 3D graphics API designed by SGI to provide a higher layer of programming for →OpenGL. Its main goals are better programmer convenience and efficiency.

Early history

Around 1988–1989, Wei Yen asked Rikk Carey to lead the IRIS Inventor project. Their goal was to create a toolkit that made developing 3D graphics applications easier to do. The strategy was based on the premise that people were not developing enough 3D applications with OpenGL because it was too time-consuming to do so with the low-level interface provided by OpenGL. If 3D programming were made easier, through the use of an object oriented API, then more people would create 3D applications and SGI would benefit. Therefore, the credo was always “ease of use” before “performance”, and soon the tagline “3D programming for humans” was being used widely.

Raison d'être

OpenGL (OGL) is a low level library that takes lists of simple polygons and renders them as quickly as possible. To do something more practical like “draw a house”, the programmer must break down the object into a series of simple OGL instructions and send them into the engine for rendering. One problem is that OGL performance is highly sensitive to the way these instructions are sent into the system, requiring the user to know which instructions to send and in which order, and forcing them to carefully cull the data to avoid sending in objects that aren't even visible in the resulting image. For simple programs a tremendous amount of programming has to be done just to get started.

Open Inventor (OI) was written to address this issue, and provide a common base layer to start working with. Objects could be subclassed from a number of pre-rolled shapes like cubes and polygons, and then easily modified into new shapes. The “world” to be drawn was placed in a scene graph run by OI, with

²²⁸ <http://freeglut.sourceforge.net/>

the system applying occlusion culling on objects in the graph automatically. OI also included a number of controller objects and systems for applying them to the scene, making common interaction tasks easier. Finally, OI also supplied a common file format for storing “worlds”, and the code to automatically save or load a world from these files. Basic 3D applications could then be written in a few hundred lines under OI, by tying together portions of the toolkit with “glue” code.

On the downside OI tended to be slower than hand-written code, as 3D tasks are notoriously difficult to make perform well without shuffling the data in the scene graph by hand. Another practical problem was that OI could only be used with its own file format, forcing developers to write converters to and from the internal system.

TGS Open Inventor

Open Inventor was later opened for 3rd-party licensing, which is when it switched from “IRIS” to “Open”. It was licensed to two third party developers, Template Graphics Software (TGS)²²⁹ and Portable Graphics. TGS later bought Portable Graphics, making them the sole licensee. In 2004 TGS was acquired by Mercury Computer Systems²³⁰, who continue to develop and support OI.

Performer

About a year into the Inventor project, a different philosophy began to emerge. Instead of simply making it easy to write applications on SGI systems, now the goal was to make it difficult to write slow applications. Members of the Inventor team left to form their own group, forming the basis of the OpenGL Performer project (then still known as IRIS Performer). Performer was also based on an internal scene graph, but was allowed to modify it for better speed as it saw fit, even dropping “less important” objects and polygons in order to maintain guaranteed performance levels. Performer also used a number of processes to run tasks in parallel for added performance, allowing it to be run (in one version) on multiple processors. Unlike Inventor, Performer remained proprietary so that SGI would have the agility to modify the API as needed to keep in step with the latest hardware enhancements.

²²⁹ <http://www.mc.com/tgs>

²³⁰ <http://www.mc.com>

Mid 1990s

At some point in the mid-1990s it was realized that there was no good reason that the two systems could not be combined, resulting in a single high-level API with both performance and programmability. SGI started work on yet another project aimed at merging the two, eventually culminating in Cosmo 3D. However Cosmo had a number of practical problems that could have been avoided with better design.

Eventually all of these ideas would come together to create the OpenGL++ effort, along with Intel, IBM and DEC. Essentially a cleaned up and more “open” version of Cosmo 3D, work on Cosmo ended and SGI turned to OpenGL++ full time. The OpenGL++ effort would drag on and eventually be killed, and SGI then tried again with Microsoft with the similar Fahrenheit project, which also died. During this time SGI ignored OI, and eventually spun it off completely to TGS.

Recent history

After many years of Inventor being solely available under proprietary licensing from TGS, it was released under an open source license in August 2000, which is available from SGI.

At approximately the same time, an API clone library called “Coin”²³¹ was released by the company Systems in Motion²³². The Coin library had been written in a clean room fashion from scratch, sharing no code with the original SGI Inventor library, but implementing the same API for compatibility reasons. Systems in Motion’s Coin library is released under a dual licensing scheme, available both under the GNU GPL (for Free Software development) and a commercially sold license for proprietary software development.

The open source version from SGI has since fallen into obscurity, as SGI has not shown any commitment to do further development of the library, and seems to spend little resources even on maintenance.

Systems in Motion’s Coin library and TGS’s Inventor are still thriving under active development, and both have added numerous improvements to the original Inventor API like extensive support for the VRML standard.

Despite its age, the Open Inventor API is still widely used for a wide range of scientific and engineering visualization systems around the world, having proven itself well designed for effective development of complex 3D application software.

²³¹ <http://www.coin3d.org>

²³² <http://www.sim.no>

External links

- Official SGI Open Inventor Site²³³
- Coin3D: independent implementation of the API²³⁴

Source: http://en.wikipedia.org/wiki/Open_Inventor

Principal Authors: Maury Markowitz, Mortene, RedWolf, ENGIMa, Engwar

OpenRT

The goal of the "OpenRT Real Time Ray Tracing Project" is to develop ray tracing to the point where it offers an alternative to the current rasterization based approach for interactive 3D graphics. Therefore the project consists of several parts: a highly optimized ray tracing core, the OpenRT-API which is similar to →OpenGL and many applications ranging from dynamically animated massive models and global illumination, via high quality prototype visualization to computer games.

External links

- Official site of the OpenRT project²³⁵
- Official site of inTrace GmbH, distributor of OpenRT²³⁶

See also

- ray tracing hardware

Source: <http://en.wikipedia.org/wiki/OpenRT>

²³³ <http://oss.sgi.com/projects/inventor/>

²³⁴ <http://www.coin3d.org>

²³⁵ <http://www.openrt.de/>

²³⁶ <http://www.inTrace.com/>

Painter's algorithm

The **painter's algorithm** 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 at some point necessary to decide which polygons are visible and which are hidden.

The name "painter's algorithm" refers to a simple-minded painter who paints the distant parts of a scene at first and then covers them by those parts which are nearer. The painter's algorithm sorts all the polygons in a scene by their depth and then paints them in this order. It will over-paint the parts that are normally not visible and thus solves the visibility problem.



Figure 64 The distant mountains are painted first, followed by the closer meadows; finally, the closest objects in this scene, the trees, are painted.

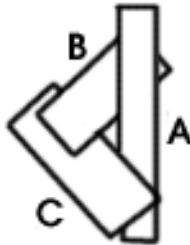


Figure 65 Overlapping polygons can cause the algorithm to fail

The algorithm can fail in certain cases. In this example, Polygons A, B and C overlap each other. It's not possible to decide which polygon is above the others or when two polygons intersect one another in three dimensions. In this case the offending polygons must be cut in some way to allow sorting to occur. Newell's algorithm proposed in 1972 gives a method for cutting such polygons. Numerous methods have also been proposed in the field of computational geometry.

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 made to 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 normal 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 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.

Source: http://en.wikipedia.org/wiki/Painter%27s_algorithm

Principal Authors: Fredrik, Sverdrup, Finell, RadRafe, Bryan Derksen

Parallax mapping

Parallax Mapping (also, **Photonic Mapping**, **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 wooden floorboards) will have more apparent depth and realism with less of an influence on the speed of the game.

Parallax mapping is done by displacing the texture coordinates such that the texture occludes itself in accordance with a height map. Next-generation 3D applications may employ parallax mapping as new graphics algorithms are developed.



Figure 66 From F.E.A.R., a bullet hole with Parallax mapping used to create the illusion of depth.



Figure 67 From F.E.A.R., same as before, but viewed at an angle.

Parallax mapping



Figure 68 From F.E.A.R., when viewed at an extreme angle, the illusion of depth disappears.

An easy way to understand this concept is to close one eye, take a pencil, point it at your eye, and move your head left and right. Parallax mapping takes that pixel on the far left of the pencil when it was facing you directly and stretches it accordingly to simulate your angle in comparison to the pencil.

Parallax mapping is also a way of faking displacement mapping where the actual geometric position of points surface is displaced along the surface normal according to the values stored into the texture: in Parallax mapping, like in normal and bump mapping, the silhouette of the object is unaffected.

See also

- Parallax

External links

- Detailed Shape Representation with Parallax Mapping²³⁷
- Parallax mapping implementation in DirectX, forum topic²³⁸

²³⁷ <http://vrsj.t.u-tokyo.ac.jp/ic-at/ICAT2003/papers/01205.pdf>

²³⁸ http://www.gamedev.net/community/forums/topic.asp?topic_id=387447

Source: http://en.wikipedia.org/wiki/Parallax_mapping

Principal Authors: Tommstein, Tnikkel, Charles Matthews, Jitse Niesen, Thepcnerd

Particle system

This article is about 3D computer graphics. For the computer game developer, see Particle Systems Ltd.

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 to produce realistic game physics. Examples of such phenomena which are commonly done with particle systems include fire, explosions, smoke, flowing water, sparks, falling leaves, clouds, fog, snow, dust, meteor tails, or abstract visual effects like glowy trails, etc.

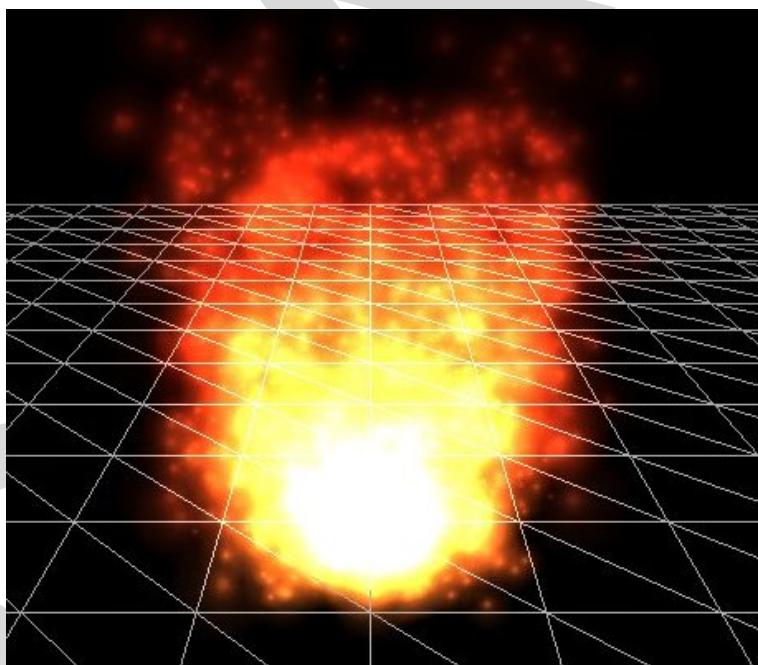


Figure 69 A particle system used to render a fire

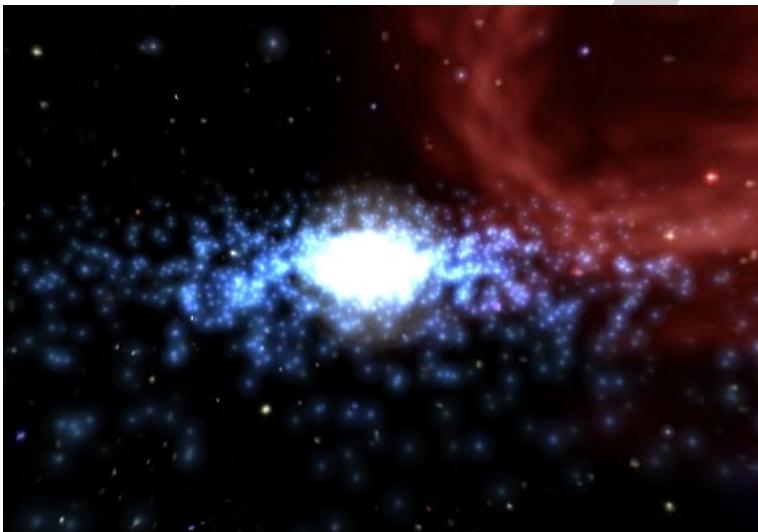


Figure 70 A particle system used to render a galaxy

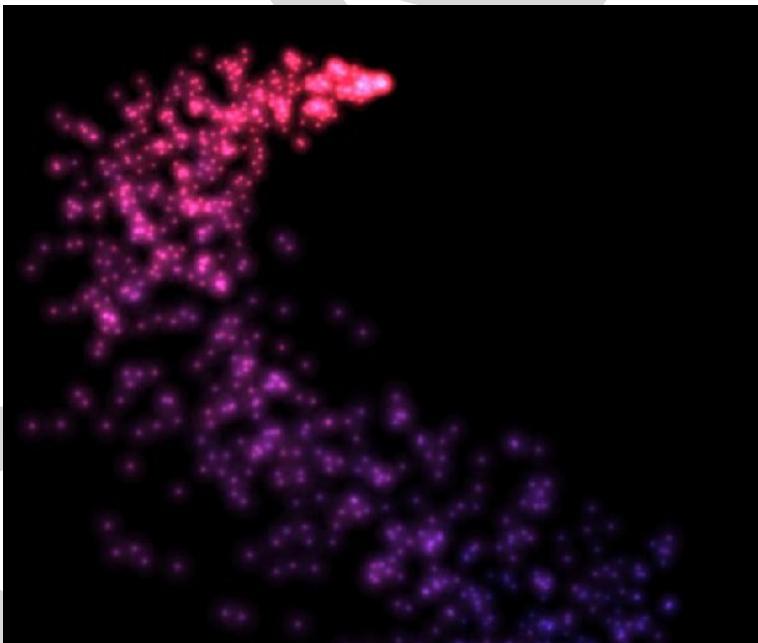


Figure 71 An abstract trail of particles

Particle system

Typical implementation

Typically the particle system's position in 3D space or motion therein is controlled by what is called the **emitter**.

The **emitter** is mainly characterized by a set of particle behavior parameters and a position in 3D space. The particle behavior parameters might include spawning rate (how many particles are generated per unit of time), particle initial velocity vector (i.e. which way will it go when it is emitted), particle life (how much time does each individual particle live before being extinguished), particle color and variations throughout its life, and many more. It is common for all or most of these parameters to be fuzzy, in the sense that instead of absolute values, some central value and allowable random variation is specified.

A typical particle system 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 amount of new particles that must be created is calculated based on spawning rates and interval between updates, and each of them is spawned in a specific position in 3D space based on emitter position and spawning area specified. Also each of the particle's parameters, like velocity, color, life, etc, are initialized based on the emitter's parameters. Then all the existing particles are checked to see if they have exceeded their lifetime, in which case they are removed from the simulation. Otherwise their position and other characteristics are modified based on some sort of physical simulation, which can be as simple as adding the velocity to the current position, and maybe accounting for friction by modulating the velocity, or as complicated as performing physically-accurate trajectory calculations taking into account external forces. Also it is common to perform some sort of collision checking with specified 3D objects in order to make the particles bounce off obstacles in the environment. However particle-particle collisions are rarely used, as they are computationally expensive and not really useful for most of the simulations.

A particle system has its own rules that are applied to every particle. Often these rules involve interpolating values over the lifetime of a particle. For example, many systems have particles fade out to nothingness by interpolating the particle's alpha value (opacity) during the lifetime of the particle.

Rendering stage

After the update is complete, each particle is rendered usually in the form of a textured billboard quad (i.e. a quadrilateral that is always facing the

viewer). However, this is not necessary, the particle may be rendered as just a single pixel in small resolution/limited processing power environments, or even as a metaball in off-line rendering (isosurfaces computed from particle-metaballs make quite convincing liquid surfaces). Finally 3D meshes can also be used to render the particles.

Artist-friendly particle system tools

Particle systems can be created and modified natively in many 3D modeling and rendering packages including 3D Studio Max, Maya and Blender. These editor programs allow artists to have instant feedback on how a particle system might look given properties and rules that they can specify. There is also plug-in software available that provide enhanced particle effects, such as After-Burn and RealFlow (for liquids). Compositing software such as Combustion or specialized, particle-only software such as Particle Studio, can be used for the creation of particle systems for film and video.

External links

- Particle Systems: A Technique for Modeling a Class of Fuzzy Objects²³⁹ – William T. Reeves (ACM Transactions on Graphics, April 1983)

Source: http://en.wikipedia.org/wiki/Particle_system

Principal Authors: Jtsiomb, Ashlux, Mrwojo, Sideris, The Merciful, MarSch

Path Tracing

Path tracing is a technique by James Kajiya when he presented his paper on the Rendering Equation in the 1980s. The main goal of path tracing is to fully solve the rendering equation.

A form of ray tracing whereby each ray is recursively traced along a path until it reaches a light emitting source where the light contribution along the path is calculated. This recursive tracing helps for solving the lighting equation more accurately than conventional ray tracing.

A simple path tracing pseudocode might look something like this:

²³⁹ <http://portal.acm.org/citation.cfm?id=357320>

```

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 cost=DotProduct(newRay.direction,r.normalWhereObjWasHit);

    Color BRDF=m.reflectance/PI;
    float scale=1.0*PI;
    Color reflected=TracePath(newRay,depth+1);

    return emittance + ( BRDF * scale * cost * reflected );
}

```

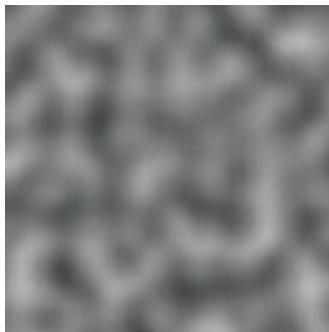
In the above example if every surface of a closed space emitted and reflected (0.5,0.5,0.5) then every pixel in the image would be white.

A variation of this algorithm is to trace rays in the opposite direction, from light sources to the camera, this is called light tracing. Furthermore these two algorithms can be combined to enhance the image results and which is called bi-directional path tracing.

Source: http://en.wikipedia.org/wiki/Path_Tracing

Principal Authors: Icairns, RJHall

Perlin noise



Perlin noise is a function which uses interpolation between a large number of pre-calculated gradient vectors to construct a value that varies pseudo-randomly over space and/or time. It resembles band-limited white noise, and is often used in CGI to make computer-generated objects more natural-looking, by imitating the pseudo-randomness of nature.

It resulted from the work of Ken Perlin, who invented it to generate textures for *Tron*. He won a special Academy Award for Perlin noise in 1997, although *Tron* was denied the 1982 Academy Award for visual effects, because it "cheated" by using computer-generated imagery.

Ken Perlin improved the implementation in 2002, suppressing some visual artifacts. see the links.

Perlin noise is widely used in computer graphics for effects like fire, smoke and clouds. It is also frequently used to generate textures when memory is extremely limited, such as in demos.

See also

- Fractal landscape
- Simplex noise

External links

- a Ken perlin talk on noise, a very nice introduction²⁴⁰
- Ken Perlin's homepage²⁴¹

²⁴⁰ <http://www.noisemachine.com/talk1/>

²⁴¹ <http://mrl.nyu.edu/~perlin/>

- Ken perlin's improved noise (2002) java source code²⁴²
- Ken Perlin's Academy Award page²⁴³
- Matt Zucker's Perlin noise math FAQ²⁴⁴
- A very good explanation and implementation of Perlin noise, with pseudo code²⁴⁵

Source: http://en.wikipedia.org/wiki/Perlin_noise

Principal Authors: Michael Hardy, Charles Matthews, Reedbeta, Maa, Saga City, Gordmoo

Per-pixel lighting

In computer graphics, **per-pixel lighting** is commonly used to refer to a set of methods for computing illumination at each rendered pixel of an image. These generally produce more realistic images than vertex lighting, which only calculates illumination at each vertex of a →3D model and then interpolates the resulting values to calculate the per-pixel color values.

Per-pixel lighting is commonly used with other computer graphics techniques to help improve render quality, including bump mapping, specularity, phong shading and shadow volumes.

Real-time applications, such as computer games, which use modern graphics cards will normally implement per-pixel lighting algorithms using pixel shaders. Per-pixel lighting is also performed on the CPU in many high-end commercial rendering applications which typically do not render at interactive framerates.

See also

- →Phong shading
- Stencil shadow volumes
- →Unified lighting and shadowing

Source: http://en.wikipedia.org/wiki/Per-pixel_lighting

²⁴² <http://mrl.nyu.edu/~perlin/noise/>

²⁴³ <http://www.mrl.nyu.edu/~perlin/doc/oscar.html>

²⁴⁴ <http://www.cs.cmu.edu/~mzucker/code/perlin-noise-math-faq.html>

²⁴⁵ http://freespace.virgin.net/hugo.elias/models/m_perlin.htm

Principal Authors: Mikkalai, Alphonze, Jheriko, David Wahler

Phong reflection model

Not to be confused with →Phong shading.

The **Phong reflection model** is an illumination and shading model, used in →3D computer graphics for assigning shades to points on a modeled surface. It was developed by →Bui Tuong Phong in his University of Utah Ph.D. dissertation "Illumination for Computer Generated Pictures" in 1973, in conjunction with a method for interpolating the calculation for each individual pixel that is rasterized from a polygonal surface model; the interpolation technique is known as →Phong shading, even when it is used with a reflection model other than Phong's.

The Phong reflection model can be treated as a simplification of the more general rendering equation; it takes advantage of the following simplifications when deciding the shade of a point on a surface:

1. It is a *local* reflection model, i.e. it doesn't account for second-order reflections, as do raytracing or radiosity. In order to compensate for the loss of some reflected light, an extra *ambient lighting* term is added to the scene that is rendered.
2. It divides the reflection from a surface into three subcomponents, *specular* reflection, *diffuse* reflection, and *ambient* reflection.

If we first define, for each light source in the scene to be rendered, the components i_s and i_d , where these are the intensities (often as RGB values) of the specular and diffuse components of the light sources respectively. A single i_a term controls the ambient lighting; it is sometimes computed as a sum of contributions from the light sources.

If we then define, for each *material* (which is typically assigned 1-to-1 for the object surfaces in the scene):

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

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 decides how "evenly" light is reflected from a shiny spot

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

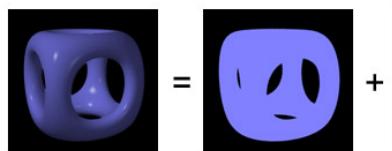
Then the shade value for each surface point I_p is calculated using this equation, which is the *Phong reflection model*:

$$I_p = k_a i_a + \sum_{\text{lights}} (k_d (L \cdot N) i_d + k_s (R \cdot V)^\alpha i_s)$$

The diffuse term does not use the direction towards the viewer (V), as the diffuse term is equal in all directions from the point, including the direction of the viewer. The specular term, however, is large only when the reflection vector R is nearly aligned with viewpoint vector V , as measured by the α power of the cosine of the angle between them, which is the dot product of the normalized direction vectors R and V . When α is large, representing an almost mirror-like reflection, the specular reflection will be very small because the high power of the cosine will go rapidly to zero with any viewpoint not aligned with the reflection.

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

Phong reflection is an empirical model, which is not based on a physical description of light interaction, but instead on informal observation. Phong observed that for very shiny surfaces the specular highlight was small and the intensity fell off rapidly, while for duller surfaces it was larger and fell off more slowly.



This equation can be represented in a graphic way:

color and ambient

Phong reflection model

Here the "color and ambient" represents a colored ambient light (diffuse and from all directions). The object shown here is gray, but is placed in a blue environment. Interpret the figure accordingly.

Phong shading interpolation method

Main article: →Phong shading

Along with the reflection model for computing color at a surface point, →Bui Tuong Phong also developed a method of interpolation to compute colors at every pixel in a rasterized triangle representing a surface patch. These topics are sometimes treated together under the term →*Phong shading*, but here the latter term is used only for the interpolation method.

See also

- →Bui Tuong Phong : Read about this shading model creator's life and work.

Source: http://en.wikipedia.org/wiki/Phong_reflection_model

Principal Authors: Dicklyon, Bignotter, Csl77, Nixdorf, Srleffler

Phong shading

An application of the →Phong reflection model.

Phong shading is an interpolation method in →3D computer graphics, using interpolation of surface normals in rasterizing polygons, to get better resolution of specular reflections such as those generated by the →Phong reflection model.

Since the inventor's publications combined the interpolation technique with his reflection model, the term *Phong shading* is also commonly used to refer to the reflection model or to the combination of the reflection model and the interpolation method.

These methods were developed by →Bui Tuong Phong, who published them in his 1973 Ph.D. dissertation at the University of Utah.

Phong illumination or reflection model

Main article: →Phong reflection model

Phong reflection is a local illumination model that can produce a certain degree of realism in three-dimensional objects by combining three elements: diffuse, specular, and ambient lighting for each considered point on a surface.

The reflection model has nothing specific to polygons or pixels, unlike the interpolation method.

Phong interpolation

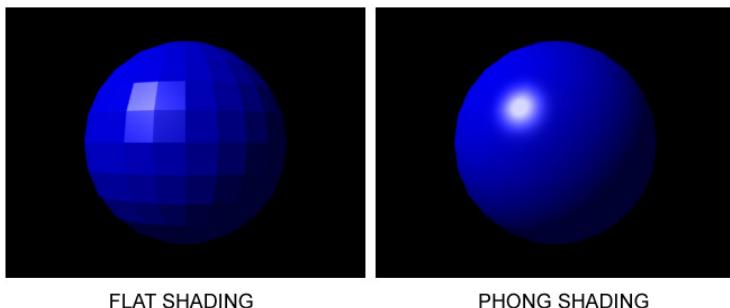


Figure 72 Phong shading interpolation example

As a rendering method, Phong shading can be regarded as an improvement on →Gouraud shading that provides a better approximation to a point-by-point application of an underlying reflection model by assuming a smoothly varying surface normal vector. The Phong interpolation method works better than Gouraud shading when applied to the Phong reflection model or to any reflection model that has small specular highlights.

The main problem with Gouraud shading is that when a specular highlight occurs near the center of a large triangle, it will usually be missed entirely, due to the interpolation of colors between vertices. This problem is fixed by Phong shading.

We are given three vertices in two dimensions, v_1 , v_2 and v_3 , as well as surface normals for each vertex n_1 , n_2 and n_3 ; we assume these are of unit length. Unlike →Gouraud shading, which interpolates colors across triangles, in Phong shading we linearly interpolate a normal vector \mathbf{N} across the surface of the triangle, from the three given normals. This is done for each pixel in the triangle, and at each pixel we normalize \mathbf{N} and use it in the Phong illumination model to obtain the final pixel color.

Phong shading

In some modern hardware, variants of this algorithm are called "pixel shading." It usually means that the lighting calculations can be done per-pixel, and that the lighting variables (including surface normals or some approximately equivalent vector) are interpolated across the polygon.

See also

- →Blinn–Phong shading model — Phong shading modified to trade precision with computing efficiency
- →Bui Tuong Phong
- →Phong reflection model
- →Specular highlight — Other specular lighting equations

Source: http://en.wikipedia.org/wiki/Phong_shading

Principal Authors: T-tus, Karada, Csl77, Dicklyon, Michael Hardy

Photon mapping



Figure 73 A crystal ball with caustics

In computer graphics, **photon mapping** is a global illumination algorithm based on ray tracing 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 interreflections between illuminated objects, and some of the effects caused by particulate matter such as smoke or water vapor. It was developed by Henrik Wann Jensen.

In the context of the refraction of light through a transparent medium, the desired effects are called caustics. A caustic is a pattern of light that is focused on a surface after having had the original path of light rays bent by an intermediate surface. An example is a glass of wine on a table. As light rays pass through the glass and the liquid, they are refracted and focused on the table the glass is standing on. The wine in the glass also produces interesting effects, changing the pattern of light as well as its color.

With photon mapping, light packets (photons) are sent out into the scene from the light source. Whenever a photon intersects with a surface, the intersection point, incoming direction, and energy of the photon are stored in a cache called the *photon map*. As each photon is bounced or refracted by intermediate surfaces, the energy gets absorbed until no more is left. We can then stop tracing the path of the photon. Often we stop tracing the path after a predefined number of bounces in order to save time. One of the great advantages of photon mapping is the independence of the scene's description. That is, the scene can be modelled using any type of geometric primitive as for instance triangles, spheres, etc.

Another technique is to send out groups of photons instead of individual photons. In this case, each group of photons always has the same energy, thus the photon map need not store energy. When a group intersects with a surface, it is either completely transmitted or completely absorbed. This is a Monte Carlo method called *Russian roulette*.

To avoid emitting unneeded photons, the direction of the outgoing rays is often constrained. Instead of simply sending out photons in random directions, they are sent in the direction of a known object that we wish to use as a photon manipulator to either focus or diffuse the light. There are many other refinements that can be made to the algorithm like deciding how many photons to send, and where and in what pattern to send them.

Photon mapping is generally a preprocess and is carried out before the main rendering of the image. Often the photon map is stored on disk for later use. Once the actual rendering is started, every intersection of an object by a ray is tested to see if it is within a certain range of one or more stored photons and if so, the energy of the photons is added to the energy calculated using a

standard illumination equation. The slowest part of the algorithm is searching the photon map for the nearest photons to the point being illuminated.

The BRL-CAD ray tracer includes an open source implementation of photon mapping. Although photon mapping was designed to work primarily with ray tracers, it can also be used with scanline renderers.

External links

- *Realistic Image Synthesis Using Photon Mapping*²⁴⁶ ISBN 1568811470
- Photon mapping introduction²⁴⁷ from Worcester Polytechnic Institute
- *Global Illumination using Photon Maps - Henrik Wann Jensen*²⁴⁸

Source: http://en.wikipedia.org/wiki/Photon_mapping

Principal Authors: Flamurai, MichaelGensheimer, Curps, Phrood, Patrick, Brlcad, Fnielsen, Nilmberg

Photorealistic (Morph)

Photorealistic computer graphics can be created by taking an original 3D rendering, which resembles a photograph, and morphing the image in photoshop. This creates a life-like 3D model, using a real image, that is a time consuming chore for 3D animators.

An example of a photorealistic morph looks like this image below



²⁴⁶ <http://graphics.ucsd.edu/~henrik/papers/book/>

²⁴⁷ http://www.cs.wpi.edu/~emmanuel/courses/cs563/write_ups/zackw/photon_mapping/PhotonMapping.html

²⁴⁸ <http://graphics.ucsd.edu/~henrik/papers/ewr7/egwr96.pdf>

See also

- 3D Rendering
- photoshopping

Source: http://en.wikipedia.org/wiki/Photorealistic_%28Morph%29

PLIB

PLIB is a suite of Open Sourced portable computer game libraries, originally written by Steve Baker in 1997 and licensed under the LGPL.

PLIB includes sound effects, music, a complete 3D engine, font rendering, a simple windowing library, a game scripting language, a GUI, networking, 3D math library and a collection of utility functions. All are 100% portable across nearly all modern computing platforms. Each library component is fairly independent of the others - so if you want to use SDL, GTK+, GLUT, or FLTK instead of PLIB's 'PW' windowing library, you can.

PLIB is used by many projects (not all games - and not all OpenSourced).

See also

- Allegro
- OpenML
- Simple DirectMedia Layer
- DirectX
- →OpenGL
- ClanLib

External links

- [PLIB website²⁴⁹](#)

Source: <http://en.wikipedia.org/wiki/PLIB>

²⁴⁹ <http://plib.sourceforge.net/>

Polygonal modeling

In →3D computer graphics, **polygonal modeling** is an approach for modeling objects by representing or approximating their surfaces using polygons. Polygonal modeling is well suited to scanline rendering and is therefore the method of choice for real-time computer graphics. Alternate methods of representing 3D objects include NURBS surfaces, subdivision surfaces, and equation-based representations used in ray tracers.

Geometric Theory and Polygons

The basic object used in mesh modeling is a vertex, a point in three dimensional space. Two vertices connected by a straight line become an edge. Three vertices, connected to each other by three edges, define a triangle, which is the simplest polygon in Euclidean space. More complex polygons can be created out of multiple triangles, or as a single object with more than 3 vertices. Four sided polygons (generally referred to as quads) and triangles are the most common shapes used in polygonal modeling. A group of polygons, connected to each other by shared vertices, is generally referred to as an **element**. Each of the polygons making up an element is called a **face**.

In Euclidean geometry, any three points determine a plane. For this reason, triangles always inhabit a single plane. This is not necessarily true of more complex polygons, however. The flat nature of triangles makes it simple to determine their surface normal, a three-dimensional vector perpendicular to the triangle's edges. Surface normals are useful for determining light transport in ray tracing, and are a key component of the popular →Phong shading model. Some rendering systems use vertex normals instead of surface normals to create a better-looking lighting system at the cost of more processing. Note that every triangle has two surface normals, which face away from each other. In many systems only one of these normals is considered valid – the other side of the polygon is referred to as a **backface**, and can be made visible or invisible depending on the programmer's desires.

Many modeling programs do not strictly enforce geometric theory; for example, it is possible for two vertices to have two distinct edges connecting them, occupying the exact same spatial location. It is also possible for two vertices to exist at the same spatial coordinates, or two faces to exist at the same location. Situations such as these are usually not desired by the user, and must be fixed manually.

A group of polygons which are connected together by shared vertices is referred to as a **mesh**. In order for a mesh to appear attractive when rendered, it is desirable that it be **non-self-intersecting**, meaning that no edge passes through a polygon. Another way of looking at this is that the mesh cannot pierce itself. It is also desirable that the mesh not contain any errors such as doubled vertices, edges, or faces. For some purposes it is important that the mesh be a **manifold** – that is, that it does not contain holes or singularities (locations where two distinct sections of the mesh are connected by a single vertex).

Construction of Polygonal Meshes

Although it is possible to construct a mesh by manually specifying vertices and faces, it is much more common to build meshes using a variety of tools. A wide variety of 3d graphics software packages are available for use in constructing polygon meshes.

One of the more popular methods of constructing meshes is box modeling, which uses two simple tools:

- The **subdivide** tool splits faces and edges into smaller pieces by adding new vertices. For example, a square would be subdivided by adding one vertex in the center and one on each edge, creating four smaller squares.
- The **extrude** tool is applied to a face or a group of faces. It creates a new face of the same size and shape which is connected to each of the existing edges by a face. Thus, performing the **extrude** operation on a square face would create a cube connected to the surface at the location of the face.

A second common modeling method is sometimes referred to as **inflation modeling** or **extrusion modeling**. In this method, the user creates a 2d shape which traces the outline of an object from a photograph or a drawing. The user then uses a second image of the subject from a different angle and extrudes the 2d shape into 3d, again following the shape's outline. This method is especially common for creating faces and heads. In general, the artist will model half of the head and then duplicate the vertices, invert their location relative to some plane, and connect the two pieces together. This ensures that the model will be symmetrical.

Another common method of creating a polygonal mesh is by connecting together various **primitives**, which are predefined polygonal meshes created by the modeling environment. Common primitives include:

- Cubes
- Pyramids

- Cylinders
- 2D primitives, such as squares, triangles, and disks
- Specialized or esoteric primitives, such as the Utah Teapot or Suzanne, Blender's monkey mascot.
- Spheres - Spheres are commonly represented in one of two ways:
 - **Icospheres** are icosahedrons which possess a sufficient number of triangles to resemble a sphere.
 - **UV Spheres** are composed of quads, and resemble the grid seen on some globes - quads are larger near the "equator" of the sphere and smaller near the "poles," eventually terminating in a single vertex.

Finally, some specialized methods of constructing high or low detail meshes exist. Sketch based modeling is a user-friendly interface for constructing low-detail models quickly, while 3d scanners can be used to create high detail meshes based on existing real-world objects in almost automatic way. These devices are very expensive, and are generally only used by researchers and industry professionals but can generate high accuracy sub-millimetric digital representations.

Extensions

Once a polygonal mesh has been constructed, further steps must be taken before it is useful for games, animation, etc. The model must be texture mapped to add colors and texture to the surface and it must be given an inverse kinematics skeleton for animation. Meshes can also be assigned weights and centers of gravity for use in physical simulation.

In order to display a model on a computer screen outside of the modeling environment, it is necessary to store that model in one of the file formats listed below, and then use or write a program capable of loading from that format. The two main methods of displaying 3d polygon models are →OpenGL and →Direct3D. Both of these methods can be used with or without a graphics card.

Advantages and disadvantages

There are many disadvantages to representing an object using polygons. Polygons are incapable of accurately representing curved surfaces, so a large number of them must be used to approximate curves in a visually appealing manner. The use of complex models has a cost in lowered speed. In scanline conversion, each polygon must be converted and displayed, regardless of size, and there are frequently a large number of models on the screen at any given time.

Often, programmers must use multiple models at varying levels of detail to represent the same object in order to cut down on the number of polygons being rendered.

The main advantage of polygons is that they are faster than other representations. While a modern graphics card can show a highly detailed scene at a frame rate of 60 frames per second or higher, raytracers, the main way of displaying non-polygonal models, are incapable of achieving an interactive frame rate (10 fps or higher) with a similar amount of detail.

File Formats

A variety of formats are available for storing 3d polygon data. The most popular are:

- .3ds, .max, which is associated with 3D Studio Max
- .mb and .ma, which are associated with Maya
- .lwo, which is associated with Lightwave
- .obj (Wavefront's "The Advanced Visualizer")
- .C4d associated with Cinema 4D
- .dxf, .dwg, .dwf, associated with AutoCAD
- .md3, .md2, associated with the Quake series of games
- .fbx (Alias)
- .rwx (Renderware)
- .wrl (VRML 2.0)

External links

- Free models in a variety of formats: 3dcafe²⁵⁰
- Computer graphics latest news, reviews, tutorials and interviews: InsideCG²⁵¹
- Computer graphics news: CG Channel²⁵²
- Computer graphics forums, reviews, and contests: CG Society²⁵³
- Open Source 3d modeling program: Blender²⁵⁴
- Simple, small and open source 3D modelling program: Wings 3D²⁵⁵
- Open Source 3D modeling program: AutoQ3D²⁵⁶

²⁵⁰ <http://www.3dcafe.com/>

²⁵¹ <http://www.insidecg.com/>

²⁵² <http://www.cgchannel.com/>

²⁵³ <http://www.cgsociety.org/>

²⁵⁴ <http://www.blender.org/>

²⁵⁵ <http://www.Wings3D.com/>

²⁵⁶ <http://autoq3d.ecuadra.com/>

References

OpenGL SuperBible (3rd ed.), by Richard S Wright and Benjamin Lipchak ISBN 0672326019

OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 1.4, Fourth Edition by OpenGL Architecture Review Board ISBN 0321173481

OpenGL(R) Reference Manual : The Official Reference Document to OpenGL, Version 1.4 (4th Edition) by OpenGL Architecture Review Board ISBN 032117383X

Blender documentation: <http://www.blender.org/cms/Documentation.628.0.html>

Maya documentation: packaged with Alias Maya, <http://www.alias.com/eng/index.shtml>

See also

- Finite element analysis / method
- →Polygon mesh
- Vector graphics

Source: http://en.wikipedia.org/wiki/Polygonal_modeling

Principal Authors: Furykef, Jreynaga, KenArthur, Peter L, TenOfAllTrades, RJFJR

Polygon (computer graphics)

Polygons are used in computer graphics to compose images that are three-dimensional in appearance. Usually (but not always) triangular, polygons arise when an object's surface is modeled, vertices are selected, and the object is rendered in a wire frame model. This is quicker to display than a shaded model; thus the polygons are a stage in computer animation. The *polygon count* refers to the number of polygons being rendered per frame.

Competing methods for rendering polygons that avoid seams

- point

- floating point
- Fixed-point
- polygon
- because of rounding, every scanline has its own direction in space and may show its front or back side to the viewer.
- Fraction (mathematics)
 - Bresenham's line algorithm
 - polygons have to be split into triangles
 - the whole triangle shows the same side to the viewer
 - the point numbers from the →Transform and lighting stage have to converted to Fraction (mathematics)
- Barycentric coordinates (mathematics)
 - used in raytracing

See also

- Polygon, for general polygon information
- →Polygon mesh, for polygon object representation
- Polygon modeling

Source: http://en.wikipedia.org/wiki/Polygon_%28computer_graphics%29

Principal Authors: Michael Hardy, Arnero, Orderud, SimonP, BlazeHedgehog

Polygon mesh

A **Polygon mesh** is a *collection of vertices and polygons* that define the shape of an object in →3D computer graphics.

Meshes usually consists of triangles, quadrilaterals or other simple convex polygons, since this simplifies rendering, but they can also contain objects made of general polygons with optional holes.

Examples of internal mesh structure representations:

- Simple list of vertices with a list of indices describing which vertices are linked to form polygons; additional information can describe a list of holes
- List of vertices + list of edges (pairs of indices) + list of polygons that link edges
- Winged edge data structure

The choice of the data structure is governed by the application: it's easier to deal with triangles than general polygons, especially in computational geometry. For optimized algorithms it is necessary to have a fast access to topological informations such as edges or neighbouring faces; this requires more complex structures such as the winged-edge representation.

