Concepts and Applications of

Computer Graphics & Vector Graphics

Hortensia Perea

Madelyn Gorham

First Edition, 2012

ISBN 978-81-323-1270-3

© All rights reserved.

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

Table of Contents

Chapter 1 - Introduction to Vector Graphics

Chapter 2 - Scalable Vector Graphics

Chapter 3 - OpenVG and Portable Document Format

Chapter 4 - Subpixel Rendering and Vector Markup Language

Chapter 5 - Shading

Chapter 6 - Gouraud Shading and Phong Shading

Chapter 7 - Blinn–Phong Shading Model and Oren–Nayar Reflectance Model

Chapter 8 - Shader and ARB (GPU Assembly Language)

Chapter 9 - Ambient Occlusion, Bloom (Shader Effect) and Cel-Shaded Animation

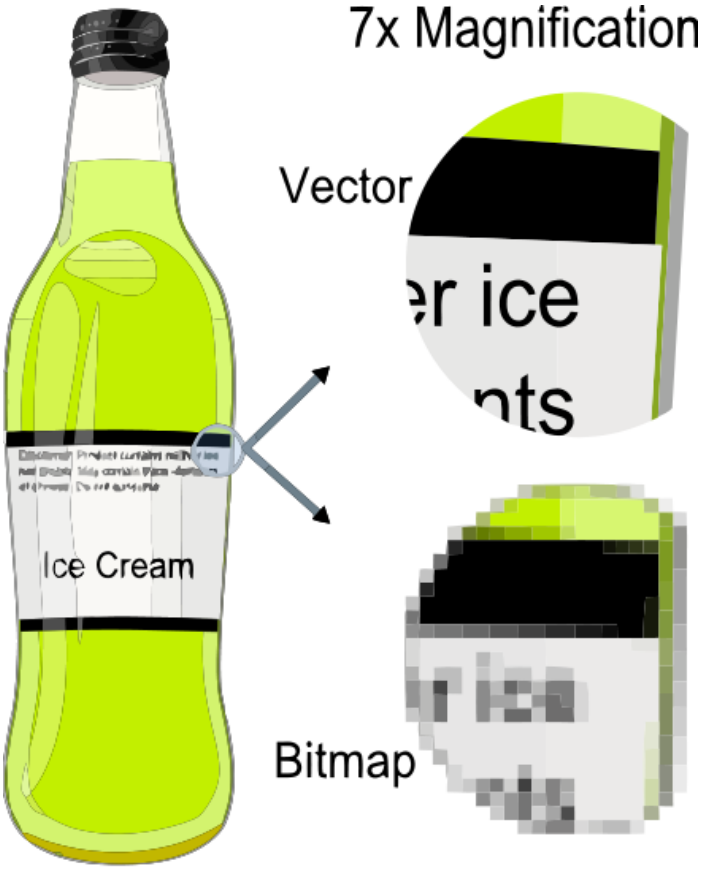
Chapter 10 - Cg (Programming Language) and Deferred Shading

Chapter 11 - GLSL and High Level Shader Language

Chapter 12 - Pixel Shader, Shadow Mapping and Shadow Volume

Chapter 1

Introduction to Vector Graphics



Example showing effect of vector graphics versus raster graphics. The original vector-based illustration is at the left. The upper-right image illustrates magnification of 7x as a vector image. The lower-right image illustrates the same magnification as a bitmap image. Raster images are based on pixels and thus scale with loss of clarity, while vector-based images can be scaled indefinitely without degrading quality.

Vector graphics is the use of geometrical primitives such as points, lines, curves, and shapes or polygon(s), which are all based on mathematical equations, to represent images in computer graphics.

Vector graphics formats are complementary to raster graphics, which is the representation of images as an array of pixels, as it is typically used for the representation of photographic images. There are instances when working with vector tools and formats is the best practice, and instances when working with raster tools and formats is the best practice. There are times when both formats come together. An understanding of the advantages and limitations of each technology and the relationship between them is most likely to result in efficient and effective use of tools.

Overview

Computer displays are made up from grids of small rectangular cells called pixels. The picture is built up from these cells. The smaller and closer the cells are together, the better the quality of the image, but the bigger the file needed to store the data. If the number of pixels is kept constant, the size of each pixel will grow and the image becomes grainy (pixellated) when magnified, as the resolution of the eye enables it to pick out individual pixels.

Vector graphics files store the lines, shapes and colours that make up an image as mathematical formulae. A vector graphics program uses these mathematical formulae to construct the screen image, building the best quality image possible, given the screen resolution. The mathematical formulae determine where the dots that make up the image should be placed for the best results when displaying the image. Since these formulae can produce an image scalable to any size and detail, the quality of the image is limited only by the resolution of the display, and the file size of vector data generating the image stays the same. Printing the image to paper will usually give a sharper, higher resolution output than printing it to the screen but can use exactly the same vector data file.

Editing vector graphics

Vector graphic drawing software is used for creating and editing vector graphics. The image can be changed by editing screen objects which are then saved as modifications to the mathematical formulae. Mathematical operators in the software can be used to stretch, twist, and colour component objects in the picture or the whole picture, and these tools are presented to the user intuitively through the graphical user interface of the computer. It is possible to save the screen image produced as a bitmap/raster file or generate a bitmap of any resolution from the vector file for use on any device.

The size of the file generated will depend on the resolution required, but the size of the vector file generating the bitmap/raster file will always remain the same. Thus, it is easy to convert from a vector file to a range of bitmap/raster file formats but it is much more difficult to go in the opposite direction, especially if subsequent editing of the vector picture is required. It might be an advantage to save an image created from a vector source file as a bitmap/raster format, because different systems have different (and incompatible) vector formats, and some might not support vector graphics at all. However, once a file is converted from the vector format, it is likely to be bigger, and it loses the advantage of scalability without loss of resolution. It will also no longer be

possible to edit individual parts of the image as discrete objects. The file size of vector graphic depends on the number of graphic elements it contains.

Vector formats are not always appropriate in graphics work. For example, devices such as cameras and scanners produce raster graphics that are impractical to convert into vectors, and so for this type of work, the editor will operate on the pixels rather than on drawing objects defined by mathematical formulae. Comprehensive graphics tools will combine images from vector and raster sources, and may provide editing tools for both, since some parts of an image could come from a camera source, and others could have been drawn using vector tools.

Standards

The W3C standard for vector graphics is SVG. The standard is complex and has been relatively slow to be established at least in part owing to commercial interests. Many web browsers now have some support for rendering SVG data but full implementations of the standard are still comparatively rare.

An original reference photograph before vectorization

Detail can be added or removed from vector art. Vector illustrations can have their own colours, allowing artists to achieve desired results.

Applications

One of the first uses of vector graphic displays was the US SAGE air defense system. Vector graphics systems were only retired from U.S. en route air traffic control in 1999, and are likely still in use in military and specialised systems. Vector graphics were also used on the TX-2 at the MIT Lincoln Laboratory by computer graphics pioneer Ivan Sutherland to run his program Sketchpad in 1963.

Subsequent vector graphics systems, most of which iterated through dynamically modifiable stored lists of drawing instructions, include Digital's GT40. There was a home

gaming system that used vector graphics called Vectrex as well as various arcade games like *Asteroids* and *Space Wars*. Storage scope displays, such as the Tektronix 4014, could display vector images but not modify them without first erasing the display.

Modern vector graphics displays can sometimes be found at laser light shows, where two fast-moving X-Y mirrors are used to rapidly draw shapes and text on a screen.

The term "vector graphics" is mainly used today in the context of two-dimensional computer graphics. It is one of several modes an artist can use to create an image on a raster display. Other modes include text, multimedia, and 3D rendering. Virtually all modern 3D rendering is done using extensions of 2D vector graphics techniques. Plotters used in technical drawing still draw vectors directly to paper.

Information

For example, consider a circle of radius *r*. The main pieces of information a program needs in order to draw this circle are

1. an indication that what is to be drawn is a circle
2. the radius *r*
3. the location of the center point of the circle
4. stroke line style and colour (possibly transparent)
5. fill style and colour (possibly transparent)

Advantages to this style of drawing over raster graphics:

- This minimal amount of information translates to a much smaller file size compared to large raster images (the size of representation does not depend on the dimensions of the object), though a vector graphic with a small file size is often said to lack detail compared with a real world photo.
- Correspondingly, one can indefinitely zoom in on e.g. a circle arc, and it remains smooth. On the other hand, a polygon representing a curve will reveal being not really curved.
- On zooming in, lines and curves need not get wider proportionally. Often the width is either not increased or less than proportional. On the other hand, irregular curves represented by simple geometric shapes may be made proportionally wider when zooming in, to keep them looking smooth and not like these geometric shapes.
- The parameters of objects are stored and can be later modified. This means that moving, scaling, rotating, filling etc. doesn't degrade the quality of a drawing. Moreover, it is usual to specify the dimensions in device-independent units, which results in the best possible rasterization on raster devices.
- From a 3-D perspective, rendering shadows is also much more realistic with vector graphics, as shadows can be abstracted into the rays of light from which they are formed. This allows for photo realistic images and renderings.

Typical primitive objects

- Lines and polylines
- Polygons
- Circles and ellipses
- Bézier curves
- Bezigons
- Text (in computer font formats such as TrueType where each letter is created from Bézier curves)

This list is not complete. There are various types of curves (Catmull-Rom splines, NURBS etc.), which are useful in certain applications.

Often, a bitmap image is considered as a primitive object. From the conceptual view, it behaves as a rectangle.

Vector operations

Vector graphics editors typically allow rotation, movement, mirroring, stretching, skewing, affine transformations, changing of z-order and combination of primitives into more complex objects.

More sophisticated transformations include set operations on closed shapes (union, difference, intersection, etc.).

Vector graphics are ideal for simple or composite drawings that need to be device-independent, or do not need to achieve photo-realism. For example, the PostScript and PDF page description languages use a vector graphics model.

Printing

Vector art is key for printing. Since the art is made from a series of mathematical curves it will print very crisply even when resized. For instance, one can print a vector logo on a small sheet of copy paper, and then enlarge the same vector logo to billboard size and keep the same crisp quality. A low-resolution raster graphic would blur or pixelate excessively if it were enlarged from business card size to billboard size. (The precise resolution of a raster graphic necessary for high-quality results depends on the viewing distance; e.g., a billboard may still appear to be of high quality even at low resolution if the viewing distance is large enough.)

If we regard typographic characters as images, then the same considerations that we have made for graphics apply even to composition of written text for printing (typesetting). Older character sets were stored as bitmaps, therefore to achieve maximum print quality they had to be used at a given resolution only; these font formats are said to be non-scalable. High quality typography is nowadays based on character drawings (fonts) which

are typically stored as vector graphics, and as such are scalable to any size. Examples of these vector formats for characters are Postscript fonts and TrueType fonts.

Vector illustration

Vector illustration is a popular technique of many digital illustrators worldwide. Some of the greatest internationally acclaimed artists in the field are Catalina Estrada, Petra Stefankova, Nathan Jurevicius, J. Otto Seibold, Leo Blanchette and others.

3D modelling

In 3D computer graphics, vectorized surface representations are most common (bitmaps can be used for special purposes such as surface texturing, height-field data and bump mapping). At the low-end, simple meshes of polygons are used to represent geometric detail in applications where interactive frame rates or simplicity are important. At the high-end, where one is willing to trade-off higher rendering times for increased image quality and precision, smooth surface representations such as Bézier patches, NURBS or Subdivision surfaces are used. One can, however, achieve a smooth surface rendering from a polygonal mesh through the use of shading algorithms such as Phong and Gouraud.

Formats

One example of vector graphics format is SVG (Scalable Vector Graphics), an open standard created and developed by the World Wide Web Consortium to address the need (and attempts of several corporations) for a versatile, scriptable and all-purpose vector format for the web and otherwise. Another example is VML, a proposed standard that was adopted by Microsoft. Arguably, the AI format, the native format for Adobe Illustrator, is also standard in vector graphics.

The SWF Adobe's (formerly Macromedia's) file format is also a vector based container used to store animation. Web pages created in Flash can be enlarged to fit any monitor size while retaining the same graphic quality.

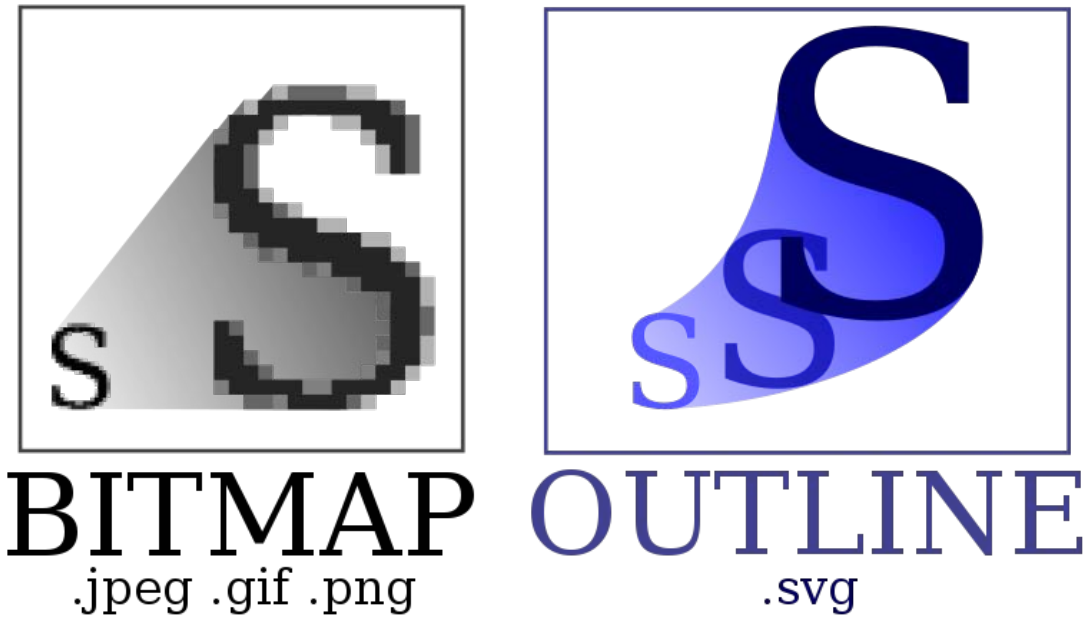
Adobe Fireworks, which is also (mostly) a vector-based application, saves its files in PNG format; vectors created in Adobe Fireworks can be transferred to Adobe Illustrator, and vice versa.

Chapter 2

Scalable Vector Graphics

Scalable Vector Graphics

Filename extension	`.svg, .svgz`
Internet media type	`image/svg+xml`
Developed by	World Wide Web Consortium
Initial release	4 September 2001
Latest release	Tiny 1.2 / 22 December 2008; 23 months ago
Type of format	vector image format
Extended from	XML
Open format?	Yes



This image illustrates the difference between bitmap and vector images. The bitmap image is composed of a fixed set of dots, while the vector image is composed of a fixed set of shapes. In the picture, scaling the bitmap reveals the dots and scaling the vector image preserves the shapes.

Scalable Vector Graphics (**SVG**) is a family of specifications of an XML-based file format for describing two-dimensional vector graphics, both static and dynamic (i.e. interactive or animated).

The SVG specification is an open standard that has been under development by the World Wide Web Consortium (W3C) since 1999.

SVG images and their behaviors are defined in XML text files. This means that they can be searched, indexed, scripted and, if required, compressed. Since they are XML files, SVG images can be created and edited with any text editor, but specialized SVG-based drawing programs are also available.

All major modern web browsers except Microsoft Internet Explorer (IE), support and render SVG markup directly. The Internet Explorer 9 beta supports SVG.

Overview

SVG has been in development since 1999 by a group of companies within the W3C after the competing standards Precision Graphics Markup Language (PGML) – developed from Adobe's PostScript – and Vector Markup Language (VML) – developed from Microsoft's RTF – were submitted to W3C in 1998. SVG drew on experience from the designs of both those formats.

SVG allows three types of graphic objects:

- Vector graphics
- Raster graphics
- Text

Graphical objects, including PNG and JPEG raster images, can be grouped, styled, transformed, and composited into previously rendered objects. SVG does not directly support z-indices that separate drawing order from document order for overlapping objects, unlike some other vector markup languages like VML. Text can be in any XML namespace suitable to the application, which enhances search ability and accessibility of the SVG graphics. The feature set includes nested transformations, clipping paths, alpha masks, filter effects, template objects and extensibility.

Since 2001, the SVG specification has been updated to version 1.1 (current Recommendation) and 1.2 (still a Working Draft). The **SVG Mobile** Recommendation introduced two simplified *profiles* of SVG 1.1, **SVG Basic** and **SVG Tiny**, meant for devices with reduced computational and display capabilities. SVG Tiny later became an autonomous Recommendation (current version 1.2) and the basis for SVG 1.2. In addition to these variants and profiles, the **SVG Print** specification (still a Working Draft) contains guidelines for printable SVG 1.2 and SVG Tiny 1.2 documents.

Although not directly comparable to SVG since it is a bitmap based format, the Canvas element in HTML provides an approach to rendering dynamic graphics in HTML that's procedural rather than declarative: instead of specifying the shapes to draw in XML, the author executes drawing commands from a script. Canvas doesn't allow for static rendering, and drawn elements are not identifiable in a DOM-like way.

Printing

Though the SVG Specification primarily focuses on vector graphics markup language, its design includes the basic capabilities of a page description language (PDL), like Adobe's PDF. It contains provisions for rich graphics, and is also compatible with CSS for styling purposes. It is unlike XHTML, whose primary purpose is communication of content, not presentation, and therefore specifies objects to be displayed but not where to place such objects. Conversely SVG is an ideal PDL for print-oriented uses, as it contains all the functionality required to place each glyph and image in a chosen location on the final page. A much more print-specialized subset of SVG (SVG Print, authored by Canon, HP, Adobe and Corel) is currently a W3C Working Draft.