Source: http://en.wikipedia.org/wiki/Polygon_mesh

Principal Authors: Orderud, Furrykef

Precomputed Radiance Transfer

Precomputed Radiance Transfer (PRT) is a technique used to render a scene in real time with complex light interactions precomputed. →Radiosity can be used to determine the diffuse lighting of the scene, however PRT offers a method to dynamically change the lighting environment.

In essence, PRT computes the illumination of a point as a linear combination of incident irradiance. An efficient method must be used to encode this data, such as Spherical harmonics.

Reference: Sloan, Peter-Pike, Jan Kautz, and John Snyder. "Precomputed Radiance Transfer for Real-Time Rendering in Dynamic, Low-Frequency Lighting Environments". ACM Transactions on Graphics, Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH), pp. 527-536. New York, NY: ACM Press, 2002.

Source: http://en.wikipedia.org/wiki/Precomputed_Radiance_Transfer

Principal Authors: Bhouston

Pre-rendered

Pre-rendered graphics, in computer graphics, is a video footage which is not being rendered in real-time by the hardware that is outputting or playing back the video. Instead, the video is a recording of a footage that was previously rendered on a different equipment (typically one that is more powerful than the hardware used for playback). The advantage of pre-rendering is the ability to use graphic models that are more complex and computationally intensive than what can be rendered in real-time, due to the possibility of using multiple computers over extended periods of time to render the end results. The disadvantage of pre-rendering, in the case of video game graphics, is a generally lower level of interactivity, if any, with the player.

CG movies such as *Toy Story*, *Shrek* and *Final Fantasy: The Spirits Within* are entirely pre-rendered. Pre-rendered graphics are used primarily as cut scenes in modern video games, where they are also known as full motion video.

See also

- →Rendering (computer graphics)
- FMV game

Source: <http://en.wikipedia.org/wiki/Pre-rendered>

Procedural generation

Procedural generation is a widely used term to indicate the possibility to create content on the fly, as opposed to creating it before distribution. This is often related to computer graphics applications.

Procedural synthesis is actually the correct term for this kind of approach, recalling as "electronic sounds" have been (and still are) generated by synthesizers from nothing but electricity and user's creativity.^[citation needed] In this document, the terms "synthesis" and "generation" are used interchangeably.

More generally, the term 'procedural' is strictly related to a procedure used to compute particular functions. This concept is similar to a fractal, although the latter is considered to be much more math-involved. Most common procedurally generated content include textures and meshes. Sound is seldom procedurally generated in PC applications. While procedural generation techniques

have been employed for years in countless games, few actually used this approach extensively. The exception is Will Wright's *Spore*, a highly-anticipated video game title populated entirely with procedurally generated content. Some "procedurally generated" elements also appeared in previous games: the first *Soldier of Fortune* from Raven Software used simple routines to add random detail to enemy models. To a certain degree, it could be said that the lighting in *Doom 3* from id Software is "procedurally generated" because it does not rely on lightmaps precomputed using a radiosity process.

The modern demoscene uses procedural generation for squeezing a lot of impressive audiovisual content into very small executable programs. Farbrausch is a team famous for its achievements in this area, although many similar techniques were already implemented by The Black Lotus in the 1990's.

Procedural generation as an application of functional programming

Procedurally generated content such as textures and landscapes may exhibit variation, but the generation of a particular item or landscape must be identical from frame to frame. Accordingly, the functions used must be referentially transparent, always returning the same result for the same point, so that they may be called in any order and their results freely cached as necessary. This is similar to lazy evaluation in functional programming languages.

Procedural generation in video games

The earliest computer games were severely limited by memory constraints. This forced content like maps to be generated algorithmically on the fly: there simply wasn't enough space to store premade levels and artwork. Today, most games include thousands of times as much data in terms of memory as algorithmic mechanics. For example, all of the buildings in the large gameworld of *Grand Theft Auto* were individually designed and placed by artists.

In a typical modern video game, game content such as textures and character and environment models are created by artists beforehand, then rendered in the game engine. As the technical capabilities of computers and video game consoles increases, the amount of work required by artists also increases. First, high-end gaming PCs and next-generation game consoles like the Xbox 360 and PlayStation 3 are capable of rendering scenes containing many very detailed objects with high-resolution textures in high-definition. This means that artists must invest a great deal more time in creating a single character, vehicle, building, or texture, since gamers will tend to expect ever-increasingly detailed environments.

Secondly, the number of unique objects displayed in a video game is increasing. In addition to highly detailed models, players expect a variety of models that appear substantially different from one another. In older games, a single character or object model might have been used over and over again throughout a game. With the increased visual fidelity of modern games, however, it is very jarring (and threatens the suspension of disbelief) to see many copies of a single object, while the real world contains far more variety. Again, artists would be required to complete orders of magnitude more work in order to create many different varieties of a particular object. The need to hire larger art staffs is one of the reasons for skyrocketing game development budgets.

Some initial approaches to procedural synthesis attempted to solve these problems by shifting the burden of content generation from the artists to programmers who can create code which automatically generates different meshes according to input parameters. Although sometimes this still happens, what has been recognized is that applying a purely procedural model is often hard at best, requiring huge amounts of time to evolve into a functional, usable and realistic-looking method. Instead of writing a procedure that completely builds content procedurally, it has been proven to be much cheaper and more effective to rely on artist created content for some details. For example, SpeedTree is a middleware used to generate trees procedurally with a large variety yet leaf textures can be fetched from regular files often representing digitally acquired real foliage. Other effective methods to generate hybrid content is to procedurally merge different pre-made assets or to procedurally apply some distortions to them.

Supposing, however, a single algorithm can be envisioned to generate a realistic-looking tree, the algorithm could be called to generate random trees, thus filling a whole forest at runtime, instead of storing all the vertices required by the various models. This would save storage media space and reduce the burden on artist, while providing richer experience. The same method would require far more processing power in respect to a mere disk loading, but with CPUs getting faster, the problem is gradually becoming smaller. Please note, developing such algorithm is non-trivial for a single tree, let alone for a variety of species (compare Sumac, Birch, Maple and its species), moreover assembling a forest could not be done by just assembling trees because in real world this introduces interactions between the various trees which dramatically change their appearance (although this is probably a minor detail).

In 2004, a PC first-person shooter called *.kkrieger* was released that made heavy use of procedural synthesis: while quite short and very simple, the advanced video effects were packed into just 96 Kilobytes. In contrast, many modern games are released across several CDs, often exceeding 2 gigabytes in

size, more than 20,000 times larger. Several upcoming commercial titles for the PC, such as Will Wright's *Spore*, will also make use of procedural synthesis, calling for special hardware support in the console world: the Xbox 360 and the Playstation 3 have impressive procedural synthesis capabilities.

Next-generation console support and speculations

The Xbox 360's CPU, known as Xenon, is a 3-core design. When running procedural synthesis algorithms.²⁵⁷, one of the Xenon CPU's cores may "lock" a portion of the 1 MB shared L2 cache. When locked, a segment of cache no longer contains any prefetched instructions or data for the CPU, but is instead used as *output* space for the procedural synthesis thread. The graphics processing unit, called Xenos, can then read directly from this locked cache space and render the procedurally generated objects. The rationale behind this design is that procedurally generated game content can be streamed directly from CPU to GPU, without incurring additional latency by being stored in system RAM as an intermediary step. The downside to this approach is that when part of the L2 cache is locked, there is even less data immediately available to keep the 3 symmetric cores in the Xenon CPU running at full efficiency (1 MB of shared L2 is already a rather small amount of cache for 3 symmetric cores to share, especially considering that the Xenon CPU does not support out-of-order execution to more efficiently use available clock cycles).

The Cell processor contains an implementation-dependent number of vector processing units called SPEs. PlayStation 3, to date the largest use of the Cell processor, will contain 8 SPEs, only seven of which are active. The eighth is to be used in the event that another SPE malfunctions. The SPEs can stream data to each other, with each SPE performing a different operation on a large set of data, thus creating a highly efficient chain which performs a sequence of operations on a data set of arbitrary size (they are virtually assembled in a "hyper-pipeline", allowing high amount of parallelism). Each SPE can execute a different task in the algorithm and pass the result on to another SPE which does the next piece of work. The cell can thus generate game objects very quickly by dividing the algorithm into smaller tasks and setting up a chain of SPEs to do the work on an arbitrarily large number of objects.

For example, to generate a boulder, an SPE may initialize the basic vertex positions to a spherical object, a second SPE may displace those points to add

²⁵⁷ <http://arstechnica.com/articles/paedya/cpu/xbox360-1.ars>

randomness, a third to generate normals, tangent and binormals for bumpmapping while a fourth SPE may add volume texture coordinates (possibly in parallel with the third SPE). Note that this example assumes the compiler can guess correct parallelism opportunity.

The Cell's counterpart to Xenon "cache locking" is each SPE's 256 KiB local store, essentially L2 cache memory which can be accessed directly, rather than being used to speed up access to main memory. This approach has both perks and flaws, as it allows each SPE to work extremely quickly, but only on a small subset of the total data of a program. There is also a direct 35 GB/s link between the Cell and the PS3's "Reality Synthesizer" GPU, allowing objects to be sent directly to the GPU to be rendered as soon as they are generated.

See also

- →Procedural texture
- Procedural animation
- Fractal landscape
- L-System

Software using procedural generation

- Elite (everything about the universe - planet positions, names, politics and general descriptions - is generated procedurally; Ian Bell has released the algorithms in C as text elite²⁵⁸⁾
- Exile
- Frontier: Elite 2
- Noctis
- StarFlight
- The Sentinel (computer game)
- Most Roguelikes procedurally generate gameworlds.
- Dozens of demos procedurally generate complex textures, sounds and 3D objects.
- SpeedTree is a widely known middleware to procedurally generate trees at various levels in the production pipeline or at runtime using L-systems.
 - The Elder Scrolls IV: Oblivion
 - Many games using the upcoming Unreal Engine 3.
- Spore
- Jade Cocoon (The merged minions were procedurally generated based on body part sizes)

²⁵⁸ <http://www.iancgbell.clara.net/elite/text/index.htm>

External links

- Real-time procedural generation of ‘pseudo infinite’ cities²⁵⁹ - ACM
- The Future Of Content²⁶⁰ - Will Wright keynote on Spore & procedural generation at the Game Developers Conference 2005. (Scroll down 3/4 of page, registration required to view video).
- Procedural 3D Content Generation²⁶¹ - Dean Macri and Kin Pallister - Intel - An excellent two-part article with an accompanying demo (including full C++ source code).
- Darwinia²⁶² - development diary²⁶³ procedural generation of terrains and trees.

Source: http://en.wikipedia.org/wiki/Procedural_generation

Principal Authors: Jacoplane, MaxDZ8, Ashley Y, Viznut, Lapinmies, Tlogmer, Praetor alpha, Jess-martin, ChopMonkey, Lupin

Procedural texture

A **procedural texture** is a computer generated image created using an algorithm intended to create a realistic representation of natural elements such as wood, marble, granite, metal, stone, and others.

Usually, the natural look of the rendered result is achieved by the usage of fractal noise and turbulence functions. These functions are used as a numerical representation of the “randomness” found in everything that surrounds us.

In general, these noise and fractal functions are simply used to “disturb” the texture in a natural way such as the undulations of the veins of the wood. In other cases, like marbles’ textures, they are based on the graphical representation of fractal noise.

²⁵⁹ <http://portal.acm.org/citation.cfm?id=604490>

²⁶⁰ <http://www.pqhp.com/cmp/gdctv/>

²⁶¹ <http://www.intel.com/cd/ids/developer/asmo-na/eng/20247.htm>

²⁶² <http://www.darwinia.co.uk/>

²⁶³ <http://www.darwinia.co.uk/extras/development.html>

²⁶⁴ <http://www.spiralgraphics.biz/gallery.htm>



Figure 74 A procedural floor grate texture generated with the texture editor Genetica²⁶⁴.

Example of a procedural marble texture:

(Taken from The Renderman Companion Book, by Steve Upstill)

```
/* Copyrighted Pixar 1988 */
/* From the RenderMan Companion p.355 */
/* Listing 16.19 Blue marble surface shader */

/*
 * blue_marble(): a marble stone texture in shades of blue
 * surface
 */
blue_marble(
    float    Ks      = .4,
    Kd      = .6,
    Ka      = .1,
    roughness = .1,
```

```

        txtscale = 1;
        color    specularcolor = 1)
{
    point PP;           /* scaled point in shader space */
    float csp;          /* color spline parameter */
    point Nf;           /* forward-facing normal */
    point V;            /* for specular() */
    float pixelsize, twice, scale, weight, turbulence;

    /* Obtain a forward-facing normal for lighting calculations.
*/
    Nf = faceforward( normalize(N), I);
    V = normalize(-I);

    /*
     * Compute "turbulence" a la [PERLIN85].  Turbulence is a sum
of
     * "noise" components with a "fractal" 1/f power spectrum.
It gives the
     * visual impression of turbulent fluid flow (for example, as
in the
     * formation of blue_marble from molten color splines!).  Use
the
     * surface element area in texture space to control the num-
ber of
     * noise components so that the frequency content is appro-
priate
     * to the scale.  This prevents aliasing of the texture.
    */
    PP = transform("shader", P) * txtscale;
    pixelsize = sqrt(area(PP));
    twice = 2 * pixelsize;
    turbulence = 0;
    for (scale = 1; scale > twice; scale /= 2)
        turbulence += scale * noise(PP/scale);

    /* Gradual fade out of highest-frequency component near limit
*/
    if (scale > pixelsize) {
        weight = (scale / pixelsize) - 1;
        weight = clamp(weight, 0, 1);
}

```

```

turbulence += weight * scale * noise(PP/scale);
}

/*
 * Magnify the upper part of the turbulence range 0.75:1
 * to fill the range 0:1 and use it as the parameter of
 * a color spline through various shades of blue.
 */
csp = clamp(4 * turbulence - 3, 0, 1);
Ci = color spline(csp,
color (0.25, 0.25, 0.35), /* pale blue */ 
color (0.25, 0.25, 0.35), /* pale blue */ 
color (0.20, 0.20, 0.30), /* medium blue */ 
color (0.20, 0.20, 0.30), /* medium blue */ 
color (0.20, 0.20, 0.30), /* medium blue */ 
color (0.25, 0.25, 0.35), /* pale blue */ 
color (0.25, 0.25, 0.35), /* pale blue */ 
color (0.15, 0.15, 0.26), /* medium dark blue */ 
color (0.15, 0.15, 0.26), /* medium dark blue */ 
color (0.10, 0.10, 0.20), /* dark blue */ 
color (0.10, 0.10, 0.20), /* dark blue */ 
color (0.25, 0.25, 0.35), /* pale blue */ 
color (0.10, 0.10, 0.20) /* dark blue */ );
};

/* Multiply this color by the diffusely reflected light. */
Ci *= Ka*ambient() + Kd*diffuse(Nf);

/* Adjust for opacity. */
Oi = Os;
Ci = Ci * Oi;

/* Add in specular highlights. */
Ci += specularcolor * Ks * specular(Nf,V,roughness);
}

```

This article was taken from The Photoshop Roadmap²⁶⁵ with written authorization

²⁶⁵ <http://www.photoshoproadmap.com>

See also

- →Perlin noise
- →Procedural generation

Source: http://en.wikipedia.org/wiki/Procedural_texture

Principal Authors: MaxDZ8, Wikedit, TimBentley, Volfy, Viznut

Pyramid of vision

Pyramid of vision is a →3D computer graphics term: the infinite pyramid into the real world, with an apex at the observer's eye and faces passing through the edges of the viewport ("window").

Source: http://en.wikipedia.org/wiki/Pyramid_of_vision

Qualitative invisibility

In CAD/CAM, **Qualitative Invisibility** (QI) is the number of solid bodies obscuring a point in space as projected onto a plane (called [vector] hidden line removal used by the Vector format). Oftentimes, when a CAD engineer creates a projection of his model into a plane [the drawing], he wishes to denote the edges which are visible by a solid segment and those which are hidden by dashed or dimmed segments. The idea of keeping track of the number of obscuring bodies gives rise to an algorithm which propagates the quantitative invisibility throughout the model. This technique uses edge coherence to speed up the calculations in the algorithm. However, QI really only works well when bodies are larger solids, non-interpenetrating, and not transparent. A technique like this would fall apart when trying to render soft organic tissue as found in the human body, because there is not always a clear delineation of structures. Also, when images become too cluttered and intertwined, then the contribution of this algorithm is marginal.

Qualitative Invisibility is a term coined by Arthur Appel of the graphics group at IBM Watson Research and used in several of his papers.

External link

- Vector Hidden Line Removal and Fractional Quantitative Invisibility²⁶⁶

Reference

Appel, A., "The Notion of Quantitative Invisibility and the Machine Rendering of Solids," Proceedings ACM National Conference, Thompson Books, Washington, DC, 1967, pp. 387-393, pp. 214-220.

Source: http://en.wikipedia.org/wiki/Qualitative_invisibility

Principal Authors: Wheger, Ulayiti, Patrick, Frencheigh

Quaternions and spatial rotation

The main article on quaternions describes the history and purely mathematical properties of the algebra of quaternions. The focus of this article is the practical application of these mathematical ideas in engineered physical systems.

The algebra of quaternions is a useful mathematical tool for formulating the composition of arbitrary spatial rotations, and establishing the correctness of algorithms founded upon such compositions. These methods find broad application in computer generated graphics, robotics, global navigation, and the spatial orientation of instruments. (citation: Quaternions and rotation Sequences: a Primer with Applications to Orbits, Aerospace, and Virtual Reality, Kuipers, Jack B., Princeton University Press copyright 1999)

Familiarity with the definition of a quaternion and the associated arithmetical rudiments is assumed. These may be gleaned from a cursory examination of a few paragraphs in the article on quaternions.

Introduction

The 1909 edition of Webster's unabridged dictionary (citation: Webster's New International Dictionary, G. & C. Merriam Co. copyright 1909, pp 1752) defines a quaternion as

²⁶⁶ <http://wheger.tripod.com/vhl/vhl.htm>

5. Math. The quotient of two vectors ... Such is the view of the inventor, Sir Wm. Rowan Hamilton, and his disciple, Prof. P. G. Tait; but authorities are not yet quite agreed as to what a quaternion is or ought to be.

This definition together with the excised "..." is technically correct if the definition of "vector" is restricted to its three dimensional conception, but the wagish remark serves to anecdotally illustrate the aura of arcane and possibly irrational esotericism that encumbered the mathematical discipline in its early days. To some degree, the disrepute of those early perceptions persist today.

The purpose of this introduction is to demystify the quaternions by demonstrating that the underlying concepts are pervasive in the everyday experience of ordinary people. This will establish a solid intuitive foundation upon which the mathematical and algorithmic content of the remainder of the article may be constructed. In fact, when approached from a geometric perspective, the quaternion algebra is as easy to understand, utilize, and apply as the algebra of complex numbers.

Non-commutativity

When considered as a purely numerical algebraic system, the non-commutativity of quaternion multiplication strikes many initiates as capricious and counter-intuitive. The following simple exercise demonstrates otherwise.

Consider an asymmetrical object, for example, a book. Set it on a table so that the spine is to the left and the front cover is visible with the title readable. Define the three orthonormal spatial axes as follows: positive z extends up from the table, and x and y oriented as usual, i.e. x extending from left to right, and y extending away from the viewer.

The application of two rotation operators to the book will demonstrate non-commutativity. First, rotate the book 90 degrees clockwise about the z axis, so that the spine is facing away. Next rotate the book 180 degrees clockwise about the y axis, so that now the front cover faces down and the spine faces away.

The application of these same two rotation operators in reverse order, first yields the book in a position with the cover facing down and spine facing *right*. After applying the z rotation, the book rests in its final position with the cover facing down and the spine facing *forward*, demonstrating $zy \neq yz$. In fact, the composition zy represents a single rotation of 180 degrees about the $[-1 \ 1 \ 0]$ axis, and yz represents a single rotation of 180 degrees about the $[1 \ 1 \ 0]$ axis.

In general, the composition of two different rotations about two distinct spatial axes will not commute. This is a truth that all of us subconsciously understand

and apply every day. Quaternion algebra is simply a mathematical formulation of this truth.

Double covering

This second exercise demonstrates the property of a quaternion rotation operator that for each complete turn of the angular parameter, the image of the object to which the operator is applied turns twice.

In a sitting position extend the forearm from the waist with the palm open and up. Place a small object, such as a coin on the open palm, so that only gravity holds the coin in the palm. When this exercise is complete, the palm (and coin) will have rotated 720 degrees or two turns, without dropping or squeezing the coin.

Define the three orthonormal spatial axes as follows: positive z extends up from the palm, and x and y oriented as usual, i.e. x extending from left to right through the shoulder sockets, and y extending away from the viewer.

Four pivot axes are of interest. The first is attached to the shoulder socket and aligned with the y axis. The second is attached to the elbow, and is aligned orthogonal to the upper arm and forearm. The third is aligned with the forearm, and the fourth is aligned orthogonal to this at the wrist in the plane of the palm.

First rotate the palm clockwise 180 degrees (counter-clockwise for left handed people). The various pivot axes will turn in synchrony in order to maintain the palm facing up, so that the palm comes to rest over the shoulder with the fingers pointing back. Next rotate the palm another 180 degrees clockwise, with the gimbals modulating so that the palm comes to rest with the fingers pointing forward and the upper arm extended upward with the forearm angled up.

Similarly, continue with another 180 degrees, so that the palm lingers with the fingers pointing back under the armpit with the elbow akimbo up and back with the upper arm extended horizontally. Finally rotate the last 180 degrees, bringing the arm to rest in the original position. It is much easier to do this than to describe it.

The movement is a composition of continuously applied incremental rotations about the four pivot axes, which notably, do not have fixed alignment with the orthonormal coordinate system. Each pivot traverses, in various combinations, 180 degrees clockwise, and 180 degrees counterclockwise, for a total of 360 degrees. For example, the wrist first flexes 90 degrees back, then 180 degrees forward, and finally 90 degrees back. In contrast, the elbow closes 90 degrees,

then opens 90 degrees, and then repeats the sequence. The palm (and coin), which is the image of the operation, traverses 720 degrees clockwise about the z coordinate axis.

Quaternion algebra provides an intuitive means for modelling these operations using the naturally defined variably oriented axes established by the physical circumstances. Any other method requires grappling with a tangle of trigonometric identities which convert the motions to the unnatural orthonormal reference frame.

Chirality

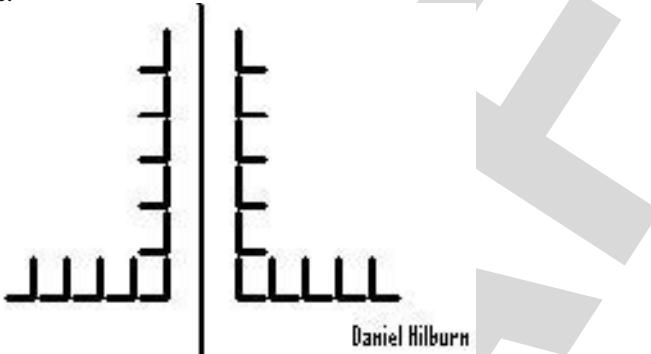
If a person endeavors to rotate the left hand so that it coincides with the right hand - palms facing in the same direction, thumbs and corresponding fingers aligned, the impossibility of the effort will soon become clear. The left and right hands are mirror images of each other, and a reflection is required to bring them into coincidence. Everyone understands instinctively that this is not possible in three dimensions. The generalization of this concept of "handedness" is known as chirality.

Chiral properties depend on the dimension of the space objects are constrained to occupy. For example consider the letter "L" located at the origin in the x-y plane. It is not possible to form the reflection of the letter across the y axis by using rotations constrained to lie entirely in the x-y plane. It is, however, possible to rotate the letter about the y axis, out of the plane 180 degrees, so that the image returns to lie in the plane.

A two dimensional reflection can be seen to be the two dimensional projection of a 180 degree three dimensional rotation. From the perspective of a three dimensional observer, the image is not a reflection, since it is possible for such an observer to go around behind the plane, where looking at the back side of the image, the "L" appears normal. A three dimensional chiral object must have three asymmetries.

So for example, if it were possible to rotate the left hand 180 degrees through the fourth dimension and back into three dimensional space, the left hand could be made to coincide with the right hand simply by applying a rotation. Of course from the perspective of a four dimensional observer, the image of the hand is not a reflection, for the observer can go around "behind" the three space, where looking at the "back" side, the hand looks normal. Naturally it is difficult to visualize this, except by analogy. A four dimensional chiral object must have four dimensions of asymmetry. This process can be extrapolated ad infinitum, but for the purposes of this article, four dimensions suffice.

It is worth noticing that if the aforementioned left hand is biological, it is not possible to bring it into coincidence with the right hand at the molecular level by rotating it through the fourth dimension. The reason is that all of the right handed DNA will be converted to left handed DNA by the rotation operation. To see this clearly, consider constructing the two-dimensional "L" out of right handed L-shaped tiles, nested together, say four along the short leg, and six along the long leg. Once reflected in the y axis, each tile will be left handed, and so will not coincide with a left handed "L" constructed from right handed tiles.



Chirality gives a four dimensional insight into the nature of double covering, demonstrates the fundamental importance of the simple concept of a reflection, and illustrates that three dimensional rotations are just a subset of fully general four dimensional rotations. The algebra of quaternions is four dimensional because it models four dimensional rotation operations, which necessarily include all three dimensional operations as a proper subset.

Definitions

In keeping with the empirical perspective of this article the presentation focuses on the practical needs of an implementor of a technological subsystem embodying quaternion algebra. Typically, such subsystems are software systems executing on a digital computer. In contrast to a pure mathematician, a practitioner is always concerned with the low level detail of a representation, for these details ultimately determine whether a system works correctly or fails. Wherever possible, the details will be hidden by an abstraction which approaches a pre-existing mathematical ideal, but due to the finite nature of computing, such abstractions necessarily fall short of pure mathematics.

In keeping with this approach, the terminology and notation is chosen to reflect modern mathematical use wherever possible, and deviates to reflect modern

technological use in aspects where details are important or known to be troublesome. These sections define the concepts, terminology, and notation useful in this presentation.

Concepts

It is important to clearly define the distinct mathematical concepts in order to separate their utilization and incompatibilities when implementing a geometrical algorithm. The resulting taxonomy can be implemented with the class definition facilities available in more modern programming languages. In practice, an amalgam of different, incompatible concepts is necessary in a given application, and the transition between them is accomplished by ad hoc methods. Careful definition minimizes the potential for operational error.

Euclidean space

Euclidean space is the set of absolute positions in n-dimensional space, typically represented by an n-tuple referred to an orthonormal basis. In three dimensions, it is a representation of the classical physical world we live in, and in four dimensions, it can represent various abstract extensions of that world, including the physicist's space-time, other non-Euclidean geometries, or the four dimensional analogue of our world.

Addition of two positions is not well defined in Euclidean space. This can be seen by considering that an n-tuple representation of an object depends upon the choices of origin and basis orientation, which are arbitrary. Therefore, the sum of two positions in one representation produces an absolute position, which will not equal the absolute position implied by the sum produced in another representation. This problem arises when simply translating the origin some small distance from the original representation. (illustrative figure here) The positions are unique *absolute* physical positions, whereas the n-tuple is merely a symbolic abstraction.

The addition of a *difference* of two positions is, however, well defined, yielding a unique absolute position independent of coordinate representation. Such a difference is traditionally called a *vector* in contradistinction to a *position*. It has a fixed length, which is called the *distance* between two positions. This length is easily computed using the familiar Euclidean metric. The Euclidean metric is invariant under rigid transformations of the space, i.e. under rotations and translations.

Any two positions $[a]$ and $[b]$ define two vectors $[v] = [a - b]$, and $-[v] = [b - a]$. While the specific n-tuple defining $[v]$ varies with coordinate representation, the addition of $[v]$ to a position referred to the same basis produces an

n-tuple referred to that basis, representative of an absolute position which is independent of the particular basis chosen for performing the computations.

In practice, Euclidean space is the world or rendering space of a computer graphics system, or the space in which servos and sensors operate in a control system. It is the space in which object definitions are instantiated by placing them at specific positions in the world space. It is the target space of a series of computations.

Affine space

Affine space is the set of differentials on Euclidean space. Since all of the elements are differences, addition of two elements produces another difference, so addition is well defined, independent of the coordinate system chosen for the underlying Euclidean space. Affine space is commonly called a vector space, and admits scalar multiplication as well as the Euclidean metric, although these are optional.

In practice, affine space is the space in which objects are defined, referred to a basis which varies from object to object, and which is typically chosen for its convenience for defining the object at hand.

Projective space

Projective space is one possible non-Euclidean affine space. As the name implies, projective space is a set well suited to defining and manipulating projective transformations, more generally called möbius transformations. These include, in particular, the all important perspective transformations of computer graphics and imaging systems. The Euclidean metric is not preserved by möbius transformations, but the cross-ratio, which measures the relative proportion of a differential to two other differentials, is invariant.

There are several ways to represent projective space, but perhaps the most common is the homogeneous representation. This method establishes an equivalence class among points on a line through the origin in a space with dimension one greater than the target space. Thus, for example, a point in three dimensional projective space is represented by the quadruple $[w \ x \ y \ z]$ with $[1 \ x/w \ y/w \ z/w]$ considered to be the canonical representation. Two points are equivalent if and only if their canonical representations are equal. More detailed descriptions of projective geometries may be found in (citation: Projective Geometry, H.S.M. Coxeter, Blaisdell Publishing Company, copyright 1964) (citation: The Real Projective Plane, H.S.M. Coxeter, Cambridge University Press, copyright 1955 & Springer-Verlag, copyright 1993) (citation: Geometry - A Comprehensive Course, Dan Pedoe, Cambridge University Press, copyright 1970)

A projective geometry is a dual space in which a one to one equivalence is established between points in the space and the set of n-1 dimensional subspaces. One consequence is that a projective space is not orientable. For example, in two dimensions, the projective space can be created from the Euclidean plane by adjoining to the Euclidean set a line at infinity, which is dual to the origin. Subsequently, it is not possible to say which "side" of a line a point is on, since it is always possible to go around through the line at infinity, and the absence of the Euclidean metric makes the notion of distance meaningless. A practical consequence of non-orientability in a graphics system is that objects behind the viewpoint can wrap around through infinity and show up as background clutter in a rendering. This problem is usually accommodated by including a rear clipping plane during rendering.

In practice, three dimensional projective space is the space of choice for performing the coordinate transformations necessary to instantiate a database of object descriptions in a world space. The Euclidean metric and orientability are usually implemented in various ad hoc ways - see for example (citation: Oriented Projective Geometry: a Framework for Geometric Computations, Jorge Stolfi, Academic Press, copyright 1991) (citation: Curves and Surfaces for Computer Aided Geometric Design: a Practical guide, Gerald E. Farin, Academic Press, copyright 1988).

Operator space

An operator space is a set of endomorphic mappings on an object space. The composition of two operators yields another mapping on the object space, so the composition is contained within the operator space.

For example, the set of 3x3 matrices is an operator space on three dimensional Euclidean space, and the set of 4x4 matrices is an operator space on three dimensional projective space. Composition can be matrix multiplication or addition. Application of the operator is achieved via matrix multiplication. If the object space is considered a row vector, the operator is right applicative, i.e. it is a postfix operator. If the object space is considered a column vector, the operator is left applicative, i.e. it is a prefix operator. In these two examples, the operator and object spaces are disjoint, and the only interface between the two is the definition of operator application.

This need not be the case. The quaternions form an operator space on 4-vectors. Composition can be quaternion multiplication or quaternion addition. Application, in the most general sense, is quaternion multiplication or addition, and may be right or left applicative. Most importantly, there is a one to one correspondence between each 4-vector and each quaternion operator, i.e. each

4-vector may be interpreted as an operator and vice versa. The identity between these two sets can lead to much confusion, so it is important to separate the definitions.

The set of 4-vectors is simply an affine vector space. The quaternions are a division algebra. In the parlance of computer programming, they are two different data types. The interface between the types includes the definitions of operator application, and a pair of explicit type conversion functions. Mixed expressions outside these limits will cause implicit type conversion errors during compilation in strongly typed programming languages and produce execution exceptions or incorrect operation in other languages.

The larger purpose of this article is to establish the equivalence of traditional matrix operator representations and the quaternion operator representation, together with conversion algorithms. In practice, a quaternion representation is more adaptable to creating intuitive user interfaces, while the matrix representation is more computationally efficient. User interface routines are therefore typically implemented with quaternions for code clarity and ease of modification, while execution engines are implemented with matrices for performance. Robust and correct conversion procedures are therefore necessary.

[This article is under reconstruction. The preceding text is essential complete, except for minor editing, such as checking spelling, formatting, and link insertion. The following text still needs attention]

Terminology

Historical

Some of the historical terminology has its origin in the writings of Sir William Rowan Hamilton, the inventor of quaternions. Other terms simply illustrate the evolution of language in the diverse, sometimes incompatible, endeavors of a broad topical area such as mathematics. The only historical terms that are used in this article are **norm** and the usage of **conjugate** with origins in complex analysis.

Tensor

Derived from the Latin tendere, tensum - *to stretch* the term tensor has a modern usage completely disjoint from its usage in early papers on quaternions, where it was synonymous with the magnitude of a quaternion. The 1909 Webster's defines the quaternion tensor

2. *Geom.* The ratio of the mere lengths of two vectors; - so called as merely stretching one vector into another. The tensor, denoted by T_q , of

a quaternion $w+ix+jy+kz$ is $\sqrt{w^2+x^2+y^2+z^2}$, the square root of its norm.

Vessor

Derived from the Latin *vertere, versum* - *to turn* the term vessor may be encountered in early papers on quaternions and refers to what is now called a unit quaternion, i.e. a quaternion whose magnitude is unity. The term vessor is now obsolete, and completely anachronistic.

Conjugate

Used as a noun, the term refers to the counterpart of a given quaternion whose scalar component is identical to the given scalar component, and whose spatial component is the given spatial component negated. Used in this sense, a quaternion conjugate is analogous to the complex conjugate of complex number theory. The formation of a conjugate is the complementary operation of a reflection: a reflection negates a one dimensional subspace and fixes the remainder; in contrast, a conjugate fixes one dimension, and negates the remainder. Given $[q]$, $[q^*]$ denotes the conjugate of $[q]$.

Used as a verb, it may mean to form a quaternion conjugate, a meaning which has its origins in complex analysis. It may also mean to form $[q^*vq]$, or $[pvq]$ where $[p]$ and $[q]$ are unit quaternions, and $[v]$ is a 3-vector in the former case, and a 4-vector in the latter. This second meaning has its origins in group theory, and may be expressed "conjugate $[v]$ by $[q]$ ". In order to avoid confusion, this article avoids this second usage and expresses the same idea by the phrase "apply $[q]$ to $[v]$ ".

Norm

The Oxford English Dictionary (citation: Oxford English Dictionary, Oxford University Press, copyright 1933, Vol. N, pp207) defines norm as

b. Algebra. (See *quot.*) **1866** Brande & Cox *Dict. Sci., etc.* II 228/2 the product $a^2 + b^2$ of a complex number $a + b\sqrt{-1}$ and its conjugate $a - b\sqrt{-1}$ is called its norm.

Notice this definition reinforces the implicit definition in the citation under "tensor" above, and shows the usage in most modern works, which defines the norm as synonymous with magnitude, is a relatively recent change

in terminology. Notably, John Conway, in his recent work (citation: On Quaternions and Octonions: their geometry, arithmetic, and symmetry, Conway, John H. & Smith, Derek A., A. K. Peters, copyright 2003) resurrects this older usage to good effect. This article conforms to this older usage in order to unambiguously distinguish between the concepts of norm and magnitude as well as to clarify the presentation of the generalized pythagorean theorem.

Scalar part

The real number component of a quaternion.

Imaginary part

Analogous to a complex number $a+bi$, a quaternion may be written as $a+b\mathbf{u}$ where $[\mathbf{u}]$ is a unit 3-vector. The set $a+b\mathbf{u}$ where $[\mathbf{u}]$ is fixed and a and b are arbitrary real numbers is isomorphic to the complex numbers. In particular , note that $[\mathbf{u}^2]=-1$. The quaternions therefore comprise infinitely many copies of the complex numbers, each defined by a vector $[\mathbf{u}]$ on the surface of the unit sphere in three dimensional space. As such, this article refers to the imaginary component as the spatial component, or a 3-vector.

Pure quaternion

A quaternion whose scalar component is zero, i.e. is purely imaginary. In this article, such an entity is simply called a 3-vector.

Modern

n-tuple

a linear array of n real numbers $[a_0 \ a_1 \ \dots \ a_{(n-1)}]$ with $a(i)$ typically represented as a floating point number.

orthonormal basis

n mutually orthogonal unit vectors which span an n -dimensional space.

vector

An n -tuple interpreted as the coordinates of a spatial position referred to an orthonormal basis.

3-vector

A three dimensional spatial position referred to the orthonormal basis

- $\mathbf{i} = [0 \ 1 \ 0 \ 0]$
- $\mathbf{j} = [0 \ 0 \ 1 \ 0]$
- $\mathbf{k} = [0 \ 0 \ 0 \ 1]$

4-vector

A four dimensional spatial position referred to the 3-vector's basis augmented by the unit vector

- $\mathbf{h} = [1 \ 0 \ 0 \ 0]$

quaternion

a 4-vector resulting from the polynomial product of two 3-vectors where $[\mathbf{i}^2] = [\mathbf{j}^2] = [\mathbf{k}^2] = [\mathbf{ijk}] = -1$ and $[\mathbf{h}^2] = [\mathbf{h}]$, i.e. $[\mathbf{h}] = 1$. A quaternion may be thought of as the sum of a scalar (the $[\mathbf{h}]$ coordinate) and a 3-vector.

Notation

The notation used herein is an amalgam of various mathematical notations in common use, and notations which have proven convenient in programming systems. The decision to use a particular element in the notation was governed by the following considerations in decreasing priority.

- Established precedent in existing mathematical discourse, leading to readability
- Unambiguous representation of concepts, thereby avoiding confusion
- Harmonious cohesion with other elements, allowing concise expression
- Practical specification of detail, providing utility
- Linear denotation, for ease of editing

Apart from the specific amalgam, there is one feature that distinguishes the notation from other mathematical notations: the notation forms a context free grammar. Expressions in the notation may therefore easily be interpreted by machine, or compiled into executable code. Moreover, the notation may easily be adapted to any existing programming language, by running it through a pre-processor which converts the notation into the appropriate, but less readable, series of library function calls.

Elements

There are two essential attributes of any data object: the actual representation of the object, and the address where it is stored. The former may be either an atomic element, specific to a particular architecture, e.g. a byte of memory, or a linear array of such atoms, i.e. an n-tuple. The specific meaning or formatting associated with these essentials is an interpretation layered on top. What follows is an explication of semantics associated with the typographical representation of these essentials.

Scalars

are denoted by italic text, e.g. a , b , c , s , etc. In keeping with mathematical notation and existing programming conventions of static and automatic allocation, the address of a scalar's storage location is implicit.

n-tuples

are typically dynamically allocated, and access to addresses (pointers) is important. The address of the storage location of an n-tuple is denoted by bold text, e.g. \mathbf{u} , \mathbf{v} , \mathbf{w} , \mathbf{p} , \mathbf{q} , etc. The actual data value is denoted by **[address]**, e.g. $[\mathbf{v}]$, $[\mathbf{q}]$, etc. This usage is an amalgam of the long standing textual denotation of matrices and vectors found in mathematical tracts, and the usage of a similar notation for dereferencing addresses found in many assembly language syntaxes. It also reflects the notation for array subscripting used in many higher level languages. The symbols $[]$ may be thought of as a dereferencing operator, more fully described below.

Rectangular arrays

are n-tuples with $n = (l)(m)$ and an interpretation imposed which views successive l-tuples as the rows of an $l \times m$ array. These arrays are denoted by bold capital text in the same fashion as n-tuples, e.g. $[\mathbf{A}]$, $[\mathbf{B}]$, $[\mathbf{M}]$, $[\mathbf{N}]$, etc.

Arithmetic operations

Multiplication

e.g. $(a)(b)$, $a[\mathbf{b}] = [\mathbf{ab}]$, $[\mathbf{a}][\mathbf{b}] = [\mathbf{q}]$.

Division

e.g. a/b , $[\mathbf{a}/\mathbf{b}] = [\mathbf{q}]$.

Addition

e.g. $a+b$, $[a+b]$.

Subtraction

e.g. $a-b$, $[a-b]$.

Bracket operations

$[a]$

the n-tuple pointed to by a , i.e. a dereferenced. $[]$ has an operator precedence higher than all other operators, and like parentheses, may be nested, and is evaluated from innermost to outer most.

$[a]$

the n-tuple $[a]$ interpreted as a row vector.

$|b|$

the n-tuple $[b]$ interpreted as a column vector, i.e. the transpose of a row vector.

$[a|b]$

the inner product (i.e. matrix product) of the row vector $[a]$ and the column vector $|b|$, which reflects the notation used by physicist's in a way which is harmonious with the notation for the fundamental semantic here: finite n-tuples.

$[A|$

the rectangular array $[A]$.

$|A|$

the transpose of the rectangular array $[A]$. The transposition operator is second in priority only to the dereferencing operator $[]$.

$[A|A]$

the matrix product of the array $[A]$ and its transpose $|A|$.

[a

the wedge product of [a] and [b]. In three dimensions, this is the cross product, and reflects the notation used in many European mathematical tracts. It also follows the notation of Grassmann algebras, where in four dimensions, [a**b**c] denotes a generalized cross product, i.e. a vector orthogonal to the parallelepiped defined by [a], [b], and [c], with magnitude equal to its volume.

[A B]

the matrix product of two compatible square arrays.

||a||

the norm of the n-tuple [a], i.e. the determinant of [a|a].

||A||

the determinant of [A|A]

|a|

the absolute value or magnitude of the n-tuple [a], i.e. the Euclidean metric equal to the square root of the norm.

|A|

the square root of ||A||, where [A] is an l x m array, i.e. the m-volume of the parallelepiped defined by the m rows of [A] in l dimensional space. (citation: k-Volume in Rn and the Generalized Pythagorean Theorem, Gerald J. Porter, American Mathematical Monthly vol. 103 #3 pp252, Mathematical Association of America, copyright March 1996)

Conventions

Example

A typical three dimensional projective space viewing transformation in left applicative form is denoted by:

$$[w] = [P:T:R:S|v]$$

Where P, T, R, S are perspective, translational, rotational, and scaling matrices respectively. The same operation denoted in right applicative form is:

$$[\mathbf{w}] = [\mathbf{v}|\mathbf{S}:\mathbf{R}:\mathbf{T}:\mathbf{P}]$$

That is, each matrix is transposed immediately following dereferencing, so the stored representation of each matrix is identical for both computations, and the resulting n-tuple is identical.

Reflections

Geometrically, a reflection in an n dimensional space is defined as the operator which fixes an (n-1) dimensional subspace and negates the corresponding mutually exclusive one dimensional subspace. This mutually exclusive one dimensional subspace is called the axis of reflection.

A reflection may be thought of as negating a single coordinate in a given representation, keeping all of the remaining coordinates unchanged. Since the choice of origin and basis are arbitrary, any arbitrary reflection may be so defined by a suitable choice of origin, and selecting one coordinate, say the first in the n-tuple, to represent the orthogonal projection of each vector onto the axis of reflection.

In two dimensions, the line orthogonal to the axis of reflection is called the mirror axis. In three dimensions, the plane orthogonal to the axis of reflection is called the mirror plane. In higher dimensions, the n-1 dimensional subspace orthogonal to the axis of reflection is called the mirror space. Vectors contained within the mirror space project onto the axis of reflection with zero magnitude. Such vectors are therefore unchanged by the reflection, and so it is clear why the (n-1) dimensional mirror space is called the fixed space of a reflection.

Analytic form of a reflection

Arbitrary vectors may be uniquely decomposed into the sum of a vector in the fixed space and a vector on the axis of reflection. The tool for computing such a decomposition is the inner product $[\mathbf{a}|\mathbf{v}]$. A reflection may therefore be concretely represented by a vector $[\mathbf{a}]$ along the axis of reflection. For computational convenience, $[\mathbf{a}]$ is assumed to have unit magnitude. There are two such unit vectors for each reflection, and for the purposes of computing a simple reflection, both are equivalent.

Define $\mathbf{m}(\mathbf{a}, \mathbf{v})$ to be the reflection represented by $[\mathbf{a}]$ applied to an arbitrary vector $[\mathbf{v}]$. Then

$$\mathbf{m}(\mathbf{a}, \mathbf{v}) = [\mathbf{v} - 2[\mathbf{a}|\mathbf{v}]\mathbf{a}]$$

That is, geometrically, subtracting from $[\mathbf{v}]$ the projection of $[\mathbf{v}]$ onto $[\mathbf{a}]$ moves $[\mathbf{v}]$ into the fixed space, and subtracting the same quantity again yields the mirror image of $[\mathbf{v}]$ on the other side of the mirror space.

Rotations

Geometrically, a rotation in an n dimensional space is defined as the operator which fixes an $(n-2)$ dimensional subspace. The corresponding mutually exclusive two dimensional subspace is said to be rotated if distances within this plane are also preserved. This mutually exclusive two dimensional subspace is called the plane of rotation. The set of rotations of a space form a proper subset of the rigid motions of the space, which comprises all reflections, rotations, and translations.

In two dimensions the point orthogonal to the plane of rotation is called the center of rotation. In three dimensions the line orthogonal to the plane of rotation is called the axis of rotation. Logically, in higher dimensions the $(n-2)$ dimensional subspace orthogonal to the plane of rotation would be called the subspace of rotation, but this terminology is apt to be confused with the plane of rotation, and will be avoided. It will suffice to refer to this orthogonal subspace as the fixed space of a rotation.

The progression of terminology is instructive, however, for it makes plain that the essence of a rotation, like a reflection, is the space which moves, and the fixed space is more ambiguous, depending on the dimension of the objects under consideration. Given human immersion in an ostensibly three dimensional Euclidean space, rotations are commonly thought of in terms of the axis of rotation, but this is a misdirection. Spatial rotations are fundamentally four dimensional quantities, which move a plane and fix an orthogonal plane. This truth is an implication of the Hurwitz-Frobenius Theorem, which states that the only division algebras over the real numbers are the real numbers, the complex numbers, the quaternions, and the octonians. There is no three dimensional division algebra, so even though we can not perceive a fixed fourth dimension in any physical rotation, it implicitly exists in the underlying mathematical abstractions.

Rotation: the composition of two reflections

Quaternion representation of a rotation

General rotations in four dimensional space

Canonical form

[This article is under reconstruction. The preceding text is in transition. The following text still embodies concepts not yet incorporated in the revision.]

Algebraic rules

Every quaternion $z = a + bi + cj + dk$ can be viewed as a sum $a + \mathbf{u}$ of a real number a (called the “real part” of the quaternion) and a 3-vector $\mathbf{u} = (b, c, d) = bi + cj + dk$ in \mathbb{R}^3 (called the “imaginary part”).

Two such quaternions are added by adding the real parts and the imaginary parts separately:

$$(a + \mathbf{u}) + (b + \mathbf{v}) = (a + b) + (\mathbf{u} + \mathbf{v})$$

The multiplication of quaternions translates into the following rule:

$$(a + \mathbf{u})(b + \mathbf{v}) = (ab - \langle \mathbf{u}, \mathbf{v} \rangle) + (av + b\mathbf{u} + \mathbf{u} \times \mathbf{v})$$

Here, $\langle \mathbf{u}, \mathbf{v} \rangle$ denotes the scalar product and $\mathbf{u} \times \mathbf{v}$ the vector product of \mathbf{u} and \mathbf{v} .

This formula shows that two quaternions z and w commute, i.e., $zw = wz$, if and only if their imaginary parts are collinear vectors (because, in this case, the vector product of their imaginary parts will simply be equal to the zero vector).

Other properties

Consider the quaternions with modulus 1. They form a multiplicative group, acting on \mathbb{R}^3 : for any such quaternion $z = \cos \frac{\alpha}{2} + \sin \frac{\alpha}{2} \hat{\mathbf{v}}$, the mapping $f(\mathbf{x}) = z \mathbf{x} z^*$ is a counterclockwise rotation through an angle α about an axis \mathbf{v} ; $-z$ is the same rotation. Composition of arbitrary rotations in \mathbb{R}^3 corresponds to the fairly simple operation of quaternion multiplication.

A pair of quaternions also allows for compact representations of rotations in 4D space; this is because the four-dimensional rotation group $SO(4)$ may be written as a semi-direct product of the three-dimensional rotation group $SO(3)$. The quaternions are, in turn, closely related to the double covering of $SO(3)$ by $SU(2)$. Also closely related are the Lorentz group $SL(2, \mathbb{C})$ and the Poincaré group.

Quaternion rotation

It is well known that the vector product is related to rotation in space. The goal then is to find a formula which expresses rotation in 3D space using quaternion multiplication, similar to the formula for a rotation in 2D using complex multiplication,

$$f(w) = zw,$$

where

$$z = e^{\alpha i}$$

is used for rotation by an angle α .

The formula in 3D cannot be a simple multiplication with a quaternion, because rotating a vector should yield a vector. Multiplying a vector with a non-trivial quaternion yields a result with non-zero real part, and thus not a vector.

It turns out that we can cancel the real part if we multiply by a quaternion from one side and with its inverse from the other side. Let $z = a + \mathbf{u}$ be a non-zero quaternion, and consider the function

$$f(\mathbf{v}) = z \mathbf{v} z^{-1}$$

where z^{-1} is the multiplicative inverse of z and \mathbf{v} is a vector, considered as a quaternion with zero real part. The function f is known as *conjugation by z* . Note that the real part of $f(\mathbf{v})$ is zero, because in general zw and wz have the same real part for any quaternions z and w , and so

$$\Re(z \mathbf{v} z^{-1}) = \Re(\mathbf{v} z^{-1} z) = \Re(\mathbf{v} 1) = 0$$

(note that this proof requires the associativity of quaternion multiplication). Furthermore, f is \mathbb{R} -linear and we have $f(\mathbf{v}) = \mathbf{v}$ if and only if \mathbf{v} and the imaginary part \mathbf{u} of z are collinear (because $f(\mathbf{v}) = \mathbf{v}$ means $\mathbf{v} z = z \mathbf{v}$). Hence f is a rotation whose axis of rotation passes through the origin and is given by the vector \mathbf{u} .

Note that conjugation with z is the equivalent to conjugation with rz for any real number r . We can thus restrict our attention to the quaternions of absolute value 1, the so-called *unit quaternions*. Note that even then z and $-z$ represent the same rotation. (The absolute value $|z|$ of the quaternion $z = a + \mathbf{v}$ is defined as the square root of $a^2 + \|\mathbf{v}\|^2$, which makes it multiplicative: $|zw| = |z| |w|$.) Inverting unit quaternions is especially easy: If $|z| = 1$, then $z^{-1} = z^*$ (the conjugate z^* of the quaternion $z = a + \mathbf{v}$ is defined as $z^* = a - \mathbf{v}$) and this makes our rotation formula even easier.

It turns out that the angle of rotation α is also easy to read off if we are dealing with a unit quaternion $z = a + \mathbf{v}$: we have

$$a = \cos \frac{\alpha}{2}.$$

To summarize, a counterclockwise rotation through an angle α about an axis \mathbf{v} can be represented via conjugation by the unit quaternion z

$$z = \cos \frac{\alpha}{2} + \sin \frac{\alpha}{2} \hat{\mathbf{v}}$$

where $\hat{\mathbf{v}}$ is the *normalized vector*

$$\hat{\mathbf{v}} = \frac{\mathbf{v}}{\|\mathbf{v}\|}.$$

The composition of two rotations corresponds to quaternion multiplication: if the rotation f is represented by conjugation with the quaternion z and the rotation g is represented by conjugation with w , then the composition $f \blacksquare g$ is represented by conjugation with zw .

If one wishes to rotate about an axis that doesn't pass through the origin, then one first translates the vectors into the origin, conjugates, and translates back.

The angle between two quaternions should not be confused with the angle of rotation involved in the rotation between the orientations corresponding to these quaternions: the former is half of the latter (or 180° minus half the latter). The angle between the axes of two rotations is again different.

For example the quaternion for the identity is ± 1 and for a 180° rotation about the z -axis is $\pm k$. The angle between the two quaternions is 90° . The angle between the axes of the two rotations is in this case undefined.

An example

Consider the rotation f around the axis $\mathbf{u} = i + j + k$, with a rotation angle of 120° , or $\frac{2\pi}{3}$ radians.

$$\alpha = \frac{2\pi}{3} = 120^\circ$$

The length of \mathbf{u} is $\sqrt{3}$, the half angle is $\pi/3$ (60°) with cosine $\frac{1}{2}$ ($\cos 60^\circ = 0.5$) and sine $\frac{\sqrt{3}}{2}$ ($\sin 60^\circ = 0.866$). We are therefore dealing with a conjugation by the unit quaternion

$$z = \cos \frac{\alpha}{2} + \sin \frac{\alpha}{2} \hat{\mathbf{u}}$$

$$z = \cos 60^\circ + \sin 60^\circ \hat{\mathbf{u}}$$

$$z = \frac{1}{2} + \frac{\sqrt{3}}{2} \cdot \hat{\mathbf{u}}$$

$$z = \frac{1}{2} + \frac{\sqrt{3}}{2} \cdot \frac{(i+j+k)}{\sqrt{3}}$$

$$z = \frac{1+i+j+k}{2}.$$

Concretely,

$$f(ai + bj + ck) = z (ai + bj + ck) z^*.$$

Note that $z^* = 1/z$, as z has unit modulus; here $z^* = (1-i-j-k)/2$. This can be simplified, using the ordinary rules for quaternion arithmetic, to

$$f(ai + bj + ck) = ci + aj + bk,$$

as expected: the rotation corresponds to keeping a cube held fixed at one point, and rotating it 120° about the long diagonal through the fixed point (observe how the three axes are permuted cyclically).

Quaternions versus other representations of rotations

The representation of a rotation as a quaternion (4 numbers) is more compact than the representation as an orthogonal matrix (9 numbers). Furthermore, for a given axis and angle, one can easily construct the corresponding quaternion, and conversely, for a given quaternion one can easily read off the axis and the angle. Both of these are much harder with matrices or Euler angles.

In computer games and other applications, one is often interested in “smooth rotations,” meaning that the scene should slowly rotate and not in a single step. This can be accomplished by choosing a curve such as the spherical linear interpolation in the quaternions, with one endpoint being the identity transformation 1 (or some other initial rotation) and the other being the intended final rotation. This is more problematic with other representations of rotations.

When composing several rotations on a computer, rounding errors necessarily accumulate. A quaternion that’s slightly off still represents a rotation after being normalised—a matrix that’s slightly off need not be orthogonal anymore and therefore is harder to convert back to a proper orthogonal matrix.

The orthogonal matrix corresponding to a rotation by the unit quaternion $z = a + bi + cj + dk$ (with $|z| = 1$) is given by

$$\begin{pmatrix} a^2 + b^2 - c^2 - d^2 & 2bc - 2ad & 2ac + 2bd \\ 2ad + 2bc & a^2 - b^2 + c^2 - d^2 & 2cd - 2ab \\ 2bd - 2ac & 2ab + 2cd & a^2 - b^2 - c^2 + d^2 \end{pmatrix}$$

(Compare the equivalent general formula for a 3×3 rotation matrix in terms of the axis and the angle.)

See also: Charts on SO(3), Euler angles

Pairs of unit quaternions as rotations in 4D space

A pair of unit quaternions z_l and z_r can represent any rotation in 4D space. Given a four dimensional vector \mathbf{v} , and pretending that it is a quaternion, we can rotate the vector \mathbf{v} like this:

$$f(v) = z_l v z_r = \begin{pmatrix} a_l & -b_l & -c_l & -d_l \\ b_l & a_l & -d_l & c_l \\ c_l & d_l & a_l & -b_l \\ d_l & -c_l & b_l & a_l \end{pmatrix} \begin{pmatrix} a_r & -b_r & -c_r & -d_r \\ b_r & a_r & d_r & -c_r \\ c_r & -d_r & a_r & b_r \\ d_r & c_r & -b_r & a_r \end{pmatrix} \begin{pmatrix} w \\ x \\ y \\ z \end{pmatrix}$$

It is straightforward to check that for each matrix M $M^T = I$, that is, that each matrix (and hence both matrices together) represents a rotation. Note that since $(z^l v) z^r = z^l (v z^r)$, the two matrices must commute. Therefore, there are two commuting subgroups of the set of four dimensional rotations. Arbitrary four dimensional rotations have 6 degrees of freedom, each matrix represents 3 of those 6 degrees of freedom.

Since an infinitesimal four-dimensional rotation can be represented by a pair of quaternions (as follows), all (non-infinitesimal) four-dimensional rotations can also be represented.

$$z_l v z_r = \begin{pmatrix} 1 & -dt_{ab} & -dt_{ac} & -dt_{ad} \\ dt_{ab} & 1 & -dt_{bc} & -dt_{bd} \\ dt_{ac} & dt_{bc} & 1 & -dt_{cd} \\ dt_{ad} & dt_{bd} & dt_{cd} & 1 \end{pmatrix} \begin{pmatrix} w \\ x \\ y \\ z \end{pmatrix}$$

$$z_l = \left(1 + \frac{dt_{ab}+dt_{cd}}{2} i + \frac{dt_{ac}-dt_{bd}}{2} j + \frac{dt_{ad}+dt_{bc}}{2} k \right)$$

$$z_r = \left(1 + \frac{dt_{ab}-dt_{cd}}{2} i + \frac{dt_{ac}+dt_{bd}}{2} j + \frac{dt_{ad}-dt_{bc}}{2} k \right)$$

Quaternions are used in computer graphics and related fields because they allow for compact representations of rotations, or correspondingly, orientations, in 3D space:

See also

- Slerp — spherical linear interpolation
- conversion between quaternions and Euler angles
- rotation group
- coordinate rotations
- Clifford algebras
- spinor group

- covering map
- 3-sphere

External links and resources

- [ftp://ftp.cis.upenn.edu/pub/graphics/shoemake/quatut.ps.Z] Shoemake. Quaternion tutorial
- Hart, Francis, Kauffman. Quaternion demo²⁶⁷
- Dam, Koch, Lillholm. Quaternions, Interpolation and Animation²⁶⁸
- Byung-Uk Lee, Unit Quaternion Representation of Rotation²⁶⁹

Source: http://en.wikipedia.org/wiki/Quaternions_and_spatial_rotation

Principal Authors: Patrick, Cyp, Fropuff, Enosch, AxelBoldt, Orderud, Michael Hardy

Radiosity

Radiosity is a global illumination algorithm used in →3D computer graphics rendering. Unlike direct illumination algorithms (such as ray tracing), which tend to simulate light reflecting only once off each surface, global illumination algorithms such as Radiosity simulate the many reflections of light around a scene, generally resulting in softer, more natural shadows.

As a rendering method, Radiosity was introduced in 1984 by researchers at Cornell University (C. Goral, K. E. Torrance, D. P. Greenberg and B. Battaile) in their paper "Modeling the interaction of light between diffuse surfaces". The theory had been in use in engineering to solve problems in radiative heat transfer since about 1950.

Notable commercial Radiosity engines have been Lightscape (now incorporated into the Autodesk 3D Studio Max internal render engine), and Radiozity by Auto*Des*Sys. Radiance (<http://radsite.lbl.gov/radiance/>), an open source Synthetic Image System that seeks physical accurate lightning effects, also makes use of the Radiosity method.

²⁶⁷ <http://graphics.stanford.edu/courses/cs348c-95-fall/software/quatdemo/>

²⁶⁸ <http://www.diku.dk/publikationer/tekniske.rapporter/1998/98-5.ps.gz>

²⁶⁹ <http://home.ewha.ac.kr/~bulee/quaternion.pdf>

Visual characteristics

The inclusion of radiosity calculations in the rendering process often lends an added element of realism to the finished scene, because of the way it mimics real-world phenomena. Consider a red ball sitting on a white floor.

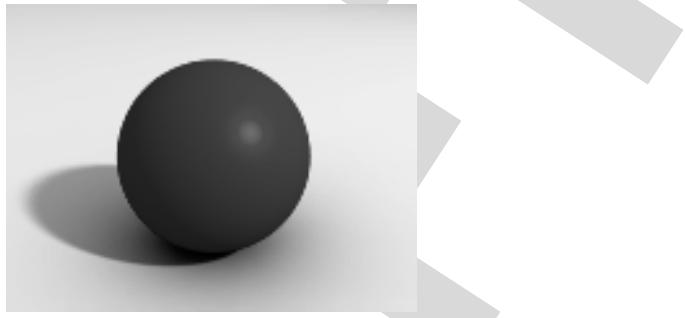


Figure 75 Demonstration of a hybrid of radiosity rendering and ray tracing

Light strikes the ball, casting a shadow, as well as reflecting a tiny amount of red light to surrounding objects - in this case, the floor. This phenomenon gives the white floor, in the vicinity of the ball, a reddish hue. The effect is subtle, but since the human eye is accustomed to its counterpart in the real world, it helps create the illusion of realism.

Physical characteristics

The basic radiosity method has its basis in the theory of thermal radiation, since radiosity relies on computing the amount of light energy transferred between two surfaces. In order to simplify the calculations, the radiosity algorithm assumes that this amount is constant across the surfaces (perfect or ideal Lambertian surfaces); this means that to compute an accurate image, geometry in the scene description must be broken down into smaller areas, or patches, which can then be recombined for the final image.

After this breakdown, the amount of light energy transfer can be computed by using the known reflectivity of the reflecting patch, combined with the *form factor* of the two patches. This dimensionless quantity is computed from the geometric orientation of two patches, and can be thought of as the fraction of the total possible emitting area of the first patch which is covered by the second patch.

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

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

Where:

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

The reciprocity:

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

gives:

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

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

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

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

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

Limitations

As radiosity only deals with the global radiance transfer between objects, position-dependent effects such as reflection (including specular lighting) and

refraction cannot be simulated directly with this method. However, some systems use hybrid approaches, making use of radiosity for illumination and ray tracing (or some other technique) for position-dependent effects.

External links

- Radiosity, by Hugo Elias²⁷⁰ (also provides a general overview of lighting algorithms, along with programming examples)
- Radiosity, by Allen Martin²⁷¹ (a slightly more mathematical explanation of radiosity)

Source: <http://en.wikipedia.org/wiki/Radiosity>

Principal Authors: Wapcaplet, Osmaker, Sallymander, Snorbaard, Uriyan, Mintleaf

Ray casting

Ray casting is not a synonym for ray tracing, but can be thought of as an abridged, and significantly faster, version of the ray tracing algorithm. Both are image order algorithms used in computer graphics to render three dimensional scenes to two dimensional screens by following rays of light from the eye of the observer to a light source. Ray casting does not compute the new tangents a ray of light might take after intersecting a surface on its way from the eye to the source of light. This eliminates the possibility of accurately rendering reflections, refractions, or the natural fall off of shadows – however all of these elements can be faked to a degree, by creative use of texture maps or other methods. The high speed of calculation made ray casting a handy method for the rendering in early real-time 3D video games.

In nature, a light source emits a ray of light which travels, eventually, to a surface that interrupts its progress. One can think of this "ray" as a stream of photons travelling along the same path. At this point, any combination of three things might happen with this light ray: absorption, reflection, and refraction. The surface may reflect all or part of the light ray, in one or more directions. It might also absorb part of the light ray, resulting in a loss of intensity of the

²⁷⁰ <http://freespace.virgin.net/hugo.elias/radiosity/radiosity.htm>

²⁷¹ <http://web.cs.wpi.edu/~matt/courses/cs563/talks/radiosity.html>

reflected and/or refracted light. If the surface has any transparent or translucent properties, it refracts a portion of the light beam into itself in a different direction while absorbing some (or all) of the spectrum (and possibly altering the color). Between absorption, reflection, and refraction, all of the incoming light must be accounted for, and no more. A surface cannot, for instance, reflect 66% of an incoming light ray, and refract 50%, since the two would add up to be 116%. From here, the reflected and/or refracted rays may strike other surfaces, where their absorptive, refractive, and reflective properties are again calculated based on the incoming rays. Some of these rays travel in such a way that they hit our eye, causing us to see the scene and so contribute to the final rendered image. Attempting to simulate this real-world process of tracing light rays using a computer can be considered extremely wasteful, as only a minuscule fraction of the rays in a scene would actually reach the eye.

The first ray casting (versus ray tracing) algorithm used for rendering was presented by A. Appel in 1968. The idea behind ray casting is to shoot rays from the eye, one per pixel, and find the closest object blocking the path of that ray - think of an image as a screen-door, with each square in the screen being a pixel. This is then the object the eye normally sees through that pixel. Using the material properties and the effect of the lights in the scene, this algorithm can determine the shading of this object. The simplifying assumption is made that if a surface faces a light, the light will reach that surface and not be blocked or in shadow. The shading of the surface is computed using traditional 3D computer graphics shading models. One important advantage ray casting offered over older scanline algorithms is its ability to easily deal with non-planar surfaces and solids, such as cones and spheres. If a mathematical surface can be intersected by a ray, it can be rendered using ray casting. Elaborate objects can be created by using solid modelling techniques and easily rendered.

Ray casting for producing computer graphics was first used by scientists at Mathematical Applications Group, Inc., (MAGI) of Elmsford, New York, New York. In 1966 the company was created to perform radiation exposure calculations for the Department of Defense. MAGI's software calculated not only how the gamma rays bounced off of surfaces (ray casting for radiation had been done since the 1940s), but also how they penetrated and refracted within. These studies helped the government to determine certain military applications ; constructing military vehicles that would protect troops from radiation, designing re-entry vehicles for space exploration. Under the direction of Dr. Philip Mittelman, the scientists developed a method of generating images using the same basic software. In 1972 MAGI became a commercial animation studio. This studio used ray casting to generate 3-D computer animation for

television commercials, educational films, and eventually feature films – they created much of the animation in the film *Tron* using ray casting exclusively. MAGI went out of business in 1985.

Wolfenstein 3d

The world in Wolfenstein 3d is built from a square based grid of uniform height walls meeting solid coloured floors and ceilings. In order to draw the world, a single ray is traced for every column of screen pixels and a vertical slice of wall texture is selected and scaled according to where in the world the ray hits a wall and how far it travels before doing so.

The purpose of the grid based levels is twofold - ray to wall collisions can be found more quickly since the potential hits become more predictable and memory overhead is reduced.

Duke Nukem 3d

Duke Nukem 3d uses an algorithm similar to Wolfenstein 3d but abandons the grid based levels, preferring a sector based approach. Levels are designed as a series of convex polyhedral sectors which share edges to form non-convex shapes. The shared edges are known as portals. Rays are cast as in Wolfenstein 3d to determine texture and wall height in every column of the display, but the sector that the ray is in is kept track of in order to minimise collision tests.

Ray casting is used to produce flat floor and ceiling graphics in a very similar way, except that viewer rotation around the y axis is taken into account when drawing pixel rows.

Doom

Doom's use of ray casting is very limited. The Doom engine's primary drawing algorithm is based on a binary space partitioning tree system that determines which walls and floors are visible from any particular position. Ray casting is used to determine texture position and scaling on walls but only after the walls that the rays will hit are already known. This has the advantage of replacing thousands of expensive divide operations with trigonometric multiplies.

'Mode 7'

'Mode 7' is a popular name for the display of textured horizontal surfaces, named after the display mode that allows their pixel plotting to be done in hardware on the Super Nintendo. These require knowledge of texture offset and scale per pixel row and are often implemented using the same trigonometric means as the other algorithms described here.

References and External links

- Wolfenstein style Ray-Casting Tutorial by F. Permadji²⁷²
- Programming a Raycasting Engine Tutorial(QBasic) by Joe King²⁷³
- James D. Foley : *Computer Graphics: Principles and Practice*, Addison-Wesley 1995, ISBN 02-018-4840-6

Source: http://en.wikipedia.org/wiki/Ray_casting

Principal Authors: Mikhajist, RzR, DaveGorman, Pinbucket, Iamhove, Silsor, Reedbeta

Ray tracing

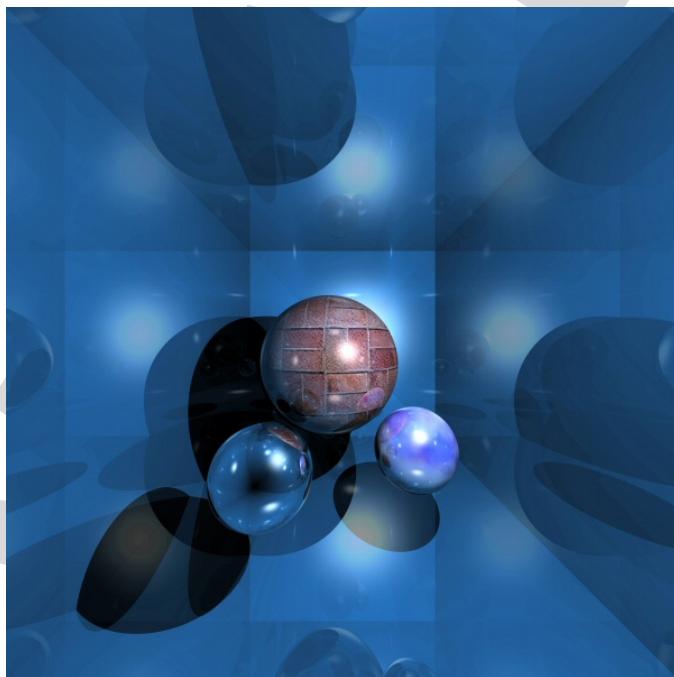


Figure 76 A ray-traced scene

272 <http://www.permadi.com/tutorial/raycast/>

273 <http://killfest.sytes.net/deltacode/tutorials1.shtml>

Ray tracing is a general technique from geometrical optics of modelling the path taken by light by following rays of light as they interact with optical surfaces. It is used in the design of optical systems, such as camera lenses, microscopes, telescopes and binoculars. The term is also applied to mean a specific rendering algorithmic approach in →3D computer graphics, where mathematically-modelled visualisations of programmed scenes are produced using a technique which follows rays from the eyepoint outward, rather than originating at the light sources. It produces results similar to ray casting and scanline rendering, but facilitates more advanced optical effects, such as accurate simulations of reflection and refraction, and is still efficient enough to frequently be of practical use when such high quality output is sought.

Broad description of ray tracing computer algorithm

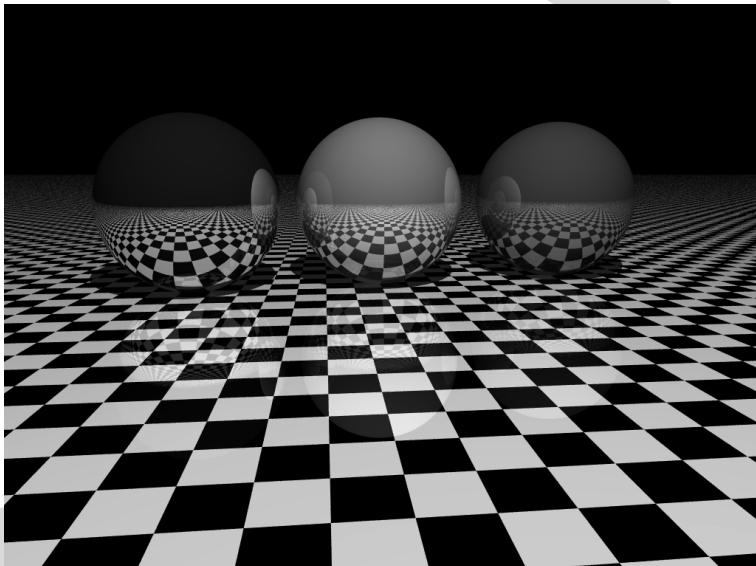


Figure 77 Three spheres, that reflect off the floor and each other

Ray tracing describes a more realistic method than either ray casting or scanline rendering, for producing visual images constructed in →3D computer graphics environments. It works by tracing in reverse, a path that could have been taken by a ray of light which would intersect the imaginary camera lens. As the scene is traversed by following in reverse the path of a very large number

of such rays, visual information on the appearance of the scene as viewed from the point of view of the camera, and in lighting conditions specified to the software, is built up. The ray's reflection, refraction, or absorption are calculated when it intersects objects and media in the scene.

Scenes in raytracing are described mathematically, usually by a programmer, or by a visual artist using intermediary tools, but they may also incorporate data from images and models captured by various technological means, for instance digital photography.

Following rays in reverse is many orders of magnitude more efficient at building up the visual information than would be a genuine simulation of light interactions, since the overwhelming majority of light rays from a given light source do not wind up providing significant light to the viewer's eye, but instead may bounce around until they diminish to almost nothing, or bounce off to infinity. A computer simulation starting with the rays emitted by the light source and looking for ones which wind up intersecting the viewpoint is not practically feasible to execute and obtain accurate imagery.

The obvious shortcut is to pre-suppose that the ray ends up at the viewpoint, then trace backwards. After a stipulated number of maximum reflections has occurred, the light intensity of the point of last intersection is estimated using a number of algorithms, which may include the classic rendering algorithm, and may perhaps incorporate other techniques such as radiosity.

Detailed description of ray tracing computer algorithm and its genesis

What happens in nature

In nature, a light source emits a ray of light which travels, eventually, to a surface that interrupts its progress. One can think of this "ray" as a stream of photons travelling along the same path. In a perfect vacuum this ray will be a straight line. In reality, any combination of three things might happen with this light ray: absorption, reflection, and refraction. A surface may reflect all or part of the light ray, in one or more directions. It might also absorb part of the light ray, resulting in a loss of intensity of the reflected and/or refracted light. If the surface has any transparent or translucent properties, it refracts a portion of the light beam into itself in a different direction while absorbing some (or all) of the spectrum (and possibly altering the color). Between absorption, reflection, and refraction, all of the incoming light must be accounted for, and no more. A surface cannot, for instance, reflect 66% of an incoming light ray, and refract

50%, since the two would add up to be 116%. From here, the reflected and/or refracted rays may strike other surfaces, where their absorptive, refractive, and reflective properties are again calculated based on the incoming rays. Some of these rays travel in such a way that they hit our eye, causing us to see the scene and so contribute to the final rendered image. Attempting to simulate this real-world process of tracing light rays using a computer can be considered extremely wasteful, as only a minuscule fraction of the rays in a scene would actually reach the eye.

Ray casting algorithm

The first ray casting (versus ray tracing) algorithm used for rendering was presented by Arthur Appel in 1968. The idea behind ray casting is to shoot rays from the eye, one per pixel, and find the closest object blocking the path of that ray - think of an image as a screen-door, with each square in the screen being a pixel. This is then the object the eye normally sees through that pixel. Using the material properties and the effect of the lights in the scene, this algorithm can determine the shading of this object. The simplifying assumption is made that if a surface faces a light, the light will reach that surface and not be blocked or in shadow. The shading of the surface is computed using traditional 3D computer graphics shading models. One important advantage ray casting offered over older scanline algorithms is its ability to easily deal with non-planar surfaces and solids, such as cones and spheres. If a mathematical surface can be intersected by a ray, it can be rendered using ray casting. Elaborate objects can be created by using solid modelling techniques and easily rendered.

Ray casting for producing computer graphics was first used by scientists at Mathematical Applications Group, Inc., (MAGI) of Elmsford, New York, New York. In 1966, the company was created to perform radiation exposure calculations for the Department of Defense. MAGI's software calculated not only how the gamma rays bounced off of surfaces (ray casting for radiation had been done since the 1940s), but also how they penetrated and refracted within. These studies helped the government to determine certain military applications ; constructing military vehicles that would protect troops from radiation, designing re-entry vehicles for space exploration. Under the direction of Dr. Philip Mittelman, the scientists developed a method of generating images using the same basic software. In 1972, MAGI became a commercial animation studio. This studio used ray casting to generate 3-D computer animation for television commercials, educational films, and eventually feature films – they created much of the animation in the film *Tron* using ray casting exclusively. MAGI went out of business in 1985.

Ray tracing algorithm

The next important research breakthrough came from Turner Whitted in 1979. Previous algorithms cast rays from the eye into the scene, but the rays were traced no further. Whitted continued the process. When a ray hits a surface, it could generate up to three new types of rays: reflection, refraction, and shadow. A reflected ray continues on in the mirror-reflection direction from a shiny surface. It is then intersected with objects in the scene; the closest object it intersects is what will be seen in the reflection. Refraction rays traveling through transparent material work similarly, with the addition that a refractive ray could be entering or exiting a material. To further avoid tracing all rays in a scene, a shadow ray is used to test if a surface is visible to a light. A ray hits a surface at some point. If the surface at this point faces a light, a ray (to the computer, a line segment) is traced between this intersection point and the light. If any opaque object is found in between the surface and the light, the surface is in shadow and so the light does not contribute to its shade. This new layer of ray calculation added more realism to ray traced images.

Advantages of ray tracing

Ray tracing's popularity stems from its basis in a realistic simulation of lighting over other rendering methods (such as scanline rendering or ray casting). Effects such as reflections and shadows, which are difficult to simulate using other algorithms, are a natural result of the ray tracing algorithm. Relatively simple to implement yet yielding impressive visual results, ray tracing often represents a first foray into graphics programming.

Disadvantages of ray tracing

A serious disadvantage of ray tracing is performance. Scanline algorithms and other algorithms use data coherence to share computations between pixels, while ray tracing normally starts the process anew, treating each eye ray separately. However, this separation offers other advantages, such as the ability to shoot more rays as needed to perform anti-aliasing and improve image quality where needed. Although it does handle interreflection and optical effects such as refraction accurately, traditional Ray Tracing is also not necessarily photorealistic. True photorealism occurs when the rendering equation is closely approximated or fully implemented. Implementing the rendering equation gives true photorealism. As the equation describes every physical effect of light flow. However, this is usually infeasable given the computing resources required. The realism of all rendering methods, then, must be evaluated as an

approximation to the equation, and in the case of Ray Tracing, it is not necessarily the most realistic. Other methods, including photon mapping, are based upon raytracing for certain parts of the algorithm, yet give far better results.

Reversed direction of traversal of scene by the rays

The process of shooting rays from the eye to the light source to render an image is sometimes referred to as *backwards ray tracing*, since it is the opposite direction photons actually travel. However, there is confusion with this terminology. Early ray tracing was always done from the eye, and early researchers such as James Arvo used the term *backwards ray tracing* to refer to shooting rays from the lights and gathering the results. As such, it is clearer to distinguish *eye-based* versus *light-based* ray tracing. Research over the past decades has explored combinations of computations done using both of these directions, as well as schemes to generate more or fewer rays in different directions from an intersected surface. For example, radiosity algorithms typically work by computing how photons emitted from lights affect surfaces and storing these results. This data can then be used by a standard recursive ray tracer to create a more realistic and physically correct image of a scene. In the context of global illumination algorithms, such as photon mapping and →Metropolis light transport, ray tracing is simply one of the tools used to compute light transfer between surfaces.

Algorithm: classical recursive ray tracing

```

For each pixel in image {
    Create ray from eyepoint passing through this pixel
    Initialize NearestT to INFINITY and NearestObject to NULL

    For every object in scene {
        If ray intersects this object {
            If t of intersection is less than NearestT {
                Set NearestT to t of the intersection
                Set NearestObject to this object
            }
        }
    }

    If NearestObject is NULL {
        Fill this pixel with background color
    } Else {
        Shoot a ray to each light source to check if in shadow
    }
}

```

```

    If surface is reflective, generate reflection ray: recurse
    If surface is transparent, generate refraction ray: re-
recurse
    Use NearestObject and NearestT to compute shading function
    Fill this pixel with color result of shading function
}
}
}
```

Ray tracing in real time

There has been some effort for implementing ray tracing in real time speeds for interactive 3D graphics applications such as computer and video games.

The →OpenRT project includes a highly-optimized software core for ray tracing along with an →OpenGL-like API in order to offer an alternative to the current rasterization based approach for interactive 3D graphics.

→Ray tracing hardware, such as the experimental Ray Processing Unit developed at the Saarland University, has been designed to accelerate some of the computationally intensive operations of ray tracing.

Some real-time software 3D engines based on ray tracing have been developed by hobbyist demo programmers since the late 1990's. The ray tracers used in demos, however, often use inaccurate approximations and even cheating in order to attain reasonably high frame rates.²⁷⁴

In optical design

Ray tracing in computer graphics derives its name and principles from a much older technique used for lens design since the 1900s. *Geometric ray tracing* is used to describe the propagation of light rays through a lens system or optical instrument, allowing the image-forming properties of the system to be modeled. This is used to optimize the design of the instrument (e.g. to minimize effects such as chromatic and other aberrations) before it is built. Ray tracing is also used to calculate optical path differences through optical systems, which are used to calculate optical wavefronts, which in turn are used to calculate system diffraction effects such as point spread function, MTF, and Strehl ratio. It is not only used for designing lenses, as for photography, but can also be used for longer wavelength applications such as designing microwave or even radio systems, and for shorter wavelengths, such as ultraviolet and X-ray optics.

²⁷⁴ <http://www.acm.org/tog/resources/RTNews/demos/overview.htm>

The principles of ray tracing for computer graphics and optical design are similar, but the technique in optical design usually uses much more rigorous and physically correct models of how light behaves. In particular, optical effects such as dispersion, diffraction and the behaviour of optical coatings are important in lens design, but are less so in computer graphics.

Before the advent of the computer, ray tracing calculations were performed by hand using trigonometry and logarithmic tables. The optical formulas of many classic photographic lenses were optimized by rooms full of people, each of whom handled a small part of the large calculation. Now they are worked out in optical design software such as OSLO or TracePro from Lambda Research, Code-V or Zemax. A simple version of ray tracing known as ray transfer matrix analysis is often used in the design of optical resonators used in lasers.