Scripting and animation

SVG drawings can be dynamic and interactive. Time-based modifications to the elements can be described in SMIL, or can be programmed in a scripting language (e.g., ECMAScript or JavaScript). The W3C explicitly recommends SMIL as the standard for

animation in SVG. A rich set of event handlers such as *onmouseover* and *onclick* can be assigned to any SVG graphical object.

Compression

SVG images, being XML, contain many repeated fragments of text and are thus particularly suited to compression by gzip, though other compression methods may be used effectively. Once an SVG image has been compressed it may be referred to as an "SVGZ" image, with the corresponding filename extension. The resulting file may be as small as 20% of the original size.

Development history

SVG was developed by the W3C SVG Working Group starting in 1998, after Macromedia and Microsoft introduced VML whereas Adobe Systems and Sun Microsystems submitted a competing format known as PGML. The working group was chaired by Chris Lilley of the W3C.

- SVG 1.0 became a W3C Recommendation on 4 September 2001.
- SVG 1.1 became a W3C Recommendation on 14 January 2003. The SVG 1.1 specification is modularized in order to allow subsets to be defined as profiles. Apart from this, there is very little difference between SVG 1.1 and SVG 1.0.
    - SVG Tiny and SVG Basic (the Mobile SVG Profiles) became W3C Recommendations on 14 January 2003. These are described as profiles of SVG 1.1.
- SVG Tiny 1.2 became a W3C Recommendation on 22 December 2008.
- SVG Full 1.2 has had a W3C Working Draft in process for years, but now will be dropped soon in favor of a SVG 2.0. SVG Tiny 1.2 was initially released as a profile, and later refactored to be a complete specification, including all needed parts of SVG 1.1 and SVG 1.2. SVG 1.2 Full adds modules onto the SVGT 1.2 core.
- SVG Print adds syntax for multi-page documents and mandatory color management support.
- SVG 1.1 Second Edition, which includes all the errata and clarifications, but no new features to the original SVG 1.1 was given as a Last Call publication on June 22, 2010. It is expected to be approved later this year.
- SVG 2.0 will completely rework draft 1.2 with more integration with new web features such as CSS, HTML and WOFF.

The MPEG-4 Part 20 standard - *Lightweight Application Scene Representation (LASeR) and Simple Aggregation Format (SAF)* is based on SVG Tiny. It was developed by MPEG (ISO/IEC JTC1/SC29/WG11) and published as ISO/IEC 14496-20:2006. SVG capabilities are enhanced in MPEG-4 Part 20 with key features for mobile services, such as dynamic updates, binary encoding, state-of-art font representation. SVG was also accommodated in MPEG-4 Part 11, in the Extensible MPEG-4 Textual (XMT) format - a textual representation of the MPEG-4 multimedia content using XML.

Mobile profiles

Because of industry demand, two mobile profiles were introduced with SVG 1.1: *SVG Tiny* (SVGT) and *SVG Basic* (SVGB). These are subsets of the full SVG standard, mainly intended for user agents with limited capabilities. In particular, SVG Tiny was defined for highly restricted mobile devices such as cellphones, and SVG Basic was defined for higher-level mobile devices, such as PDAs.

In 2003, the 3GPP adopted SVG Tiny as the mandatory vector graphics media format for next-generation phones. SVGT is the required vector graphics format and support of SVGB is optional for Multimedia Messaging Service (MMS) and Packet Switch Stream (PSS). It was later added as required format for vector graphics in 3GPP IP Multimedia Subsystem (IMS). Neither mobile profile includes support for the full DOM, while only SVG Basic has optional support for scripting, but because they are fully compatible subsets of the full standard most SVG graphics can still be rendered by devices which only support the mobile profiles.

SVGT 1.2 adds a microDOM (µDOM), allowing all mobile needs to be met with a single profile.

Functionality

The SVG 1.1 specification defines 14 important functional areas or feature sets:

Paths
: Simple or compound shape outlines drawn with curved or straight lines can be filled in or outlined (or used as a clipping path) and are expressed in a highly compact coding in which, for example, **M** (from 'move to') precedes the initial numeric X and Y coordinates and **L** (line to) will precede a subsequent point to which a line should be drawn. Further command letters (**C**, **S**, **Q**, **T** and **A**) precede data that is used to draw various Bézier and elliptical curves. **Z** is used to close a path. In all cases, absolute coordinates follow capital letter commands and relative coordinates are used after the equivalent lower-case letters.

Basic Shapes
: Straight-line paths or paths made up of a series of connected straight-line segments (polylines), as well as closed polygons, circles and ellipses can be drawn. Rectangles and round-cornered "rectangles" are other standard elements.

Text
: Unicode character text included in an SVG file is expressed as XML character data. Many visual effects are possible, and the SVG specification automatically handles bidirectional text (as when composing a combination of English and Arabic text, for example), vertical text (as Chinese was historically written) and characters along a curved path (such as the text around the edges of the Great Seal of the United States).

Painting

SVG shapes can be filled and/or outlined (painted with a colour, a gradient or a pattern). Fills can be opaque or have various degrees of transparency. "Markers" are end-of-line features, such as arrowheads, or symbols which can appear at the vertices of a polygon.

Colour

Colours can be applied to all visible SVG elements, either directly or via the 'fill', 'stroke' and other properties. Colours are specified in the same way as in CSS2, i.e. using names like `black` or `blue`, in hexadecimal such as `#2f0` or `#22ff00`, in decimal like `rgb(2,2,127)` or as percentages of the form `rgb(100%,100%,0%)`.

Gradients and Patterns

SVG shapes can be filled or outlined with solid colours as above, or with colour gradients or with repeating patterns. Colour gradients can be linear or radial (circular), and can involve any number of colours as well as repeats. Opacity gradients can also be specified. Patterns are based on predefined raster or vector graphic objects, which can be repeated in x and/or y directions. Gradients and patterns can be animated and scripted.

Since 2008, there has been discussion among professional users of SVG that either gradient meshes or preferably diffusion curves could usefully be added to the SVG specification. It is said that a "simple representation [using diffusion curves] is capable of representing even very subtle shading effects" and that "Diffusion curve images are comparable both in quality and coding efficiency with gradient meshes, but are simpler to create (according to several artists who have used both tools), and can be captured from bitmaps fully automatically."

Clipping, Masking and Compositing

Graphic elements, including text, paths, basic shapes and combinations of these, can be used as outlines to define both 'inside' and 'outside' regions that can be painted (with colors, gradients and patterns) independently. Fully opaque *clipping paths* and semi-transparent *masks* are *composited* together to calculate the color and opacity of every pixel of the final image, using simple alpha blending.

Filter Effects

Interactivity

SVG images can interact with users in many ways. In addition to hyperlinks as mentioned below, any part of an SVG image can be made to trigger events representing changes in focus, mouse clicks, scrolling or zooming the image and other pointer, keyboard and document events. Event handlers may start, stop or alter animations and trigger any other scripts in response to these events.

Linking

SVG images can contain hyperlinks to other documents, using XLink. URLs of SVG images can specify geometrical transforms in the fragment section.

Scripting

All aspects of an SVG document can be accessed and manipulated using scripts in a similar way to HTML. The default scripting language is ECMAScript (closely related to JavaScript) and there are defined Document Object Model (DOM) objects for every SVG element and attribute. Scripts are enclosed in `<script>`

elements. They can run in response to pointer events, keyboard events and document events as required.

Animation

SVG content can be animated using the built-in animation elements such as `<animate>`, `<animateMotion>` and `<animateColor>`. Animation is also possible by manipulating the DOM using ECMAScript, using the scripting language's built-in timers. SVG animation has been designed to be compatible with current and future versions of Synchronized Multimedia Integration Language (SMIL). Animations can be continuous, they can loop and repeat and they can respond to user events, as mentioned above.

Fonts

As with HTML and CSS, text in SVG may reference external font files, such as system fonts. If the required font files do not exist on the machine where the SVG file is rendered, the text may not appear as intended. To overcome this limitation, text can be displayed in an 'SVG font', where the required glyphs are defined in SVG as a font that is then referenced from the `<text>` element.

Metadata

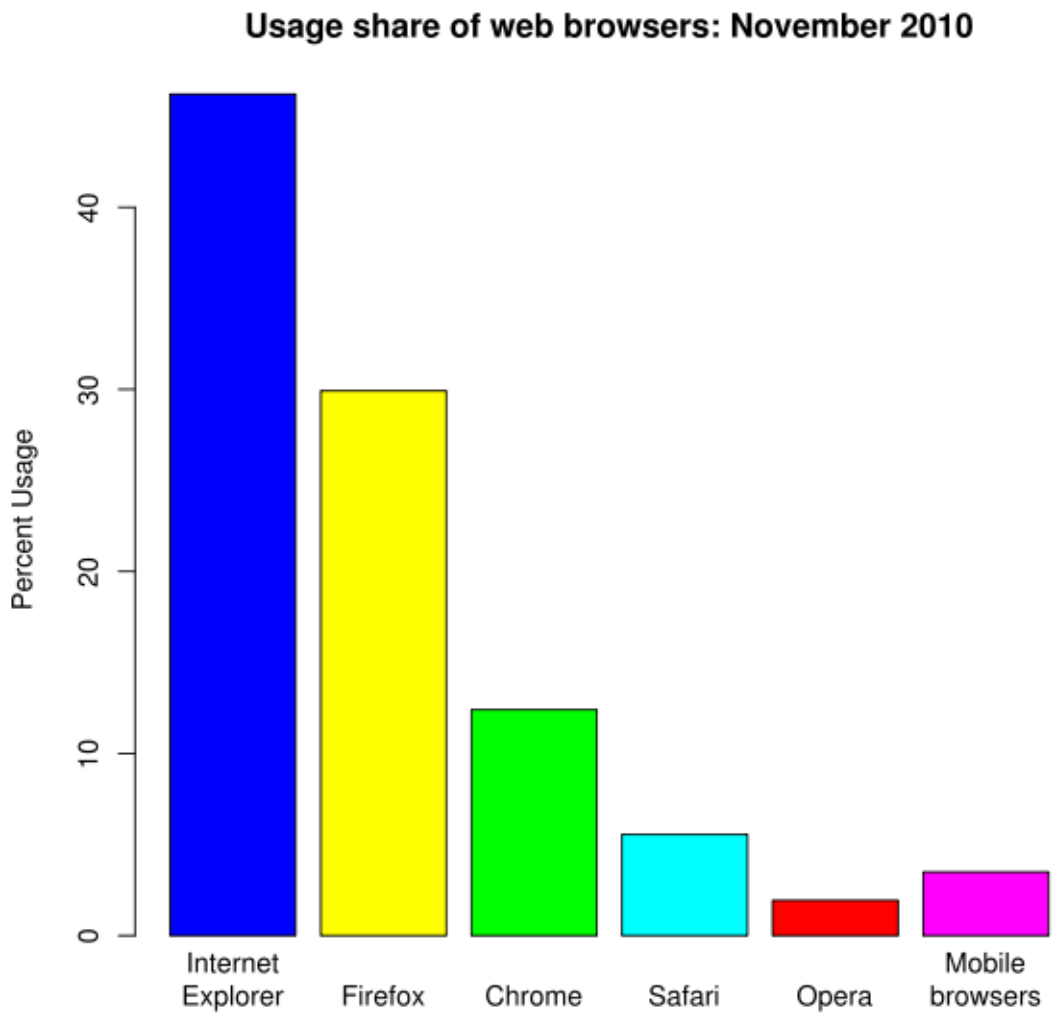
In accord with the W3C's Semantic Web initiative, SVG allows authors to provide metadata about SVG images. The main facility is the `<metadata>` element, where the document can be described using Dublin Core metadata properties (e.g., title, creator/author, subject, description, etc). Other metadata schemas may also be used. In addition, SVG defines `<title>` and `<desc>` elements where authors can provide further plain-text descriptive material within an SVG image to help indexing, searching and retrieval by a number of means.

An SVG image can define components and use them repeatedly. SVG image can also contain raster graphics (usually PNG and JPEG images) and other SVG images.

SVG on the web

Google announced on 31 August 2010 that it had begun to index SVG content on the web, whether it is in standalone files or embedded in HTML. They said that Google search users will start to see such content listed among their search results.

Support for SVG in web browsers



The usage share of web browsers.
Source: Median values from world wide summary table. Internet Explorer (46.22%) Mozilla Firefox (29.92%; Usage by version number) Google Chrome (12.40%) Safari (.%) Opera (1.93%) Mobile browsers (3.49%)

The use of SVG on the web is still limited by the lack of support in Internet Explorer which (as of December 2010) is the most widely-used browser. The most widely deployed version of IE (version 8) does not support SVG. However, Microsoft has announced that IE9 will support SVG. Many web sites that serve SVG images.

Native support

As of 2010, all major desktop browsers, and many minor ones, have some level of SVG support, except for Internet Explorer. Other browsers' implementations are not yet complete. As of 2010, only Opera, Safari and Google Chrome supported embedding via the `<img>` HTML element. Mozilla Firefox and some other browsers that can display SVG graphics currently need them embedded in `<object>` or `<iframe>` elements to display them integrated as parts of an HTML webpage. However, SVG images may be included in XHTML pages using XML namespaces.

Tim Berners-Lee, the inventor of the Web, has been critical of Internet Explorer for its failure to support SVG. All currently supported graphical browsers on Linux systems and the Macintosh have implemented some level of SVG support.

- Opera (since 8.0) has support for the SVG 1.1 Tiny specification while Opera 9 includes SVG 1.1 Basic support and some of SVG 1.1 Full. Since Opera 9. has partial SVG Tiny 1.2 support. It also supports SVGZ (compressed SVG).
- Browsers based on the Gecko layout engine (such as Firefox, Flock, Camino and SeaMonkey) all have had incomplete support for the SVG 1.1 Full specification since 200. The Mozilla site has an overview of the modules which are supported in Firefox and of the modules which are in progress in the development. Gecko 1.9, included in Firefox 3.0, adds support for more of the SVG specification (including filters).
- Browsers based on WebKit (such as Apple's Safari, Google Chrome, and The Omni Group's OmniWeb) have had incomplete support for the SVG 1.1 Full specification since 2006.
- Amaya has partial SVG support.

There are several advantages to native and full support: plugins are not needed, SVG can be freely mixed with other content in a single document, and rendering and scripting become considerably more reliable.

SVG and Microsoft Internet Explorer

As of April 2010, Internet Explorer, up to and including IE8, the latest release, is the only current major browser not to provide native SVG support, although native support is partially available in Internet Explorer 9 Platform Preview: Current IE versions require a plug-in to render SVG content. There are a number of plug-ins available to assist including:

- Google Chrome Frame from Google can support all web elements supported by WebKit, including SVG 1.0 and partially SVG 1.1.

- Renesis Player for Internet Explorer from examotion GmbH, supports SVG 1.1 on IE 6 and 7 and is available for download free of charge.

- Ssrc SVG Plugin from Savarese Software Research Corporation is a plug-in for rendering SVG in Internet Explorer versions 6, 7, and 8 on Windows XP, Server 2003, Vista, Server 2008, and 7. In addition, the plug-in enables Internet Explorer to load Mozilla XUL applications.

- Batik, a widely deployed Java plugin

- Bitflash, an SVG viewer intended for mobile devices, which can be installed as an IE plugin

- Ikivo, an SVG viewer intended for mobile devices, which can be installed as an IE plugin

- GPAC, targets SVGT 1.2

- Adobe SVG Viewer from Adobe Systems plugin supports most of SVG 1.0/1.1. Zooming and panning of the image are also supported to enable viewing of any area of the SVG lying outside the visible area of its containing window, but scrolling is not supported. Adobe SVG Viewer is available for download from Adobe Systems free of charge. However, on 1 January 2009, Adobe Systems discontinued support for Adobe SVG Viewer.

- Corel SVG Viewer(*discontinued*)

Ample SDK Open-Source JavaScript GUI Framework provides partial support for SVG 1.1, SMIL, DOM and style scripting in Internet Explorer (. - 8.0) too. It is not dependent on any plugins and relies on presence of alternative Vector Graphics format VML in Internet Explorer.

SVG Web is a JavaScript library for Web developers, targeted at Internet Explorer and dependent on the presence of an installed Adobe Flash plugin on the client machine. SVG Web provides partial support for SVG 1.1, SVG Animation (SMIL), Fonts, Video and Audio, DOM and style scripting.

On January, 2010, a senior manager of the Internet Explorer team at Microsoft announced on his official blog that Microsoft had just requested to join the SVG Working Group of the W3C in order to "take part in ensuring future versions of the SVG spec will meet the needs of developers and end users," although no plans for SVG support in Internet Explorer were mentioned at that time. During Microsoft's MIX 2010 developer conference, it was announced that IE9 would support SVG 1.1.

Software and support in applications

Images can be rasterised using a library such as ImageMagick, which provides a quick but incomplete implementation of SVG, or Batik, which implements nearly all of SVG 1.1 and much of SVG Tiny 1.2 but requires the Java Runtime Environment.