Example

As a demonstration of the principles involved in raytracing, let us consider how one would find the intersection between a ray and a sphere. The general equation of a sphere, where \mathbf{I} is a point on the surface of the sphere, \mathbf{C} is its centre and r is its radius, is $|\mathbf{I} - \mathbf{C}|^2 = r^2$. Equally, if a line is defined by its starting point \mathbf{S} (consider this the starting point of the ray) and its direction \mathbf{d} (consider this the direction of that particular ray), each point on the line may be described by the expression

$$\mathbf{S} + t\mathbf{d},$$

where t is a constant defining the distance along the line from the starting point (hence, for simplicity's sake, \mathbf{d} is generally a unit vector). Now, in the scene we know \mathbf{S} , \mathbf{d} , \mathbf{C} , and r . Hence we need to find t as we substitute in for \mathbf{I} :

$$|\mathbf{S} + t\mathbf{d} - \mathbf{C}|^2 = r^2.$$

Let $\mathbf{V} \equiv \mathbf{S} - \mathbf{C}$ for simplicity, then

$$|\mathbf{V} + t\mathbf{d}|^2 = r^2$$

$$\mathbf{V}^2 + t^2\mathbf{d}^2 + 2\mathbf{V} \cdot \mathbf{d}t = r^2$$

$$d^2t^2 + 2\mathbf{V} \cdot \mathbf{d}t + \mathbf{V}^2 - r^2 = 0.$$

Now this quadratic equation has solutions

$$t = \frac{-2\mathbf{V} \cdot \mathbf{d} \pm \sqrt{(2\mathbf{V} \cdot \mathbf{d})^2 - 4d^2(V^2 - r^2)}}{2d^2}.$$

This is merely the math behind a straight ray-sphere intersection. There is of course far more to the general process of raytracing, but this demonstrates an example of the algorithms used.

See also

- Actual state
- →Beam tracing
- BRL-CAD
- →Cone tracing
- Distributed ray tracing
- →Global illumination
- Line-sphere intersection
- Pencil tracing
- Philipp Slusallek
- →Photon mapping
- POV-Ray
- Powerwall
- →Radiosity
- Radiance (software)
- Target state
- YafRay

References

- Glassner, Andrew (Ed.) (1989). *An Introduction to Ray Tracing*. Academic Press. ISBN 0-12-286160-4.
- Shirley, Peter and Morley Keith, R. (2001) *Realistic Ray Tracing, 2nd edition*. A.K. Peters. ISBN 1-56881-198-5.
- Henrik Wann Jensen. (2001) *Realistic image synthesis using photon mapping*. A.K. Peters. ISBN 1-56881-147-0.
- Pharr, Matt and Humphreys, Greg (2004). *Physically Based Rendering : From Theory to Implementation*. Morgan Kaufmann. ISBN 0-12-553180-X.

External links

- The Ray Tracing News²⁷⁵ - short research articles and new links to resources
- Games using realtime raytracing²⁷⁶
- A series of raytracing tutorials for the implementation of an efficient ray-tracer using C++²⁷⁷
- Mini ray tracers written equivalently in various languages²⁷⁸

Raytracing software

- POV-Ray²⁷⁹
- PBRT²⁸⁰ - a Physically Based Raytracer
- Tachyon²⁸¹
- Rayshade²⁸²
- OpenRT²⁸³ - realtime raytracing library
- Raster3D²⁸⁴
- RealStorm Engine²⁸⁵ - a realtime raytracing engine
- BRL-CAD²⁸⁶
- More ray tracing source code links²⁸⁷
- Zemax²⁸⁸
- Radiance²⁸⁹
- Yafray²⁹⁰
- OSLO²⁹¹ - Lens design and optimization software; OSLO-EDU is a free download
- TracePro²⁹² - Straylight and illumination software with a CAD-like interface

²⁷⁵ <http://www.raytracingnews.org/>

²⁷⁶ <http://graphics.cs.uni-sb.de/RTGames/>

²⁷⁷ http://www.devmaster.net/articles/raytracing_series/part1.php

²⁷⁸ http://www.ffconsultancy.com/free/ray_tracer/languages.html

²⁷⁹ <http://www.povray.org/>

²⁸⁰ <http://www.pbrt.org>

²⁸¹ <http://jedi.ks.uiuc.edu/~johns/raytracer>

²⁸² <http://graphics.stanford.edu/~cek/rayshade>

²⁸³ <http://www.openrt.de/>

²⁸⁴ <http://www.bmsc.washington.edu/raster3d/raster3d.html>

²⁸⁵ <http://www.realstorm.com>

²⁸⁶ <http://brlcad.org/>

²⁸⁷ <http://www.acm.org/tog/Software.html#ray>

²⁸⁸ <http://www.zemax.com/>

²⁸⁹ <http://radsite.lbl.gov/radiance/>

²⁹⁰ <http://www.yafray.org>

²⁹¹ <http://www.lambdares.com/products/oslo/>

²⁹² <http://www.lambdares.com/products/tracepro/>

Source: http://en.wikipedia.org/wiki/Ray_tracing

Principal Authors: Pinbucket, DrBob, Srleffler, Jawed, Osmaker, Carrionluggage, ToastieIL, Stevertigo, Viznut, Ed g2s

Ray tracing hardware

Ray tracing hardware is a special purpose computer hardware design for accelerating real-time ray tracing.

Ray Tracing vs. Rasterization

The problem of rendering 3D graphics can be conceptually presented as finding all intersections between a set of "primitives" (typically triangles or polygons) and a set of "rays" (typically one or more per pixel).

Typical graphic acceleration boards, so called GPUs use rasterization algorithm. In this approach, in each step, the GPU finds all intersections of a single primitive and all rays of the screen. Well known brands such as intel, NVIDIA and ATI have developed very efficient algorithms to do so.

The ray tracing algorithm solves the rendering problem in a different way. In each step, it finds all intersections of a single ray with all primitives of the scene. The ray tracing algorithm has been around for some decades, but only recent research from the Universities of Saarland and Utah, as well as inTrace GmbH have led to the first interactive ray tracing application that is able to compute this algorithm at several frames per second.

Both approaches have different benefits and drawbacks. Rasterization can be performed in a stream-like manner, one triangle at the time, and access to complete scene is not needed. The drawback of rasterization is that non-local effects, like reflections, refraction, shadows and realistic lighting, are very difficult to compute.

The ray tracing algorithm can also be parallelized very well, if one considers each ray separately, however its memory bandwidth is crucial as various parts of the scene need to be accessed. But it can easily compute various kinds of physically correct effects, providing much more realistic impression than rasterization.

An additional advantage is logarithmic complexity of ray tracing in number of scene objects and possibility of using more complex primitives without their

prior triangulation. The biggest drawback for real time ray tracing is the need for quite costly recomputations of spatial index structures in case of highly dynamic scenes. Some problems remain to be solved, such as ray tracing of lines and text.

Implementations

Today there are two existing implementations of ray tracing hardware:

- There is the ART VPS company, situated in the UK, that sells ray tracing hardware for offline rendering.
- There is the Saarland University, striving to develop a real-time ray tracing chip, so far they have published two designs: SaarCOR and the RPU.

See also

- →OpenRT Only available real time ray tracing software

External links

- ART VPS²⁹³

Source: http://en.wikipedia.org/wiki/Ray_tracing_hardware

Reflection mapping

In Computer Graphics, **reflection mapping** is an efficient method of simulating a complex mirroring surface by means of a precomputed texture image. The texture is used to store the image of the environment surrounding the rendered object. There are several ways of storing the surrounding environment; the most common ones are the **Standard Environment Mapping** in which a single texture contains the image of the surrounding as reflected on a mirror ball, or the **Cubic Environment Mapping** in which the environment is *unfolded* onto the six faces of a cube and stored therefore as six square textures.

This kind of approach is more efficient than the classical ray tracing approach of computing the exact reflection by shooting a ray and following its optically

²⁹³ <http://www.art-render.com/>



Figure 78 An example of reflection mapping. Observe the crudeness of the output.

exact path, but it should be noted that these are (sometimes crude) approximation of the real reflection. A typical drawback of this technique is the absence of self reflections: you cannot see any part of the reflected object inside the reflection itself.

Types of Reflection Mapping

Standard Environment Mapping

Standard environment mapping, more commonly known as **spherical environment mapping**, involves the use of a textured sphere infinitely far away from the object that reflects it. By creating a spherical texture using a fisheye lens or via prerendering or with a light probe, this texture is mapped to a hollow sphere, and the texel colors are determined by calculating the light vectors from the points on the object to the texels in the environment map. This technique is similar to raytracing, but incurs less of a performance hit because all of the colors of the points to be referenced are known beforehand by the GPU, so all it has to do is to calculate the angles of incidence and reflection.

There are a few glaring limitations to spherical mapping. For one thing, due to the nature of the texture used for the map, there is an abrupt point of singularity on the backside of objects using spherical mapping. **Cube mapping**

(see below) was developed to address this issue. Since cube maps (if made and filtered correctly) have no visible seems, they are an obvious successor to the archaic sphere maps, and nowadays spherical environment maps are almost nonexistent in certain contemporary graphical applications, such as video game graphics. However, since cubic environment maps require six times the image data of sphere maps (a cube has six sides), some people are still reluctant to adopt them.

Cubic Environment Mapping

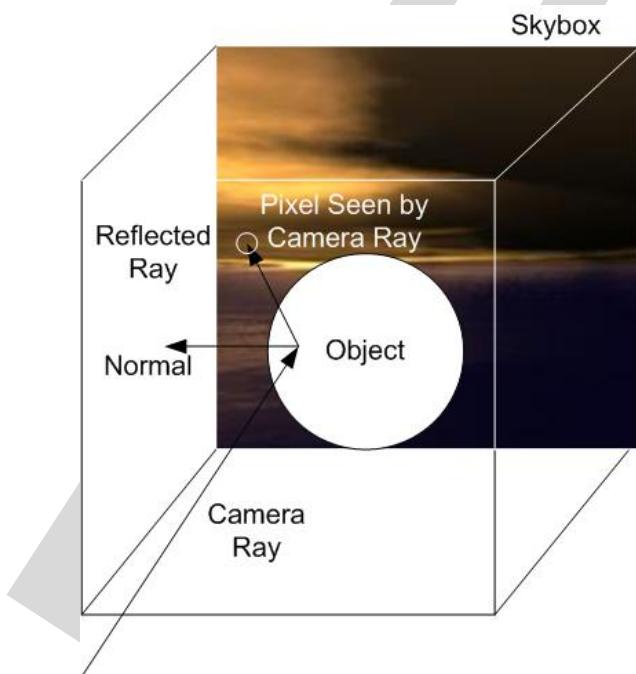


Figure 79 A diagram depicting how cube mapped reflection works.

Cube mapped reflection is a technique that uses cube mapping to make objects look like they reflect the environment around them. Generally, this is done with the same skybox that is used in outdoor renderings. Though this is not a true reflection since objects around the reflective one will not be seen in the reflection, the desired effect is usually achieved.

Cube mapped reflection is done by determining the vector that the object is being viewed at. This **camera ray** is reflected about the surface normal of

where the camera vector intersects the object. This results in the **reflected ray** which is then passed to the cube map to get the texel which the camera then sees as if it is on the surface of the object. This creates the effect that the object is reflective.

Application in Real-Time 3D Graphics

Standard Environment Mapping

Cubic Environment Mapping

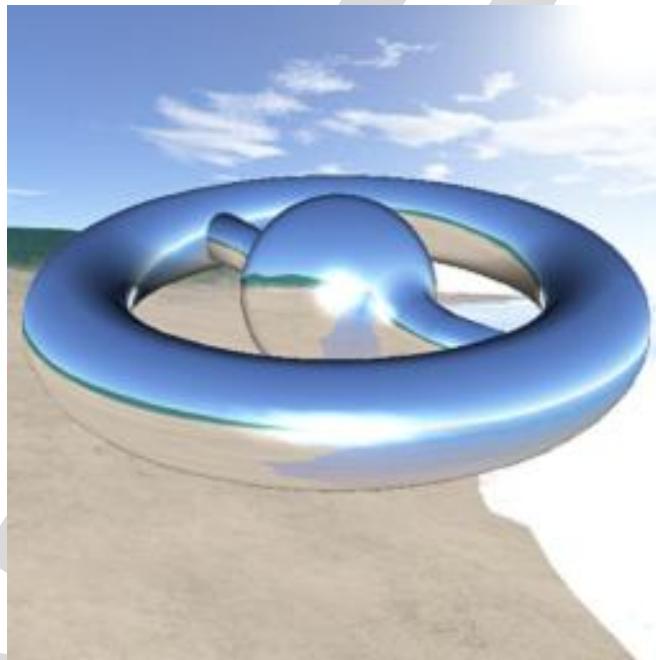


Figure 80 Example of a three-dimensional model using cube mapped reflection

Cube mapped reflection, when used correctly, may be the fastest method of rendering a reflective surface. To increase the speed of rendering, each vertex calculates the position of the reflected ray. Then, the position is interpolated across polygons to which the vertex is attached. This eliminates the need for recalculating every pixel's reflection.

See Also

- The Story of Reflection mapping²⁹⁴ cured by Paul Debevec
- Skybox (video games)
- →Cube mapping

Source: http://en.wikipedia.org/wiki/Reflection_mapping

Principal Authors: ALoopingIcon, Gaius Cornelius, Abdull, Srleffler, Freeformer

Rendering (computer graphics)

Rendering is the process of generating an image from a model, by means of a software program. The model is a description of three dimensional objects in a strictly defined language or data structure. It would contain geometry, viewpoint, texture and lighting information. The image is a digital image or raster graphics image. The term may be by analogy with an "artist's rendering" of a scene. 'Rendering' is also used to describe the process of calculating effects in a video editing file to produce final video output.

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

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

In the case of 3D graphics, rendering may be done slowly, as in pre-rendering, or in real time. Pre-rendering is a computationally intensive process that is typically used for movie creation, while real-time rendering is often done for

²⁹⁴ <http://www.debevec.org/ReflectionMapping/>

3D video games which rely on the use of graphics cards with 3D hardware accelerators.

Usage

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

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

Features

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

- **shading** — how the color and brightness of a surface varies with lighting
- **texture-mapping** — a method of applying detail to surfaces
- **bump-mapping** — a method of simulating small-scale bumpiness on surfaces
- **fogging/participating medium** — how light dims when passing through non-clear atmosphere or air
- **shadows** — the effect of obstructing light
- **soft shadows** — varying darkness caused by partially obscured light sources
- **reflection** — mirror-like or highly glossy reflection
- **transparency** — sharp transmission of light through solid objects
- **translucency** — highly scattered transmission of light through solid objects
- **refraction** — bending of light associated with transparency
- **indirect illumination** — surfaces illuminated by light reflected off other surfaces, rather than directly from a light source
- **caustics** (a form of indirect illumination) — reflection of light off a shiny object, or focusing of light through a transparent object, to produce bright highlights on another object
- **depth of field** — objects appear blurry or out of focus when too far in front of or behind the object in focus
- **motion blur** — objects appear blurry due to high-speed motion, or the motion of the camera

- **photorealistic morphing** — photoshopping 3D renderings to appear more life-like
- **non-photorealistic rendering** — rendering of scenes in an artistic style, intended to look like a painting or drawing

Techniques

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

Tracing every ray of light in a scene would be impractical and would take gigantic amounts of time. Even tracing a portion large enough to produce an image takes an inordinate amount of time if the sampling is not intelligently restricted.

Therefore, four loose families of more-efficient light transport modelling techniques have emerged: **rasterisation**, including **scanline rendering**, considers the objects in the scene and projects them to form an image, with no facility for generating a point-of-view perspective effect; **ray casting** considers the scene as observed from a specific point-of-view, calculating the observed image based only on geometry and very basic optical laws of reflection intensity, and perhaps using Monte Carlo techniques to reduce artifacts; **radiosity** uses finite element mathematics to simulate diffuse spreading of light from surfaces; and **ray tracing** is similar to ray casting, but employs more advanced optical simulation, and usually uses Monte Carlo techniques, to obtain more realistic results, at a speed which is often orders of magnitude slower.

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

Scanline rendering and rasterisation

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

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

Rasterization is frequently faster than pixel-by-pixel rendering. First, large areas of the image may be empty of primitives; rasterization will ignore these areas, but pixel-by-pixel rendering must pass through them. Second, rasterization can improve cache coherency and reduce redundant work by taking advantage of the fact that the pixels occupied by a single primitive tend to be contiguous in the image. For these reasons, rasterization is usually the approach of choice when interactive rendering is required; however, the pixel-by-pixel approach can often produce higher-quality images and is more versatile because it does not depend on as many assumptions about the image as rasterization.

Rasterization exists in two main forms, not only when an entire face (primitive) is rendered but when the vertices of a face are all rendered and then the pixels on the face which lie between the vertices rendered using simple blending of each vertex colour to the next, this version of rasterization has overtaken the old method as it allows the graphics to flow without complicated textures (a rasterized image when used face by face tends to have a very block like effect if not covered in complex textures, the faces aren't smooth because there is no gradual smoothness from one pixel to the next,) this means that you can utilise the graphics card's more taxing shading functions and still achieve better performance because you have freed up space on the card because complex textures aren't necessary. sometimes people will use one rasterization method on some faces and the other method on others based on the angle at which that face meets other joined faces, this can increase speed and not take away too much from the images overall effect.

Ray casting

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

The geometry which has been modelled is parsed pixel by pixel, line by line, from the point of view outward, as if casting rays out from the point of view. Where an object is intersected, the colour value at the point may be evaluated using several methods. In the simplest, the colour value of the object at the point of intersection becomes the value of that pixel. The colour may be determined from a texture-map. A more sophisticated method is to modify the colour value by an illumination factor, but without calculating the relationship

to a simulated light source. To reduce artifacts, a number of rays in slightly different directions may be averaged.

Rough simulations of optical properties may be additionally employed: commonly, making a very simple calculation of the ray from the object to the point of view. Another calculation is made of the angle of incidence of light rays from the light source(s). And from these and the specified intensities of the light sources, the value of the pixel is calculated.

Or illumination plotted from a radiosity algorithm could be employed. Or a combination of these.

Radiosity

→Radiosity is a method which attempts to simulate the way in which reflected light, instead of just reflecting to another surface, also illuminates the area around it. This produces more realistic shading and seems to better capture the 'ambience' of an indoor scene. A classic example used is of the way that shadows 'hug' the corners of rooms.

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

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

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

Due to the iterative/recursive nature of the technique, complex objects are particularly slow to emulate. Advanced radiosity calculations may be reserved for calculating the *ambiance* of the room, from the light reflecting off walls, floor and ceiling, without examining the contribution that complex objects make to the radiosity – or complex objects may be replaced in the radiosity calculation with simpler objects of similar size and texture.

If there is little rearrangement of radiosity objects in the scene, the same radiosity data may be reused for a number of frames, making radiosity an effective way to improve on the flatness of ray casting, without seriously impacting the overall rendering time-per-frame.

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

Ray tracing

Ray tracing is an extension of the same technique developed in scanline rendering and ray casting. Like those, it handles complicated objects well, and the objects may be described mathematically. Unlike scanline and casting, ray tracing is almost always a Monte Carlo technique, that is one based on averaging a number of randomly generated samples from a model.

In this case, the samples are imaginary rays of light intersecting the viewpoint from the objects in the scene. It is primarily beneficial where complex and accurate rendering of shadows, refraction or reflection are issues.

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

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

In some cases, at each point of intersection, multiple rays may be spawned.

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

However, efforts at optimising to reduce the number of calculations needed in portions of a work where detail is not high or does not depend on raytracing features have led to a realistic possibility of wider use of ray tracing. There is now some hardware accelerated ray tracing equipment, at least in prototype

phase, and some game demos which show use of realtime software or hardware ray tracing.

Optimisation

Optimisations used by an artist when a scene is being developed

Due to the large number of calculations, a work in progress is usually only rendered in detail appropriate to the portion of the work being developed at a given time, so in the initial stages of modelling, wireframe and ray casting may be used, even where the target output is ray tracing with radiosity. It is also common to render only parts of the scene at high detail, and to remove objects that are not important to what is currently being developed.

Common optimisations for real time rendering

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

There are some lesser known approaches to rendering, such as spherical harmonics. These techniques are lesser known often due to slow speed, lack of practical use or simply because they are in early stages of development; maybe some will offer a new solution.

Sampling and filtering

One problem that any rendering system must deal with, no matter which approach it takes, is the **sampling problem**. Essentially, the rendering process tries to depict a continuous function from image space to colors by using a finite number of pixels. As a consequence of the Nyquist theorem, the scanning frequency must be twice the dot rate, which is proportional to image resolution. In simpler terms, this expresses the idea that an image cannot display details smaller than one pixel.

If a naive rendering algorithm is used, high frequencies in the image function will cause ugly aliasing to be present in the final image. Aliasing typically manifests itself as jaggies, or jagged edges on objects where the pixel grid is visible. In order to remove aliasing, all rendering algorithms (if they are to produce good-looking images) must filter the image function to remove high frequencies, a process called antialiasing.

See also

- the painter's algorithm
- Scanline algorithms like Reyes
- Z-buffer algorithms
- →Global illumination
- →Radiosity
- →Ray tracing
- →Volume rendering

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

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

Movie type rendering software includes:

- RenderMan compliant renderers
- Mental Ray
- Brazil
- Blender (may also be used for modeling)
- LightWave (includes modelling module)

Academic core

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

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

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

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

The rendering equation

Main article: Rendering equation

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

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

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

The Bidirectional Reflectance Distribution Function

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

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

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

Geometric optics

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

Visual perception

Though it receives less attention, an understanding of human visual perception is valuable to rendering. This is mainly because image displays and human perception have restricted ranges. A renderer can simulate an almost infinite range

of light brightness and color, but current displays — movie screen, computer monitor, etc. — cannot handle so much, and something must be discarded or compressed. Human perception also has limits, and so doesn't need to be given large-range images to create realism. This can help solve the problem of fitting images into displays, and, furthermore, suggest what short-cuts could be used in the rendering simulation, since certain subtleties won't be noticeable. This related subject is tone mapping.

Mathematics used in rendering includes: linear algebra, calculus, numerical mathematics, signal processing, monte carlo.

Chronology of important published ideas

- 1970 **Scan-line algorithm** (Bouknight, W. J. (1970). A procedure for generation of three-dimensional half-tone computer graphics presentations. *Communications of the ACM*)
- 1971 **Gouraud shading** (Gouraud, H. (1971). Computer display of curved surfaces. *IEEE Transactions on Computers* **20** (6), 623–629.)
- 1974 **Texture mapping** (Catmull, E. (1974). A subdivision algorithm for computer display of curved surfaces. *PhD thesis*, University of Utah.)
- 1974 **Z-buffer** (Catmull, E. (1974). A subdivision algorithm for computer display of curved surfaces. *PhD thesis*)
- 1975 **Phong shading** (Phong, B-T. (1975). Illumination for computer generated pictures. *Communications of the ACM* **18** (6), 311–316.)
- 1976 **Environment mapping** (Blinn, J.F., Newell, M.E. (1976). Texture and reflection in computer generated images. *Communications of the ACM* **19**, 542–546.)
- 1977 **Shadow volumes** (Crow, F.C. (1977). Shadow algorithms for computer graphics. *Computer Graphics (Proceedings of SIGGRAPH 1977)* **11** (2), 242–248.)
- 1978 **Shadow buffer** (Williams, L. (1978). Casting curved shadows on curved surfaces. *Computer Graphics (Proceedings of SIGGRAPH 1978)* **12** (3), 270–274.)
- 1978 **Bump mapping** (Blinn, J.F. (1978). Simulation of wrinkled surfaces. *Computer Graphics (Proceedings of SIGGRAPH 1978)* **12** (3), 286–292.)
- 1980 **BSP trees** (Fuchs, H. Kedem, Z.M. Naylor, B.F. (1980). On visible surface generation by a priori tree structures. *Computer Graphics (Proceedings of SIGGRAPH 1980)* **14** (3), 124–133.)
- 1980 **Ray tracing** (Whitted, T. (1980). An improved illumination model for shaded display. *Communications of the ACM* **23** (6), 343–349.)

- 1981 **Cook shader** (Cook, R.L. Torrance, K.E. (1981). A reflectance model for computer graphics. *Computer Graphics (Proceedings of SIGGRAPH 1981)* **15** (3), 307–316.)
- 1983 **Mipmaps** (Williams, L. (1983). Pyramidal parametrics. *Computer Graphics (Proceedings of SIGGRAPH 1983)* **17** (3), 1–11.)
- 1984 **Octree ray tracing** (Glassner, A.S. (1984). Space subdivision for fast ray tracing. *IEEE Computer Graphics & Applications* **4** (10), 15–22.)
- 1984 **Alpha compositing** (Porter, T. Duff, T. (1984). Compositing digital images. *Computer Graphics (Proceedings of SIGGRAPH 1984)* **18** (3), 253–259.)
- 1984 **Distributed ray tracing** (Cook, R.L. Porter, T. Carpenter, L. (1984). Distributed ray tracing. *Computer Graphics (Proceedings of SIGGRAPH 1984)* **18** (3), 137–145.)
- 1984 **Radiosity** (Goral, C. Torrance, K.E. Greenberg, D.P. Battaile, B. (1984). Modelling the interaction of light between diffuse surfaces. *Computer Graphics (Proceedings of SIGGRAPH 1984)* **18** (3), 213–222.)
- 1985 **Hemi-cube radiosity** (Cohen, M.F. Greenberg, D.P. (1985). The hemi-cube: a radiosity solution for complex environments. *Computer Graphics (Proceedings of SIGGRAPH 1985)* **19** (3), 31–40.)
- 1986 **Light source tracing** (Arvo, J. (1986). Backward ray tracing. *SIGGRAPH 1986 Developments in Ray Tracing course notes*)
- 1986 **Rendering equation** (Kajiya, J.T. (1986). The rendering equation. *Computer Graphics (Proceedings of SIGGRAPH 1986)* **20** (4), 143–150.)
- 1987 **Reyes algorithm** (Cook, R.L. Carpenter, L. Catmull, E. (1987). The reyes image rendering architecture. *Computer Graphics (Proceedings of SIGGRAPH 1987)* **21** (4), 95–102.)
- 1991 **Hierarchical radiosity** (Hanrahan, P. Salzman, D. Aupperle, L. (1991). A rapid hierarchical radiosity algorithm. *Computer Graphics (Proceedings of SIGGRAPH 1991)* **25** (4), 197–206.)
- 1993 **Tone mapping** (Tumblin, J. Rushmeier, H.E. (1993). Tone reproduction for realistic computer generated images. *IEEE Computer Graphics & Applications* **13** (6), 42–48.)
- 1993 **Subsurface scattering** (Hanrahan, P. Krueger, W. (1993). Reflection from layered surfaces due to subsurface scattering. *Computer Graphics (Proceedings of SIGGRAPH 1993)* **27** (), 165–174.)
- 1995 **Photon mapping** (Jensen, H.J. Christensen, N.J. (1995). Photon maps in bidirectional monte carlo ray tracing of complex objects. *Computers & Graphics* **19** (2), 215–224.)

See also

- →Pre-rendered
- →Graphics pipeline
- Virtual model

Books and summaries

- Foley; Van Dam; Feiner; Hughes (1990). *Computer Graphics: Principles And Practice*. Addison Wesley. ISBN 0201121107.
- Glassner (1995). *Principles Of Digital Image Synthesis*. Morgan Kaufmann. ISBN 1558602763.
- Pharr; Humphreys (2004). *Physically Based Rendering*. Morgan Kaufmann. ISBN 012553180X.
- Dutre; Bala; Bekaert (2002). *Advanced Global Illumination*. AK Peters. ISBN 1568811772.
- Jensen (2001). *Realistic Image Synthesis Using Photon Mapping*. AK Peters. ISBN 1568811470.
- Shirley; Morley (2003). *Realistic Ray Tracing* (2nd ed.). AK Peters. ISBN 1568811985.
- Glassner (1989). *An Introduction To Ray Tracing*. Academic Press. ISBN 0122861604.
- Cohen; Wallace (1993). *Radiosity and Realistic Image Synthesis*. AP Professional. ISBN 0121782700.
- Akenine-Moller; Haines (2002). *Real-time Rendering* (2nd ed.). AK Peters. ISBN 1568811829.
- Gooch; Gooch (2001). *Non-Photorealistic Rendering*. AKPeters. ISBN 1568811330.
- Strothotte; Schlechtweg (2002). *Non-Photorealistic Computer Graphics*. Morgan Kaufmann. ISBN 1558607870.
- Blinn (1996). *Jim Blinns Corner - A Trip Down The Graphics Pipeline*. Morgan Kaufmann. ISBN 1558603875.
- Description of the 'Radiance' system²⁹⁵

²⁹⁵ <http://radsite.lbl.gov/radiance/papers/sg94.1/>

External links

- SIGGRAPH²⁹⁶ The ACMs special interest group in graphics — the largest academic and professional association and conference.
- Ray Tracing News²⁹⁷ A newsletter on ray tracing technical matters.
- Real-Time Rendering resources²⁹⁸ A list of links to resources, associated with the *Real-Time Rendering* book.
- <http://www.graphicspapers.com/> Database of graphics papers citations.
- <http://www.cs.brown.edu/~tor/> List of links to (recent) siggraph papers (and some others) on the web.
- <http://www.pointzero.nl/rendererers/> List of links to all kinds of renderers.
- 'Radiance' renderer.²⁹⁹ A highly accurate ray-tracing software system.
- Pixie³⁰⁰ An efficient and free RenderMan compatible OpenSource renderer.
- 'Aqsis' renderer³⁰¹ A free RenderMan compatible OpenSource REYES renderer.
- <http://www.povray.org/> A free ray tracer.
- 'jrMan' renderer³⁰² A RenderMan compatible OpenSource REYES renderer written in Java.

Source: http://en.wikipedia.org/wiki/Rendering_%28computer_graphics%29

Principal Authors: Hxa7241, Pinbucket, Reedbeta, Imroy, The Anome, Patrick, Lindosland, Bendman, Pixelbox, P3d0

²⁹⁶ <http://www.siggraph.org/>
²⁹⁷ <http://www.raytracingnews.org/>
²⁹⁸ <http://www.realtimerendering.com/>
²⁹⁹ <http://radsite.lbl.gov/radiance/>
³⁰⁰ <http://pixie.sourceforge.net/>
³⁰¹ <http://www.aqsis.org/>
³⁰² <http://www.jrman.org/>

Render layers

What are Render Passes?

When creating computer-generated imagery or →3D computer graphics, final scenes appearing in movies and television productions are usually produced by →Rendering (computer graphics) more than one "layer" or "pass," which are multiple images designed to be put together through digital compositing to form a completed frame.

Rendering in passes is based on traditions in Motion control photography which pre-date CGI. As an example, in motion control photography for a visual effects shot, a camera could be programmed to move past a physical model of a spaceship in one pass to film the fully lit beauty pass of the ship, and then to repeat the exact same camera move passing the ship again to photograph additional elements such as the illuminated windows in the ship or its thrusters. Once all of the passes were filmed, they could then be optically printed together to form a completed shot.

The terms "Render Layers" and "Render Passes" are sometimes used interchangeably. However, rendering in layers refers specifically to separating different objects into separate images, such as a foreground characters layer, a sets layer, a distant landscape layer, and a sky layer. Rendering in passes, on the other hand, refers to separating out different aspects of the scene, such as shadows, highlights, or reflections, each into a separate image.

Render Layer Overview (still needs editing)

Render Layers allows objects to be rendered into separate plates to save on render time, re-rendering, render crashing. But render layers can render more than just individual objects. Render layers can also render different visual aspects and qualities of objects. For instance a render layer can consist solely of an object's specularity, transparency (alpha), or even its occlusion and reflectivity.

Simple render passes include: Beauty, alpha, shadow, reflectivity and specularity. Though many artists and studios may use more this set of five will cover many aspects involved with successfully utilizing render layers. Other render passes include: depth, color, diffuse and occlusion.

Beauty: This is the most common render pass as it encompasses the colors of objects, the specularity and shadowing of the objects as well.

Alpha / Geometry Mattes: This render layer, most often seen as a side effect of rendering a scene with

Shadow: This render layer encompasses the shadows cast from one objects onto other objects. A large reason to render the shadows as a separate pass is to later control the density and range of shadows in post production.

Reflectivity: Any colors or objects reflected onto the object is now rendered inside this layer. This is often used when adding a slight dreamy feel to a sequence by slightly blurring the reflections off objects.

Specularity: Though this is already rendered in the Color and Beauty render layers, it is often also required to separate out the specularity from the colors of the objects.

Other Render Layer Types

Depth: In order for a compositing package to understand how to layers objects in post production, often times depth masks (or channels) are rendered, making it easier to sort objects in post without requiring alpha channel trickery.

Color: This render layer describes the color of the rendered objects. In Maya 7 it covers both the diffuse and specular colors of the given objects without the object's shadowing.

Diffuse: This layer is responsible for rendering strictly the color, having nothing to do with specularity or self-shadowing.

Occlusion: An occlusion pass is a visual stunner. It is often used to add a higher level of depth in the imagery. In essence, its purpose is to darken all the nooks and crannies of the objects. The closer objects are to each other the darker those parts will appear in the occlusion pass. This simulates the effect of light being blocked out by narrow spaces and close objects.

Uses: Rendering in layers saves a lot of time when rendering. Although more rendering is done, and more images are created, the renderer deals with a smaller number of objects and render qualities when rendering each image. Render Layers make it easier to add visual effects in post production, where it is much cheaper (by cost of time and cpu intensity) to create such effects. Render Layers also make it easier to create a desired visual effect in post production. Post production is where color, tonal, density, luminance, chroma, saturation and other such visual qualities are edited and perfected creating an overall, cohesive look to the image.

Lighting: Creating render layers also allows post production artists to have more control of scene lighting by rendering objects into separate layers as defined by the lights. For instance, if all objects are put into 3 render layers

each render layer can have its own light. Thereby allowing the post production artists to have control of the lighting in post production.

Note: This essay written with a practice and understanding Maya 7. Other 3D rendering packages may have differences in their terminology or execution.

Source: http://en.wikipedia.org/wiki/Render_layers

Retained mode

Retained mode refers to a programming style for 3D graphics where a persistent representation of graphical objects, their spatial relationships, their appearance and the position of the viewer, is held in memory and managed by a library layer. The programmer performs less low-level work in loading, managing, culling and rendering the data and can focus on higher application level functions.

See also

- →OpenGL
- →Scene graph

Source: http://en.wikipedia.org/wiki/Retained_mode

S3 Texture Compression

S3 Texture Compression (S3TC) (sometimes also called **DXTn** or **DXTc**) is a group of related image compression algorithms originally developed by S3 Graphics, Ltd. for use in their Savage 3D computer graphics accelerator. Unlike previous image compression algorithms, S3TC's fast random access to individual pixels made it uniquely suited for use in compressing textures in hardware accelerated →3D computer graphics. Its subsequent inclusion in Microsoft's DirectX 6.0 led to widespread adoption of the technology among hardware and software makers. While S3 Graphics is no longer a leading competitor in the graphics accelerator market, license fees are still levied and collected for the use of S3TC technology, for example in consoles.

Codecs

There are five variations of the S3TC algorithm (named **DXT1** through **DXT5**, referring to the FOURCC code assigned by Microsoft to each format), each designed for specific types of image data. All convert a 4x4 block of pixels to a 64-bit or 128-bit quantity, resulting in compression ratios of 8:1 or 4:1 with 32-bit RGBA input data. S3TC is a lossy compression algorithm, resulting in image quality degradation, but for most purposes the resulting images are more than adequate. (The notable exception is the compression of normal map data, which usually results in annoyingly visible artifacts. ATI's →3Dc compression algorithm is a modification of DXT5 designed to overcome S3TC's shortcomings in this area.)

Like many modern image compression algorithms, S3TC only specifies the method used to decompress images, allowing implementers to design the compression algorithm to suit their specific needs. The early compression routines were not optimal, and although since greatly improved, hindered early adoption of S3TC by developers. The nVidia GeForce 1 through to GeForce 4 cards also used 16 bit interpolation to render DXT1 textures, which resulted in banding when unpacking textures with color gradients. Again, this created an unfavorable impression of texture compression, not related to the fundamentals of the codec itself.

DXT1

DXT1 is the smallest variation of S3TC, storing 16 input pixels in 64 bits of output, consisting of two 16-bit RGB 5:6:5 color values and a 4x4 two bit lookup table.

If the first color value (c_0) is numerically greater than the second color value (c_1), then two other colors are calculated, such that $c_2 = \frac{2}{3}c_0 + \frac{1}{3}c_1$ and $c_3 = \frac{1}{3}c_0 + \frac{2}{3}c_1$.

Otherwise, if $c_0 \leq c_1$, then $c_2 = \frac{1}{2}c_0 + \frac{1}{2}c_1$ and c_3 is transparent.

The lookup table is then consulted to determine the color value for each pixel, with a value of 0 corresponding to c_0 and a value of 3 corresponding to c_3 . DXT1 does not support texture alpha data.

DXT2 and DXT3

DXT2 and DXT3 converts 16 input pixels into 128 bits of output, consisting of 64 bits of alpha channel data (four bits for each pixel) followed by 64 bits of color data, encoded the same way as DXT1 (with the exception that the 4 color version of the DXT1 algorithm is always used instead of deciding which version to use based on the relative values of c_0 and c_1). In DXT2, the color data is interpreted as being premultiplied by alpha, in DXT3 it is interpreted as not having been premultiplied by alpha. Typically DXT2/3 are well suited to images with sharp alpha transitions, between translucent and opaque areas.

DXT4 and DXT5

DXT4 and DXT5 converts 16 input pixels into 128 bits of output, consisting of 64 bits of alpha channel data (two 8 bit alpha values and a 4x4 3 bit lookup table) followed by 64 bits of color data (encoded the same way as DXT2 and DXT3).

If $\alpha_0 > \alpha_1$, then six other alpha values are calculated, such that $\alpha_2 = \frac{6\alpha_0+1\alpha_1}{7}$, $\alpha_3 = \frac{5\alpha_0+2\alpha_1}{7}$, $\alpha_4 = \frac{4\alpha_0+3\alpha_1}{7}$, $\alpha_5 = \frac{3\alpha_0+4\alpha_1}{7}$, $\alpha_6 = \frac{2\alpha_0+5\alpha_1}{7}$, and $\alpha_7 = \frac{1\alpha_0+6\alpha_1}{7}$.

Otherwise, if $\alpha_0 \leq \alpha_1$, four other alpha values are calculated such that $\alpha_2 = \frac{4\alpha_0+1\alpha_1}{5}$, $\alpha_3 = \frac{3\alpha_0+2\alpha_1}{5}$, $\alpha_4 = \frac{2\alpha_0+3\alpha_1}{5}$, and $\alpha_5 = \frac{1\alpha_0+4\alpha_1}{5}$ with $\alpha_6 = 0$ and $\alpha_7 = 255$.

The lookup table is then consulted to determine the alpha value for each pixel, with a value of 0 corresponding to α_0 and a value of 7 corresponding to α_7 . DXT4's color data is premultiplied by alpha, whereas DXT5's is not. Because DXT4/5 use an interpolated alpha scheme, they generally produce superior results for alpha (transparency) gradients than DXT2/3. Some consider DXT5 to be the most flexible general purpose compression codec.

S3TC Format Comparison

FOURCC	Description	Alpha premultiplied?	Compression ratio	Texture Type
DXT1	Opaque / 1-bit Alpha	N/A	8:1 / 6:1	Simple non-alpha
DXT2	Explicit alpha	Yes	4:1	Sharp alpha
DXT3	Explicit alpha	No	4:1	Sharp alpha
DXT4	Interpolated alpha	Yes	4:1	Gradient alpha
DXT5	Interpolated alpha	No	4:1	Gradient alpha

See also

- →3Dc
- FXT1
- DirectDraw Surface

External links

- NVIDIA Texture Tools³⁰³
- ATI Developer: Tools³⁰⁴
- MSDN Compressed Texture Resources³⁰⁵
- Comparison between S3TC and FXT1 texture compression³⁰⁶
- The Truth about S3TC³⁰⁷ Note: This article used an early S3TC compression engine, not nVidia's or ATI's updated codecs.

Source: http://en.wikipedia.org/wiki/S3_Texture_Compression

Principal Authors: Timharwoodx, NJM, Paul-Jan, N00body, Ilya K

Scanline rendering

Scanline rendering is a rendering technique, or family of algorithms, in →3D computer graphics that works on a row-by-row basis rather than a polygon-by-polygon or pixel-by-pixel basis. All of the polygons to be rendered are first sorted by the top y coordinate at which they first appear, then each row or scan line of the image is computed using the intersection of a scan line with the polygons on the front of the sorted list, while the sorted list is updated to discard no-longer-visible polygons as the active scan line is advanced down the picture.