- 3dpaintbrush can export 3D scenes/objects to SVG files.
- ABViewer "?" viewing, editing and conversion.
- ACD Canvas supports importing and exporting SVG.
- Adobe Acrobat (since version .1) supports SVG 1.0 only in so called "alternate presentations" in PDF format.
- Adobe Illustrator supports both the import and export of SVG images. When writing SVG files it has the option to embed a copy of the associated proprietary Illustrator format for later editing. This often results in changes being lost if another editor is used when the file is reopened in Illustrator because it ignores everything but the embedded Illustrator file.
- Altsoft Xml2PDF allows converting SVG files to PDF, PS, various GDI+ formats.
- Batik SVG Toolkit can be used by Java programs to render, generate, and manipulate SVG graphics.
- Blender will import SVG graphics as paths. As of 2.3 it will also now export UV Maps as SVG files.
- Cairo is a vector graphics based library which can generate SVG. It has bindings for many programming languages including C++, Haskell, Java, Perl, Python, Scheme, Smalltalk and several others.
- CorelDRAW has an SVG export and import filter.
- Ecava IntegraXor created SAGE (SCADA Animation Graphic Editor) which allows SVG graphic to be used in SCADA application.
- GIMP allows SVG images to be imported as paths or rendered bitmaps. *GIMP paths* are compatible with SVG.
- The GNOME project has had integrated SVG support throughout the desktop since 2000.
- GNU LilyPond can export musical scores to SVG.
- Go-oo Draw (OpenOffice.org variant) can open and export SVG files.
- Google Docs can export a drawing into SVG files.
- Inkscape is an open source vector graphics program for Linux, Microsoft Windows and Mac OS X that uses SVG as its native image format. Inkscape also contains tools to convert pixel-based graphics to SVG. Inkscape can import PDF files and thus can transform content in PDF to SVG.
- ItsNat is a Java based web framework with built-in support of native SVG, Adobe SVG Viewer, Renesis, Savarese Ssrc, Batik applet and SVGWeb plugins.
- Joffice is a Java based Office suite with built-in support of SVG.
- Mappetizer is a commercial converter for spatial data to SVG.
- Microsoft Visio since version 2003 can import and save files in the SVG format as well as the SVG compressed format.
- OpenOffice.org Draw can export SVG drawings. Import extensions are available to import SVG images into OpenOffice.org Draw.
- Opera SVG Viewer is an Opera widget image browser that allows to preview and scale all SVG files from a directory.
- OxygenOffice Draw (OpenOffice.org variant) can open and export SVG files.

- PDFTron's PDF2SVG is a commercial PDF to SVG conversion software. Available as a Command-line application and as an SDK on multiple platforms, including Windows, Mac OS X and Linux.
- The Processing programming language has native support for the SVG format.
- QuickLook is a feature of Mac OS X that can display a variety of files, including SVG.
- RENESIS Player (Examotion) is a free SVG browser for windows. Its Windows Thumbnail Plugin allows the generation of thumbnail previews in any Windows Explorer window.
- Safari is a Web browser made by Apple that can display SVG files.
- Scribus can import SVG files and export separate pages in SVG.
- ShareCAD, free on-line SVG viewer.
- Sheetster Supports creation and display of SVG spreadsheet charts.
- sK1 is a free software vector graphics program for Linux, supports both the import and export of SVG images.
- Sketsa (Kiyut) is a mature commercial SVG editor.
- SVG Kit for Adobe Creative Suite is a plug-in, which adds full-functional support of SVG images to Adobe InDesign and Adobe Photoshop.
- SVG Pony is a commercial converter from SVG to SVG Tiny.
- SVG-edit is a lightweight, web-based, Javascript-driven SVG editor that works in any modern browser.
- Universal Converter is a virtual printer that allows conversion of files to SVG format and also converts SVG files to other file formats.
- ViewerSvg is a SVG to XAML converter. The exported XAML can be used in WPF or Silverlight applications.
- Virtual Mechanics SiteSpinner applications export to SVG.
- Xara Xtreme has an SVG export and import filter in its free/open-source LINUX version only and pay versions for Windows .
- xfig and gnuplot allow export of SVG drawings.
- VectorPen is an OpenSource Project, which allows one to convert the files created on digital notepads to vector based pdf or svg

Mobile support

The most popular implementations for mobile phones are by Ikivo and Bitflash, while for PDAs, Bitflash and Intesis have implementations. Adobe Flash Lite has optionally supported SVG Tiny since version 1.1. At the SVG Open 200 conference, Sun demonstrated a mobile implementation of SVG Tiny 1.1 for the Connected Limited Device Configuration (CLDC) platform.

Mobile SVG players from Ikivo and BitFlash come pre-installed, i.e., the manufacturers burn the SVG player code into their mobiles before shipping to the customers. Mobiles can also include full web browsers (such as Opera Mobile and the iPhone's Safari) that include SVG support.

The level of SVG Tiny support available varies from mobile to mobile, depending on the SVG engine installed. Many newer mobile products support additional features beyond SVG Tiny 1.1, like gradient and opacity; this standard is often referred as SVGT 1.1+. The iPhone, for example, supports declarative animation but not interactivity.

Nokia's S60 platform has built-in support for SVG. For example, icons are generally rendered using the platform's SVG engine. Nokia has also led the JSR 226: Scalable 2D Vector Graphics API expert group which defines Java ME API for SVG presentation and manipulation. This API has been implemented in S60 Platform 3rd Edition Feature Pack 1 onward. Some Series 40 phones also support SVG (such as Nokia 6280).

Most Sony Ericsson phones beginning with K700 (by release date) support SVG Tiny 1.1. Phones beginning with K70 also support such features as opacity and gradients. Phones with Sony Ericsson Java Platform-8 have support for JSR 226.

SVG is also supported on various mobile devices from Motorola, Samsung, LG, and Siemens mobile/BenQ-Siemens.

Hardware rendering

Implementations of SVG that uses the OpenVG API will be hardware accelerated on devices which have hardware acceleration of OpenVG. Examples include Unix-like operating systems with Gallium3D based graphics drivers and several GPUs intended for handheld devices.

Chapter 3

OpenVG and Portable Document Format

OpenVG

OpenVG

Developer(s)	Khronos Group
Stable release	1.1 / December 2008
Operating system	Cross-platform
Type	API
License	Various

OpenVG is a standard API designed for hardware-accelerated 2D vector graphics. It is aimed primarily at mobile phones, media and gaming consoles such as the PlayStation 3, and other consumer electronic devices. It will help manufacturers create flashier user interfaces that are less dependent on energy-hungry CPUs. OpenVG is well suited to accelerating Flash or SVG sequences.

History

The OpenVG group was formed on July 6, 2004 by a selection of major firms including 3Dlabs, Bitboys, Ericsson, Hybrid Graphics, Imagination Technologies, Motorola, Nokia, PalmSource, Symbian, and Sun Microsystems. Other firms including chip manufacturers ATI, LG Electronics, Mitsubishi Electric, NVIDIA, and Texas Instruments and software- and/or IP vendors DMP, Esmertec, ETRI, Falanx Microsystems, Futuremark, HI Corporation, Ikivo, HUONE (formerly MTIS), Superscape, and Wow4M have also

participated in the working group. The first draft specification from the group was made available at the end of 2004, and the 1.0 version of the specification was released on August 1, 200.

On December 9, 2008, the Khronos Group publicly released the OpenVG 1.1 Specification. This latest revision includes glyph rendering for accelerated text, improved anti-aliasing, and Flash support. An updated reference implementation is also provided, as well as a conformance test suite.

On January 16, 2007, Zack Rusin announced the start of an independent Open Source implementation of OpenVG built on top of QtOpenGL.

Shortly after, Ivan Leben has started another Open Source project to implement an ANSI C implementation of the specification purely on top of OpenGL.

Since February 27, 2007 the OpenVG Sample Reference Implementation is available from the Khronos Website under MIT open source license.

On May 1, 2009 Zack Rusin from Tungsten Graphics added OpenVG state tracker to Mesa 3D, which enables SVG vector graphics to be hardware accelerated by any Gallium3D-based driver.

Implementations

In hardware

- AMD/ATI Z160 and Z180 OpenVG 1.x Graphics core. Implemented in the Freescale i.MX3, i.MX1 and i.MX3. IP sold to Qualcomm and rebranded "Adreno" for Snapdragon cores.
- ARM Mali Graphics Processing Units
- Broadcom BCM4760 SoC - Announced 01/June/09. BCM4760 Contains “an OpenGL(R) ES 1.1/OpenVG(TM) 1.0-compliant graphics processor”
- Imagination PowerVR VGX10 OpenVG 1.1
- Imagination PowerVR SGX Series - API support includes OpenGL ES 1.1/2.0, OpenVG 1.1, OpenGL 2.0/3.0 and DirectX9/10.1
- Takumi GV00 - GV300, GV00 Graphics Accelerator IP-Core
- Vivante GC300 series - 2D only (GC300, GC30), (2D+3D (GC00, GC600, GC800, GC1000).

For GPUs

- Mesa Gallium3D from VMware/Tungsten Graphics provides an OpenVG implementations for cards with Gallium drivers (all Intel, nVidia and AMD/ATI, along with VMware virtual GPU have Gallium drivers either working, in development or planned)

For media accelerators

- HuOne AlexVG-forma - Accelerating Vector Graphics API on Media acceleration hardware. Using horse power of Multimedia or 2D bitmap graphics hardware.

On OpenGL ES 1.x

- Hooked OpenVG - Hooked Wireless OpenVG complete implementation of both OpenVG 1.01 and OpenVG 1.1, implemented on top of OpenGL ES, and is fully conformant. Claim "It leverages proprietary technology from Hooked in the areas of tessellation and triangulation for shapes and curve." First released in Feb/2008.
- HuOne AlexVG-forge - An OpenVG engine which is 2D vector graphic standard by using the graphic chip that supports OpenGL/OpenGL ES.
- Mazatech AmanithVG GLE - Built on top of OpenGL 1.x. Claim "achieving better performance than software rasterizers in terms of high resolution animations and complex special effects (transparencies, fading, realtime rotoscaling and many others)."

On OpenGL ES 2.x

- Mazatech AmanithVG GLE - using shaders… not clear… need to find more info
- MonkVG

In software

- HuOne AlexVG - First released in Sep/200
- Khronos OpenVG Reference implementation
- Mazatech AmanithVG SRE - Claim "Thanks to its original polygon rasterization algorithm and dedicated optimized scanline fillers, this engine constitutes the fastest OpenVG software rendering solution available on the market."
- ShivaVG - open source
- Vega - Gallium3D state tracker

Portable Document Format

Portable Document Format

Filename extension	.pdf
Internet media type	application/pdf application/x-pdf application/x-bzpdf application/x-gzpdf
Type code	'PDF ' (including a single space)
Uniform Type Identifier	com.adobe.pdf
Magic number	%PDF
Developed by	Adobe Systems
Initial release	1993
Latest release	1.7
Standard(s)	ISO/IEC 32000-1:2008

Portable Document Format (**PDF**) is an open standard for document exchange. The file format created by Adobe Systems in 1993 is used for representing two-dimensional documents in a manner independent of the application software, hardware, and operating system. Each PDF file encapsulates a complete description of a fixed-layout 2D document that includes the text, fonts, images, and 2D vector graphics which compose the documents. Lately, 3D drawings can be embedded in PDF documents with Acrobat 3D using U3D or PRC and various other data formats.

In 1991 Adobe Systems co-founder John Warnock outlined a system called "Camelot" that evolved into the Portable Document Format (PDF).

Originally a proprietary format, PDF was officially released as an open standard on July 1, 2008, and published by the International Organization for Standardization as ISO/IEC 32000-1:2008.

History

PDF's adoption in the early days of the format's history was slow. Adobe Acrobat, Adobe's suite for reading and creating PDF files, was not freely available; early versions of PDF had no support for external hyperlinks, reducing its usefulness on the Internet; the larger size of a PDF document compared to plain text required longer download times over the slower modems common at the time; and rendering PDF files was slow on the less powerful machines of the day. Additionally, there were competing formats such as Envoy, Common Ground Digital Paper, Farallon Replica and even Adobe's own PostScript format (.ps); in those early years, PDF was popular mainly in desktop publishing workflows. In 199, AT&T Labs commenced work on another electronic document standard targeted at libraries and archives for preserving their books and documents, DjVu. This standard has evolved into the .djv/ .djvu format, which has had growing success and penetration in the online world for eBooks, catalogs, and image-sharing.

Adobe soon started distributing its Acrobat Reader (now Adobe Reader) program at no cost, and continued supporting the original PDF, which eventually became the de facto standard for printable documents on the web (a standard web document).

The PDF specification has changed several times and continues to evolve as new versions of Adobe Acrobat are released. There have been nine versions of PDF with corresponding Acrobat releases:

- (1993) – PDF 1.0 / Acrobat 1.0
- (1994) – PDF 1.1 / Acrobat 2.0
- (1996) – PDF 1.2 / Acrobat 3.0
- (1999) – PDF 1.3 / Acrobat 4.0
- (2001) – PDF 1.4 / Acrobat .0
- (2003) – PDF 1. / Acrobat 6.0
- (200) – PDF 1.6 / Acrobat 7.0
- (2006) – PDF 1.7 / Acrobat 8.0
- (2008) – PDF 1.7, Adobe Extension Level 3 / Acrobat 9.0
- (2009) – PDF 1.7, Adobe Extension Level / Acrobat 9.1

The ISO 32000-1:2008 PDF open standard was published by the ISO on July 1, 2008. PDF is now a published ISO standard, titled *Document management—Portable document format—Part 1: PDF 1.7*

According to the ISO PDF standard abstract:

ISO 32000-1:2008 specifies a digital form for representing electronic documents to enable users to exchange and view electronic documents independent of the environment in which they were created or the environment in which they are viewed or printed. It is intended for the developer of software that creates PDF files (conforming writers), software that reads existing PDF files and interprets their contents for display and

interaction (conforming readers) and PDF products that read and/or write PDF files for a variety of other purposes (conforming products).

A new version named "PDF 2.0" is currently under development (ISO/NP 32000-2 - Document management—Portable document format—Part 2: PDF 2.0).

Technical foundations

Anyone may create applications that can read and write PDF files without having to pay royalties to Adobe Systems; Adobe holds patents to PDF, but licenses them for royalty-free use in developing software complying with its PDF specification.

The PDF combines three technologies:

- A subset of the PostScript page description programming language, for generating the layout and graphics.
- A font-embedding/replacement system to allow fonts to travel with the documents.
- A structured storage system to bundle these elements and any associated content into a single file, with data compression where appropriate.

PostScript

PostScript is a page description language run in an interpreter to generate an image, a process requiring many resources. It can handle not just graphics, but standard features of programming languages such as `if` and `loop` commands. PDF is largely based on PostScript but simplified to remove flow control features like these, while graphics commands such as `lineto` remain.

Often, the PostScript-like PDF code is generated from a source PostScript file. The graphics commands that are output by the PostScript code are collected and tokenized; any files, graphics, or fonts to which the document refers also are collected; then, everything is compressed to a single file. Therefore, the entire PostScript world (fonts, layout, measurements) remains intact.

As a document format, PDF has several advantages over PostScript:

- PDF contains tokenized and interpreted results of the PostScript source code, for direct correspondence between changes to items in the PDF page description and changes to the resulting page appearance.
- PDF (from version 1.4) supports true graphic transparency; PostScript does not.
- PostScript is an imperative programming language with an implicit global state, so instructions accompanying the description of one page can affect the appearance of any following page. Therefore, all preceding pages in a PostScript document must be processed in order to determine the correct appearance of a given page, whereas each page in a PDF document is unaffected by the others. As

a result, PDF viewers allow the user to quickly jump to the final pages of a long document, whereas a Postscript viewer needs to process all pages sequentially before being able to display the destination page (unless the optional PostScript Document Structuring Conventions have been carefully complied with).

Technical overview

File structure

A PDF file consists primarily of *objects*, of which there are eight types:

- Boolean values, representing *true* or *false*
- Numbers
- Strings
- Names
- Arrays, ordered collections of objects
- Dictionaries, collections of objects indexed by Names
- Streams, usually containing large amounts of data
- The null object

Objects may be either *direct* (embedded in another object) or *indirect*. Indirect objects are numbered with an *object number* and a *generation number*. An index table called the *xref table* gives the byte offset of each indirect object from the start of the file. This design allows for efficient random access to the objects in the file, and also allows for small changes to be made without rewriting the entire file (*incremental update*). Beginning with PDF version 1, indirect objects may also be located in special streams known as *object streams*. This technique reduces the size of files that have large numbers of small indirect objects and is especially useful for *Tagged PDF*.

There are two layouts to the PDF files—non-linear (not "optimized") and linear ("optimized"). Non-linear PDF files consume less disk space than their linear counterparts, though they are slower to access because portions of the data required to assemble pages of the document are scattered throughout the PDF file. Linear PDF files (also called "optimized" or "web optimized" PDF files) are constructed in a manner that enables them to be read in a Web browser plugin without waiting for the entire file to download, since they are written to disk in a linear (as in page order) fashion. PDF files may be optimized using Adobe Acrobat software or QPDF.

Imaging model

The basic design of how graphics are represented in PDF is very similar to that of PostScript, except for the use of transparency, which was added in PDF 1.4.

PDF graphics use a device independent Cartesian coordinate system to describe the surface of a page. A PDF page description can use a matrix to scale, rotate, or skew graphical elements. A key concept in PDF is that of the *graphics state*, which is a

collection of graphical parameters that may be changed, saved, and restored by a *page description*. PDF has (as of version 1.6) 24 graphics state properties, of which some of the most important are:

- The *current transformation matrix* (CTM), which determines the coordinate system
- The *clipping path*
- The *color space*
- The *alpha constant*, which is a key component of transparency

Vector graphics

Vector graphics in PDF, as in PostScript, are constructed with *paths*. Paths are usually composed of lines and cubic Bézier curves, but can also be constructed from the outlines of text. Unlike PostScript, PDF does not allow a single path to mix text outlines with lines and curves. Paths can be stroked, filled, or used for clipping. Strokes and fills can use any color set in the graphics state, including *patterns*.

PDF supports several types of patterns. The simplest is the *tiling pattern* in which a piece of artwork is specified to be drawn repeatedly. This may be a *colored tiling pattern*, with the colors specified in the pattern object, or an *uncolored tiling pattern*, which defers color specification to the time the pattern is drawn. Beginning with PDF 1.3 there is also a *shading pattern*, which draws continuously varying colors. There are seven types of shading pattern of which the simplest are the *axial shade* (Type 2) and *radial shade* (Type 3).

Raster images

Raster images in PDF (called *Image XObjects*) are represented by dictionaries with an associated stream. The dictionary describes properties of the image, and the stream contains the image data. (Less commonly, a raster image may be embedded directly in a page description as an *inline image*.) Images are typically *filtered* for compression purposes. Image filters supported in PDF include the general purpose filters

- **ASCII8Decode** a deprecated filter used to put the stream into 7-bit ASCII
- **ASCIIHexDecode** similar to ASCII8Decode but less compact
- **FlateDecode** a commonly used filter based on the DEFLATE or Zip algorithm
- **LZWDecode** a deprecated filter based on LZW Compression
- **RunLengthDecode** a simple compression method for streams with repetitive data using the Run-length encoding algorithm

and the image-specific filters

- **DCTDecode** a lossy filter based on the JPEG standard
- **CCITTFaxDecode** a lossless filter based on the CCITT fax compression standard

- **JBIG2Decode** a lossy or lossless filter based on the JBIG2 standard, introduced in PDF 1.4
- **JPXDecode** a lossy or lossless filter based on the JPEG 2000 standard, introduced in PDF 1.

Normally all image content in a PDF is embedded in the file. But PDF allows image data to be stored in external files by the use of *external streams* or *Alternate Images*. Standardized subsets of PDF, including PDF/A and PDF/X, prohibit these techniques.

Text

Text in PDF is represented by *text elements* in page content streams. A text element specifies that *characters* should be drawn at certain positions. The characters are specified using the *encoding* of a selected *font resource*.

Fonts

A font object in PDF is a description of a digital typeface. It may either describe the characteristics of a typeface, or it may include an embedded *font file*. The latter case is called an *embedded font* while the former is called an *unembedded font*. The font files that may be embedded are based on widely used standard digital font formats: **Type 1** (and its compressed variant **CFF**), **TrueType**, and (beginning with PDF 1.6) **OpenType**. Additionally PDF supports the **Type 3** variant in which the components of the font are described by PDF graphic operators.

Encodings

Within text strings, characters are shown using *character codes* (integers) that map to glyphs in the current font using an *encoding*. There are a number of predefined encodings, including *WinAnsi*, *MacRoman*, and a large number of encodings for East Asian languages, and a font can have its own built-in encoding. (Although the WinAnsi and MacRoman encodings are derived from the historical properties of the Windows and Macintosh operating systems, fonts using these encodings work equally well on any platform.) PDF can specify a predefined encoding to use, the font's built-in encoding or provide a lookup table of differences to a predefined or built-in encoding (not recommended with TrueType fonts). The encoding mechanisms in PDF were designed for Type 1 fonts, and the rules for applying them to TrueType fonts are complex.

For large fonts or fonts with non-standard glyphs, the special encodings *Identity-H* (for horizontal writing) and *Identity-V* (for vertical) are used. With such fonts it is necessary to provide a *ToUnicode* table if semantic information about the characters is to be preserved.

Transparency

The original imaging model of PDF was, like PostScript's, *opaque*: each object drawn on the page completely replaced anything previously marked in the same location. In PDF 1.4 the imaging model was extended to allow transparency. When transparency is used, new objects interact with previously marked objects to produce blending effects. The addition of transparency to PDF was done by means of new extensions that were designed to be ignored in products written to the PDF 1.3 and earlier specifications. As a result, files that use a small amount of transparency might view acceptably in older viewers, but files making extensive use of transparency could view completely wrongly in an older viewer without warning.

The transparency extensions are based on the key concepts of *transparency groups*, *blending modes*, *shape*, and *alpha*. The model is closely aligned with the features of Adobe Illustrator version 9. The blend modes were based on those used by Adobe Photoshop at the time. When the PDF 1.4 specification was published the formulas for calculating blend modes were kept secret by Adobe. They have since been published.

The concept of a transparency group in PDF specification is independent of existing notions of "group" or "layer" in applications such as Adobe Illustrator. Those groupings reflect logical relationships among objects that are meaningful when editing those objects, but they are not part of the imaging model.

Interactive elements

PDF files may contain interactive elements such as annotations and form fields.

Interactive Forms is a mechanism to add forms to the PDF file format.

PDF currently supports two different methods for integrating data and PDF forms. Both formats today coexist in PDF specification:

- **AcroForms** (also known as **Acrobat forms**), introduced in the PDF 1.2 format specification and included in all later PDF specifications.
- **Adobe XML Forms Architecture (XFA)** forms, introduced in the PDF 1. format specification. The XFA specification is not included in the PDF specification, it is only referenced as an optional feature. Adobe XFA Forms are not compatible with AcroForms.

AcroForms

AcroForms were introduced in the PDF 1.2 format. AcroForms permit using objects (text boxes, radiobuttons, etc.) and some code (JavaScript).

Alongside the standard PDF action types, Interactive forms (AcroForms) support submitting, resetting, and importing data. Submit action transmits the names and values

of selected interactive form fields to a specified uniform resource locator (URL). Interactive form field names and values may be submitted in any of the following formats, (depending on the settings of the action's ExportFormat, SubmitPDF, and XFDF flags):

- HTML Form format (HTML 4.01 Specification since PDF 1.; HTML 2.0 since 1.2)
- Forms Data Format (FDF)
- XML Forms Data Format (XFDF) (external XML Forms Data Format Specification, Version 2.0; supported since PDF 1.; it replaced the "XML" form submission format defined in PDF 1.4.)
- PDF (the entire document can be submitted rather than individual fields and values). (defined in PDF 1.4)

AcroForms can keep form field values in external stand-alone files containing key:value pairs. The external files may use Forms Data Format (FDF) and XML Forms Data Format (XFDF) files. The usage rights (UR) signatures define rights for import form data files in FDF, XFDF and text (CSV/TSV) formats, and export form data files in FDF and XFDF formats.

Forms Data Format (FDF)

Forms Data Format (FDF)

Filename extension	`.fdf`
Internet media type	`application/vnd.fdf`
Type code	'FDF '
Developed by	Adobe Systems
Initial release	1996 (PDF 1.2)
Extended from	PDF
Extended to	XFDF
Standard(s)	ISO/IEC 32000-1:2008

The Forms Data Format (FDF) is based on PDF, it uses the same syntax and has essentially the same file structure, but is much simpler than PDF, since the body of an FDF document consists of only one required object. Forms Data Format is defined in PDF format specification (since PDF 1.2). The Forms Data Format can be used when submitting form data to a server, receiving the response, and incorporating into the interactive form. It can also be used to export form data to stand-alone files that can be imported back into the corresponding PDF interactive form. Beginning in PDF 1.3, FDF can be used to define a container for annotations that are separate from the PDF

document to which they apply. FDF is typically used to encapsulate information such as X.09 certificates, requests for certificates, directory settings, timestamp server settings, and embedded PDF files for network transmission. The FDF uses the MIME content type application/vnd.fdf, filename extension .fdf and on Mac OS it uses file type 'FDF '. A support for importing and exporting FDF stand-alone files is not widely implemented in free or freeware PDF software. For example, there is no support in Evince, Okular, KPDF or Sumatra PDF. Import support for stand-alone FDF files is implemented in Adobe Reader; export and import support (including saving of FDF data in PDF) is for example implemented in Foxit Reader and PDF-XChange Viewer Free; saving of FDF data in a PDF file is also supported in pdftk.

XML Forms Data Format (XFDF)

XML Forms Data Format (XFDF)

Filename extension	`.xfdf`
Internet media type	`application/vnd.adobe.xfdf`
Type code	'XFDF'
Developed by	Adobe Systems
Latest release	2.0
Extended from	PDF, FDF, XML

XML Forms Data Format (XFDF) is the XML version of Forms Data Format, but the XFDF implements only a subset of FDF containing forms and annotations. There are not XFDF equivalents for some entries in the FDF dictionary - such as the Status, Encoding, JavaScript, Pages keys, EmbeddedFDFs, Differences and Target. The XFDF specification is referenced (but not included) in PDF 1. specification (and in later versions). It is described separately in *XML Forms Data Format Specification*. The PDF 1.4 specification allowed form submissions in XML format, but this was replaced by submissions in XFDF format in PDF 1. specification. XFDF conforms to the XML standard. XFDF can be used the same way as FDF - e.g. form data is submitted to a server, modifications are made, then sent back and the new form data is imported in an interactive form. It can also be used to export form data to stand-alone files that can be imported back into the corresponding PDF interactive form. A support for importing and exporting FDF stand-alone files is not widely implemented in free or freeware PDF software. Import of XFDF is implemented in Adobe Reader and later versions; import and export is implemented in PDF-XChange Viewer Free; embedding of XFDF data in PDF form is implemented in pdftk (pdf toolkit).

Adobe XML Forms Architecture (XFA)

In the PDF 1. format, Adobe Systems introduced a new, proprietary format for forms, namely Adobe XML Forms Architecture (XFA) forms. The XFA 2.02 is referenced in the PDF 1. specification (and also in later versions) but is described separately in *Adobe XML Forms Architecture (XFA) Specification*, which has several versions. Adobe XFA Forms are not compatible with AcroForms. Creating XFA Forms for use in Adobe Reader requires Adobe LiveCycle Forms Designer or Avoka SmartForm Composer. Adobe Reader contains "disabled features" for use of XFA Forms, that will activate only when opening a PDF document that was created using enabling technology available only from Adobe. The XFA Forms are not compatible with Adobe Reader prior to version 6.

XFA forms can be created and used as PDF files or as XDP (XML Data Package) files. The format of an XFA resource in PDF is described by the XML Data Package Specification. The XDP may be a standalone document or it may in turn be carried inside a PDF document. XDP provides a mechanism for packaging form components within a surrounding XML container. An XDP can also package a PDF file, along with XML form and template data. PDF may contain XFA (in XDP format), but also XFA may contain PDF. When the XFA (XML Forms Architecture) grammars used for an XFA form are moved from one application to another, they must be packaged as an XML Data Package.

When the PDF and XFA are combined, the result is a form in which each page of the XFA form overlays a PDF background. This architecture is sometimes referred to as XFAF (XFA Foreground). The alternative is to express all of the form, including boilerplate, directly in XFA. It is sometimes called *full* XFA.

Starting with PDF 1., the text contents of variable text form fields, as well as markup annotations may include formatting information (style information). These rich text strings are XML documents that conform to the rich text conventions specified for the XML Forms Architecture specification 2.02, which is itself a subset of the XHTML 1.0 specification, augmented with a restricted set of CSS2 style attributes. In PDF 1.6, PDF supports the rich text elements and attributes specified in the XML Forms Architecture (XFA) Specification, 2.2. In PDF 1.7, PDF supports the rich text elements and attributes specified in the XML Forms Architecture (XFA) Specification, 2.4

Logical structure and accessibility

A PDF may contain structure information to enable better text extraction and accessibility. When published, PDF/UA, now ISO/AWI 14289, will provide definitive information on how the contents of PDF files are to be tagged with accurate structure information.

Security and signatures

A PDF file may be encrypted for security, or digitally signed for authentication.

The standard security provided by Acrobat PDF consists of two different methods and two different passwords, "user password" and "owner password". A PDF document may be protected by password to open ('user' password) and the document may also specify operations that should be restricted even when the document is decrypted: printing; copying text and graphics out of the document; modifying the document; and adding or modifying text notes and AcroForm fields (using 'owner' password). However, all operations (except the document open password protection, if applicable) which are restricted by "owner" or "user" passwords are trivially circumvented by many commonly available "PDF cracking" software and even freely online, and thus these restrictions are obviously ineffective in letting the author control what can and cannot be done with the pdf file he or she created, once it is distributed. This warning is also displayed when applying such restrictions using Adobe Acrobat software to create or edit PDF files.

Even without removing the password, most freeware or open source PDF readers will ignore the digital rights management "protections" and will allow the user to print or make copy of excerpts of the text as if the document were not limited by password protection.

Usage rights

Beginning with PDF 1., Usage rights (UR) signatures are used to enable additional interactive features that are not available by default in a particular PDF viewer application. The signature is used to validate that the permissions have been granted by a bonafide granting authority. For example, it can be used to allow a user:

- to save the PDF document along with modified form and/or annotation data
- import form data files in FDF, XFDF and text (CSV/TSV) formats
- export form data files in FDF and XFDF formats
- submit form data
- instantiate new pages from named page templates
- apply a digital signature to existing digital signature form field
- create, delete, modify, copy, import, export annotations

For example, Adobe Systems grants permissions to enable additional features in Adobe Reader, using public-key cryptography. Adobe Reader will verify that the signature uses a certificate from an Adobe-authorized certificate authority. The PDF 1. specification declares that other PDF viewer applications are free to use this same mechanism for their own purposes.

File attachments

PDF files can have document-level and page-level file attachments, which the reader can access and open or save to their local filesystem. PDF attachments can be added to existing PDF files for example using pdftk. Adobe Reader provides support for attachments, and poppler based readers like Evince or Okular also have some support for document-level attachments.

Metadata

PDF files can contain two types of metadata. The first is the Document Information Dictionary, a set of key/value fields such as author, title, subject, creation and update dates. This is stored in the optional Info trailer of the file. A small set of fields is defined, and can be extended with additional text values if required.

Later, in PDF 1.4, support was added for the Metadata Streams, using the Extensible Metadata Platform (XMP) to add XML standards-based extensible metadata as used in other file formats. This allows metadata to be attached to any stream in the document, such as information about embedded illustrations, as well as the whole document (attaching to the document catalog), using an extensible schema.

Subsets

Proper subsets of PDF have been, or are being, standardized under ISO for several constituencies:

- PDF/X for the printing and graphic arts as ISO 1930 (working in ISO TC130)
- PDF/A for archiving in corporate/government/library/etc environments as ISO 1900 (work done in ISO TC171)
- PDF/E for exchange of engineering drawings (work done in ISO TC171)
- PDF/UA for universally accessible PDF files

A PDF/H variant (PDF for Healthcare) is being developed. However, it may consist more of a set of "best practices" than of a specific format or subset.

Mars

Adobe is exploring an XML-based next-generation PDF code named Mars.

The format of graphic elements of Mars is sometimes described simply as "SVG", but according to the version 0.8 draft specification of November 2007 (§3 Mars SVG Support) the format is actually merely similar to SVG: it contains both additions to and subtractions from SVG, so it is in general neither viewable by nor creatable with standard SVG tools: some things will look noticeably different between SVG viewers and Mars viewers.

Technical issues

Accessibility

PDF files can be created specifically to be accessible for disabled people. Current PDF file formats can include tags (XML), text equivalents, captions, audio descriptions, et cetera. Some software can automatically produce tagged PDFs, however this feature is not always enabled by default. Leading screen readers, including JAWS, Window-Eyes,

Hal, and Kurzweil 1000 and 3000 can read tagged PDFs; current versions of the Acrobat and Acrobat Reader programs can also read PDFs aloud. Moreover, tagged PDFs can be re-flowed and magnified for readers with visual impairments. Problems remain with adding tags to older PDFs and those that are generated from scanned documents. In these cases, accessibility tags and re-flowing are unavailable, and must be created either manually or with OCR techniques. These processes are inaccessible to some disabled people. PDF/UA, the PDF/Universal Accessibility Committee, an activity of AIIM, is working on a specification for PDF accessibility based on ISO 32000.

One of the significant challenges with PDF accessibility is that PDF documents have three distinct views, which, depending on the document's creation, can be inconsistent with each other. The three views are (i) the physical view, (ii) the tags view, and (iii) the content view. The physical view is displayed and printed (what most people consider a PDF document). The tags view is what screen readers read (useful for people with poor eyesight). The content view is displayed when the document is re-flowed to Acrobat (useful for people with mobility disability). For a PDF document to be accessible, the three views must be consistent with each other.

Security

PDF attachments carrying viruses were first discovered in 2001. This virus, named "OUTLOOK.PDFWorm" or "Peachy", uses Microsoft Outlook to send itself as an attachment to an Adobe PDF file. It was activated with Adobe Acrobat, but not with Acrobat Reader.

From time to time, new vulnerabilities are discovered in various versions of Adobe Reader, prompting the company to issue security fixes. One aggravating factor is that Adobe Reader can be configured to start automatically if a web page has an embedded PDF file, providing a vector for attack. If a malicious web page contains an infected PDF file that takes advantage of a vulnerability in Adobe Reader, the system may be compromised even if the browser is secure.

On March 30, 2010 security researcher Didier Stevens reported an Adobe Reader and Foxit Reader "exploit". The exploit results in an arbitrary executable being run if the user accepts a "launch executable" warning prompt.

Usage restrictions and monitoring

PDFs may be encrypted so that a password is needed to view or edit the contents. The PDF Reference defines both 40-bit and 128-bit encryption, both making use of a complex system of RC4 and MD. The PDF Reference also defines ways in which third parties can define their own encryption systems for use in PDF.

PDF files may also contain embedded DRM restrictions that provide further controls that limit copying, editing or printing. The restrictions on copying, editing, or printing depend on the reader software to obey them, so the security they provide is limited.

The PDF Reference has technical details. Like HTML files, PDF files may submit information to a web server. This could be used to track the IP address of the client PC, a process known as phoning home. After update 7.0. to Acrobat Reader, the user will be notified "via a dialogue box that the author of the file is auditing usage of the file, and be offered the option of continuing."

Through its LiveCycle Policy Server product, Adobe provides a method to set security policies on specific documents. This can include requiring a user to authenticate and limiting the timeframe a document can be accessed or amount of time a document can be opened while offline. Once a PDF document is tied to a policy server and a specific policy, that policy can be changed or revoked by the owner. This controls documents that are otherwise "in the wild." Each document open and close event can also be tracked by the policy server. Policy servers can be set up privately or Adobe offers a public service through Adobe Online Services.

Default display settings

PDF documents can contain display settings, including the page display layout and zoom level. Adobe Reader will use these settings to override the user's default settings when opening the document. The free Adobe Reader cannot remove these settings.

Content

A PDF file is often a combination of vector graphics, text, and raster graphics. The basic types of content in a PDF are:

- text stored as such
- vector graphics for illustrations and designs that consist of shapes and lines
- raster graphics for photographs and other types of image

In later PDF revisions, a PDF document can also support links (inside document or web page), forms, JavaScript (initially available as plugin for Acrobat 3.0), or any other types of embedded contents that can be handled using plug-ins.

PDF 1.6 supports interactive 3D documents embedded in the PDF - 3D drawings can be embedded using U3D or PRC and various other data formats.