The asset of this method is that it is not necessary to translate the coordinates of all vertices from the main memory into the working memory—only vertices

³⁰³ http://developer.nvidia.com/object/nv_texture_tools.html

³⁰⁴ <http://www.ati.com/developer/tools.html>

³⁰⁵ http://msdn.microsoft.com/library/default.asp?url=/library/en-us/directx9_c/directx/graphics/ProgrammingGuide/GettingStarted/Direct3DTextures/compressed/compressedtextreresources.asp

³⁰⁶ <http://www.digit-life.com/articles/reviews3tcfxt1/>

³⁰⁷ <http://web.archive.org/web/20030618083605/www.hardwarecentral.com/hardwarecentral/reports/140/1/>

defining edges that intersect the current scan line need to be in active memory, and each vertex is read in only once. The main memory is often very slow compared to the link between the central processing unit and cache memory, and thus avoiding re-accessing vertices in main memory can provide a substantial speedup.

This kind of algorithm can be easily integrated with the →Phong reflection model, the Z-buffer algorithm, and many other graphics techniques.

Scanline rendering is used by most modern graphics cards and is typically accessed by the programmer using some 3D API such as →OpenGL or →Direct3D.

History

The first publication of the scanline rendering technique was probably by Wylie, Romney, Evans, and Erdahl in 1967.³⁰⁸

Other early developments of the scanline rendering method were by Bouknight in 1969,³⁰⁹ and Newell, Newell, and Sancha in 1972.³¹⁰ Much of the early work on these methods was done in Ivan Sutherland's graphics group at the University of Utah, and at the Evans & Sutherland company in Salt Lake City, Utah.

References

External Links

[<http://accad.osu.edu/~waynec/history/tree/utah.html>] University of Utah Graphics Group History

See also

- Scan line
- Pixel
- Raster scan

Source: http://en.wikipedia.org/wiki/Scanline_rendering

Principal Authors: Dicklyon, Samwisefoxburr, Nixdorf, Kc9cqj, Timo Honkasalo

³⁰⁸ Wylie, C, Romney, G W, Evans, D C, and Erdahl, A, "Halftone Perspective Drawings by Computer," Proc. AFIPS FJCC 1967, Vol. 31, 49

³⁰⁹ Bouknight W.J, "An Improved Procedure for Generation of Half-tone Computer Graphics Representation," UI, Coordinated Science Laboratory, Sept 1969

³¹⁰ Newell, M E, Newell R. G, and Sancha, T.L, "A New Approach to the Shaded Picture Problem," Proc ACM National Conf. 1972

Scene graph

A **scene-graph** is a general data structure commonly used by vector-based graphics editing applications and modern computer games. Examples of such programs include AutoCAD, Adobe Illustrator and CorelDRAW.

The scene-graph is an object-oriented structure that arranges the logical and often (but not necessarily) spatial representation of a graphical scene. The definition of a scene-graph is fuzzy, due to the fact that programmers who implement scene-graphs in applications and in particular the games industry take the basic principles and adapt these to suit a particular application. This means there is no hard and fast rule what a scene-graph should be or shouldn't be.

Scene-graphs are a collection of nodes in a graph or tree structure. This means that a node may have many children but often only a single parent, the effect of a parent is apparent to all its child nodes - An operation applied to a group automatically propagates its effect to all of its members. In many programs, associating a geometrical transformation matrix (see also transformation and matrix) at each group level and concatenating such matrices together is an efficient and natural way to process such operations. A common feature, for instance, is the ability to group related shapes/objects into a compound object which can then be moved, transformed, selected, etc. as easily as a single object.

Scene-graphs in graphics editing tools

In vector-based graphics editing, each node in a scene graph represents some atomic unit of the document, usually a shape such as an ellipse or Bezier path. Although shapes themselves (particularly paths) can be decomposed further into nodes such as spline nodes, it is practical to think of the scene graph as composed of shapes rather than going to a lower level of representation.

Another useful and user-driven node concept is the layer. A layer acts like a transparent sheet upon which any number of shapes and shape groups can be placed. The document then becomes a set of layers, any of which can be conveniently made invisible, dimmed, and/or locked (made read-only). Some applications place all layers in a linear list while others support sublayers (i.e., layers within layers, to any desired depth).

Internally, there may be no real structural difference between layers and groups at all, since they are both just nested scenegraphs. If differences are needed, a common type declaration in C++ would be to make a generic scenegraph

class, and then derive layers and groups as subclasses. A visibility member, for example, would be a feature of a layer but not necessarily of a group.

Scene-graphs in games and 3D applications

Scene graphs are ideal for modern games using 3D graphics and increasingly large worlds or levels. In such applications, nodes in a scene-graph (generally) represent entities or objects in the scene.

For instance, a game might define a logical relationship between a knight and a horse so that the knight is considered an extension to the horse. The scene graph would have a 'horse' node with a 'knight' node attached to it.

As well as describing the logical relationship, the scene-graph may also describe the spatial relationship of the various entities: the knight moves through 3D space as the horse moves.

In these large applications, memory requirements are major considerations when designing a scene-graph. For this reason many large scene-graph systems use instancing to reduce memory costs and increase speed. In our example above, each knight is a separate scene node, but the graphical representation of the knight (made up of a 3D mesh, textures, materials and shaders) is instanced. This means that only a single copy of the data is kept, which is then referenced by any 'knight' nodes in the scene-graph. This allows a reduced memory budget and increased speed, since when a new knight node is created, the appearance data does not need to be duplicated.

Scene-graph Implementation

The simplest form of scene graph uses an array or linked list data structure, and displaying its shapes is simply a matter of linearly iterating the nodes one by one. Other common operations, such as checking to see which shape intersects the mouse pointer (e.g., in a GUI-based applications) are also done via linear searches. For small scenegraphs, this tends to suffice.

Larger scenegraphs cause linear operations to become noticeably slow and thus more complex underlying data structures are used, the most popular being a tree. This is the most common form of scene-graph. In these scene-graphs the composite design pattern is often employed to create the hierarchical representation of group-nodes and leaf-nodes.

Group Nodes - Can have any number of child nodes attached to it. Group nodes includes transformations and switch nodes.

Leaf Nodes - Are nodes that are actually rendered or see the effect of an operation. These include objects, sprites, sounds, lights and anything that could be considered 'rendered' in some abstract sense.

Scene-graph Operations and Dispatch

In order to apply an operation to a scene-graph some way of dispatching an operation based upon what node is currently being considered is needed. For example in a render operation a transformation group-node would do nothing more than accumulate its transformation (generally this is matrix multiplication but could involve operations with vector displacement and quaternions or Euler angles instead). Whereas an object leaf-node would send the object off for rendering to the renderer (some implementations might render the object directly but this can integrate the underlying rendering API - e.g. OpenGL or DirectX too tightly and rigidly - it is better to separate the scene-graph and renderer systems as this promotes good OOP style and extensibility).

In order to dispatch differently for different node types several different approaches can be taken, each have pros and cons and are widely disputed among programmers arguing which is best.

In Object-Oriented languages such as C++ this can easily be achieved by virtual functions, the node base class has virtual functions for every operation that can be performed on the nodes. This is simple to do but prevents the addition of new operations by other programmers that don't have access to the source.

Alternatively the **visitor pattern** can be used - this is relatively simple and faster than virtual functions where the operation to be performed is decided by multiple dispatch. This has a similar disadvantage in that it is similarly difficult to add new node types.

Other techniques involve the use of RTTI (Run-Time-Type-Information) the operation can be realised as a class which is passed the current node, it then queries the nodes type (RTTI) and looks up the correct operation in an array of callbacks or functors. This requires that at initialisation the user/system registers functors/callbacks with the different operations so they can be looked up in the array. This system offers massive flexibility, speed and extensibility of new nodes and operations.

Variations on these techniques exist and new methods can offer added benefits - one alternative is scene-graph re-building where the scene-graph is re-built for each of the operations performed, this however can be very slow but produces a highly optimised scene-graph. This demonstrates that a good scene-graph implementation depends heavily on the application it is used in.

Traversals

Traversals are the key to the power of applying operations to scene-graphs. A traversal generally consists of starting at some arbitrary node (often the root of the scene-graph), applying the operation(s) often the updating and rendering operations are applied one after the other, and recursively moving down the scene-graph(tree) to the child nodes, until a leaf node is reached. At this point many scene-graphs then traverse back up the tree possibly applying a similar operation. As an example - Rendering: While recursively traversing down the scene-graph hierarchy the operations applies a PreRender operation, once reaching a leaf node it begins traversing back up the tree applying a PostRender operation - This nicely allows certain nodes push a state for child nodes and pop the state afterward.

Some scene-graph operations are actually more efficient when nodes are traversed in a different order - This is where some systems implement scene-graph re-building to reorder the scene-graph into an easier to parse format or tree.

For example:

In 2D cases, scenegraphs typically render themselves by starting at the tree's root node and then recursively drawing the child nodes. The tree's leaves represent the most foreground objects. Since drawing proceeds from back to front with closer objects simply overwriting farther ones, the process is known as employing the →Painter's algorithm. In 3D systems, which often employ depth buffers, it is more efficient to draw the closest objects first, since farther objects often need only be depth-tested instead of actually rendered.

Scene-graph and Bounding Volume Hierarchies (BVHs)

Bounding Volume Hierarchies (BVHs) are useful for numerous tasks - including efficient culling and speeding up collision detection between objects. A BVH is a spatial structure but doesn't have to partition the geometry (see spatial partitioning, below).

A BVH is a tree of bounding volumes (often spheres, AABBs or/and OBBs). At the bottom of the hierarchy the size of the volume is just large enough to encompass a single object tightly (or possibly even some smaller fraction of an object in high resolution BVHs), as you walk up the hierarchy each node has its own volume which tightly encompasses all the volumes beneath it. At the root of the tree is a volume that encompasses all the volumes in the tree (the whole scene).

BVHs are useful for speeding up collision detection between objects. If an object's bounding volume does not intersect a volume higher in the tree then

it cannot intersect any object below that node (so they are all rejected very quickly).

Obviously there are some similarities between BVHs and scene-graphs. A scene-graph can easily be adapted to include/become a BVH - if each node has a volume associated or there's purpose built 'bound node' added in at convenient location in the hierarchy. This may not be the typical view of scene-graph but there are benefits to including a BVH in a scene-graph.

Scene-graphs and Spatial Partitioning

An effective way of combining spatial partitioning and scene-graphs is by creating a scene leaf node that contains the spatial partitioning data - This data is usually static and generally contains non-moving level data in some partitioned form. Some systems may have the systems separate and render them separately, this is fine and there are no real advantages to either method. In particular it is bad to have the scene-graph contained within the spatial partitioning system, this is because the scene-graph is better thought of as the grander system to the spatial partitioning.

When it is useful to combine them

In short: Spatial partitioning will/should considerably speed up the processing and rendering time of the scene-graph.

Very large drawings, or scene graphs that are generated solely at runtime (as happens in ray tracing rendering programs), require defining of group nodes in a more automated fashion. A raytracer, for example, will take a scene description of a 3D model and build an internal representation that breaks up its individual parts into bounding boxes (also called bounding slabs). These boxes are grouped hierarchically so that ray intersection tests (as part of visibility determination) can be efficiently computed. A group box that does not intersect an eye ray, for example, can entirely skip having to test any of its members.

A similar efficiency holds in 2D applications as well. If the user has magnified a document so that only part of it is visible on his computer screen, and then scrolls said document, it is useful to use a bounding box (or in this case, a bounding rectangle scheme) to quickly determine which scenegraph elements are visible and thus actually need to be drawn.

Depending on the particulars of the application's drawing performance, a large part of the scenegraph's design can be impacted by rendering efficiency considerations. In 3D video games such as Quake, for example, binary space partitioning (BSP) trees are heavily favored to minimize visibility tests. BSP trees,

however, take a very long time to compute from design scenegraphs, and must be recomputed if the design scenegraph changes so the levels tend to remain static and dynamic characters aren't generally considered in the spatial partitioning scheme.

Scenegraphs for dense regular objects such as heightfields and polygon meshes tend to employ quadtrees and octrees, which are specialized variants of a 3D bounding box hierarchy. Since a heightfield occupies a box volume itself, recursively subdividing this box into eight subboxes (hence the 'oct' in octree) until individual heightfield elements are reached is efficient and natural. A quadtree is simply a 2D octree.

PHIGS

PHIGS was the first commercial scene-graph specification, and became an ANSI standard in 1988. Disparate implementations were provided by Unix hardware vendors. The "HOOPS 3D Graphics System"³¹¹ appears to have been the first commercial scene graph library provided by a single software vendor. It was designed to run on disparate lower-level 2D and 3D interfaces, with the first major production version (v3.0) completed in 1991. Shortly thereafter, Silicon Graphics released IRIS Inventor 1.0 (1992), which was a scene-graph built on top of the IRIS GL 3D API. It was followed up with →Open Inventor in 1994, a portable scene graph built on top of OpenGL. More 3D scenegraph libraries can be found in 3D Scenegraph APIs.

References

Books

- Leler, Wm and Merry, Jim (1996) *3D with HOOPS*, Addison-Wesley
- Wernecke, Josie (1994) *The Inventor Mentor: Programming Object-Oriented 3D Graphics with Open Inventor*, Addison-Wesley, ISBN 0-201-62495-8 (Release 2)

Web sites and articles

- Strauss, Paul (1993). "IRIS Inventor, a 3D Graphics Toolkit"³¹²
- Helman, Jim; Rohlfs, John (1994). "IRIS Performer: A High Performance Multiprocessing Toolkit for Real-Time 3D Graphics"³¹³

³¹¹ <http://www.hoops3d.com>

³¹² <http://portal.acm.org/citation.cfm?id=165889>

³¹³ <http://portal.acm.org/citation.cfm?id=192262>

- Carey, Rikk and Bell, Gavin (1997). "The Annotated VRML 97 Reference Manual"³¹⁴
- PEXTimes³¹⁵
- Bar-Zeev, Avi. "Scenegraphs: Past, Present, and Future"³¹⁶
- Open Scene Graph³¹⁷
- OpenSG³¹⁸

Source: http://en.wikipedia.org/wiki/Scene_graph

Principal Authors: Mortene, Mikkalai, Tommstein, CamTarn, WillC2 45220, Reinyday, Engwar

Shader

A **shader** is a computer program used in →3D computer graphics to determine the final surface properties of an object or image. This often includes arbitrarily complex descriptions of light absorption, diffusion, texture mapping, reflection, refraction, shadowing, surface displacement and post-processing effects.

By design, shaders are ideal candidates for parallel execution by multiple graphic processors, which are usually located on a video card, allowing for scalable multiprocessing and lessening the burden on the CPU for rendering scenes.

Because of the goals shaders are designed to address, they are usually written using a shading language, a specifically designed programming language built around the intrinsic strengths and weaknesses of the different computational model. Although limited in some ways when compared to traditional approaches, the parallel architecture exposed by shaders has been used to combine highly scalable processing power with the flexibility of programmable devices, which is a boon in addressing the growing demands for graphics quality.

The increasing performance and programmability of shader-based architectures attracted researchers trying to exploit the new parallel model for General Purpose computation on GPUs. This demonstrated that shaders could be used

³¹⁴ <http://www.jwave.vt.edu/~engineer/vrml97book/ch1.htm>

³¹⁵ <http://www.jch.com/jch/vrml/PEXTimes.txt>

³¹⁶ <http://www.realityprime.com/scenegraph.php>

³¹⁷ <http://www.openscenegraph.org>

³¹⁸ <http://www.opensg.org>

to process a large variety of information, and not just rendering-specific tasks. This new programming model, which resembles stream processing, allows high computational rates at extremely low cost that will operate on wide installed base (e.g. the common home PC).

Real-time Shader Structure

There are different approaches to shading, mainly because of the various applications of the targeted technology. Production shading languages are usually at a higher abstraction level, avoiding the need to write specific code to handle lighting or shadowing. In contrast, real-time shaders integrate light and shadowing computations. In those languages, the lights are passed to the shader itself as parameters.

There are actually two different applications of shaders in real-time shading languages. Although the feature set actually converged (so it's possible to write a vertex shader using the same functions of a fragment shader), the different purposes of computation impose limitations to be acknowledged.

Vertex Shaders

Vertex shaders are applied *for each vertex* and run on a programmable vertex processor. Vertex shaders define a method to compute vector space transformations and other linearizable computations.

A vertex shader expects various inputs:

- *Uniform variables* are constant values for each shader invocation. It is allowed to change the value of each uniform variable between different shader invocation batches. This kind of variable is usually a 3-component array but this does not need to be. Usually, only basic datatypes are allowed to be loaded from external APIs so complex structures must be broken down. Uniform variables can be used to drive simple conditional execution on a per-batch basis. Support for this kind of branching at a vertex level has been introduced in shader model 2.0.
- *Vertex attributes*, which are a special case of *variant variables*, which are essentially per-vertex data such as vertex positions. Most of the time, each shader invocation performs computation on different data sets. The external application usually does not access these variables "directly" but manages as large arrays. Besides this little detail, applications are usually capable of changing a single vertex attribute with ease. Branching on vertex attributes requires a finer degree of control which is supported with extended shader model 2.

Vertex shader computations are meant to provide following stages of the graphics pipeline with interpolable fragment attributes. Because of this, a vertex shader must output at least the transformed homogeneous vertex position (in GLSL this means the variable `gl_Position` must be written). Outputs from different vertex shader invocations from the same batch will be linearly interpolated across the primitive being rendered. The result of this linear interpolation is fetched to the next pipeline stage.

Some examples of vertex shader's functionalities include arbitrary mesh deformation (possibly faking lens effects such as fish-eye) and vertex displacements in general, computing linearizable attributes for later pixel-shaders such as texture coordinate transformations. Actually, vertex shaders **cannot** create vertices.

Pixel Shaders

Pixel shaders are used to compute properties which, most of the time, are recognized as pixel colors.

Pixel shaders are applied *for each pixel*. They are run on a pixel processor, which usually features much more processing power than its vertex-oriented counterpart. As of October 2005, some architectures are merging the two processors in a single one to increase transistor usage and provide some kind of load balancing.

As previously stated, the pixel shaders expects input from interpolated vertex values. This means there are three sources of information:

- *Uniform variables* can still be used and provide interesting opportunities. A typical example is passing an integer providing a number of lights to be processed and an array of light parameters. Textures are special cases of uniform values and can be applied to vertices as well, although vertex texturing is often more expensive.
- *Varying attributes* is a special name to indicate fragment's variant variables, which are the interpolated vertex shader output. Because of their origin, the application has no direct control on the actual value of those variables.

Branching on the pixel processor has also been introduced with an extended pixel shader 2 model but hardware supporting this efficiently is beginning to be commonplace only now (6 March 2006), usually with full pixel shader model 3 support.

A fragment shader is allowed to *discard* the results of its computation, meaning that the corresponding framebuffer position must retain its actual value.

Fragment shaders also don't need to write specific color information because this is not always wanted. Not producing color output when expected however gives undefined results in GLSL.

Fragment shaders have been employed to apply accurate lighting models, simulate multi-layer surface properties, simulating natural phenomena such as turbulence (vector field simulations in general) and applying depth of field to a scene or other color-space transformations.

Texturing

Note: the following information on texture mapping with shaders applies specifically to →GLSL. Those statements may not hold true for DirectX HLSL.

The functionality by itself continues to be applied "as usual" with shading languages providing special ad-hoc functions and opaque objects.

It has been stated that textures are special uniform variables. The shading languages define special variables to be used as textures called *samplers*. Each sampler does have a specific *lookup mode* assigned explicitly in the name. Looking up a texture actually means to get an interpolated texel color at a specified position. For example, in GLSL **sampler2D** will access a specific texture performing bidimensional texturing and filtering to be used with a **tex2D** function call. Other details are specified by the function used to actually perform the lookup. For cube map textures, a **samplerCube** would be used with a **textureCube** function call.

Understanding completely the model also needs to know a little about the previous shading model, commonly referred as multitexturing or *texture cascade*. For our purposes, we'll just assume there is a limited set of units which can be linked to specific textures and somehow produce color results, possibly combining them in a sequential order. This is redundant with the new programming model which allows much greater flexibility.

To lookup to a specific texture, the sampler really needs to know what of those texture units needs to be used with the specified lookup mode. This means samplers are really integers referring to the texture unit used to carry on the lookup. It will now be possible to bind to each texture unit an image texture just as usual. It will happen that those units are actually a subset of "legacy" texture units and are referred as *image units*. Most implementation actually allow more image units than texture units because of the lower complexity to implement them but also to push for the new programming model. In short, samplers are really linked to image units, which are bound to textures.

For final users, this extra flexibility results in both improved performance and richer content because of the better hardware utilization and resources.

Lighting & Shadowing

Considering the lighting equation, we have seen the trend to move evaluations to fragment granularity. Initially, the lighting computations were performed at vertex level (phong lighting model) but improvements in fragment processor designs allowed to evaluate much more complex lighting equations such as the *blinn lighting model*, often referred as bump mapping. In this latter technique, vertex shaders are used to set up a vertex local space (also called *tangent space*) which is then used to compute per-pixel lighting vectors. The actual math for this can be quite involved and is beyond the scope of this article.

It is well acknowledged that lighting really needs hardware support for dynamic loops (this is often referred as *DirectX Pixel Shader Model 3.0*) because this allows to process many lights of the same type with a single shader. By contrast, previous shading models would have need the application to use multi pass rendering (an expensive operation) because of the fixed loops. This approach would also have needed more complicated machinery. For example, after finding there are 13 "visible" lights, the application would have the need to use a shader to process 8 lights (suppose this is the upper hardware limitation) and another shader to process the remaining 5. If there are 7 lights the application would have needed a special 7-light shader. By contrast, with dynamic loops the application can iterate on dynamic variables thus defining a uniform array to be 13 (or 7) "lights long" and get correct results, provided this actually fits in hardware capabilities. At the time this is being written (27 October 2005) there are enough resources to evaluate over 50 lights per pass when resources are managed carefully. Compare this to old programming models.

Computing accurate shadows make this much more complicated, depending on the algorithm used. Compare stencil shadow volumes and shadow mapping. In the first case, the algorithm requires at least some care to be applied to multiple lights at once and there's no actual proof of a multi-light shadow volume based version. Shadow mapping by contrast seems to be much more well suited to future hardware improvements and to the new shading model which also evaluates computations at fragment level. Shadow maps however needs to be passed as samplers, which are limited resources: actual hardware (27 October 2005) support up to 16 samplers so this is a hard-limit, unless some tricks are

used. It is speculated that future hardware improvements and packing multiple shadow maps in a single 3D-texture will rapidly raise this resource availability.

Further reading

- Steve Upstill: *The RenderMan Companion: A Programmer's Guide to Realistic Computer Graphics*, Addison-Wesley, ISBN 0-201-50868-0
- David S. Elbert, F. Kenton Musgrave, Darwyn Peachey, Ken Perlin, Steven Worley: *Texturing and modeling: a procedural approach*, AP Professional, ISBN 0-12-228730-4. Ken Perlin is the author of →Perlin noise, an important procedural texturing primitive.
- Randima Fernando, Mark Kilgard. *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics*, Addison-Wesley Professional, ISBN 0-32119-496-9
- Randi Rost: *OpenGL Shading Language*, Addison-Wesley Professional, ISBN 0-321-19789-5

References

1. ↑ Search ARB_shader_objects³¹⁹ for the issue "32) Can you explain how uniform loading works?". This is an example of how a complex data structure must be broken in basic data elements.
2. ↑ Required machinery has been introduced in OpenGL by ARB_multitexture³²⁰ but this specification is no more available since its integration in core OpenGL 1.2.
3. ↑ Search again ARB_shader_objects³²¹ for the issue "25) How are samplers used to access textures?". You may also want to check out "Subsection 2.14.4 Samplers".
4. ↑ Search NVIDIA developer resources³²² for various papers on per-pixel lighting.
5. ↑ You'll be glad to see those limits are (at least theoretically) rather high. Check out The OpenGL® Shading Language³²³ for the string "52) How should resource limits for the shading language be defined?" and look for your favorite video card at Delphi3d.net hardware database³²⁴.

³¹⁹ http://oss.sgi.com/projects/ogl-sample/registry/ARB/shader_objects.txt

³²⁰ <http://oss.sgi.com/projects/ogl-sample/registry/ARB/multitexture.txt>

³²¹ http://oss.sgi.com/projects/ogl-sample/registry/ARB/shader_objects.txt

³²² <http://developer.nvidia.com>

³²³ <http://www.opengl.org/documentation/glsl/>

³²⁴ <http://www.delphi3d.net/hardware/index.php>

Source: <http://en.wikipedia.org/wiki/Shader>

Principal Authors: T-tus, Flamurai, MaxDZ8, Parameter, Rufous

Shading language

A **shading language** is a special programming language adapted to easily map on shader programming. Those kind of languages usually have special data types like color and normal. Because of the various target markets of 3D graphics, different shading languages have been developed: a brief overview is given below.

Production rendering

These kind of shading languages are geared towards maximum image quality. Material properties are totally abstracted, little programming skill and no hardware knowledge is required. These kind of shaders are often developed by artists to get the right "look", just as texture mapping, lighting and other facets of their work.

Processing these kinds of shaders is usually a time-consuming process. The computational power required to get this kind of shading to work can be rather expensive because of their ability to produce photorealistic results. Most of the time, production rendering is run on large computer clusters.

RenderMan Shading Language

RenderMan Shading Language, which is defined in the *RenderMan Interface Specification*, is the most common shading language for production-quality rendering. RenderMan by Rob Cook, is currently used in all of Pixar's products. It's also one of the first shading languages ever implemented.

The language actually defines six major shader types:

- *Light source shaders* compute the color of the light emitted from a point on the light source towards a point on the surface being illuminated.
- *Surface shaders* are used to model the optical properties of the object being illuminated. They output the final color and position of the point being illuminated by taking into account the incoming light and the physical properties of the object.

- *Displacement shaders* manipulate the surface's geometry independent of its color.
- *Deformation shaders* transform the entire space that a geometry is defined in. Only one RenderMan implementation, the AIR renderer, actually implements this shader type.
- *Volume shaders* manipulate the color of a light as it passes through a volume. They are used to create effects like fog.
- *Imager shaders* describe a color transformation to final pixel values. This is much like an image filter, however the imager shader operates on pre-quantized data, which typically has a greater dynamic range than can be displayed on the output device.

Most of the time RenderMan is referenced, it is really meant to speak about *PRMan*, the RenderMan implementation from Pixar, which was the only one available for years. Further information can be found at the RenderMan repository³²⁵.

Gelato Shading Language

Developed by NVIDIA, a graphics processing unit manufacturer for its Gelato rendering software.

This language is meant to interact with hardware, providing higher computational rates while retaining cinematic image quality and functionalities.

Real-time rendering

Until recently, developers did not have the same level of control over the output from the graphics pipeline of graphics cards, but shading languages for real-time rendering are now widespread. They provide both higher hardware abstraction and a more flexible programming model when compared to previous paradigms which hardcoded transformation and shading equations. This results in both giving the programmer greater control over the rendering process, and delivering richer content at lower overhead.

Quite surprisingly those shaders, which are designed to be executed directly on the GPU at the proper point in the pipeline for maximum performance, also scored successes in general processing because of their stream programming model.

³²⁵ <http://www.renderman.org/>

This kind of shading language is usually bound to a graphics API, although some applications also provide built-in shading languages with limited functionalities.

Historically, only few of those languages were successful in both establishing themselves and maintaining strong market position; a short description of those languages follows below.

OpenGL shading language

Also known as →GLSL or *glslang*, this standardized high level shading language is meant to be used with →OpenGL.

The language featured a very rich feature set since the beginning, unifying vertex and fragment processing in a single instruction set, allowing conditional loops and (more generally) branches.

Historically, GLSL have been preceded by various OpenGL extensions such as ARB_vertex_program³²⁶, ARB_fragment_program³²⁷ and many others. Those were low-level, assembly-like languages with various limitations. Usage of those languages is now discouraged. Those two extensions were also preceded by other proposals which didn't survive in the new version.

Cg programming language

This language developed by NVIDIA has been designed for easy and efficient production pipeline integration. The language features API independence and comes with a large variety of free tools to improve asset management.

The first Cg implementations were rather restrictive because of the hardware being abstracted but they were still innovative when compared to previous methods. Cg seems to have survived the introduction of the newer shading languages very well, mainly of its established momentum in the digital content creation area, although the language is seldom used in final products.

A distinctive feature of Cg is the use of *connectors*, special data structures to link the various stages of processing. Connectors are used to define the input from application to vertex processing stage and the attributes to be interpolated as input to fragment processing.

DirectX High-Level Shader Language

This is possibly the most successful language to date, mainly because of great pressure from Microsoft, combined with the fact that it was the first C-style

³²⁶ http://oss.sgi.com/projects/ogl-sample/registry/ARB/vertex_program.txt

³²⁷ http://oss.sgi.com/projects/ogl-sample/registry/ARB/fragment_program.txt

shader language to be used for real-time rendering. The high level shader language (also called HLSL for short) was released before its main competitor, GLSL, although its feature set was later extended in two different revisions to match the GLSL feature set.

References

1. ↑ <https://renderman.pixar.com/products/rispec/>
2. ↑ NVIDIA Gelato official website, <http://film.nvidia.com/page/gelato.html>
3. ↑ Official language specification, <http://www.opengl.org/documentation/glsl/>
4. ↑ Previous vertex shading languages (in no particular order) for OpenGL include EXT_vertex_shader³²⁸, NV_vertex_program³²⁹, the aforementioned ARB_vertex_program³³⁰, NV_vertex_program2³³¹ and NV_vertex_program3³³².
5. ↑ For fragment shading nvpars³³³ is possibly the first shading language featuring high-level abstraction based on NV_register_combiners³³⁴, NV_register_combiners2³³⁵ for pixel math and NV_texture_shader³³⁶, NV_texture_shader2³³⁷ and NV_texture_shader3³³⁸ for texture lookups. ATI_fragment_shader³³⁹ did not even provide a "string oriented" parsing facility (although it has been later added by ATI_text_fragment_shader³⁴⁰). ARB_fragment_program³⁴¹, has been very successful. NV_fragment_program³⁴² and NV_fragment_program2³⁴³ are actually similar although the latter provides much more advanced functionality in respect to others.

³²⁸ http://oss.sgi.com/projects/ogl-sample/registry/EXT/vertex_shader.txt

³²⁹ http://oss.sgi.com/projects/ogl-sample/registry/NV/vertex_program.txt

³³⁰ http://oss.sgi.com/projects/ogl-sample/registry/ARB/vertex_program.txt

³³¹ http://oss.sgi.com/projects/ogl-sample/registry/NV/vertex_program2.txt

³³² http://oss.sgi.com/projects/ogl-sample/registry/NV/vertex_program3.txt

³³³ <http://developer.nvidia.com/object/nvpars.html>

³³⁴ http://oss.sgi.com/projects/ogl-sample/registry/NV/register_combiners.txt

³³⁵ http://oss.sgi.com/projects/ogl-sample/registry/NV/register_combiners2.txt

³³⁶ http://oss.sgi.com/projects/ogl-sample/registry/NV/texture_shader.txt

³³⁷ http://oss.sgi.com/projects/ogl-sample/registry/NV/texture_shader2.txt

³³⁸ http://oss.sgi.com/projects/ogl-sample/registry/NV/texture_shader3.txt

³³⁹ http://oss.sgi.com/projects/ogl-sample/registry/ATI/fragment_shader.txt

³⁴⁰ http://oss.sgi.com/projects/ogl-sample/registry/ATI/text_fragment_shader.txt

³⁴¹ http://oss.sgi.com/projects/ogl-sample/registry/ARB/fragment_program.txt

³⁴² http://oss.sgi.com/projects/ogl-sample/registry/NV/fragment_program.txt

³⁴³ http://oss.sgi.com/projects/ogl-sample/registry/NV/fragment_program2.txt

6. ↑ Official Cg home page, http://developer.nvidia.com/object/cg_toolkit.html
7. ↑ Fx composer from NVIDIA home page, http://developer.nvidia.com/object/fx_composer_home.html

Source: http://en.wikipedia.org/wiki/Shading_language

Principal Authors: Rayc, MaxDZ8, Flamura, Inike, Ecemaml

Shadow mapping

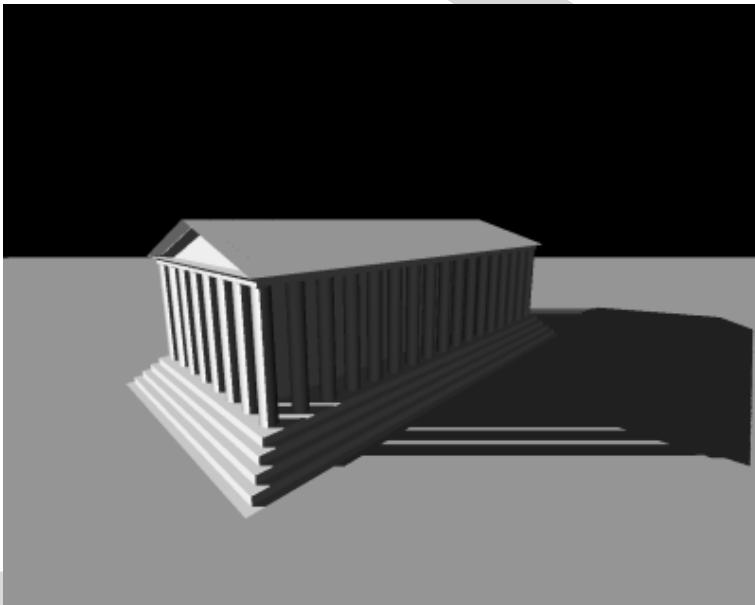


Figure 81 Scene with shadow mapping

Shadow mapping is a process, by which, shadows are added to →3D computer graphics. This concept was introduced by Lance Williams in 1978, in a paper entitled "Casting curved shadows on curved surfaces". Since then, it has been used both in pre-rendered scenes and in realtime. Shadow mapping is used by Pixar's RenderMan, and likewise, shadow mapping has been used in such films as *Toy Story*.

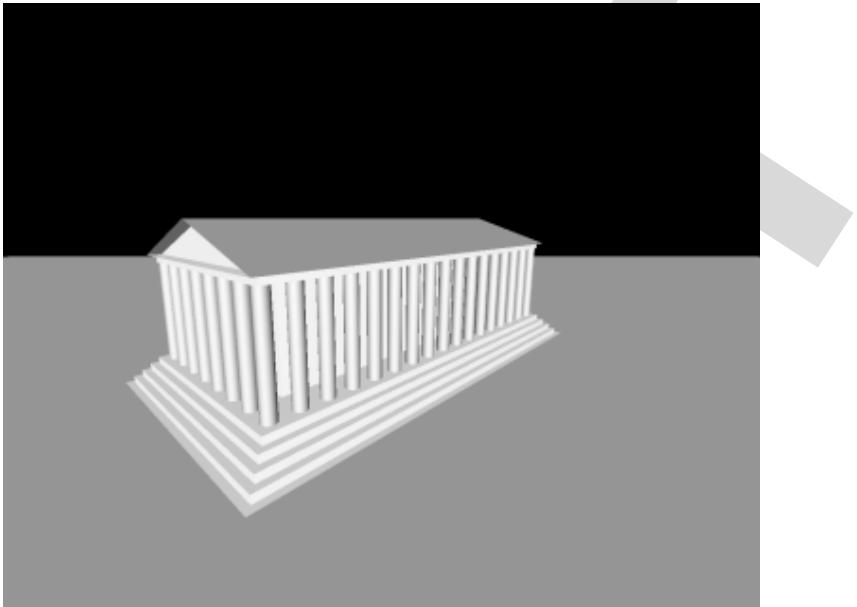


Figure 82 Scene with no shadows

Shadows are created by testing whether a pixel is visible from the light source, by comparing it to a z-buffer or *depth* image, in the form of a texture. This technique involves three passes or re-renderings, although it can be accomplished with just two.

Principle of a shadow and a shadow map

A shadow can be considered thus: Imagine you have a spot light source, that is aimed at some objects. The shadow thus cast by this light source can be visualized by viewing through the light source at the objects and marking the visible areas, and disregarding every point not seen by the light source as "shadow".

The complexity of this simple operation becomes apparent when one considers the amount of photons that are incident on the surface of the objects. The sheer number of photons incident cannot be simulated at real-time even by current fastest computers. Hence to make this effect more accessible to people via games, 3D modelling tools, visualization tools, etc, programmers employ a number of techniques to fake this complex shadow operation. The following is a partial list of algorithms that most programmers use to approach this problem:

- Shadow mapping
- →Shadow volumes
- Ray-traced shadows

Though current computer games are leaning toward a mixture of shadow volume techniques and shadow volumes, the explosion of growth in the graphics card arena has favored more toward shadow mapping, since memory is the only limitation for shadow mapping.

Algorithm overview

As in any other graphics discussion, the standard camera, spot-light are assumed to exist. The camera (usually a perspective camera) points toward the scene, and the algorithm must paint the shadowed region using appropriate color — usually by dark color, although some might replace this dark color with a texture so as to achieve the effect of a projector head onto the object.

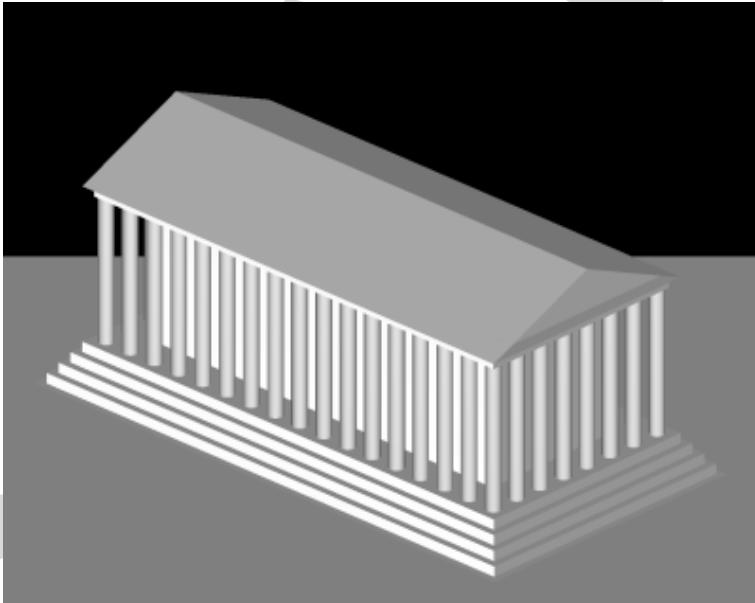


Figure 83 Pass one, render

Pass one

Pass one involves rendering the scene from the light's point of view. Since this example uses a directional source of light (e.g., the Sun), it makes more sense to have orthographic projection, instead of perspective projection for the shadow map (inferring a point source of light).

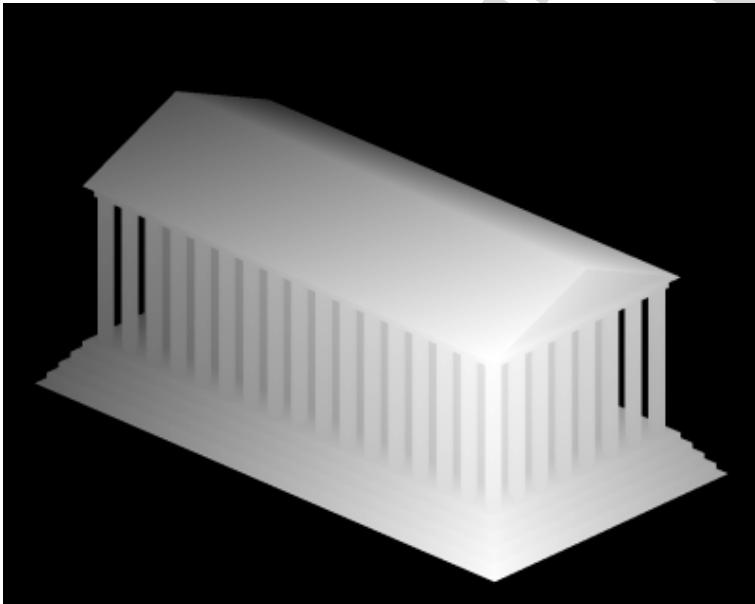


Figure 84 Pass one, depth map

Since only the needed information is used, the color buffer (and others) may be disabled, along with any color-altering schemes (e.g., lighting, shaders, texturing). It is important that only the shadow-casting objects are rendered, opposed to the objects that are shadowed. When shadow mapping is done in realtime, speed is important and therefore the less that is rendered, the better. A depth-offset is usually applied and enabled for the rendering of the shadow casters, since if this is not used, ugly stitching artifacts will result.

When the rendering of the shadow casters are complete, a texture map (image) of the depth buffer is made. This is the shadow map that will be used for the remainder of the process.