Two PDF files that look similar on a computer screen may be of very different sizes. For example, a high resolution raster image takes more space than a low resolution one. Typically higher resolution is needed for printing documents than for displaying them on screen. Other things that may increase the size of a file is embedding full fonts, especially for Asiatic scripts, and storing text as graphics.

Standard Type 1 Fonts

There are fourteen typefaces that have a special significance to PDF documents:

- Times (v3) (in regular, italic, bold, and bold italic)
- Courier (in regular, oblique, bold and bold oblique)
- Helvetica (v3) (in regular, oblique, bold and bold oblique)
- Symbol
- Zapf Dingbats

These fonts, sometimes referred to as the "base fourteen fonts" should always be present (actually present or a close substitute) and so need not be embedded in a PDF. PDF viewers must know about the metrics of these fonts. Other fonts may be substituted if they are not embedded in a PDF.

Versions

Version	Year of publication	New features	Supported by Reader version
1.0	1993		Acrobat Reader (Carousel)
1.1	1996	Passwords, encryption (MD, RC4 40bit), device-independent color, threads and links	Acrobat Reader 2.0
1.2	1996	Interactive page elements (such as radio buttons and checkboxes); interactive, fill-in forms (AcroForm); Forms Data Format (FDF) for interactive form data that can be imported, exported, transmitted and received from the Web; mouse events; support for playing movies from external files; support for playing sounds, either embedded in the PDF file or from external files ; Unicode; advanced color features and image proxying	Acrobat Reader 3.0
1.3	2000	Digital signatures; ICC and DeviceN color spaces; JavaScript actions; Embedded file streams - embedding files of any type directly within the body of the PDF file itself (e.g. used for attachments); many new annotation types; new features of the Adobe imaging model embodied in PostScript LanguageLevel 3; Masked images; Alternate representations for a single image; Smooth shading; Enhanced page numbering; Web Capture - a facility for capturing information from World Wide Web and converting it to PDF;	Acrobat Reader 4.0

		Logical structure - a facility for representating logical structure independently of its graphical structure; additional support for CIDFonts; data structures for mapping strings and numbers to PDF objects; Prepress Support - information useful in prepress production workflows; new functions - for several types of function object that represent parameterized classes of functions	
1.4	2001	JBIG2; transparency; RC4 encryption key lengths greater than 40bits (40-128bits); enhancements to interactive forms and Forms Data Format (FDF), support for XML form submissions, embedded FDF files, Unicode specification of field export values, support for remote collaboration and digital signatures in FDF files; support for accessibility to disabled users; metadata streams using XML - Extensible Metadata Platform (XMP); Tagged PDF ;Facilities for including printer's marks; Support for the display and preview of production-related page boundaries; New predefined CMaps; Alternate Presentations - alternate ways in which the document may be viewed; The ability to import content from one PDF document into another; EmbeddedFiles entry in the PDF document's name dictionary - a standard location for the embedded data; OCR text layer	Acrobat Reader .0
1.	2003	JPEG 2000; enhanced support for embedding and playback of multimedia; object streams; cross reference streams; XML Forms Data Format (XFDF) for interactive form submission (replaced the XML format in PDF 1.4); support for forms, rich text elements and attributes based on Adobe's XML Forms Architecture (XFA) 2.02; public-key security handlers using PKCS#7 (introduced in PDF 1.3 but not documented in the PDF Reference until PDF 1.), public-key encryption, permissions - usage rights (UR) signatures (does not require document encryption), PKCS#7 with SHA-1, RSA up to 4096-bits; security handler can use its own encryption and decryption algorithms; Optional Content - sections of content in a PDF document that can be selectively viewed or hidden by document authors or consumers - for items such as CAD drawings, layered artwork, maps, and	Adobe Reader 6.0

		multi-language documents; Alternate Presentations - the only type is slideshow - invoked by means of JavaScript actions (Adobe Reader supports only SVG 1.0) Support for Windows 98 dropped. To view and print newer version PDFs, such as those at the IRS website, with older versions of Reader requires downloading in Google Docs "Quick View" simplified PDF format.	
1.6	2004	3D artwork, e.g. support for Universal 3D file format; OpenType font embedding; support for XFA 2.2 rich text elements and attributes; AES encryption; PKCS#7 with SHA26, DSA up to 4096-bits; NChannel color spaces; additional support for embedded file attachments, including cross-document linking to and from embedded files; enhancements and clarifications to digital signatures related to usage rights and modification detection and prevention signatures	Adobe Reader 7.0
1.7 (ISO/IEC 32000-1:2008)	2006	increased presentation of 3D artwork; support for XFA 2.4 rich text elements and attributes; presentation of multiple file attachments (portable collections); document requirements for a PDF consumer application; new string types - PDFDocEncoded string, ASCII string, byte string; PKCS#7 with SHA384, SHA12 and RIPEMD160	Adobe Reader 8
1.7 Extension Level 3	2008	26-bit AES encryption; incorporation of XFA Datasets into a file conforming PDF/A-2; RichMedia annotations - attaching Flash applications, video (including Flash video with H.264), audio, and other multimedia with expanded functionality, two-way scripting bridge between Flash and a conforming application; support for the rich text conventions described in XFA 2. and 2.6	Adobe Reader 9
1.7 Extension Level	2009	XFA 3.0	Adobe Reader 9.1

Implementations

PDF-viewing software is generally provided free of charge, including versions by Adobe Reader, Foxit Reader, PDF-XChange Viewer, Sorax Reader and others.

There are many software options for creating PDFs, including the PDF printing capabilities built in to Mac OS X and some Linux distributions, the multi-platform OpenOffice.org, Microsoft Office 2007 (if updated to SP2), WordPerfect since version 9, Free PDF XP and numerous PDF print drivers for Microsoft Windows, the pdfTeX typesetting system, the DocBook PDF tools, applications developed around Ghostscript and Adobe Acrobat itself. Google's online office suite Google Docs also allows for uploading, and saving to the Portable Document Format.

Raster image processors (RIPs) are used to convert PDF files into a raster format suitable for imaging onto paper and other media in printers, digital production presses and prepress in a process known as rasterisation. RIPs capable of processing PDF directly include the Adobe PDF Print Engine from Adobe Systems and Jaws and the Harlequin RIP from Global Graphics.

Editing PDFs (structure)

There is also specialized software for editing PDF files, though the choices are much more limited and often expensive. As of version 0.46, Inkscape also allows PDF editing through an intermediate translation step involving Poppler. A PDF editor most similar to word processor is called Infix PDF Editor.

Annotating PDFs

Adobe Acrobat is one example of proprietary software that allows the user to annotate, highlight, add notes to already created PDF files. One UNIX application available as free software (under the GNU General Public License) is PDFedit. Another GPL-licensed application native to the Linux environment is Xournal. Xournal allows for annotating in different fonts and colours, as well as a rule for quickly underlining and highlighting lines of text or paragraphs. Xournal also has a shape recognition tool for squares, rectangles and circles. In Xournal annotations may be moved, copied and pasted. The freeware Foxit Reader allows annotating but adds a watermark on each annotated page. The commercial version of the package does not have this limitation. Tracker Software's PDF-XChange Viewer allows annotations and markups without restrictions in its freeware alternative. Apple's Mac OS X's integrated PDF viewer, Preview, does also enable annotations. For mobile annotation, iAnnotate PDF for the iPad and Aji Annotate for the iPhone, both produced by Aji, allow annotation of PDFs as well as exporting summaries of the annotations.

There are also web annotation systems which allow to annotate pdf and other documents formats, e.g. A.nnotate, crocodoc, WebNotes.

In cases where PDFs are expected to have all of the functionality of paper documents, ink annotation is required. Many programs which accept ink input from the mouse are not responsive enough for handwriting using an input tablet or tablet PC. Existing solutions on the PC include Bluebeam PDF Revu, and PDF Annotator.

Other applications and functionalities

Several applications embracing the PDF standard are now available as an online service including Scribd for viewing and storing, Pdfvue for online editing, and Zamzar for PDF Conversion.

In 1993 the Jaws RIP from Global Graphics became the first shipping prepress RIP that interpreted PDF natively without conversion to another format. The company released an upgrade to their Harlequin RIP with the same capability in 1997.

Agfa-Gevaert introduced and shipped Apogee, the first prepress workflow system based on PDF, in 1997.

Many commercial offset printers have accepted the submission of press-ready PDF files as a print source, specifically the PDF/X-1a subset and variations of the same. The submission of press-ready PDF files are a replacement for the problematic need for receiving collected native working files.

PDF was selected as the "native" metafile format for Mac OS X, replacing the PICT format of the earlier Mac OS. The imaging model of the Quartz graphics layer is based on the model common to Display PostScript and PDF, leading to the nickname "Display PDF". The Preview application can display PDF files, as can version 2.0 and later of the Safari web browser. System-level support for PDF allows Mac OS X applications to create PDF documents automatically, provided they support the Print command. The files are then exported in PDF 1.3 format according to the file header. When taking a screenshot under Mac OS X versions 10.0 through 10.3, the image was also captured as a PDF; in 10.4 and 10. the default behaviour is set to capture as a PNG file, though this behaviour can be set back to PDF if required.

Some desktop printers also support direct PDF printing, which can interpret PDF data without external help. Currently, all PDF capable printers also support PostScript, but most PostScript printers do not support direct PDF printing.

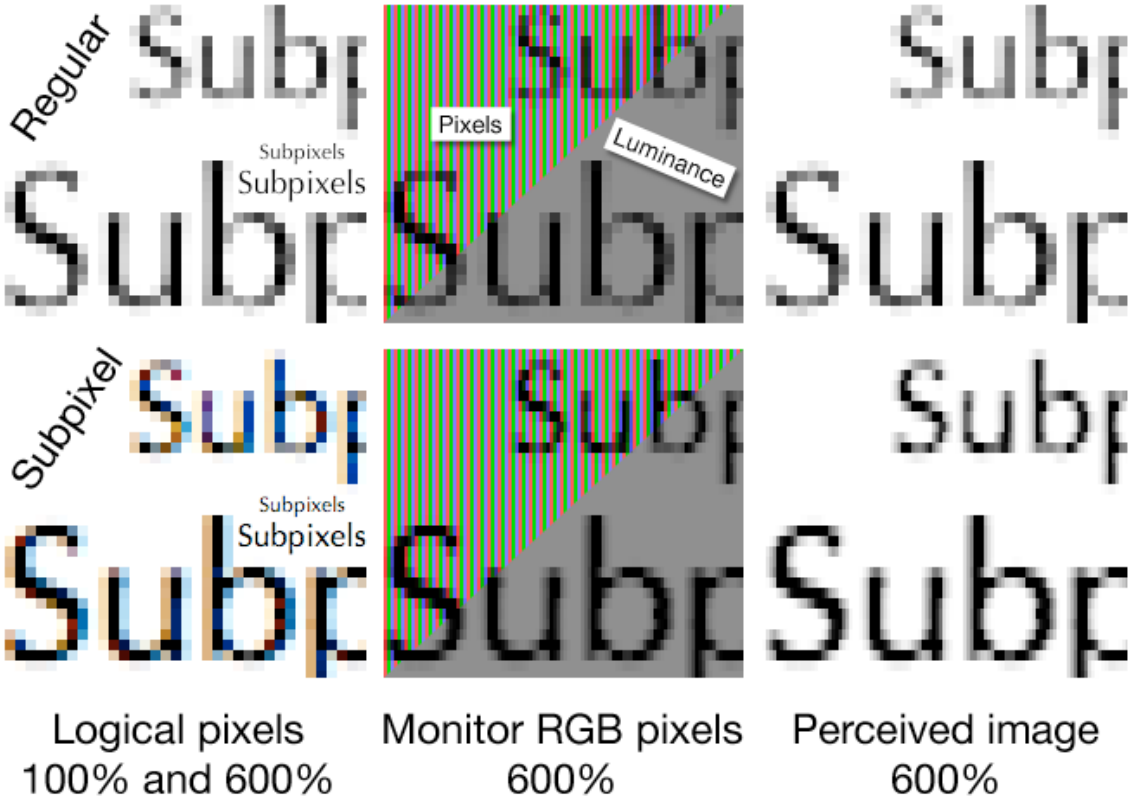
The Free Software Foundation considers one of their high priority projects to be "developing a free, high-quality and fully functional set of libraries and programs that implement the PDF file format and associated technologies to the ISO 32000 standard." The GNUpdf library has, however, not been released yet, while Poppler has enjoyed wider use in applications such as Evince, which comes with the GNOME desktop environment, at the expense of relying on the GPLv2-licensed Xpdf code base that can't be used by GPLv3 programs. There are also commercial development libraries available as listed in List of PDF software.

The Apache PDFBox project of the Apache Software Foundation is an open source Java library for working with PDF documents. PDFBox is licensed under the Apache License.

Chapter 4

Subpixel Rendering and Vector Markup Language

Subpixel rendering

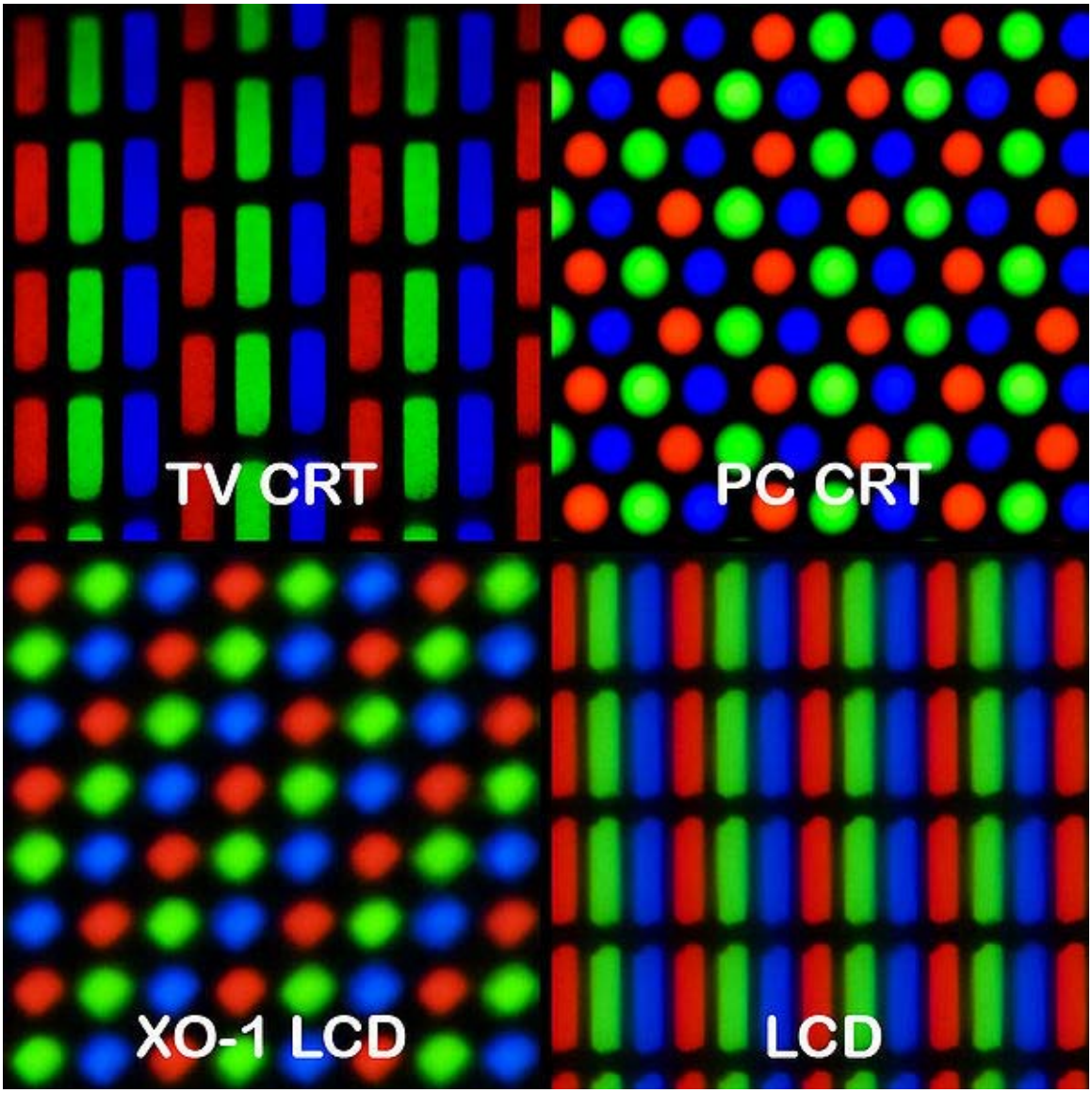


Subpixel rendering works by increasing the luminance reconstruction points of a color subpixelated screen, such as a liquid crystal display (LCD) or organic light-emitting diode (OLED) display. This thumbnail image is downsized and does not show the technique.

Subpixel rendering is a way to increase the apparent resolution of a computer's liquid crystal display (LCD) or Organic Light Emitting Diode display by rendering pixels to

take into account the screen type's physical properties. It takes advantage of the fact that each pixel on a color LCD is actually composed of individual red, green, and blue subpixel stripes to anti-alias text with greater detail or to increase the resolution of all image types on layouts which are specifically designed to be compatible with subpixel rendering.

Background



Examples of pixel geometry, showing various arrangements of pixels and subpixels, which must be considered for subpixel rendering. LCD displays (bottom right is the most typical example) are best suited to subpixel rendering.

"Aa" rendered in subpixel

The previous image, with the R, G and B channel separated

A single pixel on a color subpixelated display is made of several color primaries, typically three colored elements—ordered (on various displays) either as blue, green, and red (BGR), or as red, green, and blue (RGB). Some displays have more than three primaries, often called MultiPrimary, such as the combination of red, green, blue, and yellow (RGBY), or red, green, blue, and white (RGBW), or even red, green, blue, yellow, and cyan (RGBYC).

These pixel components, sometimes called sub-pixels, appear as a single color to the human eye because of blurring by the optics and spatial integration by nerve cells in the eye. The components are easily visible, however, when viewed with a small magnifying glass, such as a loupe. Over a certain resolution threshold the colors in the sub-pixels are not visible, but the relative intensity of the components shifts the apparent position or orientation of a line.

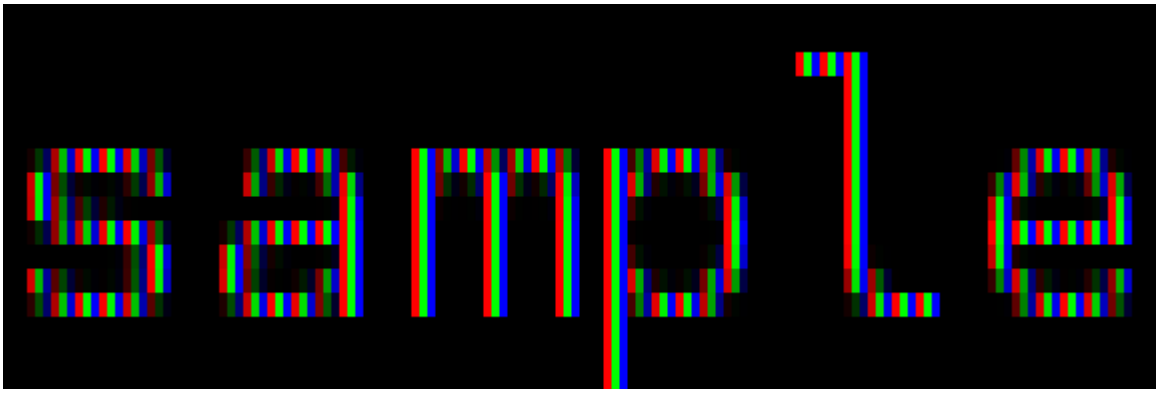
Methods that take this interaction between the display technology and the human visual system into account are called subpixel rendering algorithms. The resolution at which colored sub-pixels go unnoticed differs, however, with each user—some users are distracted by the colored "fringes" resulting from sub-pixel rendering.

Subpixel rendering is better suited to some display technologies than others. The technology is well-suited to LCDs and other technologies where each logical pixel corresponds directly to three or more independent colored sub-pixels, but less so for CRTs.

In a CRT the light from the pixel components often spreads across pixels, and the outputs of adjacent pixels are not perfectly independent. If a designer knew precisely about the

display's electron beams and aperture grille, subpixel rendering might have some advantage. But the properties of the CRT components, coupled with the alignment variations that are part of the production process, make subpixel rendering less effective for these displays.

The technique should have good application to organic light emitting diodes and other display technologies that organize pixels the same way as LCDs.



Pixels on an LCD are made up of separate red, green, and blue elements, which can be used to give finer control over rendering the curvature of text. The word would appear white on the screen because red, green, and blue light combined are indistinguishable from white light to the human visual system.

History

Originally invented by IBM in 1988, subpixel rendering was first brought to public attention by Microsoft in Windows XP as ClearType, but it was not activated by default until Windows Vista. Mac OS X uses subpixel rendering as well, except in the "Best for CRT" setting, which applies traditional anti-aliasing. In stealth mode from 1992 to 1999, Candice H. Brown Elliott was researching subpixel rendering and developing novel layouts, the PenTile Matrix family, that worked together with the subpixel rendering algorithms to increase the resolution of color flat panel displays. In 2000, she co-founded Clairvoyante, Inc. to commercialize these layouts and subpixel rendering algorithms. In 2008, Samsung purchased Clairvoyante and simultaneously funded a new company, Nouvoyance, Inc., retaining much of the technical staff, with Ms. Brown Elliott as CEO.

Subpixel Rendering and the Apple II

It is sometimes claimed (e.g. by Steve Gibson) that the Apple II supported an early form of subpixel rendering in its high-resolution (280x192) graphics mode. However, the method Gibson describes can also be viewed as a limitation of the way the machine generates color, rather than as a technique intentionally exploited by programmers to increase resolution.

The bytes that comprise the Apple II high-resolution screen buffer contain seven visible bits (each corresponding directly to a pixel) and a flag bit used to select between purple/green or blue/orange color sets. Each pixel, since it is represented by a single bit, is either on or off; there are no bits within the pixel itself for specifying color or brightness. Color is instead created as an artifact of the NTSC color encoding scheme, determined by horizontal position: pixels with even horizontal coordinates are always purple (or blue, if the flag bit is set), and odd pixels are always green (or orange). Two lit pixels next to each other are always white, regardless of whether the pair is even/odd or odd/even, and irrespective of the value of the flag bit. The foregoing is only an approximation of the true interplay of the digital and analog behavior of the Apple's video output circuits on one hand, and the properties of real NTSC monitors on the other hand. However, this approximation was what most programmers of the time would have in mind while working with the Apple's high-resolution mode.

In Gibson's example, then, the programmer is not necessarily placing purple and green pixels to increase the perceived resolution of a white line; he may also be seen as simply drawing a line two pixels wide so as to make it appear white, and this latter mental model is arguably the one that most programmers of the time used. If a diagonal line were only one pixel wide, it would appear alternately purple and green as it meandered down the screen between even and odd horizontal coordinates. While the quote from Apple II inventor Steve Wozniak on Gibson's page seems to imply that Apple II graphics programmers routinely used subpixel rendering, it is difficult to make a case that many of them thought of what they were doing in such terms.

The flag bit in each byte affects color by shifting pixels half a pixel-width to the right. This half-pixel shift was exploited by some graphics software, such as HRCG (High-Resolution Character Generator), an Apple utility that displayed text using the high-resolution graphics mode, to smooth diagonals. (Many Apple II users had monochrome displays, or turned down the saturation on their color displays when running software that expected a monochrome display, so this technique was useful.) Although it did not provide a way to address subpixels individually, it did allow positioning of pixels at fractional pixel locations and can thus be considered a form of subpixel rendering. However, this technique is not related to LCD subpixel rendering as described here.

Addressability vs. resolution

With subpixel rendering technology, the number of points that may be independently addressed to reconstruct the image is increased. When the green subpixels are reconstructing the shoulders, the red subpixels are reconstructing near the peaks and vice versa. For text fonts, increasing the addressability allows the font designer to use spatial frequencies and phases that would have created noticeable distortions had it been whole pixel rendered. The improvement is most noted on italic fonts which exhibit different phases on each row. This reduction in moiré distortion is the primary benefit of subpixel rendered fonts on the conventional RGB Stripe panel.

Although subpixel rendering increases the number of reconstruction points on the display this does not always mean that higher resolution, higher spatial frequencies, more lines and spaces, may be displayed on a given arrangement of color subpixels. A phenomenon occurs as the spatial frequency is increased past the whole pixel Nyquist limit from the Nyquist–Shannon sampling theorem. Chromatic aliasing (color fringes) may appear with higher spatial frequencies in a given orientation on the color subpixel arrangement.

Example with the common RGB stripes layout

For example, consider an RGB Stripe Panel:

```
RGBRGBRGBRGBRGBRGB              WWWWWWWWWWWWWWWWWW          R = red
RGBRGBRGBRGBRGBRGB     is       WWWWWWWWWWWWWWWWWW          G = green
RGBRGBRGBRGBRGBRGB  perceived   WWWWWWWWWWWWWWWWWW  where   B = blue
RGBRGBRGBRGBRGBRGB     as       WWWWWWWWWWWWWWWWWW          W = white
RGBRGBRGBRGBRGBRGB              WWWWWWWWWWWWWWWWWW
```

Shown below is an example of black and white lines at the Nyquist limit, but at a slanting angle, taking advantage of Subpixel rendering to use a different phase each row:

```
RGB___RGB___RGB___              WWW___WWW___WWW___          R = red
_GBR___GBR___GBR__     is       _WWW___WWW___WWW__          G = green
__BRG___BRG___BRG_  perceived   __WWW___WWW___WWW_   where  B = blue
___RGB___RGB___RGB     as       ___WWW___WWW___WWW          _ = black
____GBR___GBR___GB              ____WWW___WWW___WW          W = white
```

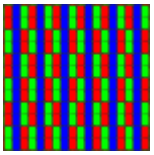
Shown below is an example of chromatic aliasing when the traditional whole pixel Nyquist limit is exceeded:

```
RG__GB__BR__RG__GB              YY__CC__MM__YY__CC          R = red    Y
= yellow
RG__GB__BR__RG__GB     is       YY__CC__MM__YY__CC          G = green  C
= cyan
RG__GB__BR__RG__GB  perceived   YY__CC__MM__YY__CC   where  B = blue   M
= magenta
RG__GB__BR__RG__GB     as       YY__CC__MM__YY__CC          _ = black
RG__GB__BR__RG__GB              YY__CC__MM__YY__CC
```

This case shows the result of attempting to place vertical black and white lines at four subpixels per cycle on the RGB Stripe architecture. One can visually see that the lines, instead of being white, are colored. Starting from the left, the first line is red combined with green to produce a yellow-colored line. The second line is green combined with blue to produce a pastel cyan-colored line. The third line is blue combined with red to produce a magenta-colored line. The colors then repeat: yellow, cyan, and magenta. This demonstrates that a spatial frequency of one cycle per four subpixels is too high. Attempts to go to a yet higher spatial frequency, such as one cycle per three subpixels, would result in a single solid color.

Example with RBG-GBR alternated stripes layout

Novel subpixel layouts have been developed to allow higher real resolution without chromatic aliasing. Shown here is one of the member of the PenTile Matrix family of layouts. Shown below is an example of how a simple change to the arrangement of color subpixels may allow a higher limit in the horizontal direction:



RBG-GBR alternated subpixel geometry (zoomed at 12:1).

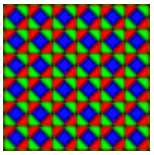
```
RBGRBGRBGRBGRBGRBG
GBRGBRGBRGBRGBRGBR
RBGRBGRBGRBGRBGRBG
GBRGBRGBRGBRGBRGBR
RBGRBGRBGRBGRBGRBG
GBRGBRGBRGBRGBRGBR
```

In this case, the red and green order are interchanged every row to create a red & green checkerboard pattern with blue stripes. Note that the vertical subpixels could be split in half vertically to double the vertical resolution as well : the current LCD panels already typically use two color LEDs (aligned vertically and displaying the same lightness, see the zoomed images below) to illuminate each vertical subpixel. This layout is one of the PenTile matrix family of layouts. When displaying the same number of black white lines, the blue subpixels are set at half brightness "b":

```
Rb_Rb_Rb_Rb_Rb_Rb_
Gb_Gb_Gb_Gb_Gb_Gb_
Rb_Rb_Rb_Rb_Rb_Rb_
Gb_Gb_Gb_Gb_Gb_Gb_
Rb_Rb_Rb_Rb_Rb_Rb_
Gb_Gb_Gb_Gb_Gb_Gb_
```

Notice that every column that turns on comprises red and green subpixels at full brightness and blue subpixels at half value to balance it to white. Now, one may display black and white lines at up to one cycle per three subpixels without chromatic aliasing, twice that of the RGB Stripe architecture.

Non-striped variants of the RBG-GBR alternated layout



RG-B-GR alternated subpixel geometry (zoomed at 12:1).

Variants of the previous layout have been proposed by Clairvoyante/Nouvoyance (and demonstrated by Samsung) as members of the PenTile Matrix Family of layouts specifically designed for subpixel rendering efficiency.

For example, taking profit of the doubled visible horizontal resolution, one could double the vertical resolution to make the definition more isotropic. However this would reduce the aperture of pixels, producing lower contrasts. A better alternative uses the fact that the blue subpixels are those that contribute the least to the visible intensity, so that they are less precisely located by the eye. Blue subpixels are then rendered just as a diamond in the center of a pixel square, and the rest of the pixel surface is split in four parts as a checker board of red and green subpixels with smaller sizes. Rendering images with this variant can use the same technique as before, except that now there's a near-isotropic geometry that supports both the horizontal and the vertical with the same geometric properties, making the layout ideal for displaying the same image details when the LCD panel can be rotated.

The doubled vertical and horizontal visual resolution allows to reduce the subpixel density of about 33%, in order to increase their aperture also of about 33%, with the same separation distance between subpixels (for their electronic interconnection), And also to reduce the power dissipation of about 0% with a white/black contrast increased of about 0% and still a visual-pixel resolution enhanced by about 33% (i.e. about 12 dpi instead of 96 dpi), but with only half the total number of subpixels for the same displayed surface.

Checkered RG-BW layout

Another variant, called the RGBW Quad, uses a checkerboard with 4 subpixels per pixel, adding a white subpixel, or more specifically, replacing one of the green subpixels of Bayer filter Pattern with a white subpixel, to increase the contrast and reduce the energy needed to illuminate white pixels (because color filters in classic RGB striped panels absorb more than 6% of the total white light used to illuminate the panel. As each subpixel is a square instead of a thin rectangle, this also increases the aperture with the same average subpixel density, and same pixel density along both axis. As the horizontal density is reduced and the vertical density remains identical (for the same square pixel density), it becomes possible to increase the pixel density of about 33%, while maintaining the contrast comparable to classic RGB or BGR panels, taking profit of the more efficient use of light and lowered absorption levels by the color filters.

It is not possible to use subpixel rendering to increase the resolution without creating color fringes similar to those seen in classic RGB or BGR striped panels, but the increased resolution compensates it (in addition, their effective visible color is reduced by the presence of "color-neutral" white subpixels.

However, this layout allows a better rendering of greys (and more exact rendering of CMYK-based colors), at the price of a lower color separation. But this is consistent with humane vision and with modern image and video compression formats (like JPEG and MPEG) used in modern HDTV transmission and in BlueRay video records.

Yet another variant, a member for the PenTile Matrix Family of subpixel layouts, alternates between subpixel order RGBW / BWRG every other row, to allow subpixel rendering to increase the resolution, without chromatic aliasing. As before, the increased transmittance using the white subpixel allows higher subpixel density, but in this case, the displayed resolution is even higher due to the benefits of subpixel rendering:

```
RGBWRGBWRGBW
BWRGBWRGBWRG
RGBWRGBWRGBW
BWRGBWRGBWRG
RGB_RGB_RGB_
_W___W___W__
RGB_RGB_RGB_
_W___W___W__
```

Visual resolution versus pixel resolution and software compatibility

Thus, not all layouts are created equal. Each particular layout may have a different "visual resolution", Modulation Transfer Function Limit (MTFL), defined as the highest number of black and white lines that may be simultaneously rendered without visible chromatic aliasing.

However, such alternate layouts are still not compatible with subpixel rendering font algorithms used in Windows, Mac OS X and Linux, which currently support only the RGB or BGR striped subpixel layouts. However, the PenTile Matrix displays have a built-in subpixel rendering engine that allows conventional RGB data sets to be converted to the layouts, providing plug'n'play compatibility with conventional layout displays. New display models should be proposed in the future that will allow monitor drivers to specify : their visual resolution separately from the full pixel resolution and the relative position offsets of visible subpixels for each color plane, as well as their respective contribution to white intensity. Such monitor drivers would allow renderers to correctly adjust their geometry transform matrixes in order to correctly compute the values of each color plane, and take the best profit of subpixel rendering with the lowest chromatic aliasing.

Patents

Microsoft has several patents in the United States on subpixel rendering technology for text rendering on RGB Stripe layouts. This has caused FreeType, the library used by most current software on the X Window System, to disable this functionality by default. Apple is able to use it in Mac OS X due to a patent cross-licensing agreement.

The patents 6,219,02, 6,239,783, 6,307,66, 6,22,973, 6,243,070, 6,393,14, 6,421,04, 6,282,327, 6,624,828 are filed between 1998-10-07 - 1999-10-07. And thus should expire in 2019-10-07.

Clairvoyante, Inc. filed for approximately 100 patents on subpixel rendering and layouts specifically designed to be compatible with subpixel rendering (the PenTile Matrix

Family). In March 2008, Samsung bought Clairvoyante and all of their patents. The co-founder of Clairvoyante, Candice Brown Elliott, then started Nouvoyance to continue development of the technology in partnership with Samsung.

Examples

Photos were taken with an Intel_Play Qx3 Microscope at 60x power on a ViewSonic VG191 display. Note that the display has RGB pixels. Displays exist in all four patterns horizontal RGB/BGR and vertical RGB/BGR but horizontal RGB is the most common. In addition, several color subpixel patterns have been developed specifically to take advantage of subpixel rendering. The best known of these is the PenTile matrix family of patterns.

Note that FreeType with the subpixel rendering functionality shown in these examples cannot be distributed in the United States due to the patents mentioned above.