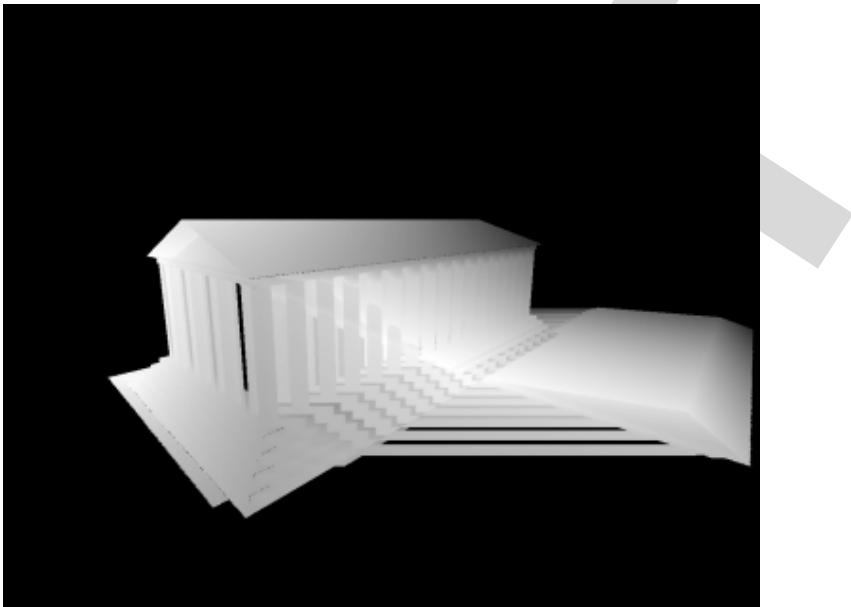


Figure 85 Pass two, projection

Pass two

The second pass involves drawing the scene from the eye's view. The shadow map is first projected over the whole scene, or at least, the shadow receivers, from the light's point of view. This is a mathematically intense part of the process, since multiple matrices are involved to generate the proper texture coordinates.

A test is applied to every pixel, to determine how far from the light the depth map says it is, and how far it really is. Usually, if its greater, the pixel fails the test, and is considered "shadowed".

Pass three

Pass three draws in the shadowed parts of the scene, since very rarely will shadows ever be totally black. This step can be done before or after pass two, but most often it is combined with pass two. A handful of ways exist, but it is most often combined by a shader that simply applies a different lighting scheme to the pixel. For this very reason alone, it is hard to find programs that use shadow mapping without shaders.

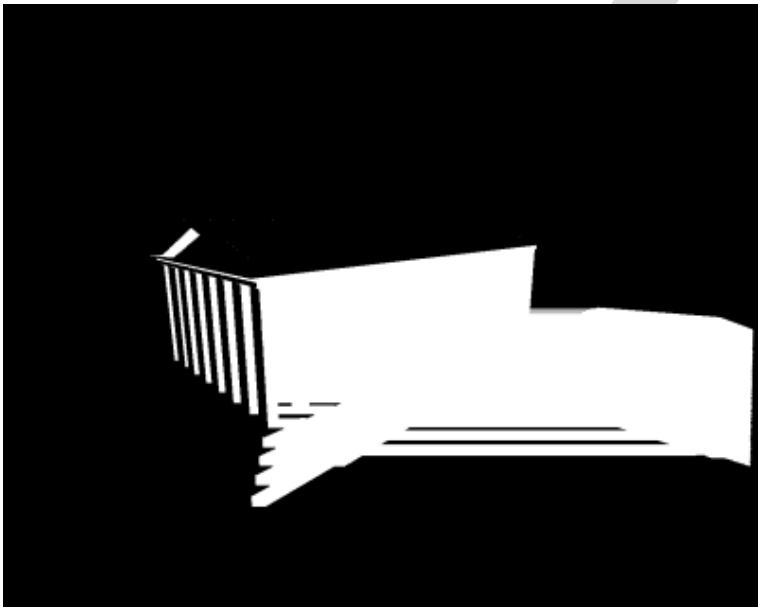


Figure 86 Pass two, failures

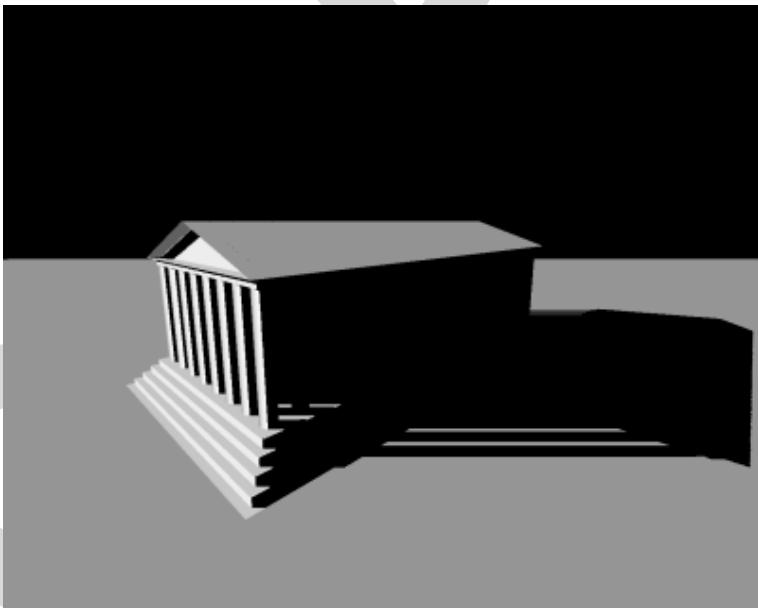


Figure 87 Pass two complete

Shadow mapping

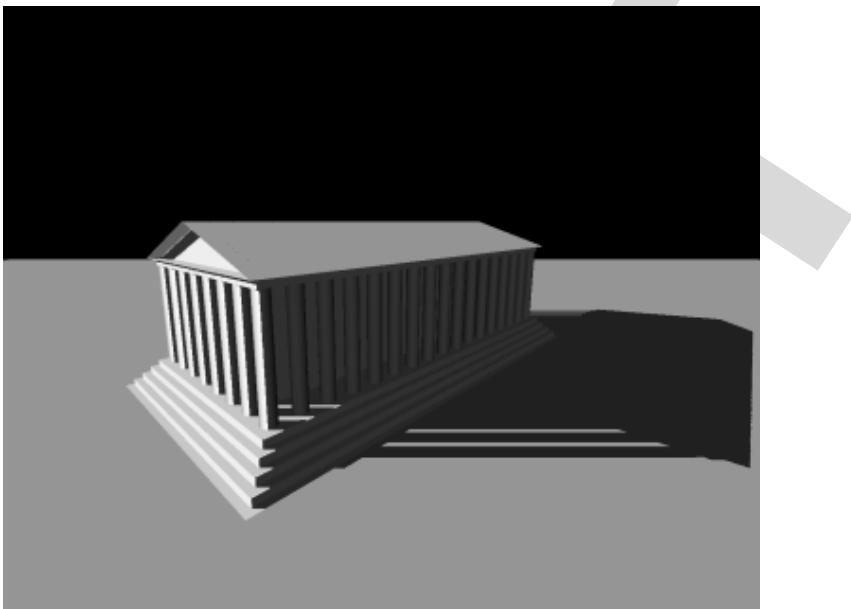


Figure 88 Final scene

This example used the →OpenGL extension *GL_ARB_shadow_ambient* to accomplish the process in two passes.

See also

- →Shadow volume, another shadowing technique

External links

- Hardware Shadow Mapping³⁴⁴, nVidia
- Shadow Mapping with Today's OpenGL Hardware³⁴⁵, nVidia

Source: http://en.wikipedia.org/wiki/Shadow_mapping

Principal Authors: Praetor alpha, Dormant25, Tommstein, Starfox, Klassobanieras

³⁴⁴ <http://developer.nvidia.com/attach/8456>

³⁴⁵ <http://developer.nvidia.com/attach/6769>

Shadow volume

Shadow volumes are a technique used in →3D computer graphics since 1977 to add shadows to a rendered scene. It is generally considered among the most practical general purpose real-time shadowing systems given the capabilities of modern 3D graphics hardware, and has been popularized by the computer game *Doom 3*.

A shadow volume divides the virtual world into two; areas that are in shadow and areas that are not.

Construction

In order to construct a shadow volume, project a line from the light through each vertex in the shadow casting object to some point (generally at infinity). These projections will together form a volume; any point inside that volume is in shadow, everything outside is lit by the light.

Actual shadow volumes are computed in the following way:

- Find all silhouette edges (edges which separate front-facing faces from back-facing faces)
- Extend all silhouette edges in the direction away from the light-source to form quadrilateral surfaces
- Add *front-cap* and *back-cap* to the surfaces to form a closed volume (may not be necessary, depending on the implementation used)

Usage

Current research and implementations focus on the use of a hardware stencil buffer to optimize the algorithm making use of hardware acceleration - see →Stencil shadow volume.

In order to test whether a given pixel in the rendered image is shadowed or not, the shadow volume itself is rendered, though to the stencil buffer only and not to the final image. For every front-facing face in the shadow volume the value in the stencil buffer is increased; for every back-facing face it is decreased.

Once all faces in the shadow volume have been rendered to the stencil buffer, any pixel with a value not equal to zero will be in shadow.

In order to understand why, think in terms of a ray of light heading back from the pixel on the screen - if it passes into the shadow volume, it will be through a front-facing face and so the stencil buffer value will be increased. If it then

passes out through the back of the volume (through a back-facing face) it will be decreased.

If, however, the pixel is in shadow then the value will only be decreased when it leaves the shadow volume at the back, and so the value will be non-zero.

Figure 1 shows a simple scene containing a camera, a light, a shadow casting object (the blue circle) and three shadow receiving objects (the green squares), all represented in 2D. The thick black lines show the outline of the shadow volume. The line projecting from the camera (smiley face) is the line of sight.

The line of sight first hits object a; at this point it has not reached any faces of the shadow volume and so the stencil buffer will have a value of 0 - not in shadow. At point 1 a front face of the shadow volume is crossed and increment the stencil buffer. Next we hit object b; at this point we have a value of 1 in the buffer so the object is in shadow. Progressing down the line of sight we reach the back face of the shadow volume at 2, and decrement the stencil buffer back to 0. Finally we hit object c, again with a value of 0 in the stencil buffer so the object is not in shadow.

One problem with this algorithm is that if the camera (eye) is itself in the shadow volume then this approach will fail - the line of sight first crosses a back face of the shadow volume, decrementing the stencil buffer value so that it is non-zero when it reaches object c, even though that object should not be shadowed.

One solution to this is to trace backwards from some point at infinity to the eye of the camera. This technique was discovered independently by a number of people, but was popularized by John Carmack, and is generally known as →Carmack's Reverse.

See also

- →Stencil shadow volume, which describes how shadow volumes are implemented using the *stencil buffer*
- →Silhouette edge
- →Shadow mapping, an alternative shadowing algorithm

External Articles

- <http://www.gamedev.net/reference/articles/article1873.asp> - this is an excellent introductory article
- <http://www.gamedev.net/reference/articles/article2036.asp> - another explanation of the basic algorithm

Source: http://en.wikipedia.org/wiki/Shadow_volume

Principal Authors: Orderud, Steve Leach, Staz69uk, Cma, LiDaobing

Silhouette edge

In computer graphics, a **silhouette edge** on a 3D body projected onto a 2D plane (display plane) is the collection of points whose outwards *surface normal is perpendicular to the view vector*. Due to discontinities in the surface normal, a silhouette edge is also an edge which separates a front facing face from a back facing face. Without loss of generality, this edge is usually chosen to be the closest one on a face, so that in parallel view this edge corresponds to the same one in a perspective view. Hence, if there is an edge between a front facing face and a side facing face, and another edge between a side facing face and back facing face, the closer one is chosen. The easy example is looking at a cube in the direction where the face normal is colinear with the view vector.

The first type of silhouette edge is sometimes troublesome to handle because it does not necessarily correspond to a physical edge in the CAD model. The reason that this can be an issue is that a programmer might corrupt the original model by introducing the new silhouette edge into the problem. Also, given that the edge strongly depends upon the orientation of the model and view vector, this can introduce numerical instabilities into the algorithm (such as when a trick like dilution of precision is considered).

Computation

To determine the silhouette edge of an object, we first have to know the plane equation of all faces. Then, by examining the sign of the *point-plane distance* from the light-source to each face

$$ax + by + cz + d = \begin{cases} > 0 & \text{front facing} \\ = 0 & \text{parallel} \\ < 0 & \text{back facing} \end{cases}$$

we can determine if the edges is front- or back facing.

The silhouette edge(s) consist of all **edges separating a front facing face from a back facing face**.

External links

Source: http://en.wikipedia.org/wiki/Silhouette_edge

Principal Authors: Orderud, Wheger, Gaius Cornelius, David Levy, RJHall

Solid modelling

Solid modelling (or **modeling**) is the unambiguous representation of the solid parts of an object, that is, models of solid objects suitable for computer processing. It is also known as **volume modelling**. Other modelling methods include surface models (used extensively in automotive and consumer product design as well as entertainment animation) and wire frame models (which can be ambiguous about solid volume).

Primary uses of solid modelling are for CAD, engineering analysis, computer graphics and animation, rapid prototyping, medical testing, product visualization and visualization of scientific research.

Basic theoretical concepts

- **Sweeping**
 - An area feature is "swept out" by moving a primitive along a path to form a solid feature. These volumes either add to the object ("extrusion") or remove material ("cutter path").
 - Also known as 'sketcher based modelling'.
 - Analogous to various manufacturing techniques such as extrusion, milling, lathe and others.
- **Boundary representation (BREP)**
 - A solid object is represented by boundary surfaces and then filled to make solid.
 - Also known as 'surfacing'.
 - Analogous to various manufacturing techniques; Injection moulding, casting, forging, thermoforming, etc.
- **Parameterized primitive instancing**.

- An object is specified by reference to a library of parameterized **primitives**.
- For example, a bolt is modelled for a library, this model is used for all bolt sizes by modifying a set of its parameters.
- **Spatial occupancy** (voxel)
 - The whole space is subdivided into regular **cells**, and the object is specified by the set of cells it occupies.
 - Models described this way lend themselves to Finite difference analysis.
 - This is usually done after a model is made, as part of automated pre-processing for analysis software.
- **Facet modeling**
 - Forming the outside surface form of the volume from any triangular planes
 - Often used in reverse engineering of physical models.
- **Decomposition**
 - Similar to "spatial occupancy", but the cells are neither regular, nor "prefabricated".
 - Models described this way lend themselves to FEA.
 - This is usually done after a model is made, as part of automated pre-processing for analysis software.
- →**Constructive solid geometry**
 - Simple objects are combined using boolean operators such as union, difference, and intersection.
- **Feature based modelling**
 - Complex combinations of objects and operators are considered together as a unit which can be modified or duplicated.
 - Order of operations is kept in a history tree, and parametric changes can propagate through the tree.
- **Parametric modelling**
 - Attributes of features are parameterized, giving them labels rather than only giving them fixed numeric dimensions, and relationships between parameters in the entire model are tracked, to make changing numeric values of parameters easier.
 - Almost always combined with features, giving parametric feature based modelling.

History

Solid modeling has to be seen in context of the whole history of CAD, the key milestones being the development of Romulus which went on to influence the development of Parasolid and ACIS and thus the mid-range Windows based feature modelers such as SolidWorks and Solid Edge and the arrival of parametric solid models system like T-Flex and Pro/Engineer.

Practical applications

Parametric Solid modelling CAD

Solid modellers have become commonplace in engineering departments in the last ten years due to faster PCs and competitive software pricing. They are the workhorse of machine designers.

Solid modelling software creates a virtual 3D representation of components for machine design and analysis. Interface with the human operator is highly optimized and includes programmable macros, keyboard shortcuts and dynamic model manipulation. The ability to dynamically re-orient the model, in real-time shaded 3-D, is emphasized and helps the designer maintain a mental 3-D image.

Design work on components is usually done within context of the whole product using assembly modelling methods.

A solid model generally consists of a group of features, added one at a time, until the model is complete. Engineering solid models are built mostly with sketcher-based features; 2-D sketches that are swept along a path to become 3-D. These may be cuts, or extrusions for example.

Another type of modelling technique is 'surfacing' (Freeform surface modelling). Here, surfaces are defined, trimmed and merged, and filled to make solid. The surfaces are usually defined with datum curves in space and a variety of complex commands. Surfacing is more difficult, but better applicable to some manufacturing techniques, like injection molding. Solid models for injection molded parts usually have both surfacing and sketcher based features.

Engineering drawings are created semi-automatically and reference the solid models.

The learning curve for these software packages is steep, but a fluent machine designer who can master these software packages is highly productive.

The modelling of solids is only the minimum requirement of a CAD system's capabilities.

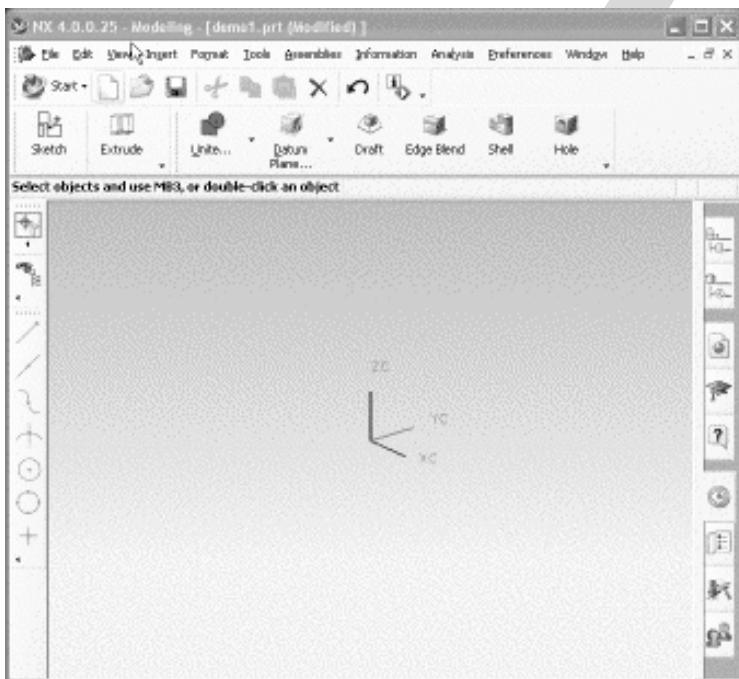


Figure 89

Parametric modelling uses parameters to define a model (dimensions, for example). The parameter may be modified later, and the model will update to reflect the modification. Typically, there is a relationship between parts, assemblies, and drawings. A part consists of multiple features, and an assembly consists of multiple parts. Drawings can be made from either parts or assemblies.

Example: A shaft is created by extruding a circle 100 mm. A hub is assembled to the end of the shaft. Later, the shaft is modified to be 200 mm long (click on the shaft, select the length dimension, modify to 200). When the model is updated the shaft will be 200 mm long, the hub will relocate to the end of the shaft to which it was assembled, and the engineering drawings and mass properties will reflect all changes automatically.

Examples of parameters are: dimensions used to create model features, material density, formulas to describe swept features, imported data (that describe a reference surface, for example).

Related to parameters, but slightly different are Constraints. Constraints are relationships between entities that make up a particular shape. For a window, the sides might be defined as being parallel, and of the same length.

Parametric modelling is obvious and intuitive. But for the first three decades of CAD this was not the case. Modification meant re-draw, or add a new cut or protrusion on top of old ones. Dimensions on engineering drawings were *created*, instead of *shown*.

Parametric modelling is very powerful, but requires more skill in model creation. A complicated model for an injection molded part may have a thousand features, and modifying an early feature may cause later features to fail. Skillfully created parametric models are easier to maintain and modify.

Parametric modelling also lends itself to data re-use. A whole family of capscrews can be contained in one model, for example.

Entertainment

Animation of a computer generated character is an example of parametric modelling. Jar Jar Binks is described by parameters which locate key body positions. The model is then built off these locations. The parameters are modified, and the model rebuilt, for each frame to create animation.

Medical solid modelling

Modern computed axial tomography and magnetic resonance imaging scanners can construct solid models of interior body features.

Uses of medical solid modelling;

- Visualization
- Visualization of specific body tissues (just blood vessels and tumor, for example)
- Creating solid model data for rapid prototyping (to aid surgeons preparing for difficult surgeries, for example)
- Combining medical solid models with CAD solid modelling (design of hip replacement parts, for example)

See also

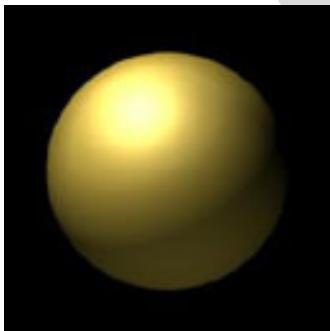
- Computer graphics
- Computational geometry
- Euler boundary representation
- Engineering drawing

- Technical drawing
- SolidWorks
- SolidEdge
- Pro/ENGINEER
- BRL-CAD
- I-DEAS
- NX (Unigraphics)
- CATIA
- AutoCAD

Source: http://en.wikipedia.org/wiki/Solid_modelling

Principal Authors: Duk, Freeformer, Mikkalai, Oleg Alexandrov, Wheger

Specular highlight



A **specular highlight** is the bright spot of light that appears on shiny objects when illuminated (for example, see image at right). Specular highlights are important in →3D computer graphics, as they provide a strong visual cue for the shape of an object and its location with respect to light sources in the scene.

Microfacets

The term *specular* means that light is perfectly reflected in a mirror-like way from the light source to the viewer. Specular reflection is visible only where the surface normal is oriented precisely halfway between the direction of incoming light and the direction of the viewer; this is called the **half-angle** direction because it bisects (divides into halves) the angle between the incoming light and the viewer. Thus, a specularly reflecting surface would show a specular

highlight as the perfectly sharp reflected image of a light source. However, many shiny objects show blurred specular highlights.

This can be explained by the existence of **microfacets**. We assume that surfaces that are not perfectly smooth are composed of many very tiny facets, each of which is a perfect specular reflector. These microfacets have normals that are distributed about the normal of the approximating smooth surface. The degree to which microfacet normals differ from the smooth surface normal is determined by the roughness of the surface.

The reason for blurred specular highlights is now clear. At points on the object where the smooth normal is close to the half-angle direction, many of the microfacets point in the half-angle direction and so the specular highlight is bright. As one moves away from the center of the highlight, the smooth normal and the half-angle direction get farther apart; the number of microfacets oriented in the half-angle direction falls, and so the intensity of the highlight falls off to zero.

The specular highlight often reflects the color of the light source, not the color of the reflecting object. This is because many materials have a thin layer of clear material above the surface of the pigmented material. For example plastic is made up of tiny beads of color suspended in a clear polymer and human skin often has a thin layer of oil or sweat above the pigmented cells. Such materials will show specular highlights in which all parts of the color spectrum are reflected equally. On metallic materials such as gold the color of the specular highlight will reflect the color of the material.

Models of Microfacets

A number of different models exist to predict the distribution of microfacets. Most assume that the microfacet normals are distributed evenly around the normal; these models are called **isotropic**. If microfacets are distributed with a preference for a certain direction along the surface, the distribution is **anisotropic**.

Phong distribution

In the →Phong reflection model, the intensity of the specular highlight is calculated as $k_{spec} = \cos^n(R, V)$, where R is the mirror reflection of the light vector off the surface, and V is the viewpoint vector.

In Blinn-Phong shading, the intensity of a specular highlight is calculated as $k_{spec} = \cos^n(N, H)$, where N is the smooth surface normal and H is the half-angle direction (the direction vector midway between L , the vector to the light, and V , the viewpoint vector).

The number n is called the Phong exponent, and is a user-chosen value that controls the apparent smoothness of the surface. These equations imply that the distribution of microfacet normals is an approximately Gaussian distribution, or approximately Pearson type II distribution, of the corresponding angle.³⁴⁶ While this is a useful heuristic and produces believable results, it is not a physically based model.

Gaussian distribution

A slightly better model of microfacet distribution can be created using a Gaussian distribution. The usual function calculates specular highlight intensity as:

$$k_{spec} = e^{-\left(\frac{\angle(N,H)}{m}\right)^2}$$

where m is a constant between 0 and 1 that controls the apparent smoothness of the surface.

Material from this section adapted from: Glassner, Andrew S. (ed). An Introduction to Ray Tracing. San Diego: Academic Press Ltd, 1989. p. 148.

Beckmann distribution

A physically based model of microfacet distribution is the Beckmann distribution. This function gives very accurate results, but is also rather expensive to compute.

$$k_{spec} = \frac{1}{4m^2 \cos^4(N,H)} e^{-\left(\frac{\tan(N,H)}{m}\right)^2}$$

where m is as before: a constant between 0 and 1 that controls the apparent smoothness of the surface.

Material from this section adapted from: Foley et al. Computer Graphics: Principles and Practice. Menlo Park: Addison-Wesley, 1990. p. 764.

Heidrich-Seidel anisotropic distribution

The Heidrich-Seidel distribution is a simple anisotropic distribution, based on the Phong model. It can be used to model surfaces that have small parallel grooves or fibers, such as brushed metal, satin, and hair. The specular highlight intensity for this distribution is:

³⁴⁶ Richard Lyon, "Phong Shading Reformulation for Hardware Renderer Simplification", Apple Technical Report #43, Apple Computer, Inc. 1993

$$k_{spec} = [\sin(L, T) \sin(V, T) - \cos(L, T) \cos(V, T)]^n$$

where n is the Phong exponent, V is the viewing direction, L is the direction of incoming light, and T is the direction parallel to the grooves or fibers at this point on the surface.

Ward anisotropic distribution

The Ward anisotropic distribution³⁴⁷ uses two user-controllable parameters α_x and α_y to control the anisotropy. If the two parameters are equal, then an isotropic highlight results. The specular term in the distribution is:

$$k_{spec} = \frac{1}{\sqrt{(N \cdot L)(N \cdot V)}} \frac{N \cdot L}{4\alpha_x \alpha_y} \exp \left[-2 \frac{\left(\frac{H \cdot X}{\alpha_x} \right)^2 + \left(\frac{H \cdot Y}{\alpha_y} \right)^2}{1 + (H \cdot N)} \right]$$

The specular term is zero if $N \cdot L < 0$ or $N \cdot E < 0$. All vectors are unit vectors. The vector V is the vector from the surface point to the eye, L is the direction from the surface point to the light, H is the half-angle direction, N is the surface normal, and X and Y are two orthogonal vectors in the normal plane which specify the anisotropic directions.

Using multiple distributions

If desired, different distributions (usually, using the same distribution function with different values of m or n) can be combined using a weighted average. This is useful for modelling, for example, surfaces that have small smooth and rough patches rather than uniform roughness.

References

See also

- →Diffuse reflection
- Retroreflector
- Reflection (physics)
- Refraction

Source: http://en.wikipedia.org/wiki/Specular_highlight

Principal Authors: Dicklyon, Reedbeta, BenFrantzDale, PlowboyLifestyle, Connelly

³⁴⁷ <http://courses.dce.harvard.edu/~cscie234/papers/Surface%20Reflection%20Models.pdf>

Specularity

Specularity is the quality used in many 3D rendering programs to set the size and the brightness of a texture's reflection to light.

Adding specularity into a render allows one to add highlights to an object.

Specularity only provides an illusion of reflection to the light. The function of specularity only provides a central hotspot and the halo surrounding it.

In some situations, specularity is preferred over reflection because of faster render speeds (due to the complexity of raytracing in reflection algorithms) and because, even though reflection mapping is more realistic, specularity algorithms often appear more realistic.

External links

- Leigh Van Der Byl Specularity Tutorial³⁴⁸

Source: <http://en.wikipedia.org/wiki/Specularity>

Principal Authors: Neonstarlight, Frap, Volfy, Megan1967, Oliver Lineham

Stencil buffer

A **Stencil buffer** is an extra buffer, in addition to the *color buffer* (pixel buffer) and *depth buffer* (z-buffer) found on modern computer graphics hardware. The buffer is per pixel, and works on integer values, usually with a depth of one byte per pixel. The depth buffer and stencil buffer often share the same area in the RAM of the graphics hardware.

In the simplest case, the stencil buffer is used to limit the area of rendering (stenciling). More advanced usage of the stencil buffer make use of the strong connection between the depth buffer and the stencil buffer in the rendering pipeline (for example, stencil values can be automatically increased/decreased for every pixel that failed or passed the depth test).

The simple combination of depth test and stencil modifiers make a vast number of effects possible (such as shadows, outline drawing or highlighting of

³⁴⁸ <http://leigh.cgcommunity.com/tutorials/part5.htm>

intersections between complex primitives) though they often require several rendering passes and, therefore, can put a heavy load on the graphics hardware.

The most typical application is still to add shadows to 3D applications.

The stencil buffer and its modifiers can be accessed in computer graphics APIs like →OpenGL and →Direct3D.

See also

- →Z-buffering (depth buffer)
- →Stencil shadow volume

Source: http://en.wikipedia.org/wiki/Stencil_buffer

Principal Authors: Cyc, Claynoik, Eep², Orderud, Ddawson

Stencil shadow volume

A **Stencil shadow volume** is a method of rendering in →3D computer graphics. Frank Crow introduced shadow volumes in 1977 as the geometry describing the 3D shape of the region occluded from a light source. Tim Heidmann later showed how to use the stencil buffer to render shadows with shadow volumes in real time. Heidmann's approach only worked when the virtual camera was itself not in shadow. Around 2000, several people discovered that Heidmann's method can be made to work for all camera positions by reversing a test based on the z-coordinate. His original method is now known as *z-pass* and the new method is called *z-fail*. Sim Dietrich's PowerPoint talk at a Creative Labs talk appears to be the first to mention this; however, Cass Everitt and Mark Kilgard's 2002 NVIDIA technical report is the first detailed analysis of the technique. John Carmack of id Software popularized the technique by using it in the Doom 3 video game, so it is often referred to as →Carmack's Reverse.

Creative Labs has recently attempted to enforce a patent on z-fail testing.

Variants

There are several ways of implementing stencil shadow volumes, all of which use the *stencil buffer* available in both →OpenGL and DirectX. The variants differ in speed, robustness and the number of stencil-bits required.

All variants require the construction of a shadow volume first, using the properties of silhouette edges.

Exclusive-Or variant

This is the fastest and simplest variant, requiring only a single additional pass. This method also only requires a single bit in the stencil buffer.

1. Render scene with only ambient lighting
2. Render entire shadow volume to the stencil buffer. Invert stencil value on *z-pass*
3. Render scene with diffuse and specular lighting in areas of zero stencil value

This variant has the drawback of intersecting shadow volumes canceling out each other. It also fails if the camera is inside a shadow volume.

Depth-pass variant

This is a more advanced *stencil counting* approach capable of handling intersecting shadow volumes.

1. Render scene with only ambient lighting
2. Render front-facing faces to the stencil buffer. Increment stencil value on *z-pass*
3. Render back-facing faces to the stencil buffer. Decrement stencil value on *z-pass*
4. Render scene with diffuse and specular lighting in areas of zero stencil value

The near-plane is a plane in front of the camera; it is used to clip geometry that is in front of that plane. The Depth-pass variant of the stencil shadow volume algorithm fails if the near-rectangle (the portion of the near-plane visible by the camera) intersects a shadow volume. It is possible, however, to 'repair' such failure, with an additional rendering pass, see ZP+ in the External links section, below.

Depth-fail variant (Carmack's Reverse)

This is an even more advanced *stencil counting* approach capable of handling both intersecting shadow volumes and cameras inside the shadow volumes.

1. Render scene with only ambient lighting
2. Render back-facing faces to the stencil buffer. Increment stencil value on *z-fail*
3. Render front-facing faces to the stencil buffer. Decrement stencil value on *z-fail*
4. Render scene with diffuse and specular lighting in areas of zero stencil value

This variant does not fail if the camera is inside a shadow volume, but **requires closed shadow volumes** which can be expensive to compute.

Common problems

- Non-convex objects may have *self-intersecting silhouettes*, leading to artifacts in the shadows if this is not accounted for (by removing all self-intersections before rendering).
- Stencil buffer uses saturation arithmetic, which may lead to problems if the initial stencil value is zero.

See also

- →Shadow volume, which describes the general algorithm
- →Silhouette edge, used to determine the shadow volume
- →Stencil buffer, the buffer used

External links

- The Theory of Stencil Shadow Volumes³⁴⁹ by Hun Yen Kwoon on GameDev.net
- Tutorial - Stenciled Shadow Volumes in OpenGL³⁵⁰ by Josh Beam on 3ddrome
- ZP+: correct Z-pass stencil shadows³⁵¹

Source: http://en.wikipedia.org/wiki/Stencil_shadow_volume

Principal Authors: Orderud, Staz69uk, Gracefool, HopeSeekr of xMule, Boredzo

³⁴⁹ <http://www.gamedev.net/reference/articles/article1873.asp>

³⁵⁰ <http://www.3ddrome.com/articles/shadowvolumes.php>

³⁵¹ <http://artis.inrialpes.fr/Publications/2005/HHLH05/>

Subdivision surface

In computer graphics, **subdivision surfaces** are used to create smooth surfaces out of arbitrary meshes. Subdivision surfaces are defined as the limit of an infinite refinement process. They were introduced simultaneously by Edwin Catmull and Jim Clark, and by Daniel Doo and Malcom Sabin in 1978. Little progress was made until 1995, when Ulrich Reif solved subdivision surfaces behaviour near extraordinary vertices.

The fundamental concept is refinement. By repeatedly refining an initial polygonal mesh, a sequence of meshes is generated that converges to a resulting subdivision surface. Each new subdivision step generates a new mesh that has more polygonal elements and is smoother.

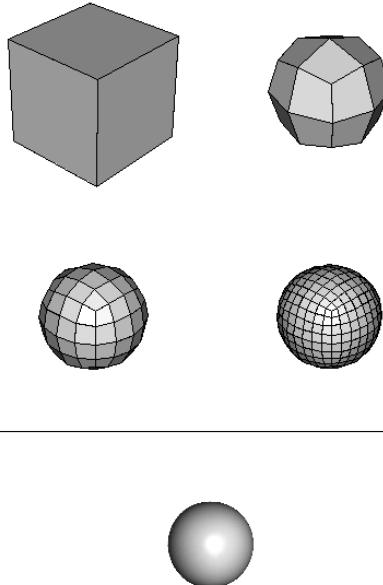


Figure 90 First three steps of Catmull-Clark subdivision of a cube with subdivision surface below

Subdivision methods

There are several refinement schemes:

- **Catmull-Clark** is a generalization of bi-cubic uniform B-splines
- **Doo-Sabin** is a generalization of bi-quadratic uniform B-splines
- **Loop**, by Charles Loop, is a generalization of quartic triangular box splines (works with triangular meshes)
- **Butterfly** named after the scheme's shape
- **Midedge**
- **Kobbelt** is a variational subdivision method that tries to overcome uniform subdivision drawbacks

Advantages over NURBS modelling

Subdivision surface modeling is now preferred over NURBS modeling in major modelers because subdivision surfaces have many benefits:

- work with more complex topology
- numerically stable
- easier to implement
- local continuity control
- local refinement
- no tessellation issue
- switch between coarser and finer refinement

B-spline relationship

B-spline curves are refinable: their control point sequence can be refined and the iteration process converges to the actual curve. This is a useless property for curves, but its generalization to surfaces yields subdivision surfaces.

Refinement process

Interpolation inserts new points while original ones remain undisturbed.

Refinement inserts new points and moves old ones in each step of subdivision.

Extraordinary points

The Catmull-Clark refinement scheme is a generalization of bi-cubic uniform B-splines. Any portion of the surface that is equivalent to a 4×4 grid of control points represents a bi-cubic uniform B-spline patch. Surface refinement is easy in those areas where control points valence is equal to four. Defining a subdivision surface at vertices with valence other than four was historically difficult;

such points are called **extraordinary points**. Similarly, extraordinary points in the Doo-Sabin scheme have a valence other than three.

Most schemes don't produce extraordinary vertices during subdivision.

External links

- Resources about subdivisions³⁵²
- Geri's Game³⁵³ : Oscar winning animation by Pixar completed in 1997 that introduced subdivision surfaces (along with cloth simulation)
- Subdivision for Modeling and Animation³⁵⁴ tutorial, SIGGRAPH 2000 course notes
- A unified approach to subdivision algorithms near extraordinary vertices³⁵⁵, Ulrich Reif (Computer Aided Geometric Design 12(2):153-174 March 1995)

Source: http://en.wikipedia.org/wiki/Subdivision_surface

Principal Authors: Orderud, Romainbehar, Furrykef, Lauciusa, Feureau

Subsurface scattering

Subsurface scattering (or SSS) is a mechanism of light transport in which light penetrates the surface of a translucent object, is scattered by interacting with the material, and exits the surface at a different point. The light will generally penetrate the surface and be reflected a number of times at irregular angles inside the material, before passing back out of the material at an angle other than the angle it would reflect at had it reflected directly off the surface. Subsurface scattering is important in →3D computer graphics, being necessary for the realistic rendering of materials such as marble, skin, and milk.

³⁵² <http://www.subdivision.org/subdivision/index.jsp>

³⁵³ <http://www.pixar.com/shorts/gg/theater/index.html>

³⁵⁴ <http://www.mrl.nyu.edu/dzorin/sig00course/>

³⁵⁵ [http://dx.doi.org/10.1016/0167-8396\(94\)00007-F](http://dx.doi.org/10.1016/0167-8396(94)00007-F)

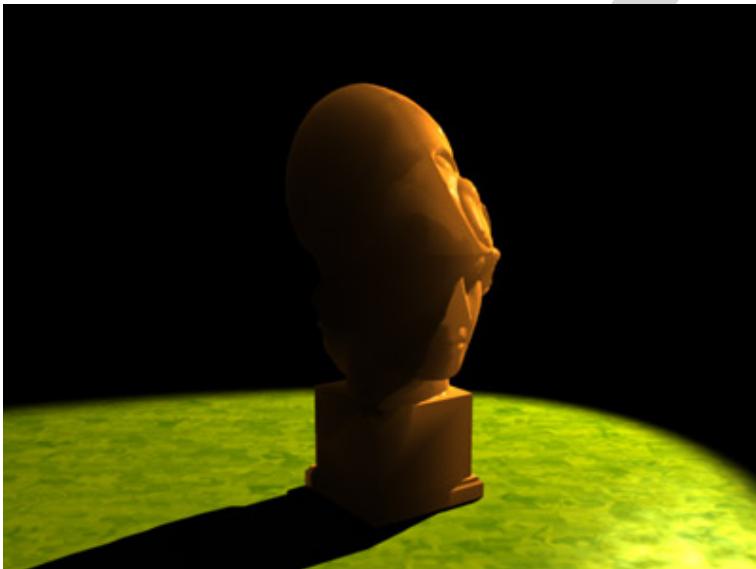


Figure 91 Three dimensional object with subsurface scattering

External links

- Henrik Wann Jensen's subsurface scattering website³⁵⁶
- An academic paper by Jensen on modelling subsurface scattering³⁵⁷
- Simple superficial surface scattering for Blender³⁵⁸

Source: http://en.wikipedia.org/wiki/Subsurface_scattering

Principal Authors: Reedbeta, T-tus, Rufous, RJHall, ALoopingIcon

³⁵⁶ <http://graphics.ucsd.edu/~henrik/images/subsurf.html>

³⁵⁷ <http://graphics.ucsd.edu/~henrik/papers/bssrdf/>

³⁵⁸ http://www.dedalo-3d.com/index.php?filename=SXCOL/mh_ssss/abstract.html

Surface caching

Surface caching is a computer graphics technique pioneered by John Carmack, first used in the computer game *Quake*. The traditional method of lighting a surface is to calculate the surface from the perspective of the viewer, and then apply the lighting to the surface. Carmack's technique is to light the surface independent of the viewer, and store that surface in a cache. The lighted surface can then be used in the normal rendering pipeline for any number of frames.

Surface caching is one of the reasons that it became practical to make a true 3D game that was reasonably fast on a 66 MHz Pentium microprocessor.

External links

- *Quake's Lighting Model: Surface Caching*³⁵⁹ - an in-depth explanation by Michael Abrash

Source: http://en.wikipedia.org/wiki/Surface_caching

Principal Authors: Fredrik, KirbyMeister, Schneelocke, Tregoweth, Hephaestos

Surface normal

A **surface normal**, or just **normal** to a flat surface is a three-dimensional vector which is perpendicular to that surface. A normal to a non-flat surface at a point p on the surface is a vector which is perpendicular to the tangent plane to that surface at p . The word *normal* is also used as an adjective as well as a noun with this meaning: a line *normal* to a plane, the *normal* component of a force, the *normal* vector, etc.

Calculating a surface normal

For a polygon (such as a triangle), a surface normal can be calculated as the vector cross product of two edges of the polygon.

For a plane given by the equation $ax + by + cz = d$, the vector (a, b, c) is a normal.

³⁵⁹ <http://www.bluesnews.com/abrash/chap68.shtml>

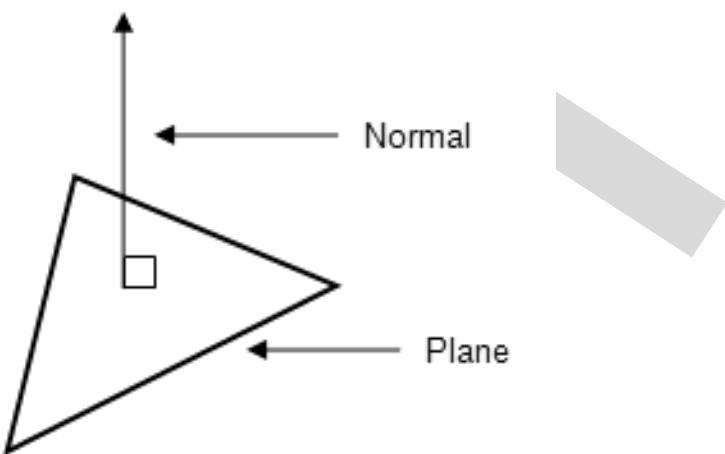


Figure 92 A polygon and one of its two normal vectors.