Lower case **e** subpixel rendered with FreeType

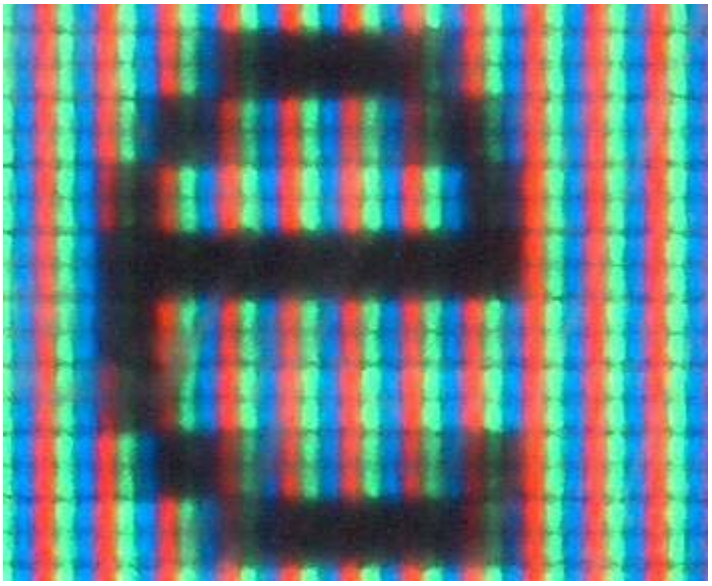


Photo of subpixel **e** rendered on LCD screen

Lower case **is** subpixel rendered with FreeType

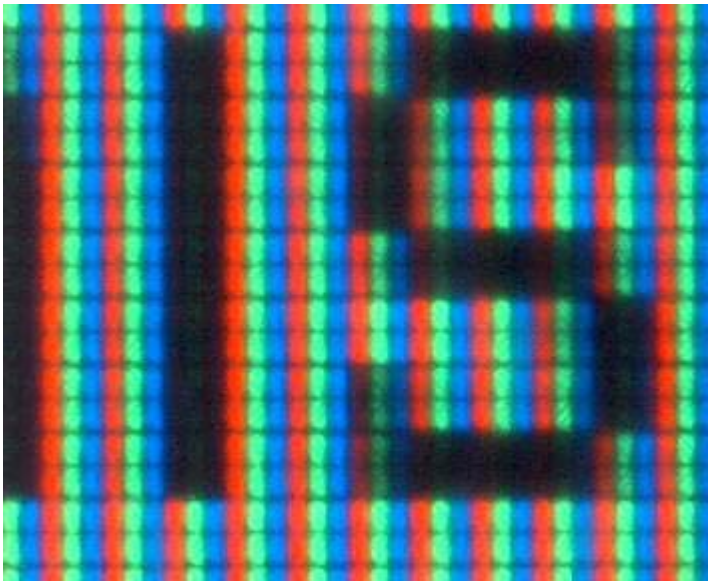
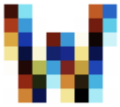


Photo of subpixel **is** rendered on LCD screen



Lower case **w** subpixel rendered with FreeType

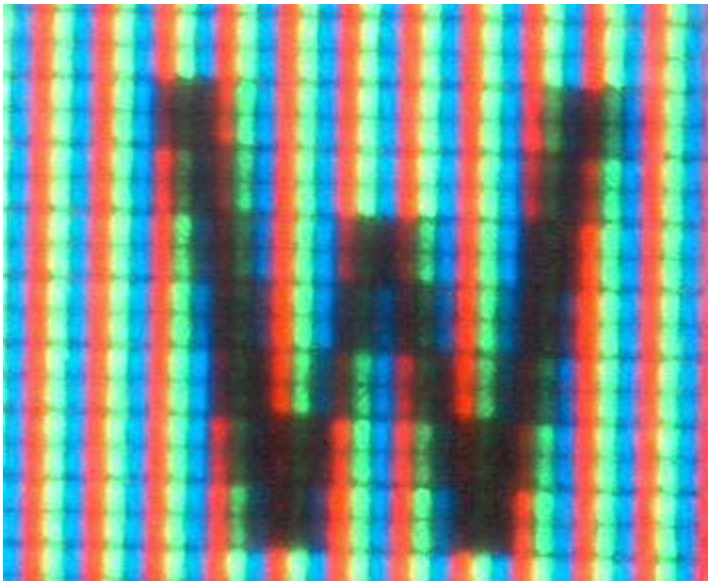
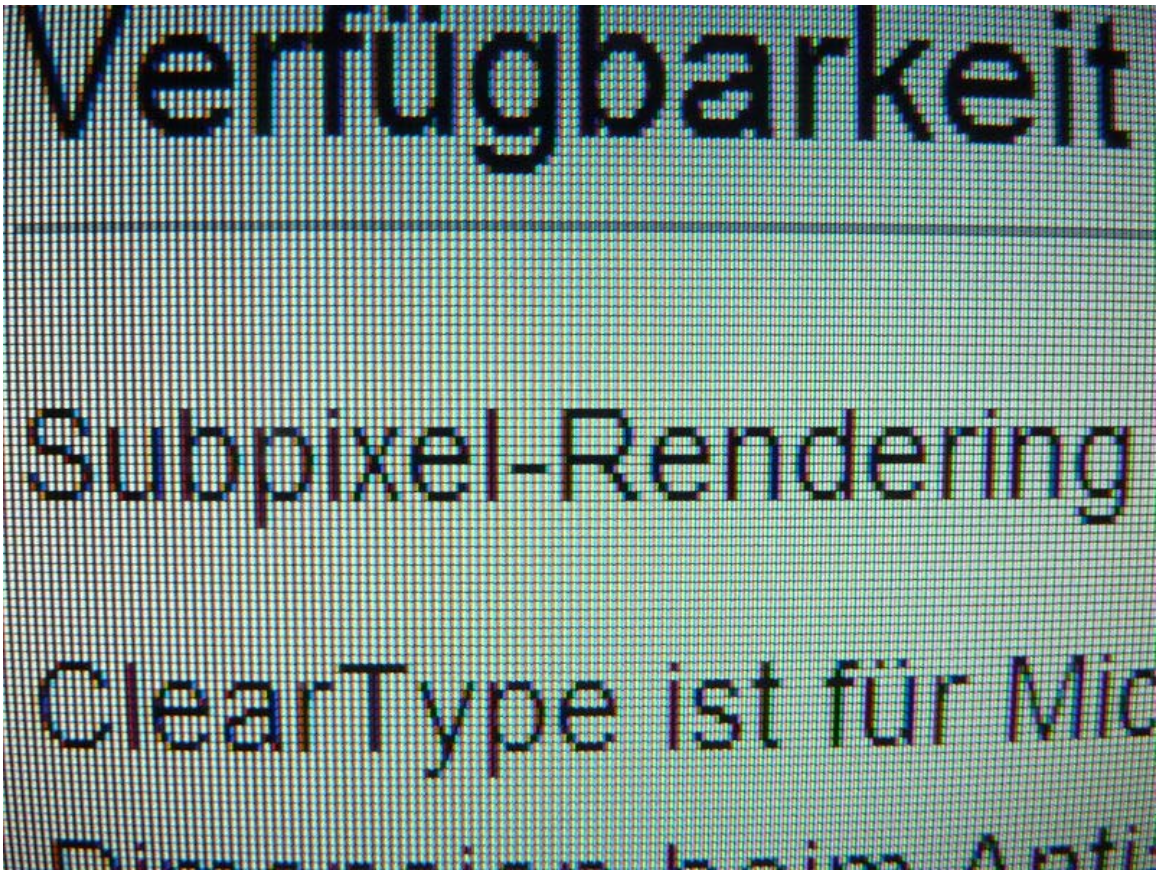
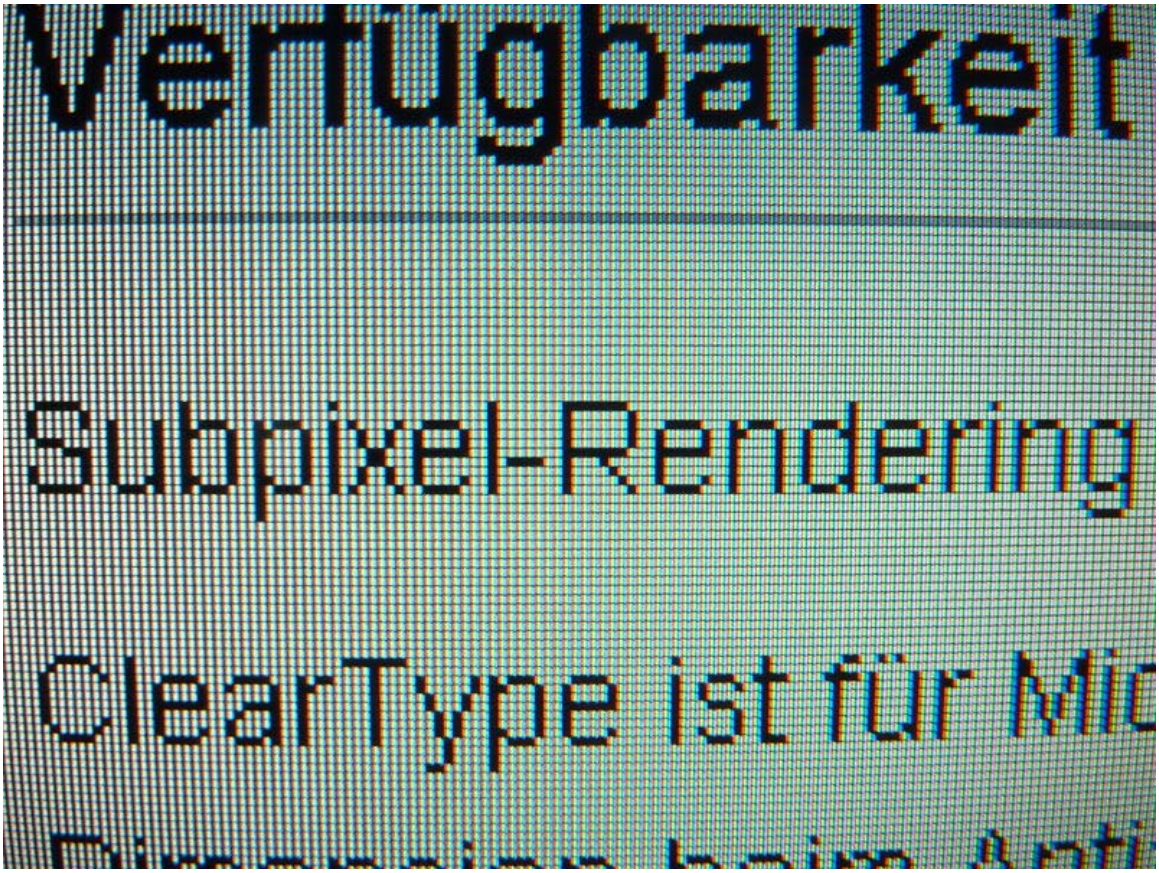


Photo of subpixel **w** rendered on LCD screen



Clear Type activated on a TFT



No Subpixel rendering on a TFT

Vector Markup Language

	Vector Markup Language
Filename extension	`.htm or .html`
Internet media type	`application/vnd.openxmlformats-officedocument.vmlDrawing`
Developed by	Microsoft
Type of format	Vector image format

Extended from	XML
Standard(s)	Part of ECMA-376 and ISO/IEC 2900:2008

Vector Markup Language (**VML**) is a deprecated XML language used to produce vector graphics.

VML was submitted as a proposed standard to the W3C in 1998 by Autodesk, Hewlett-Packard, Macromedia, Microsoft, and Visio. Around the same time other competing W3C submissions were received in the area of web vector graphics, such as PGML from Adobe Systems, Sun Microsystems, and others. As a result of these submissions, a new W3C working group was created, which produced Scalable Vector Graphics (SVG). SVG became a W3C Recommendation in 2001 as a language for describing two-dimensional vector and mixed vector/raster graphics in XML. VML has been largely deprecated in favor of other formats, such as SVG. SVG is not compatible with VML.

While Microsoft continues to document VML, development of the format ceased in 1998. VML is implemented in Internet Explorer .0 and higher and in Microsoft Office 2000 and higher. VML is still available in Internet Explorer 9 but Microsoft expects web sites to transition to SVG in the future. Version 2 of the Google Maps JavaScript API used to use VML for vector paths on Internet Explorer .+, but has been officially deprecated in favour of version 3, which does not.

The Vector Markup Language is included in Part 4 of the Office Open XML standards ISO/IEC 2900:2008 and ECMA-376. According to the specification, VML should be considered a deprecated format included in Office Open XML for legacy reasons only. However, VML is in fact very pervasive in MS Office 2007 (Word, Excel and Powerpoint) documents.

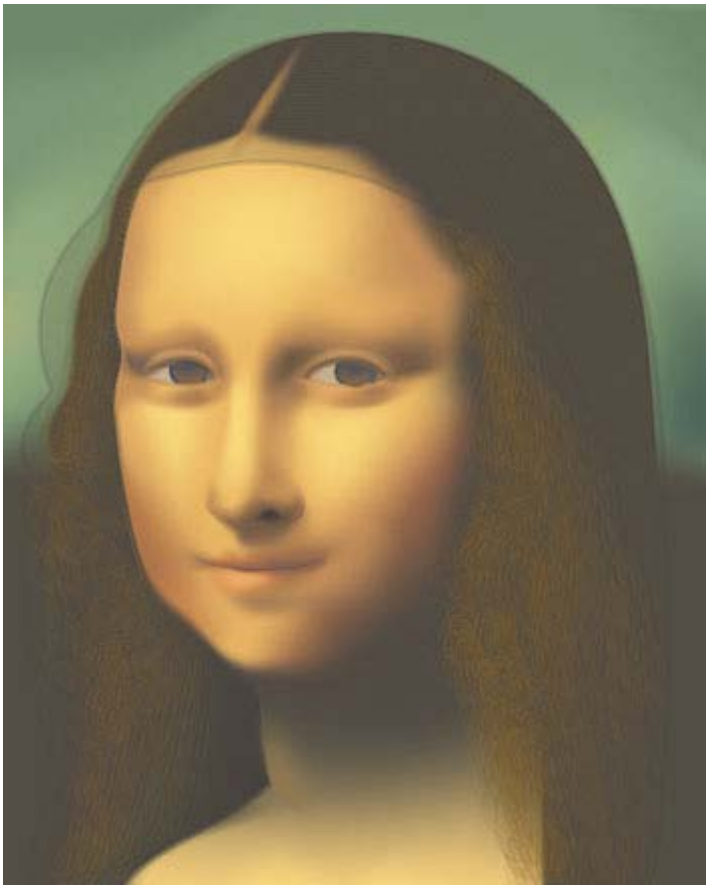
Example

VML images and their behaviours are defined within HTML markup. The following example displays an oval filled in blue in Microsoft Internet Explorer:

VML oval in Internet Explorer

```
<html xmlns:v>
 <style>v\:*{behavior:url(#default#VML);position:absolute}</style>
<body>
 <v:oval style="left:0;top:0;width:100;height:0" fillcolor="blue"
stroked="f"/>
```

```
</body>
</html>
```



VML Webart example

Software

VML is used by most Microsoft Office applications, such as Microsoft Word, Microsoft Visio, etc. within its HTML files, created using the '`Save As HTML`' option (plain HTML or MHT). Such files retain complete vector information, and can be reopened for editing using other Microsoft applications, such as Microsoft PowerPoint. VML is natively supported by Microsoft's Internet Explorer inline within HTML, using an undefined version of SGML namespaces.

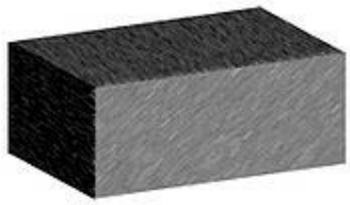
VML is not natively supported by most web browsers. Web browsers such as Mozilla Firefox, Opera, Safari or Google Chrome support SVG instead of VML.

Chapter 5

Shading

Shading refers to depicting depth perception in 3D models or illustrations by varying levels of darkness.

Drawing



Example of shading

Shading is a process used in drawing for depicting levels of darkness on paper by applying media more densely or with a darker shade for darker areas, and less densely or with a lighter shade for lighter areas. There are various techniques of shading including cross hatching where perpendicular lines of varying closeness are drawn in a grid pattern to shade an area. The closer the lines are together, the darker the area appears. Likewise, the farther apart the lines are, the lighter the area appears.

Light patterns, such as objects having light and shaded areas, help when creating the illusion of depth on paper.

Simple tutorials and lessons are available at numerous locations online to help learn the tips and tricks of shading to improve sketches.

Computer graphics

In computer graphics, **Shading** refers to the process of altering a color based on its angle to lights and its distance from lights to create a photorealistic effect. Shading is performed during the rendering process.

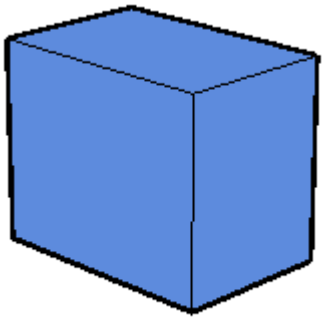
Angle to light source

Shading alters the colors of faces in a 3D model based on the angle of the surface to the sun or other light sources.

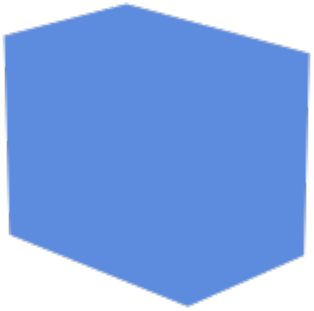
The very first image below has the faces of the box rendered, but all in the same color. Edge lines have been rendered here as well which makes the image easier to see.

The second image is the same model rendered without edge lines. It is difficult to tell where one face of the box ends and the next begins.

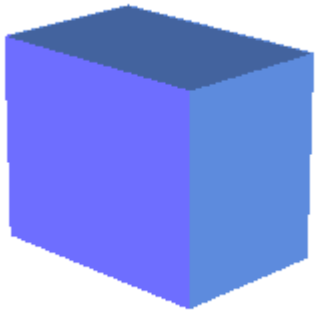
The third image has shading enabled, which makes the image more realistic and makes it easier to see which face is which.



Rendered image of a box. This image has no shading on its faces, but uses edge lines to separate the faces.

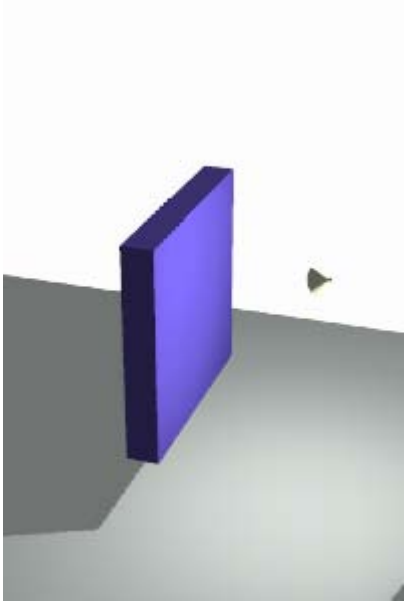


This is the same image with the edge lines removed



This is the same image rendered with shading of the faces to alter the colors of the 3 faces based on their angle to the light sources.