If a (possibly non-flat) surface S is parametrized by a system of curvilinear coordinates $\mathbf{x}(s, t)$, with s and t real variables, then a normal is given by the cross product of the partial derivatives

$$\frac{\partial \mathbf{x}}{\partial s} \times \frac{\partial \mathbf{x}}{\partial t}.$$

If a surface S is given implicitly, as the set of points (x, y, z) satisfying $F(x, y, z) = 0$, then, a normal at a point (x, y, z) on the surface is given by the gradient

$$\nabla F(x, y, z).$$

If a surface does not have a tangent plane at a point, it does not have a normal at that point either. For example, a cone does not have a normal at its tip nor does it have a normal along the edge of its base. However, the normal to the cone is defined almost everywhere. In general, it is possible to define a normal almost everywhere for a surface that is Lipschitz continuous.

Uniqueness of the normal

A normal to a surface does not have a unique direction; the vector pointing in the opposite direction of a surface normal is also a surface normal. For an oriented surface, the surface normal is usually determined by the right-hand rule.

Uses

- Surface normals are essential in defining surface integrals of vector fields.
- Surface normals are commonly used in →3D computer graphics for lighting calculations; see →Lambert's cosine law.

External link

- An explanation of normal vectors³⁶⁰ from Microsoft's MSDN

Source: http://en.wikipedia.org/wiki/Surface_normal

Principal Authors: Oleg Alexandrov, Frecklefoot, Olegalexandrov, Patrick, Subhash15

Synthespian

For the Doctor Who novel, see Synthespians™.

A **synthespian** is any synthetic actor. A portmanteau of the words *synthetic*, meaning not of natural origin, and *thespian*, meaning dramatic actor. The dinosaurs in *Jurassic Park*, for instance, were animatronic synthespians created by Stan Winston Studios. Aki Ross from the movie *Final Fantasy: The Spirits Within* was an entirely computer-generated synthespian.

The term "synthespian" was created by Jeff Kleiser and Diana Walczak of Kleiser-Walczak Construction Company.³⁶¹ When they were assembling a synthetic thespian for their project, "Nestor Sextone for President", they coined the term "synthespian".

Source: <http://en.wikipedia.org/wiki/Synthespian>

Principal Authors: RoyBoy, Pegship, Drat, Sean Black, Khaosworks

³⁶⁰ http://msdn.microsoft.com/library/default.asp?url=/library/en-us/directx9_c/directx/graphics/programmingguide/GettingStarted/3DCoordinateSystems/facevertexnormalvectors.asp

³⁶¹ <http://www.kwcc.com/works/sp/lead.html>

Tao (software)

For "The ACE ORB", see TAO (software).

The **Tao Framework** is a library giving .NET and mono developers access to popular graphics and gaming libraries like →OpenGL and SDL.

External links

- Tao homepage³⁶²
- Tao wiki³⁶³

Source: http://en.wikipedia.org/wiki/Tao_%28software%29

Principal Authors: Orderud, Suruena, Asparagus, Zondor

Texel (graphics)

A **texel**, or *texture element* (also *texture pixel*) is the fundamental unit of texture space³⁶⁴, used in computer graphics. Textures are represented by arrays of texels, just as pictures are represented by arrays of pixels.

When texturing a 3D surface, a process known as texture mapping maps texels to appropriate pixels in the output picture. On modern computers, this operation is accomplished on the graphics card.

References

Source: http://en.wikipedia.org/wiki/Texel_%28graphics%29

Principal Authors: Neckelmann, Nlu, Dicklyon, Neg, ONjA

³⁶² <http://www.mono-project.com/Tao>

³⁶³ <http://www.taoframework.com/Home>

³⁶⁴ Andrew Glassner, *An Introduction to Ray Tracing*, San Francisco: Morgan-Kaufmann, 1989

Texture filtering

In computer graphics, texture filtering is the method used to map texels (pixels of a texture) to points on a 3D object. There are many methods of texture filtering, and developers must make a tradeoff between rendering speed and image quality when choosing a texture filtering algorithm.

The purpose of texture filtering is to accurately represent a texture that is not aligned to screen coordinates. This is a common problem in 3D graphics where textures are wrapped on polygons whose surface is not orthogonal to the screen.

Different methods

The fastest method is to take a point on an object and look up the closest texel to that position. The resulting point then gets its color from that one texel. This is sometimes referred to as *nearest neighbor* filtering. It works quite well, but can result in visual artifacts when objects are small, large, or viewed from odd angles.

Antialiasing means thinking of the pixel and texel as blocks on a grid that together make up an image and using the area a texel covers on a pixel as a weight. As this is computationally expensive, a lot of approximations have been invented: mip-mapping, supersampling, anisotropic filtering.

Mip-mapping stores multiple copies of a texture at smaller resolutions. Each mip map is a quarter the resolution of the previous mip map. This speeds up texture mapping on small polygons.

In bilinear filtering, the four closest texels to the screen coordinate are sampled and the weighted average is used as the final colour. In trilinear filtering, bilinear filtering is done for the nearest two mip map levels and the results from both mip maps are averaged. This removes mip map transition lines from the rendered image.

Both bilinear and trilinear filtering do not take into account the angle at which the texture is oriented toward the screen. This produces blurry results for textures that are at receding angle to the screen.

Anisotropic filtering takes up to 16 samples based on the angle of the textured polygon. More texels are sampled at the receding end than the near end. This produces accurate results no matter which way the texture is oriented toward the screen.

See also

- →Texture mapping

Source: http://en.wikipedia.org/wiki/Texture_filtering

Principal Authors: Alanius, Arnero, Rich Farmbrough, Srleffler, Tavla

Texture mapping

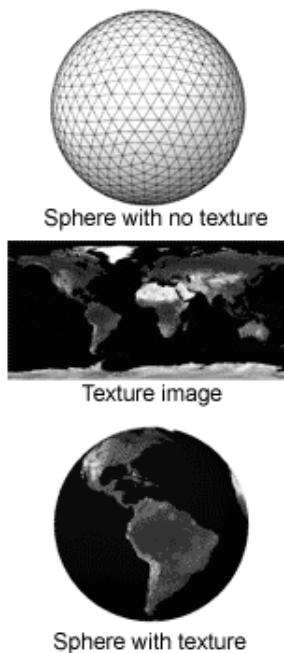


Figure 93 Spherical texture mapping

Texture mapping is a method of adding detailed colour to a computer-generated graphic. An image (the texture) is added (mapped) to a simpler

Texture mapping

shape that is generated in the scene, like a decal pasted onto its surface. This allows a complicated colouring of the surface without requiring additional polygons to represent minute details.

Multitexturing is the use of more than one texture at a time on a shape. This has various uses, sometimes as a way of applying a light map to a surface, which is faster than requiring the graphics hardware to do lighting calculation for that surface on the fly, or more recently bump mapping has become popular, which allows a texture to directly control the lighting calculations, allowing the surface to not only have detailed colouring, but detailed contours as well (bumps).

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 is commonly chosen as good tradeoff between speed and accuracy.

At the hardware level usually texture coordinates are specified at each vertex of a given triangle (any polygon may be broken down into triangles for rendering), and these coordinates are interpolated as part of a calculation that is an extension of Bresenham's line algorithm. Direct interpolation of the texture coordinates between vertices results in *affine* texture mapping, which causes a perceivable discontinuity between adjacent triangles when the 3D geometry of the specified triangle is at an angle to the plane of the screen, *perspective correction* is thus preferred where realism is important, and adjusts the texture coordinate interpolation as a function of the 3D depth of each pixel. Because perspective correction involves more calculation, it did not become commonplace in graphics hardware until recently.

See also

- Edwin Catmull
- →Texture filtering
- Texture Splatting - a technique for combining textures.

External links

- Graphics for the Masses by Paul Rademacher³⁶⁵
- High resolution textures resource³⁶⁶

³⁶⁵ <http://www.cs.unc.edu/~rademach/xroads-RT/RTarticle.html>

³⁶⁶ <http://www.mayang.com/textures/>

- High resolution images and textures resource³⁶⁷
- Texture Hound | Ultimate 3d Texture Links Directory³⁶⁸

Source: http://en.wikipedia.org/wiki/Texture_mapping

Principal Authors: Arnero, Al Fecund, T-tus, Canadacow, RJHall, AzaToth, Collabi, Viznut

Transform and lighting

Transform and lighting is a term used in computer graphics, generally used in the context of hardware acceleration (**Hardware T&L**). Transform refers to the task of converting spatial coordinates, which in this case involves moving three-dimensional objects in a virtual world and converting the coordinates to a two-dimensional view. Lighting refers to the task of taking light objects in a virtual scene, and calculating the resulting colour of surrounding objects as the light falls upon them.

In modern 3D games with complex scenes and detailed lighting effects, the high number of points to be transformed and lit is a computationally intense process, which is why 3D graphics cards offer acceleration.

Source: http://en.wikipedia.org/wiki/Transform_and_lighting

Principal Authors: Mel Etitis, GregorB, Gracefool, Peterhuh, Gothmog.es

Unified lighting and shadowing

Unified lighting and shadowing is a lighting model used in the Doom 3 game engine developed by Id Software.

Previous 3D games like Quake III used separate lighting models for determining how a light would illuminate a character or a map. Lighting and shadow information for maps would be static, pre-generated and stored, whereas lighting and shadowing information for characters would be determined at run-time.

³⁶⁷ <http://www.imageafter.com/>

³⁶⁸ <http://www.texturehound.com/>



Figure 94 Doom 3 uses unified lighting and shadowing. Shadows are calculated using a stencil shadow volume.

Doom 3 claims to use a unified model, which renders every triangle using the same lighting mechanism, regardless as to whether it originated from a model, or map geometry. This is not strictly true as some models are marked with a 'don't self shadow' flag, custom material shaders can allow different lighting mechanisms to be employed on different surfaces (most often a reflective cube map effect), and the point sprite effects (such as explosions) are totally unlit. A renderer using a truly unified lighting system would use an identical set of lighting calculations for every pixel on the screen and would not make such distinctions, although Doom 3's lighting is certainly far 'more unified' than previous games, there is still much more which can be done on recent and future hardware to improve the consistency of lighting in games.

Doom 3 does not use →OpenGL's built in system, instead, it uses its own system which gives better quality and more accurate illumination than OpenGL's default lighting model.

See also

- →Shadow mapping - a technique for rendering shadows
- →Stencil shadow volumes - an alternative method, used for shadows in Doom 3



Figure 95 Quake III used an older lighting model in which shadows are calculated differently for moving characters and backgrounds.

Source: http://en.wikipedia.org/wiki/Unified_lighting_and_shadowing

Principal Authors: AlistairMcMillan, Orderud, Jheriko, Michael Hardy, Gracefool

Utah teapot

The **Utah teapot** or **Newell teapot** is a 3D model which has become a standard reference object (and something of an in-joke) in the computer graphics community. It is a simple, round, solid, partially concave mathematical model of an ordinary teapot.

The teapot model was created in 1975 by early computer graphics researcher Martin Newell, a member of the pioneering graphics program at the University of Utah.

Utah teapot

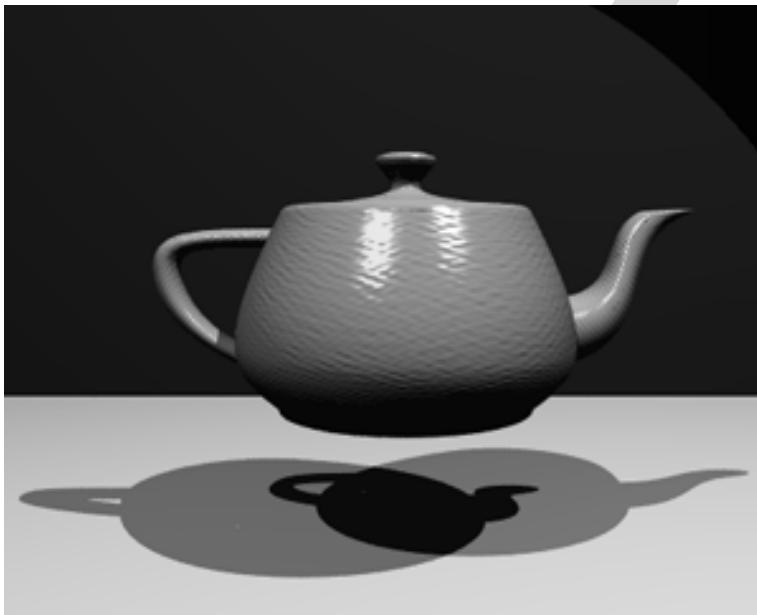


Figure 96 The Utah teapot

History

Newell needed a moderately simple mathematical model of a familiar object for his work. Sandra Newell (his wife) suggested modelling their tea service since they were sitting down to tea at the time. He got some graph paper and a pencil, and sketched the entire tea service by eye. Then, he went back to the lab and edited Bezier control points on a Tektronix storage tube, again by hand. While a cup, saucer, and teaspoon were digitized along with the famous teapot, only the teapot itself attained widespread usage. It is thought that a milk jug was also modelled - but the data for that seems to have been lost.

The teapot shape contains a number of elements that made it ideal for the graphics experiments of the time — it is round, contains saddle points, has a genus greater than zero because of the hole in the handle, can project a shadow on itself, and looks reasonable when displayed without a complex surface texture.

Newell made the mathematical data that describes the teapot's geometry (a set of three-dimensional coordinates) publicly available, and soon other researchers began to use the same data for their computer graphics experiments. These researchers needed something with roughly the same characteristics that



Figure 97 The Melitta teapot that was the prototype for the Utah teapot model.

Newell had, and using the teapot data meant they didn't have to laboriously enter geometric data for some other object. Although technical progress has meant that the act of rendering the teapot is no longer the challenge it was in 1975, the teapot continued to be used as a reference object for increasingly advanced graphics techniques.

Over the following decades, editions of computer graphics journals (such as the ACM SIGGRAPH's quarterly) regularly featured versions of the teapot: faceted or smooth-shaded, wireframe, bumpy, translucent, refractive, even leopard-skin and furry teapots were created.

The original teapot model was never intended to be seen from below and had no surface to represent the base of the teapot; later versions of the data set have fixed this.

The real teapot is noticeably taller than the computer model because Newell's frame buffer used non-square pixels. Rather than distorting the image, Newell's colleague Jim Blinn reportedly scaled the geometry to cancel out the stretching, and when the model was shared with users of other systems, the scaling stuck. Height scale factor was 1.3.

The original, physical teapot was purchased from ZCMI (a department store in Salt Lake City, Utah) in 1974. It was donated to the Boston Computer Museum in 1984 where it was on display until 1990. It now resides in the ephemera collection at the Computer History Museum in Mountain View, California where it

is catalogued as "Teapot used for Computer Graphics rendering" and bears the catalogue number X00398.1984.

Applications

Versions of the teapot model, or sample scenes containing it, are distributed with or freely available for nearly every current rendering and modeling program, including AutoCAD, POV-Ray, →OpenGL, →Direct3D, and 3D Studio Max. Along with the expected cubes and spheres, the GLUT library even provides the function `glutSolidTeapot()` as a graphics primitive, as does its →Direct3D counterpart `D3DXCreateTeapot()`. BeOS included a small demo of a rotating 3D teapot, intended to show off the platform's multimedia facilities.

Teapot scenes are commonly used for renderer self-tests and benchmarks. In particular, the *Teapot in a stadium* benchmark and problem concern the difficulty of rendering a scene with drastically different geometrical density and scale data in various parts of the scene.

Appearances

With the advent first of computer generated short films, and then of full length feature films, it has become something of an in joke to hide a Utah teapot somewhere in one of the film's scenes. Utah teapots can be found in *Toy Story*, *Monsters Inc.*, and Disney's *Beauty and the Beast*, as well as in *The Simpsons*. It is also featured in one of the levels of the video game *Super Monkey Ball 2*, in technological demo section of *Serious Sam*, and can be found in *Microsoft Train Simulator*. By using a cheat code it is possible to have a Utah teapot as an avatar in the *Star Wars: Knights of the Old Republic* PC game. In computers, the Utah teapot sometimes appears in the Pipes screensaver shipped with Microsoft Windows.

One famous ray-traced image (by Jim Arvo and Dave Kirk, from their 1987 Sig-Graph paper 'Fast Ray Tracing by Ray Classification.') shows six stone columns five of which are surmounted by the platonic solids (tetrahedron, cube, octahedron, dodecahedron, icosahedron) - and the sixth column has a teapot. The image is titled "The Six Platonic Solids" - which has lead some people to call the teapot a "Teapotahedron". This image appeared in on the covers of several books and journals.

Jim Blinn (in one of his "Project Mathematics!" videos) proves an interesting version of the Pythagorean theorem: Construct a (2D) teapot on each side of a right triangle and the area of the teapot on the hypotenuse is equal to sum of the areas of the teapots on the other two sides.

See also

- Virtual model
- Trojan room coffee pot
- Stanford Bunny
- Lenna
- →Cornell Box

External links

- Direct link to the image of Utah teapot at Computer History Museum³⁶⁹
- A brief history of the Utah teapot³⁷⁰
- Original data set (tgz)³⁷¹
- Interactive Java rendering of the teapot³⁷²

Source: http://en.wikipedia.org/wiki/Utah_teapot

Principal Authors: Finlay McWalter, SteveBaker, Tregoweth, Mikkalai, Gargaj, Dysprosia, 5994995, Flamurai

UV mapping

UV mapping is a 3D modelling process of making a 2D map representing a 3D model. This map is associated with an image known as a texture. In contrast to "X", "Y" and "Z", which are the coordinates for the original 3D object in the modelling space, "U" and "V" are the coordinates of this transformed map of the surface. Then the image is back-transformed ("wrapped") onto the surface of the 3D object. For example, it is used to give an animated character a realistic looking face, but various other applications are possible.

³⁶⁹ <http://archive.computerhistory.org/resources/still-image/Teapot/src/102630883.jpg>

³⁷⁰ <http://www.sjbaker.org/teapot/>

³⁷¹ <http://www.sjbaker.org/teapot/teaset.tgz>

³⁷² <http://mrl.nyu.edu/~perlin/experiments/teapot/>

External links

- LSCM Mapping image³⁷³ with Blender

Source: http://en.wikipedia.org/wiki/UV_mapping

Vertex

Vertex can have a number of meanings, dependent on the context.

- In geometry, a vertex (Latin: corner; plural **vertices**) is a corner of a polygon (where two sides meet) or of a polyhedron (where three or more faces and edges meet).
- In graph theory, a *graph* describes a set of connections between objects. Each object is called a node or vertex. The connections themselves are called *edges* or *arcs*.
- In optics, a vertex (or surface vertex) is the point where the surface of an optical element crosses the optical axis. It is one of the cardinal points of an optical system.
- In nuclear and particle physics, a vertex is the interaction point, where some subnuclear process occurs, changing the number and/or momenta of interacting particles or nuclei.
- In →3D computer graphics, a vertex is a point in 3D space with a particular location, usually given in terms of its x, y, and z coordinates. It is one of the fundamental structures in polygonal modeling: two vertices, taken together, can be used to define the endpoints of a line; three vertices can be used to define a planar triangle. Vertices are commonly confused with vectors because a vertex can be described as a vector from a coordinate system's origin. They are, however, two completely different things.
- In anatomy, the vertex is the highest point of the skull in the anatomical position (i.e. standing upright). It lies between the parietal bones in the median sagittal plane.

³⁷³ http://de.wikibooks.org/wiki/Bild:Blender3D_LSCM.png

- In astrology, the vertex is a point in space seen near the Eastern horizon (and the Ascendant) in the two-dimensional horoscope. In astrological theory, planet(s) close to it are supposed to lend their personality to the event(s) that the chart captures.
- Vertex is also an album by Buck 65.

In addition, there are several companies named "Vertex".

- Vertex Software develops web-based customer management solutions.
- Vertex Pharmaceuticals is a biotech company, with 2005 revenues of \$161 million, that is developing small molecule therapeutics for the treatment of HCV infection, inflammation, autoimmune diseases, cancer, pain, cystic fibrosis and other diseases, both independently and in collaboration with other pharmaceutical and biotechnology companies.

Source: <http://en.wikipedia.org/wiki/Vertex>

Principal Authors: Wapcaplet, Superborsuk, Ash211, AxelBoldt, Patrick, Nmg20, Mikkalai, V1adislav, Melaen

Viewing frustum

In →3D computer graphics, the **viewing frustum** or **view frustum** is the region of space in the modeled world that may appear on the screen; it is the field of view of the notional camera. The exact shape of this region varies depending on what kind of camera lens is being simulated, but typically it is a frustum of a rectangular pyramid. The planes that cut the frustum perpendicular to the viewing direction are called the *near plane* and the *far plane*. Objects closer to the camera than the near plane or beyond the far plane are not drawn. Often, the far plane is placed infinitely far away from the camera so all objects within the frustum are drawn regardless of their distance from the camera.

Viewing frustum culling or **view frustum culling** is the process of removing objects that lie completely outside the viewing frustum from the rendering process. Rendering these objects would be a waste of time since they are not directly visible. In ray tracing, viewing frustum culling cannot be performed because objects outside the viewing frustum may be visible when reflected off an object inside the frustum. To make culling fast, it is usually done using bounding volumes surrounding the objects rather than the objects themselves.

Source: http://en.wikipedia.org/wiki/Viewing_frustum

Principal Authors: Gdr, Poccil, Eep², Flamura, Dpv

Volume rendering

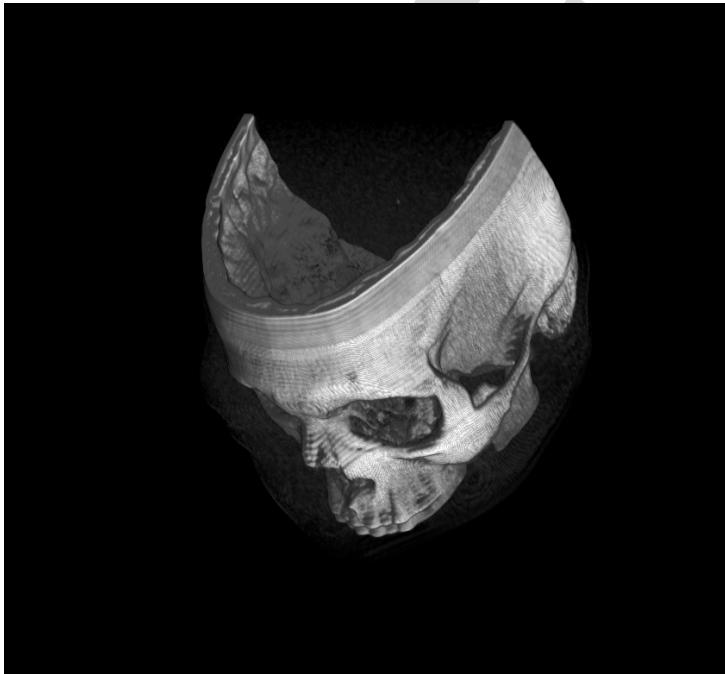


Figure 94 A volume rendered cadaver head using view-aligned texture mapping and diffuse reflection

Volume rendering is a technique used to display a 2D projection of a 3D discretely sampled data set.

A typical 3D data set is a group of 2D slice images acquired by a CT or MRI scanner. Usually these are acquired in a regular pattern (e.g., one slice every millimeter) and usually have a regular number of image pixels in a regular pattern. This is an example of a regular volumetric grid, with each volume element, or voxel represented by a single value that is obtained by sampling the immediate area surrounding the voxel.

To render a 2D projection of the 3D data set, one first needs to define a camera in space relative to the volume. Also, one needs to define the opacity and color of every voxel. This is usually defined using an **RGBA** (for red, green, blue, alpha) transfer function that defines the RGBA value for every possible voxel value.



Figure 95 Volume rendered CT scan of a forearm with different colour schemes for muscle, fat, bone, and blood.

A volume may be viewed by extracting surfaces of equal values from the volume and rendering them as polygonal meshes or by rendering the volume directly as a block of data. The Marching Cubes algorithm is a common technique for extracting a surface from volume data. Direct volume rendering is a computationally intensive task that may be performed in a several ways.

Direct Volume Rendering

A direct volume renderer requires every sample value has to be mapped to opacity and a color. This is done with a “transfer function” which can be a simple ramp, a piecewise linear function or an arbitrary table. Once converted to an **RGBA** (for red, green, blue, alpha) value, the composed RGBA result is projected on correspondent pixel of the frame buffer. The way this is done depends on the rendering technique.

A combination of these techniques is possible. For instance, a shear warp implementation could use texturing hardware to draw the aligned slices in the off-screen buffer.

Volume Ray Casting

Main article: Volume ray casting.

The simplest way to project the image is to cast rays through the volume using ray casting. In this technique, a ray is generated for each desired image pixel. Using a simple camera model, the ray starts at the center of the projection of the camera (usually the eye point) and passes through the image pixel on the imaginary image plane floating in between the camera and the volume to be rendered. The ray is clipped by the boundaries of the volume to save time then the ray is sampled at regular intervals throughout the volume. The data is interpolated at each sample point, the transfer function applied to form an RGBA sample, the sample is composited onto the accumulated RGBA of the ray, and the process repeated until the ray exits the volume. The RGBA color is converted to an RGB color and deposited in the corresponding image pixel. The process is repeated for every pixel on the screen to form the completed image. The examples of high quality ray casting volume rendering can be seen on ³⁷⁴.

Splatting

This is a technique which trades quality for speed. Here, every volume element is splatted (like snow balls) on to the viewing surface from in back to front order. These splats are rendered as disks whose properties (color and transparency) vary diametrically in normal (Gaussian) manner. Flat disks and those with other kinds of property distribution are also used depending on the application.

Shear Warp

A new approach to volume rendering was developed by Philippe Lacroute and Marc Levoy and described in the paper "Fast Volume Rendering Using a Shear-Warp Factorization of the Viewing Transformation" ³⁷⁵. In this technique, the viewing transformation is transformed such that the nearest face of the volume becomes axis aligned with an off-screen image buffer with a fixed scale of voxels to pixels. The volume is then rendered into this buffer using the far more favourable memory alignment and fixed scaling and blending factors. Once all

³⁷⁴ <http://www.fovia.com/>

³⁷⁵ <http://graphics.stanford.edu/papers/shear/>

slices of the volume have been rendered, the buffer is then warped into the desired orientation and scale in the displayed image.

This technique is relatively fast at the cost of less accurate sampling and potentially worse image quality compared to ray casting.

Texture Mapping

Many 3D graphics systems use texture mapping to apply images, or textures, to geometric objects. Commodity PC graphics cards are fast at texturing and can efficiently render slices of a 3D volume, with realtime interaction capabilities.

These slices can either be aligned with the volume and rendered at an angle to the viewer, or aligned with the viewing plane and sampled from unaligned slices through the volume. Graphics hardware support for 3D textures is needed for the second technique.

Volume aligned texturing produces images of reasonable quality, though there is often a noticeable transition when the volume is rotated. View aligned texturing creates images of similar high quality to those of ray casting, and indeed the sampling pattern is identical.

Hardware-Accelerated Volume Rendering

A recently exploited technique to accelerate rendering is the use of modern graphics cards to accelerate traditional volume rendering algorithms such as ray-casting. Starting with the programmable pixel shaders that appeared around the year 2000(?), people recognized the power of parallel operations on multiple pixels and began to perform general purpose computations on the graphics chip. The pixel shaders, once called "register-combiners" were able to read and write randomly from texture memory, perform some basic mathematical and logical calculations. These SIMD processors, now called GPUs, were harnessed to perform general calculations such as ray tracing polygons and signal processing. With OpenGL version 2.0, the pixel shaders now are able to function as MIMD processors (now able to independently branch) with as many as 48 parallel processing elements utilizing up to 1GB of texture memory and high bit depth numerical formats. With such power, virtually any algorithm such as volume ray casting or CT reconstruction can be performed with tremendous acceleration.

Optimization Techniques

Empty Space Skipping

Often, a volume rendering system will have a system for identifying regions of the volume containing no visible material. This information can be used to avoid rendering these transparent regions.

Early Ray Termination

This is a technique used when the volume is rendered in front to back order. For a ray through a pixel, once sufficient dense material has been encountered, further samples will make no significant contribution to the pixel and so may be ignored.

Octree and BSP space subdivision

The use of hierarchical structures such as octree and BSP-tree could be very helpful for both compression of volume data and speed optimization of volumetric ray casting process.

Volume Segmentation

By sectioning out large portions of the volume that one considers uninteresting before rendering, the amount of calculations that have to be made by ray casting or texture blending can be significantly reduced

Sources

- R. A. Drebin, L. Carpenter, P. Hanrahan: *Volume Rendering*. 1988

Source: http://en.wikipedia.org/wiki/Volume_rendering

Principal Authors: Sjschen, Andrewmu, Thetawave, Anilkny, Ctachme

Volumetric lighting

Volumetric lighting is a technique used in →3D computer graphics to add lighting to a rendered scene. The term seems to have been introduced from cinematography and is now widely applied to 3D modelling and rendering especially in the field of computer and video games. Basically it allows the viewer to see the beams of a light source shining through the environment; seeing sunbeams streaming through an open window can be considered an example of volumetric lighting.

In volumetric lighting, the light cone emitted by a light source is modeled as a more or less transparent object and considered as a container of a "volume": as a result, light has the capability to give the effect of passing through an actual three dimensional medium (such as fog, dust, smoke or steam) that is inside its volume, just like in the real world.

References

- Volumetric lighting tutorial at Art Head Start³⁷⁶
- 3D graphics terms dictionary at Tweak3D.net³⁷⁷

Source: http://en.wikipedia.org/wiki/Volumetric_lighting

Voxel

A **voxel** (a portmanteau of the words *volumetric* and *pixel*) is a volume element, representing a value in three dimensional space. This is analogous to a pixel, which represents 2D image data. Voxels are frequently used in the visualisation and analysis of medical and scientific data. Some true 3D displays use voxels to describe their resolution. For example, a display might be able to show $512 \times 512 \times 512$ voxels.

As with pixels, voxels themselves typically do not contain their position in space (their coordinates) - but rather, it is inferred based on their position relative to

³⁷⁶ <http://www.art-head-start.com/tutorial-volumetric.html>

³⁷⁷ <http://www.tweak3d.net/3ddictionary/>

other voxels (i.e. their position in the data structure that makes up a single volume image).

Voxel data

A voxel represents the sub-volume box with constant scalar/vector value inside which is equal to scalar/vector value of the corresponding grid/pixel of the original discrete representation of the volumetric data. The boundaries of a voxel are exactly in the middle between neighboring grids. Thus, the notion of “Voxel” is applicable only for nearest neighbor interpolation and it is not applicable for higher order of interpolation such as tri-linear, tri-cubic... etc; these cases can be represented through Cell volume subdivision.

The value of a voxel may represent various properties. In CT scans, the values are Hounsfield units, giving the opacity of material to X-rays. Different types of value are acquired from MRI or ultrasound.

Voxels can contain multiple scalar values what essentially is a vector data; in the case of ultrasound scans with B-mode and Doppler data, density, and volumetric flow rate are captured as separate channels of data relating to the same voxel positions.

Other values may be useful for immediate 3D rendering, such as a surface normal vector and color.

Uses

Visualization

A volume containing voxels can be visualised either by direct volume rendering or by the extraction of polygon iso-surfaces which follow the contours of given threshold values. The marching cubes algorithm is often used for iso-surface extraction, however other methods exist as well.

Computer gaming

- Many NovaLogic games have used voxel-based rendering technology, including the *Delta Force* series.
- Westwood Studios *Command & Conquer: Tiberian* series engine games used voxels to render the vehicles.
- Similarly *Total Annihilation* used voxels to render its vehicles.

Trivia

In the minimalist webcomic *Pixel*, in which pixels inside a computer are the main characters, one 'race' of supporting characters are the voxels, who have the "supernatural" power of moving in three dimensions.

See also

- →Volume rendering

External links

- Volex³⁷⁸, a volumetric display using LED
- Voxel3D, voxel based modeling software³⁷⁹
- Voxlap, an open source voxel engine written by Ken Silverman³⁸⁰
- HVox, another voxel-based terrain engine³⁸¹
- Iehovah, a volume based surface generation library for real-time visualization³⁸²
- Geek, a voxel terrain engine that uses perlin noise to create natural looking geometry³⁸³
- Cavernosa, a terrain/cave sculpturing tool based on a hierarchical binary voxel grid³⁸⁴
- A tutorial, that explains how to draw a voxel terrain with code in C++ such as Commanche/Outcast³⁸⁵

Source: <http://en.wikipedia.org/wiki/Voxel>

Principal Authors: Andrewmu, Pythagoras1, Maestrosync, Xezbeth, Stefanbanev, Omegatron, Retodon8, RJHall, Karl-Henner, Wlievens

³⁷⁸ <http://www.ucsi.edu.my/research/projects.html>
³⁷⁹ <http://www.everygraph.com/frame.php?contents=product&name=voxel3d>
³⁸⁰ <http://advsys.net/ken/voxlap.htm>
³⁸¹ <http://www.p0werup.de/>
³⁸² <http://www.home.zonnet.nl/petervenis>
³⁸³ <http://www.flipcode.org/cgi-bin/fcarticles.cgi?show=62853>
³⁸⁴ <http://www.btinternet.com/%7Eahcox/Cavernosa/index.html>
³⁸⁵ <http://www.massal.net/article/voxel/>

W-buffering

In computer graphics, **w-buffering** is a technique for managing image depth coordinates in three-dimensional (3-D) graphics which is alternative to z-buffering. It is used in cases where z-buffering produces artifacts. W-buffering does a much better job of quantizing the depth buffer. W-buffering provides a linear representation of distance in the depth buffer, whereas z-buffering is nonlinear and allocates more bits for surfaces that are close to the eyepoint and less bits for those farther away.

Source: <http://en.wikipedia.org/wiki/W-buffering>

Principal Authors: Lupin, Nabla, Interiot

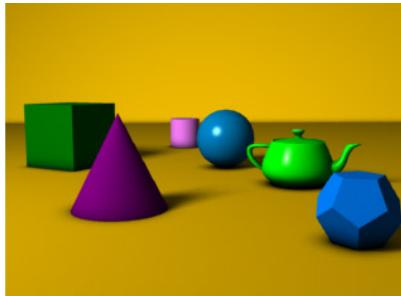
Z-buffering

In computer graphics, **z-buffering** is the management of image depth coordinates in three-dimensional (3-D) graphics, usually done in hardware, sometimes in software. It is one solution to the visibility problem, which is the problem of deciding which elements of a rendered scene are visible, and which are hidden. The painter's algorithm is another common solution which, though less efficient, can also handle non-opaque scene elements.

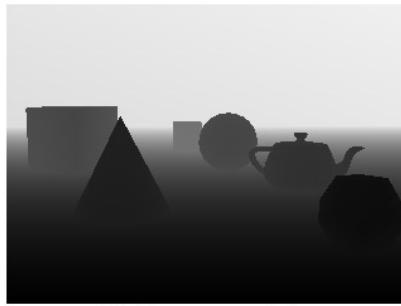
When an object is rendered by a 3D graphics card, the depth of a generated pixel (z coordinate) is stored in a buffer (the **z-buffer**). This buffer is usually arranged as a two-dimensional array (x-y) with one element for each screen pixel. If another object of the scene must be rendered in the same pixel, the graphics card compares the two depths and chooses the one closer to the observer. The chosen depth is then saved to the z-buffer, replacing the old one. In the end, the z-buffer will allow the graphics card to correctly reproduce the usual depth perception: a close object hides a farther one.

The granularity of a z-buffer has a great influence on the scene quality: a 16-bit z-buffer can result in artifacts (called "z-buffer fighting") when two objects are very close to each other. A 24-bit or 32-bit z-buffer behaves much better. An 8-bit z-buffer is almost never used since it has too little precision.

Additionally, precision in the z-buffer distance values is not spread evenly over distance. Nearer values are much more precise (and hence can display closer objects more properly) than values which are farther away. Generally, this



A simple three dimensional scene



Z-buffer representation

Figure 96 Z-buffer data

is desirable, but sometimes it will cause artifacts to appear as objects become more distant. A variation on z-buffering which results in more evenly distributed precision is called w-buffering.

At the start of a new scene, the z-buffer must be cleared to a defined value, usually 1.0, because this value is the upper limit (on a scale of 0 to 1) of depth, meaning that no object is present at this point through the viewing frustum.

The invention of the z-buffer concept is most often attributed to Edwin Catmull. Actually, also Wolfgang Straßer described this idea in his 1974 Ph.D. thesis¹.

On recent PC graphics cards (1999-2005), z-buffer management uses a significant chunk of the available memory bandwidth. Various methods have been employed to reduce the impact of z-buffer, such as lossless compression (computer resources to compress/decompress are cheaper than bandwidth) and ultra fast hardware z-clear that makes obsolete the "one frame positive, one frame negative" trick (skipping inter-frame clear altogether using signed numbers to cleverly check depths).

Mathematics

The range of depth values in camera space (See →3D projection) to be rendered is often defined between a *near* and *far* value of *z*. After a perspective transformation, the new value of *z*, or *z'*, is defined by:

$$z' = \frac{far+near}{far-near} + \frac{1}{z} \left(\frac{-2 \cdot far \cdot near}{far-near} \right)$$

Where *z* is the old value of *z* in camera space, and is sometimes called *w* or *w'*.

The resulting values of *z'* are normalized between the values of -1 and 1, where the *near* plane is at -1 and the *far* plane is at 1. Values outside of this range correspond to points which are not in the viewing frustum, and shouldn't be rendered.

To implement a z-buffer, the values of *z'* are linearly interpolated across screen space between the vertices of the current polygon, and these intermediate values are generally stored in the z-buffer in fixed point format. The values of *z'* are grouped much more densely near the *near* plane, and much more sparsely farther away, resulting in better precision closer to the camera. The closer the *near* plane is set to the camera, the less precision there is far away – having the *near* plane set too closely is a common cause of undesirable rendering artifacts in more distant objects.

To implement a w-buffer, the old values of *z* in camera space, or *w*, are stored in the buffer, generally in floating point format. However, these values cannot be linearly interpolated across screen space from the vertices – they usually have to be inverted, interpolated, and then inverted again. The resulting values of *w*, as opposed to *z'*, are spaced evenly between *near* and *far*.

Whether a z-buffer or w-buffer results in a better image depends on the application.

When using the <canvas> tag with JavaScript, you can use the following as a Z-Buffer formula.

```
//F=far, N=near, Z=final return
f=0;
n=0;
z=0;
z=((((f+n)/(f-n)))+((1/z)*((-2*f*n)/(f-n))));
```

See also

- Edwin Catmull – inventor of the z-buffer concept.
- →3D computer graphics
- →Irregular Z-buffer
- Z-order

External links

- Learning to Love your Z-buffer³⁸⁶
- Alpha-blending and the Z-buffer³⁸⁷

Notes

Note 1: see W.K. Giloi, J.L. Encarnaçāo, W. Straßer. "The Giloi's School of Computer Graphics". Computer Graphics 35 4:12–16.

->

Source: <http://en.wikipedia.org/wiki/Z-buffering>

Principal Authors: Penguin42, ToohrVyk, Zotel, Alfio, Solkoll, Zoicon5, Bookandcoffee

Z-fighting

Z-fighting is a phenomenon in 3D rendering that occurs when two or more coplanar primitives have similar values in the Z-buffer, causing random parts of the primitives to be rendered. This problem is usually caused by floating point round-off errors. Z-fighting is reduced by the use of a higher resolution depth buffer, by →W-buffering, or by simply moving the polygons further apart.

It is a relatively rare occurrence to encounter z-fighting while using a 24-bit Z-buffer. For example, with a 16-bit Z-buffer, at 10,000 units, resolving accuracy is only 1,800 units, meaning a point at 10,000 units away from the camera is given the same Z-buffer value as a point 11,799 units away. For comparison, resolving accuracy for a 24-bit Z-buffer is 6 units at the same distance.

As virtual world size increases, a greater likelihood exists that you will encounter Z-fighting between primitives.

³⁸⁶ http://www.sjbaker.org/steve/omniv/love_your_z_buffer.html

³⁸⁷ http://www.sjbaker.org/steve/omniv/alpha_sorting.html

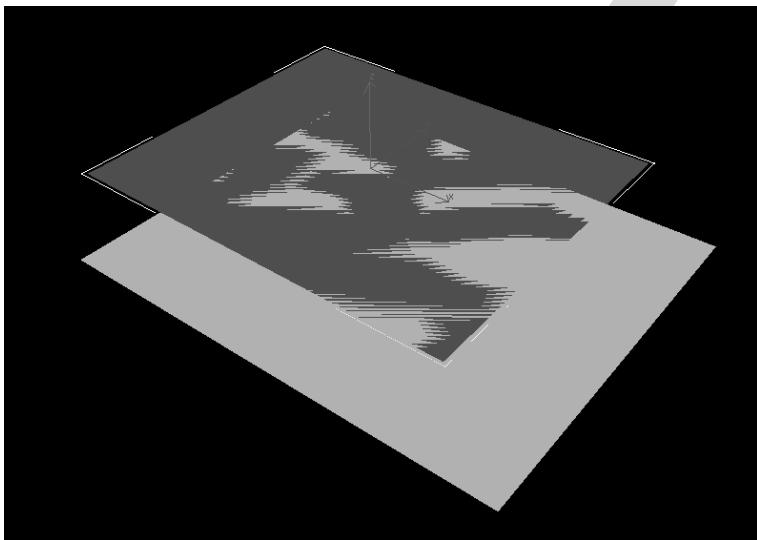


Figure 97 Coplanar polygons

Source: <http://en.wikipedia.org/wiki/Z-fighting>

Principal Authors: Chentianran, Mhoskins, Rbrwr, Reedbeta, RJHall

GNU Free Documentation License

Version 1.2, November 2002

Copyright (C) 2000,2001,2002 Free Software Foundation, Inc. 51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

— 0. PREAMBLE —

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondly, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

— 1. APPLICABILITY AND DEFINITIONS —

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as

such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

— 2. VERBATIM COPYING —

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

— 3. COPYING IN QUANTITY —

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

— 4. MODIFICATIONS —

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
- B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.
- C. State on the Title page the name of the publisher of the Modified Version, as the publisher.
- D. Preserve all the copyright notices of the Document.
- E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
- F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

- G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
- H. Include an unaltered copy of this License.
- I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
- J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
- K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
- L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
- M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.
- N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.
- O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

— 5. COMBINING DOCUMENTS —

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements."

— 6. COLLECTIONS OF DOCUMENTS —

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

— 7. AGGREGATION WITH INDEPENDENT WORKS —

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

— 8. TRANSLATION —

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

— 9. TERMINATION —

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

— 10. FUTURE REVISIONS OF THIS LICENSE —

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See <http://www.gnu.org/copyleft/>.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

List of Figures

Please note that images have various different licenses and usage constraints.

License details for a specific image can be found at:

<http://en.wikipedia.org/wiki/Image:FILENAME.JPG>

For each image we list the filename and license.

Note that license names are abbreviated like GFDL, PD, fairuse, etc. Please have a look at the corresponding page at Wikipedia for further information about the images and details of the associated licenses.

Engine_movingparts.jpg, GFDL	3
3D_Plus_3DBuilding.jpg, cc-by-sa-2.5	4
Raytraced_image_jawray.jpg, Attribution	6
Perspective_Projection_Principle.jpg, Convert to SVG Math	8
Cel_shading_OGRE3D.jpg, NoRightsReserved	9
Modeler83.png, software-screenshot	12
Modeler83.png, software-screenshot	14
Aooclude_insectambient.jpg, gfdl	23
Aooclude_insectdiffuse.jpg, gfdl	23
Aooclude_insectcombined.jpg, gfdl	24
Aooclude_insectambient.jpg, gfdl	24
Aooclude_insectdiffuse.jpg, gfdl	25
Aooclude_insectcombined.jpg, gfdl	25
Aooclude_insectambient.jpg, gfdl	26
Aooclude_insectdiffuse.jpg, gfdl	26
Aooclude_insectcombined.jpg, gfdl	27
Aooclude_hemisphere.png, GFDL-presumed	27
Aooclude_bentnormal.png, GFDL-presumed	28
Pixel_interpolation.png, PD-user Dcoetzee	28
Bloom38fx.jpg, game-screenshot	33
BoundingBox.jpg, cc-by-2.0	34
Bump-map-demo-smooth.png, GFDL	40
Bump-map-demo-bumpy.png, GFDL	40
Catmull_Clark_subdivision_of_a_cube.png, GFDL	43
Toon-shader.jpg, cc-by-2.0	45
Zelda_Wind_Waker_Deku_Leaf_float.jpg, Nintendo-screenshot	47
Spidey_animated.jpg, film-screenshot	50
Tron_Lightcycles.jpg, film-screenshot	67
Flight_dynamics.jpg, NowCommons Image:Flight_dynamics.png	70
Cornell_box.png, PD	72
Displacement.jpg, cc-by-2.0	83
CGfog.jpg, cc-by-2.0	85
Phong-shading-sample.jpg, CopyrightedFreeUse	87
Global_illumination.jpg, GFDL-presumed Longhair	96
Gouraud_shading.png, cc-by-2.5	103
Obj_lineremoval.png, GFDL	108
Old_saint_pauls_1.jpg, self2 GFDL cc-by-sa-2.5,2.0,1.0	111
HDRI-Example.jpg, self2 GFDL cc-by-2.5	112

Old saint pauls 2.jpg, FeaturedPicture	113
Grand Canyon HDR imaging.jpg, self2 GFDL cc-by-2.5	114
Farcryhdr.jpg, game-screenshot	116
Farcrynohdr.jpg, game-screenshot	117
Hl2hdrcomparison.jpg, game-screenshot	118
Dodsourceinside1.jpg, game-screenshot	120
Dodsourceinside2.jpg, game-screenshot	121
Dodsourceoutside1.jpg, game-screenshot	121
Dodsourceoutside2.jpg, game-screenshot	122
Isosurface_on_molecule.jpg, CopyrightedFreeUseProvided	135
LambertCosineLaw1.png, PD-self	139
LambertCosineLaw2.png, PD-self	140
Metaballs.gif, GFDL	146
MotionCapture.jpg, GFDL-self	153
Painters_problem.png, GFDL	163
Normal map example.png, cc-by-sa	164
OpenGL_Logo.jpg, logo Computer software logos	168
Painter_s_algorithm.png, NowCommonsThis	187
Painters_problem.png, GFDL	187
Fear-pmapping1.jpg, game-screenshot	189
Fear-pmapping2.jpg, game-screenshot	189
Fear-pmapping3.jpg, game-screenshot	190
particle_sys_fire.jpg	191
particle_sys_galaxy.jpg	192
particle_sys_trail.jpg	192
Phong-shading-sample.jpg, CopyrightedFreeUse	201
Caustics.jpg, cc-by-2.0	202
Procedural Texture.jpg, GFDL	219
Radiosity.png, PD-user Wapcaplet	246
Raytraced_image_jawray.jpg, Attribution	251
Raytracing reflection.png, PD-self	252
Spoon_fi.jpg, GFDL-self	263
cube_mapped_reflection_example.jpg	264
Cube_mapped_reflection_example_2.JPG, PD-self	265
7fin.png, GFDL-self	302
3noshadow.png, GFDL-self	303
1light.png, GFDL-self	304
2shadowmap.png, GFDL-self	305
4overmap.png, GFDL-self	306
5failed.png, GFDL-self	307
6black.png, GFDL-self	307
7fin.png, GFDL-self	308
Cad-fm01s.gif, GFDL-self	315
Catmull-Clark_subdivision_of_a_cube.png, GFDL	325
Scattering-example.jpg, cc-by-2.0	328
SurfaceNormalDrawing.PNG, PD-ineligible	330
TextureMapping.png, cc-by-2.0	334
Doom3Marine.jpg, game-screenshot	337
Quake3Screenshot.jpg, game-screenshot	338
utah_teapot.png	339
Melitta teapot.png, PD	340

CTSkullImage.png, PD-self
CTWristImage.png, PD-self
Z-buffer.jpg, cc-by-2.0
Z-fighting.png, PD-self

345
346
354
357

Index

- .kkrieger 215
 16-bit 353
 19th century 45
 2006-05-16 42
 20th century 46
 24-bit 353, 356
 27 october 296
 2d computer graphics 2, 168
 3-sphere 245
 32-bit 353
 3d animation 77, 136
 3dc 282, 284
 3d computer graphics 11, 22, 22, 42, 51, 67,
 73, 74, 76, 85, 87, 88, 95, 109, 131, 148,
 149, 152, 163, 164, 168, 187, 198, 200,
 206, 211, 222, 245, 252, 252, 266, 279,
 281, 284, 292, 302, 309, 317, 322, 327,
 331, 343, 344, 350, 356
 3d display 350
 3dfx 65, 94
 3d geometric model 91
 3d geometry 105
 3d graphics 41, 94, 281
 3d graphics card 353
 3dlabs 169, 176
 3d model 10, 132, 197
 3d modeler 10, 14, 15
 3d modeling 38
 3d modelling 342
 3d motion controller 11
 3d projection 10, 355
 3d rendering 3
 3d scanner 208
 3d studio 5
 3d studio max 12, 56, 158, 194, 209, 245,
 341
 4d 240
 6dof 156
 6 march 294
 8-bit 353
 ōban star-racers 50
 ōkami 48
- a**
- aabb 289
 absorption 248
 acceleration 153
 acis 314
 acm press 104
 acm transactions on graphics 147, 194, 212
- acquisition 92
 activision 166
 actor 331
 actual state 259
 adobe illustrator 286
 afterburn 194
 aki ross 331
 algorithm 15, 26, 67, 91, 92, 95, 148, 188,
 218, 245, 281, 284
 algorithmic 79
 algorithms 35
 aliasing 27, 67, 272
 almost everywhere 330
 alpha blending 59, 170
 alpha channel 283
 ambient occlusion 3, 10, 130
 anaglyph image 10
 analysis 92
 anatomical position 343
 anatomy 343
 angle 153, 241
 anim8or 13
 animation 11, 129, 131, 132, 161, 312, 316
 animation#styles and techniques of animation
 51
 animation control 132
 animator 132
 anime 46
 anisotropic 150
 anisotropic filtering 31, 333
 anthony a. apodaca 148
 anti-aliasing 134, 151, 255
 antialiasing 272, 333
 api 145, 178, 183, 186, 322
 apple computer 169
 appleseed 48
 applied mathematics 92
 arcade game 86
 architecture review board 62
 array 287, 293
 arthur appel 254
 artificial intelligence 74, 74
 artist 132
 art of illusion 13, 44
 ascendant 344
 assembly modelling 314
 associativity 241
 astrology 344
 ati technologies 64, 95, 169
 atomic betty 49

august 23 124
 august 9 124
 autocad 209, 286, 317, 341
 autodesk 11, 12
 autodesk, inc. 13
 auto modellista 48
 autostereogram 11
 aux 172
 avionics 181

b

b-spline 326, 326, 326
 bandwidth 354
 barycentric coordinates (mathematics) 211
 batch 293
 bbc 77
 beam tracing 67, 96, 259
 beos 341
 beyblade: the movie - fierce battle 48
 bi-directional path tracing 147, 195
 bidirectional reflectance distribution function 274
 bilinear filtering 333
 bilinear interpolation 29, 335
 binary space partitioning 250
 binocular 252
 biomedical computing 92
 birch 215
 bit 282
 bitmap 28
 bitmap textures 267
 black body 138
 blinn-phong shading model 104, 202
 blinn-phong shading 318
 bloom (shader effect) 122
 bmr_t 90
 bomberman generation 48
 bomberman jetters 48
 bone 135
 boolean 68
 boundary representation 87, 312, 316
 bounding sphere 36
 bounding volume 344
 bounding volume hierarchies 35
 box modeling 207
 box splines 326
 brain 131
 branch 300
 brazil r/s 7
 bresenham's line algorithm 211, 335
 brian paul 145
 brightness 33, 321
 brl-cad 69, 204, 259, 317

buck 65 344
 buffer 353
 bugs bunny 77
 bui tuong phong 198, 200, 200, 200, 202
 bump mapping 1, 9, 15, 80, 83, 84, 143, 164, 166, 188, 190, 197, 267, 296, 335
 bumpmapping 217
 butterfly 326
 byte 321

c

c + 173, 178, 183, 286, 288
 c4 engine 56
 cache 151
 cache coherency 269
 cache memory 285
 cad 8, 68, 69, 79, 222, 316
 calculation 131
 calculus 275
 call of duty 2 166
 caltech 93
 cam 8
 canada's worst driver 49
 carmack's reverse 310, 322
 cartoon 45
 cass everitt 322
 casting 312
 cat 28
 catia 317
 caustics 7
 cel damage 47, 48
 celluloid 46
 cellulose acetate 46
 cgl 174
 cg programming language 56
 change state 87
 character animation 51
 charts on so(3) 244
 chemistry 136
 chirality 226
 chrominance 112
 cinema 4d 13
 cinematography 111, 350
 circle 36, 91, 140
 circumference 5
 clanlib 205
 class of the titans 49
 clifford algebra 244
 clip mapping 144
 clipping (computer graphics) 106
 collinear 240, 241
 collision detection 35

- color 76, 91, 351
 color buffer 321
 comic book 45
 comm. acm 39
 command & conquer: tiberian series 351
 comparison of direct3d and opengl 80, 176
 compiler 217
 compiz 63
 compositing 8, 279
 computation 247
 computational fluid dynamics 135
 computational geometry 34, 91, 212, 316
 computed axial tomography 316
 computer 14, 15, 258
 computer-aided design 91, 92, 93
 computer-generated imagery 11, 78, 143, 279
 computer and video games 15, 109, 141,
 152, 257, 266, 350
 computer animation 51, 76, 131, 152, 158,
 210
 computer cluster 298
 computer display 119
 computer game 10, 15, 149, 243, 304, 309
 computer games 33
 computer generated image 218
 computer graphics 16, 22, 25, 33, 34, 39,
 40, 45, 54, 75, 76, 85, 88, 89, 91, 92, 100,
 103, 105, 111, 115, 135, 137, 141, 146,
 196, 197, 203, 210, 213, 213, 244, 248,
 262, 312, 316, 321, 325, 332, 333, 338,
 353, 353
 computer history museum 340
 computer program 292
 computer programming 54, 100, 141
 computer science 92
 computer software 58
 concave 338
 cone tracing 27, 96, 259
 consortium 180
 constraints 136
 constructive solid geometry 4, 87, 313
 continuous function 272
 contrast ratio 33
 conversion between quaternions and euler an-
 gles 244
 convex 250
 convex hull 37
 coordinate 5, 350
 coordinate rotation 244
 coordinates 339, 342
 coreldraw 286
 cornell box 342
 cornell university 71, 245
 cosine 138, 242
 cosmo 3d 179, 185
 covering map 245
 c programming language 52, 59, 97, 99, 99
 cpu 215
 crazyracing kartrider 48
 creative labs 42, 322
 cross-platform 168, 182
 cross platform 98
 cross product 329
 crowd psychology 74
 crystal space 54
 crytek 166
 cube 243, 341
 cube map 75, 265
 cube mapping 264, 266
 cuboid 36, 68
 curve 243
 curvilinear coordinates 330
 cut scene 213
 cylinder 36
- d**
 d.i.c.e. 49
 d3dx 341
 dark cloud 2 47, 48
 darwyn peachey 297
 data structure 134, 286, 287
 datatype 293
 david s. elbert 297
 day of defeat: source 120
 decal 335
 december 24 123
 delilah and julius 49
 demo effect 147
 demo effects 84
 demoscene 214
 density 135, 351
 depth buffer 289, 321, 356
 depth of field 7, 267
 deus ex: invisible war 33, 166
 device driver 175
 diameter 140
 diffraction 257
 diffuse inter-reflection 96
 diffuse interreflection 203
 diffuse reflection 137, 138, 320, 345
 diffusion 292
 digital 2
 digital camera 118
 digital content creation 56, 300
 digital equipment corporation 178
 digital image 91, 266

digital images 112
 digital puppetry 159
 dimension 91
 direct3d 3, 10, 58, 95, 128, 168, 175, 176,
 208, 285, 322, 341
 direct3d vs. opengl 168
 directdraw 80
 directdraw surface 284
 direct illumination 245
 directly proportional 138
 directx 10, 53, 55, 79, 84, 94, 123, 123, 205,
 273, 281, 323
 directx graphics 80
 disney's california adventure 77
 displacement mapping 41, 41, 148, 166, 190
 display generator 92
 distance fog 86
 distributed ray tracing 27, 67, 259
 doctor who 331
 dodecahedron 341
 doo-sabin 326
 doom 250
 doom 3 33, 42, 144, 144, 166, 214, 309,
 322, 336
 doom engine 250
 dot product 137, 199
 dragon ball z: budokai 2 48
 dragon booster 49
 dragon quest viii 48, 49
 duck dodgers 49
 duke nukem 3d 250
 dust 350
 dvd 274
 dxf 209
 dx5 1
 dynamical simulation 36
 dynamic link library 82
 dynamic range 111

e

earth science 15
 edge modeling 38
 educational film 250, 254
 edwin catmull 43, 325, 335, 354, 356
 eidos interactive 166
 elbow 88
 ellipse 286
 elmsford, new york 249, 254
 embedded device 180
 emergent behavior 74
 emission spectrum 71
 emulate 80
 enemy territory: quake wars 144, 144

energy 140
 engineering 92
 engineering drawing 314, 316
 entertainment 92
 environment map 23
 environment mapping 134
 eovia 13
 epic games 166
 epic megagames 116
 eric veach 147
 euclidean geometry 206
 euclidean space 206
 euler angle 243
 euler angles 69, 244, 288
 euler characteristic 87
 evans & sutherland 285
 execute buffer 59
 extrusion 312
 eye 246
 e^3 116

f

f.e.a.r. 189, 190
 f. kenton musgrave 297
 facet 313
 facial motion capture 160
 fad 46
 fahrenheit graphics api 66
 fairly oddparents 49
 family guy 49
 farbrausch 214
 far cry 54, 166
 farcry 117
 fear effect 46
 february 2 173
 field of view 25
 file format 208
 film 73, 89
 final-render 7
 final fantasy: the spirits within 213, 331
 final gathering 130
 finding nemo 77
 finite difference 313
 finite element 268
 finite element analysis 210
 finite state machine 59
 first-person shooter 215
 fisheye lens 263
 fixed-point 211
 flat shading 9, 104
 flight dynamics 70
 flight simulator 149
 floating point 16, 112, 123, 180, 211, 355,

- 356
 fltk 102, 172, 205
 fluid flow 135
 fmv game 213
 fog 299, 350
 formz 13
 fortran 90, 173
 forward kinematic animation 4, 131
 fourcc 282
 fractal 91, 213, 218
 fractal landscape 196, 217
 fraction (mathematics) 211
 fragment 300
 frame buffer 89, 106, 340
 framebuffer 294
 frame rate 209
 frank crow 322
 freeform surface modelling 314
 freeglut 182
 free software 182, 185
 frustum 344, 355
 full motion video 213
 full screen effect 112, 119
 functional programming 214
 funky cops 49
 futurama 48, 49
 fxt1 284
- g**
 g.i. joe: sigma 6 49
 gamecube 123
 game developers conference 218
 game physics 191
 game programmer 42
 gamma correction 112
 gamut 120
 gaussian distribution 319, 319
 geforce 62, 89
 geforce 2 41
 geforce 256 61
 geforce fx 124, 129
 geometric 1
 geometrical optics 252
 geometric primitive 105
 geometric space 91
 geometry 343
 geophysics 136
 ghost in the shell: s.a.c. 2nd gig 48
 ghost in the shell: stand alone complex 48
 gigabytes 215
 glass cockpit 181
 glee 172, 173
 glew 172, 173
- glide api 10, 65
 global illumination 22, 76, 130, 130, 203,
 245, 256, 259, 273
 glossy 76
 glsl 169, 176, 176, 295, 300
 glu 102, 172, 176, 182
 glucose transporter 182
 glut 101, 172, 182
 glut 101, 102, 102, 172, 174, 176, 341
 glx 174
 gnu gpl 185
 gollum 143, 159
 google earth 56, 110
 gouraud shading 9, 32, 32, 88, 201
 gpu 348
 gradient 330
 grand theft auto 3 86
 granite 218
 granularity 296
 graphic art 2
 graphics 11
 graphics card 3, 10, 22, 56, 80, 89, 94, 106,
 124, 208, 268, 285, 299, 332, 336
 graphics cards 124
 graphics hardware 105
 graphics pipeline 32, 89, 109, 169, 176, 277,
 299
 graphics processing unit 52, 97, 216, 299
 graph theory 343
 gravity 73
 grayscale 164
 green 72
 gregory ward 111
 greg ward 116
 group dynamics 74
 gtk 205
 gui 205, 287
 gundam seed 49
 gungrave 49
- h**
 half-life 2 166
 half-life 2: lost coast 116, 118
 half precision 112
 halo 2 141, 166
 hardware 10, 134, 321, 353
 hardware acceleration 79, 336
 harvest moon: save the homeland 49
 heat transfer 96
 heightfield 110, 291
 heightmap 40, 83, 84
 hemicube 247
 henrik wann jensen 203

hewlett-packard 174
 hidden face removal 108
 hidden line removal 79, 109
 high-definition 214
 high dynamic range imaging 116, 130
 high dynamic range rendering 33, 34, 113
 high level shader language 80
 history of 3d graphics 11
 hlsl 295
 homogeneous coordinates 20
 hoops3d 10
 horoscope 344
 hot wheels highway 35 world race 48
 html 39
 hue 246
 human eye 119
 hypotenuse 341

i

i-deas 317
 ian bell 217
 ibm 174, 178, 185
 icosahedron 208, 341
 id software 42, 144, 144, 166, 214, 322, 336
 ieee transactions on computers 104
 ihv 64
 illusion 321
 image 103, 266
 image-based lighting 23
 image compression 281
 image order 248
 image plane 26
 image processing 178
 image resolution 272
 immediate-mode rendering 174
 implementation 300
 implicit surface 4
 in-joke 338
 infinitesimal 244
 injection moulding 312
 innocence: ghost in the shell 48
 input device 173
 integer 123, 321
 integral 247
 intel 169, 178, 185, 218
 interaction point 343
 interactive manipulation 132
 interpolation 326
 invader zim 50
 inverse kinematic animation 4, 88, 132
 inverse kinematics 132, 136, 208
 invisibility 222
 iris 64, 119

iris inventor 175
 iris performer 175
 irradiance 138
 irregular z-buffer 356
 irrlicht 10
 isocontour 135
 isometric projection 19
 isotropic 150
 ivan sutherland 285

j

jade cocoon 217
 jaggies 272
 james d. foley 251
 james h. clark 92
 japan 48
 jar-jar binks 159
 jar jar binks 316
 java 3d 10, 173
 java opengl 176
 javascript 355
 jet set radio 46, 49
 jet set radio future 46
 jim blinn 9, 146, 340
 jim henson 77
 jitendra malik 128
 jogl 173
 johann heinrich lambert 138
 john carmack 42, 59, 144, 310, 322, 329
 joint 136
 jsr 184 10
 july 28 42
 july 29 42

k

kd-tree 134
 ken perlin 196, 297
 kermit the frog 77
 keyframing 5
 khronos group 169, 180
 killer 7 48, 49
 kilobytes 215
 kinematic decoupling 132
 kinematics 132
 kirby: right back at ya! 50
 klonoa 2 47, 49
 kluge 79
 kml 57
 knuckle 88
 kobbelt 326
 kurt akeley 54, 169

1
 l-system 217
 l2 cache 216
 lambert's cosine law 138, 331
 lambertian 165
 lambertian diffuse lighting model 103
 lambertian reflectance 76
 lance williams 151, 302
 larry gritz 148
 laser 258
 latin 149, 343
 lazy evaluation 214
 led 152
 lego exo-force 48, 51
 lenna 73, 73, 342
 lens flare 7
 leonidas j. guibas 147
 level set 135
 lgpl 205
 light 75, 76, 203, 252, 317, 321, 327
 light absorption 292
 lighting 266, 350
 lighting model 295
 lightmap 214
 light probe 263
 light tracing 195
 light transport theory 130
 lightwave 5, 14, 209
 light weight java game library 177
 line-sphere intersection 259
 linear 353
 linear algebra 166, 275
 linear interpolation 29
 linked list 287
 linux 58, 63, 98, 168, 173
 lipschitz continuous 330
 list of 3d artists 11
 list of cel-shaded video games 48, 51
 list of software patents 42
 list of vector graphics markup languages 57
 live & kicking 77
 load balancing 294
 logarithmic 258
 logical operator 53
 logluv tiff 113
 looney tunes: back in action 77
 lord of the rings 143
 lorentz group 240
 lossless compression 354
 lossy data compression 1
 low poly 143, 165
 luminance 112, 137, 138
 luminous energy 140

m
 m. c. escher 87
 m. e. newell 163
 machinima 77, 78
 mac os 58, 168
 mac os x 58, 63, 174
 magic eye 11
 magnetic resonance imaging 316
 manifold 207
 maple 215
 marble 218, 327
 marching cubes 136, 147, 351
 marc levoy 347
 mark kilgard 297, 322
 massive (animation) 74
 match moving 14, 153, 161
 mathematical 131
 mathematical applications group, inc. 249,
 254
 matrox 64
 maxon 13
 maya 56
 medical image processing 91
 medical imaging 135
 megaman nt warrior 50
 mega man x7 49
 mega man x command mission 49
 melitta 340
 mental ray 7, 13, 273
 mesa 3d 173, 176
 mesh 43, 82, 213, 294
 metaball 194
 metal 218
 metal gear acid 2 48, 49
 metal gear solid 3 123
 meteorology 136
 metropolis-hastings algorithm 147
 metropolis light transport 96, 256
 michael abrash 329
 micropolygon 84
 microscope 252
 microsoft 62, 79, 123, 123, 128, 169, 178,
 185, 281, 300
 microsoft corporation 58
 microsoft train simulator 341
 microsoft windows 168, 174, 341
 midedge 326
 milk 327
 milkshape 3d 13
 milling 312
 mimd 348
 minigl 94
 minimax 163

mip-mapping 333
 mipmap 56, 101, 180
 mipmapping 166
 mit license 145
 mobile phone 180
 mode 7 250
 modelview matrix 92
 monster house 50
 monster rancher 49
 monsters inc. 341
 monte carlo method 23, 147, 203
 motion blur 7, 267
 motion capture 77, 78, 132
 motion control photography 279
 mountain view, california 340
 movies 266
 movie studio 159
 multi-agent system 74
 multimedia 92
 multi pass rendering 296
 multiprocessing 292
 puppet 77

n

need for speed underground 2 123
 network transparency 174
 new line cinema 74
 news 175
 newtek 12
 new york 249, 254
 new zealand 111, 113
 nintendo 58, 123
 nintendo 64 85
 noctis 217
 noise 218
 non-photorealistic rendering 45, 268
 nonlinear programming 132
 normalize 355
 normal map 282
 normal mapping 41, 83, 84, 143, 188
 normal vector 25, 137
 novalogic 351
 nsopengl 174
 numerical 218
 numerical instability 79
 nunchaku 158
 nurbs 4, 101, 143, 165, 206, 326
 nvidia 1, 52, 64, 89, 95, 117, 124, 169, 299,
 300, 322
 nvidia scene graph 10
 nx (unigraphics) 317

o

obb 289
 obb tree 35
 obj 209
 object-oriented 178
 object-oriented model 91
 object oriented 183
 octahedron 341
 october 2005 294
 octree 110, 291, 349
 ogre 10
 old saint paul's 111, 113
 on the fly 213
 opcodes 59
 openal 176
 open directory project 178
 openexr 113
 .opengl 3, 10, 20, 32, 53, 55, 58, 80, 84, 92,
 93, 94, 94, 98, 99, 100, 101, 145, 145, 178,
 180, 182, 183, 186, 205, 208, 257, 273,
 281, 285, 300, 308, 322, 323, 332, 337, 341
 + 175
 opengl arb 97, 98, 145
 opengl architecture review board 169, 177
 opengl es 10, 59, 169, 176
 opengl performer 173, 178, 184
 opengl shading language 10
 .openglut 182
 open inventor 178, 291
 openml 205
 openrt 257, 262
 openscenegraph 10, 173
 opensg 10
 opensl es 176
 open source 95, 145, 145, 168, 173, 179,
 185, 205
 open standard 1, 58
 operating system 58, 173, 182
 operating systems 79
 operation 68
 optical axis 343
 optical coating 258
 optical resonator 258
 optics 76, 343
 orientation 244
 orthogonal 243
 orthogonal matrix 70, 243
 orthographic projection 101, 305
 oslo 258
 out-of-order execution 216

p

p. g. tait 224

paging 82
 paint 76
 painter's algorithm 110, 164, 273, 289, 353
 panda3d 11
 panoscan 113
 parallax 190
 parallax mapping 41, 83, 84, 166, 166
 parameter 68, 293
 parasolid 314
 parietal bone 343
 partial derivative 330
 particle physics 343
 particle system 7, 74, 82
 particle systems ltd 191
 pat hanrahan 26
 path tracing 96
 paul debaev 111, 116, 266
 paul e. debaev 128
 paul heckbert 26
 pdf 127
 pencil tracing 259
 pentium 329
 performance capture 160
 perl 173
 perlin noise 222, 297
 perpendicular 329
 personal computer 15
 personal computers 90
 perspective distortion 19, 22
 perspective projection 8, 305
 perspective transform 355
 pharmacology 136
 phigs 174
 philip mittelman 254
 philipp slusallek 259
 phonemes 5
 phong reflection model 8, 32, 32, 39, 200,
 201, 202, 285, 318
 phong shading 8, 9, 22, 32, 32, 39, 40, 88,
 104, 137, 197, 197, 198, 200, 206
 photograph 131
 photography 2
 photon 140, 248, 253
 photon mapping 72, 76, 96, 256, 256, 259
 photoshopping 205
 physical law 88
 physical simulation 208
 pixar 13, 43, 90, 143, 273, 298, 302, 327
 281, 285, 303, 333, 335, 350, 353
 pixel buffer 321
 pixel image editor 113
 pixels 131, 332, 340
 pixel shaders 80, 348
 pixologic 13
 planar 163
 plasma displays 119
 platonic solid 341
 playstation 2 123
 playstation 3 54, 56, 58, 123, 168, 181, 214,
 216, 216
 playstation portable 56
 plib 173
 plug-in 194
 plugin 11
 poincaré group 240
 pointer 99
 point spread function 257
 polarized glasses 11
 polygon 5, 14, 25, 26, 36, 87, 103, 134, 148,
 163, 187, 206, 211, 211, 211, 329, 343,
 351, 355
 polygonal modeling 4
 polygon mesh 210, 211
 polygon modeling 211
 polygons 206
 polyhedron 36, 43, 343
 polynomial 91
 polytope 36
 polyurethane 137
 portal rendering 110
 portmanteau 331, 350
 poser 14
 position 153
 post-processing 292
 pov-ray 5, 7, 13, 72, 259, 341
 povray 96
 power animator 64
 powerwall 259
 pre-rendered 277
 pro/engineer 317
 procedural animation 217
 procedural generation 222
 procedural texture 217
 procedural textures 267
 procedure 68
 product visualization 312
 profile 180
 programmer 10, 41
 programming 52
 programming language 292, 298
 projection matrix 92
 puppeteer 78
 puppeteers 77
 puppets 78
 pyramid 222

pythagorean theorem 341

q

qsdk 10
quadric 101
quadrilateral 44, 211
quadtree 291
quake 69, 94, 209
quake iii 336
quake iii arena 61
quartz compositor 63
quaternion 223, 244
quaternions 288
quesa 10

r

radeon 62
radian 242
radiance 23, 112, 138, 140
radiance (software) 259
radiation exposure 249, 254
radiative heat transfer 245
radiosity 7, 76, 96, 198, 212, 214, 253, 256, 259, 268, 270, 273
ram 216, 321
randima fernando 297
randi rost 297
rapid prototyping 312, 316
raster graphics 266
raster image 105
rasterisation 163
rasterization 105, 105, 106, 106, 106, 106, 133, 261
raster manager 92
raster scan 285
ratz 77
ratzrun 77
raven software 214
ray-sphere intersection 259
ray casting 23, 252, 268
ray processing unit 257
ray tracer 110
ray tracing 6, 7, 26, 35, 67, 96, 110, 186, 194, 203, 245, 248, 261, 262, 268, 273, 290, 344

raytracing 3, 198, 211, 263, 321
ray tracing hardware 186, 257
ray transfer matrix analysis 258
real-time 3
real-time computer graphics 206
realflow 194
reality engine 92
reality lab 80

real number 240
reboot 50
reconstruction 92
rectangle 36
red 72
reference frame 16
reference rasterizer 80
reflection 247
reflection (physics) 320
reflection mapping 321
reflectivity 76
refraction 27, 248, 248, 252, 253, 267, 292, 320
refractive index 8
render 87
renderer 89
render farm 7, 273
rendering 16, 25, 148, 290, 321
rendering (computer graphics) 11, 213, 279
rendering equation 147, 198, 255, 274
renderman 10, 13, 83, 90, 273, 298, 302
renderman pro server 90
render monkey 90
rendermorphics 80
renderware 10
rescue heroes: the movie 48
research 92
retained mode 170, 173
retroreflector 320
reverse engineering 92
rgb 165, 198
rgba 282
rhinoceros 3d 5
right-hand rule 330
right triangle 341
robocop 2 77
robotech: battlecry 49
robotics 131, 132, 136
rotation group 240, 244
rotations 244
rotoscope 156, 161
round-off error 356
rounding 211
runaway: a road adventure 49
runtime 290

s

saarcor 262
saarland university 257, 262
saddle point 339
salt lake city, utah 285, 340
saturation arithmetic 324
savage 3d 281

- scalar product 240
 scan line 285
 scanline 211, 268
 scanline rendering 5, 7, 206, 252
 scene description language 273
 scene graph 35, 170, 178, 183, 281
 scientific computing 92
 scientific visualization 178
 screensaver 341
 sculpting 2
 secondary animation 158
 sector 250
 sega 47
 sega dreamcast 46
 serious sam 341
 set 199
 shader 33, 105, 106, 107, 298, 306
 shader language 128
 shader model 293
 shaders 176
 shading 164
 shading language 292
 shadow 255, 292, 321
 shadow mapping 134, 296, 310, 337
 shadow volume 304, 308, 323, 324
 shin megami tensei: digital devil saga 49
 shin megami tensei iii: nocturne 49
 shockwave 135
 shoulder 88
 shrek 213
 side effects software 12
 siggraph 116, 147, 327, 340, 341
 silhouette 190
 silhouette edge 309, 310, 323, 324
 silicon graphics 168, 182
 silicon graphics, inc. 92
 simd 148, 348
 sim dietrich 322
 simple directmedia layer 205
 simplex noise 196
 simulation 16, 129
 sine 242
 skeletal animation 51, 151, 152
 sketch based modeling 208
 sketchup 57
 skin 327
 skull 343
 skybox (video games) 75, 266
 skyland 50
 $sl(2,c)$ 240
 slashdot 42
 slerp 244
 sly 2: band of thieves 47
 sly cooper 49
 sly cooper and the thievius raccoonus 47
 smoke 350
 $so(3)$ 240
 $so(4)$ 240
 softimage 64
 softimage xsi 56
 software 71, 353
 software application 15
 solid angle 140
 solidedge 317
 solid modeling 67, 69
 solid modelling 87, 249, 254
 solidworks 317
 sonic x 48, 50
 sony 58, 123
 sony computer entertainment 56
 sound 213
 source engine 69, 116, 127
 space 355
 spatial rotation 223
 special effects animation 51
 spectralon 137
 specular 4, 76
 specular highlight 104, 137, 201, 202
 specularity 197
 specular lighting 247
 specular reflection 75, 76, 96, 137, 138
 speedtree 215, 217
 sphere 5, 36, 68, 101, 182, 249, 254
 spherical harmonics 272
 spherical wrist 132
 spinor group 244
 splash damage 144
 square root 241
 stanford bunny 73, 73, 342
 starflight 217
 star wars: clone wars 48
 star wars: knights of the old republic 341
 steam 350
 stencil 321
 stencil buffer 309, 322, 324
 stencil shadow volume 42, 296, 309, 310, 322, 337, 337
 stencil shadow volumes 197
 steradian 76
 steve baker 205
 steven worley 297
 steve upstill 148, 297
 stream processing 293
 stream processor 106
 strehl ratio 257
 stuart little 3: call of the wild 48

- su(2) 240
 subdivision 38
 subdivision surface 4, 43, 206
 subsurface scattering 7
 sumac 215
 sun 138, 174, 305
 sun microsystems 169, 179
 superior defender gundam force 50
 superman 64 85
 super mario 64 143
 super nintendo 250
 supersampling 333
 supersonic 135
 surface 206, 329
 surface integral 331
 surface normal 38, 40, 87, 103, 138, 206, 264, 317
 suspension of disbelief 215
 swift3d 14
 symbian os 181
 synthesizer 213
 synthespians™ 331
 synthetic 331
- t**
- tachikoma days 48
 tales of symphonia 49
 tao (software) 332
 target state 259
 technical drawing 91, 317
 technological 131
 telescope 252
 tessellation 5, 84, 101
 tetrahedron 341
 texture 15, 82, 213, 250, 321
 texture filtering 333, 335, 335
 texture mapped 208
 texture mapping 9, 15, 22, 28, 80, 144, 149, 170, 292, 295, 332, 334, 345
 texture splatting 335
 the animatrix 48
 theatre 76
 the black lotus 214
 the chronicles of riddick: escape from butcher bay 166
 the elder scrolls iv: oblivion 217
 the house of the dead 3 47
 the inquirer 42
 the iron giant 48
 the jim henson hour 77
 the legend of zelda: phantom hourglass 49
 the legend of zelda: the wind waker 47, 49
 the littlest robo 50
- the lord of the rings 74
 theme park 76
 the muppets 77
 thermoforming 312
 the sentinel (computer game) 217
 the simpsons 50
 thespian 331
 the tech report 42
 thief: deadly shadows 166
 threshold 147
 thumb 88
 tim heidmann 322
 tom and jerry blast off to mars 48
 tom clancy's splinter cell: chaos theory 166
 tone mapping 34, 119, 120, 275
 tony hawk's american sk8land 49
 tool 131
 total annihilation 351
 toy story 213, 302, 341
 tracepro 258
 trademark 94
 traditional animation 45, 51
 transfer function 112
 transform and lighting 22, 92, 170, 211
 transformers cybertron 50
 transformers energon 50
 translucent 249, 253
 triangle 206, 211, 211
 triangulation 262
 trigonometry 258
 trilinear filtering 31, 31, 333
 trojan room coffee pot 342
 tron 2.0 33
 trudimension 10
 truespace 5
 turbulence 218
 turner whitted 255
 turok: dinosaur hunter 85
 turtle talk with crush 77
 typography 91
- u**
- ubisoft 117, 166
 ultrashadow 62
 ultrasound 351
 ultraviolet 257
 unified lighting and shadowing 197
 units 356
 unit vector 258
 university of utah 39, 104, 198, 200, 285, 338
 unix 58, 168
 unreal engine 56, 69, 166

usa 104
utah teapot 73, 73, 182, 208

v

v-ray 7
valence 326
valve software 166
vector field 295, 331
vector format 222
vector graphics 210
vector product 240
vectors 136
vertex 142, 151, 206, 300
vertex normals 206
vertices 38
video 14
video game 36, 159, 188, 214, 214, 264
videogame 16
video game console 58, 180
video game industry 15
video game publisher 159
video games 45, 152
vietnam 39
viewing frustum 26, 35, 106, 110, 354
viewport 222
viewtiful joe 47, 49
virtualgl 177
virtualization 82
virtual model 277, 342
virtual reality 85, 168
visibility 142
visibility function 23
visibility problem 133, 187, 353
visible spectrum 138
visitor pattern 288
visual basic 173
visualization 312
visual system 112
volume 135
volume ray casting 347
volume rendering 273, 351, 352
volumetric display 352
volumetric flow rate 351
volumetric model 91
volumetric sampling 7
voodoo 94
voxel 313, 345
vrml 7, 11, 179, 185, 273

w

w-buffering 354, 356
wacky races 46
waldo c. graphic 77

walt disney imagineering 77
walt disney world 77
wavelength 257
web design 39
web page 39
wellington 111, 113
westwood studios 351
weta digital 74
wgl 174
white 72
white noise 196
wii 58, 123
wild arms 3 49
william rowan hamilton 231
will wright 214, 216, 218
wind 73
windowing system 173
windows 95 79, 80
windows display driver model 81
windows graphics foundation 80
windows vista 80
wing 135
winged edge 211
wings 3d 13, 44
winx club 50
wireframe 15, 108, 267
wire frame model 210, 312
wolfenstein 3d 250
wood 218
worcester polytechnic institute 204
wrist 88
wxwindows 102

x

x-men legends ii: rise of apocalypse 49
x-ray 257
x11 window system 175
x3d 11, 57
xbox 47, 58, 79, 123
xbox 360 59, 79, 123, 214, 216, 216
xerography 46
xhtml 39
xml schema 56
xscreensaver 147

y

yafray 3, 13, 96, 259

z

z-buffer 20, 21, 133, 285, 321, 356
z-buffering 46, 110, 170, 322, 353
z-order 356
zemax 258

zero set 91
zoids 50

zone of the enders: the 2nd runner 49

DRAFT