Light sources



Shading effects from floodlight

There are many types of lights:

- Ambient, lights all objects within a scene equally, brightening them without adding shading
- Directional, illuminates all objects equally from a given direction, like an area light of infinite size and infinite distance from the scene; there is shading, but cannot be any distance falloff
- Point, originates from a single point, and spreads outward in all directions
- Spotlight, originates from a single point, and spreads outward in a coned direction
- Area, originates from a single plane and illuminates all objects in a given direction beginning from that plane

- Volume, an enclosed space lighting objects within that space

Shading is interpolated based on how the angle of these light sources reach the objects within a scene. Of course, these light sources can be and often are combined in a scene. The renderer then interpolates how these lights must be combined, and produces a 2d image to be displayed on the screen accordingly.

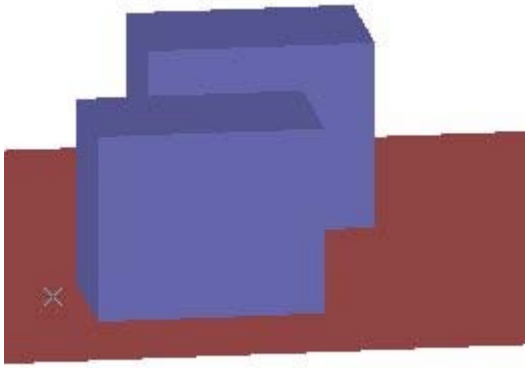
Distance falloff

Theoretically, two surfaces which are parallel, are illuminated the same amount from a distant light source, such as the sun. Even though one surface is further away, your eye sees more of it in the same space, so the illumination appears the same.

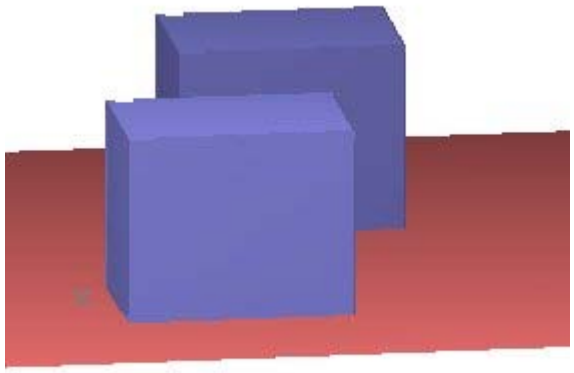
Notice in the first image that the color on the front faces of the two boxes is exactly the same. It appears that there is a slight difference where the two faces meet, but this is an optical illusion because of the vertical edge below where the two faces meet.

Notice in the second image that the surfaces on the boxes are bright on the front box and darker on the back box. Also the floor goes from light to dark as it gets farther away.

This distance falloff effect produces images which appear more realistic without having to add additional lights to achieve the same effect.



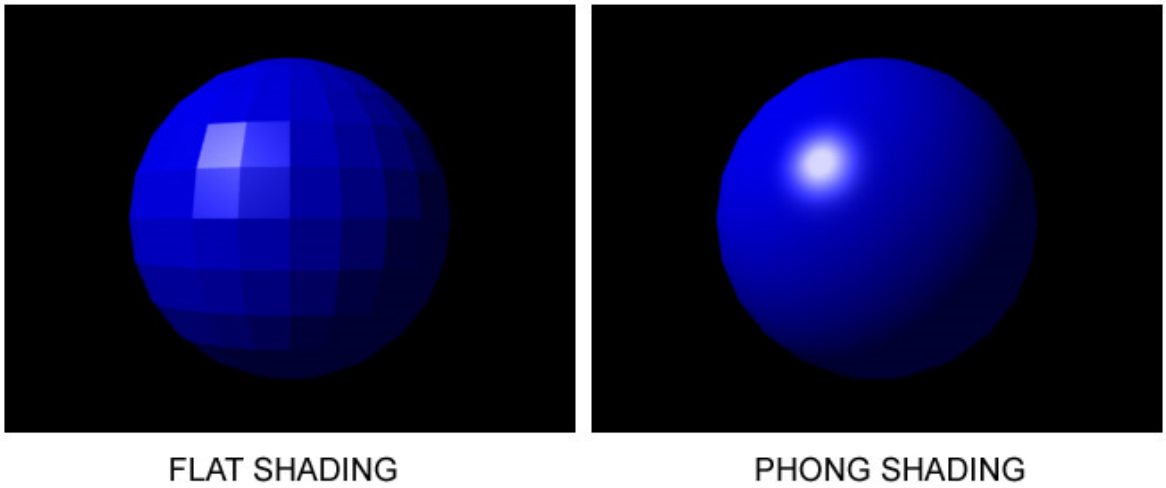
Two boxes rendered with an OpenGL renderer. Note that the colors of the two front faces are the same even though one box is further away.



The same model rendered using ARRIS CAD which implements "Distance Falloff" to make surfaces which are closer to the eye appear brighter.

Distance falloff can be calculated in a number of ways:

- None
- Linear - For every x units a given point is from the light source, the amount of light received is x units less bright.
- Quadratic - This is approximately how light works in real life. A point that is twice as far from the light source as another will receive four times less light.
- Factor of *n* - A point that is x units from a light source will receive $1/x^n$ as much light.
- Any number of other mathematical functions may also be used.



Flat shading interpolation example

Flat vs smooth shading

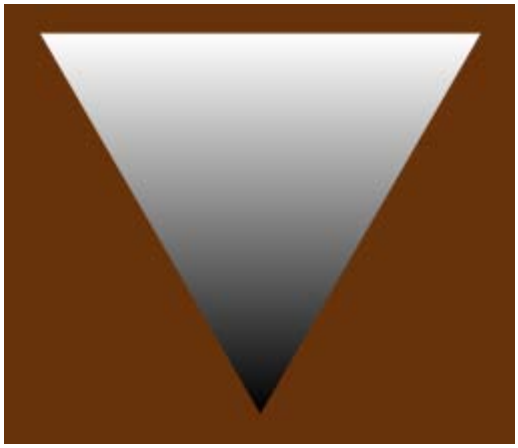
Flat shading is a lighting technique used in 3D computer graphics. It shades each polygon of an object based on the angle between the polygon's surface normal and the direction of the light source, their respective colors and the intensity of the light source. It was used for high speed rendering where more advanced shading techniques were too computationally expensive. But by the end of the 20th century affordable graphics cards were offering smooth shading that were also very fast, making flat shading for speed reasons unnecessary.

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; flat shading is only inappropriate when the object must appear with smoother curves than actually modeled. 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.

Chapter 6

Gouraud Shading and Phong Shading

Gouraud shading



Gouraud-shaded polygon

Gouraud shading, named after Henri Gouraud, is a method used in computer graphics to simulate the differing effects of light and colour 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. Gouraud first published the technique in 1971 in IEEE Transactions on Computers.

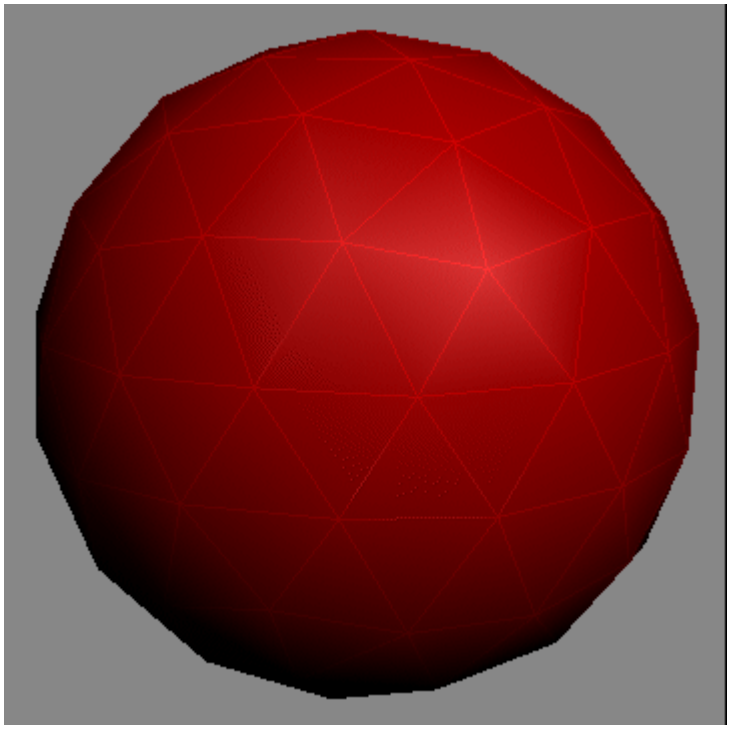
Principles behind the method

The basic principle behind the method is as follows: An estimate to the surface normal of each vertex in a 3D model is found by averaging the surface normals of polygons which meet at each vertex. Using these estimates, lighting computations based on the Phong reflection model are then performed to produce colour intensities at the vertices. Screen pixel intensities can then be bilinearly interpolated from the colour values calculated at the vertices. In a simpler sense, a gradient is formed in the area where the vertices meet.

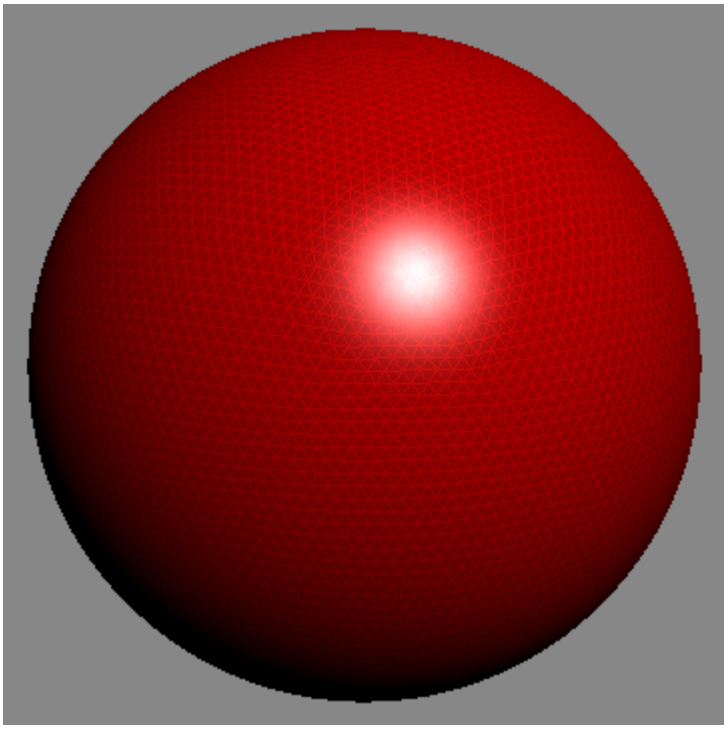
Gouraud shading's strength and weakness lies in its interpolation. Interpolating colour values from three samples of expensive lighting calculations is much less processor

intensive than performing that lighting calculation for each pixel, as is done in Phong shading. However, highly localized lighting effects (such as specular highlights, e.g. the glint of reflected light on the surface of an apple) will not be rendered correctly, and if a highlight lies in the middle of a polygon, but does not spread to the polygon's vertex, it will not be apparent in a Gouraud rendering; conversely, if a highlight occurs at the vertex of a polygon, it will be rendered correctly at this vertex (as this is where the lighting model is applied), but will be spread unnaturally across all neighboring polygons via the interpolation method. The problem is easily spotted in a rendering which ought to have a specular highlight moving smoothly across the surface of a model as it rotates. Gouraud shading will instead produce a highlight continuously fading in and out across neighboring portions of the model, peaking in intensity when the intended specular highlight passes over a vertex of the model. (This can be improved by increasing the density of vertices in the object, or alternatively an adaptive tessellation scheme might be used to increase the density only near the highlight.)

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



Gouraud-shaded sphere - note the poor behaviour of the specular highlight



The same sphere rendered with a very high polygon count

Phong shading

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

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

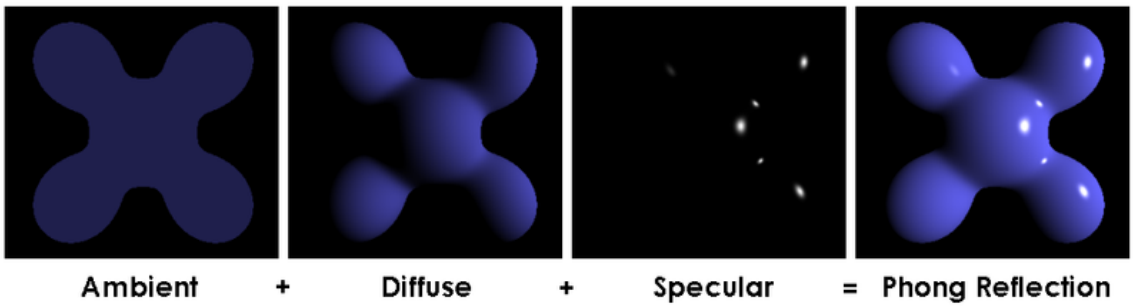
History

These methods were developed by Bui Tuong Phong at the University of Utah, who published them in his 1973 Ph.D. dissertation. Phong's shading methods were considered radical at the time of their introduction, but have evolved into a baseline shading method

for many rendering applications. Phong's methods have proven popular due to their generally parsimonious use of CPU time per rendered pixel.

Phong reflection model

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



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

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

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

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

We further define lights as the set of all light sources, L as the direction vector from the point on the surface toward each light source, N as the normal at this point on the surface,

R as the direction that a perfectly reflected ray of light would take from this point on the surface, and V as the direction pointing towards the viewer (such as a virtual camera).

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

$$I_p = k_a i_a + \sum_{\mathrm{m} \in \mathrm{lights}} (k_d (L_m \cdot N) i_d + k_s (R_m \cdot V)^\alpha i_s).$$

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

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

Although the above formulation is the common way of presenting the Phong model, a particular term in the sum should only be included if it is positive, i.e. the equation is formally incorrect.

Inverse Phong reflection model

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

The Phong reflection model contains many parameters, such as the surface diffuse reflection parameter (albedo) which may vary within the object. Thus the normals of an object in a photograph can only be determined, by introducing additive information such as the number of lights, light directions and reflection parameters.

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

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

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

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

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

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

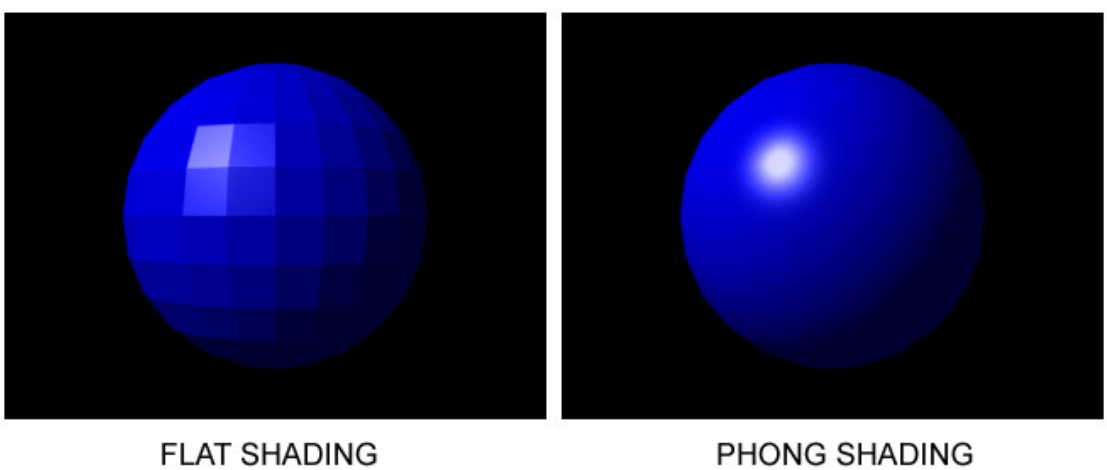
For instance if the light direction is 45 degrees above the object $L = [0.71,0.71]$ we get two equations with two unknowns.

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

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

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

Phong interpolation



Phong shading interpolation example

Phong shading improves upon Gouraud shading and provides a better approximation of the shading of a smooth surface. Phong shading assumes a smoothly varying surface normal vector. The Phong interpolation method works better than Gouraud shading when

applied to a reflection model that has small specular highlights such as the Phong reflection model.

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

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

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

Chapter 7

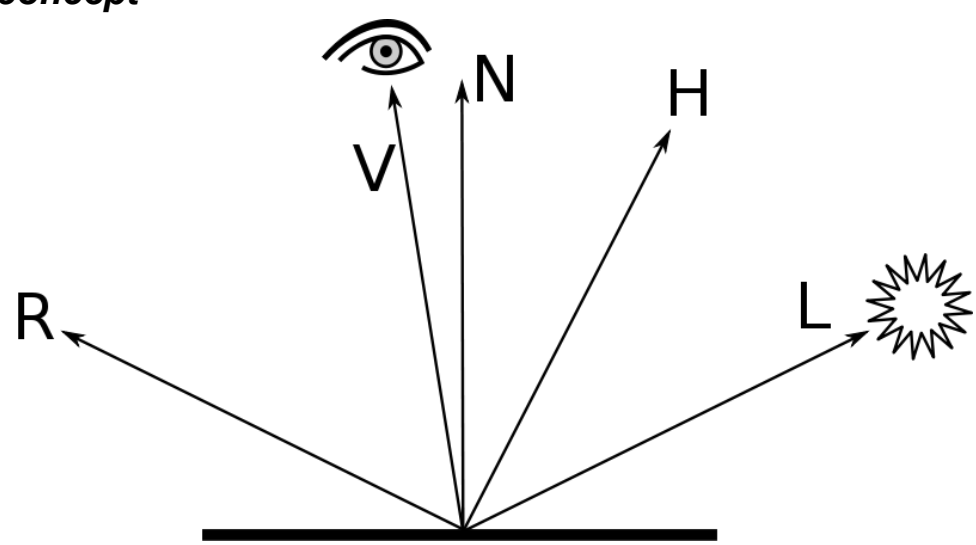
Blinn–Phong Shading Model and Oren–Nayar Reflectance Model

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 developed by Jim Blinn.

Blinn–Phong is the default shading model used in OpenGL and Direct3D's fixed-function pipeline (before Direct3D 10 and OpenGL 3.1), 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.

The concept



Vectors for calculating Phong and Blinn–Phong shading

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.

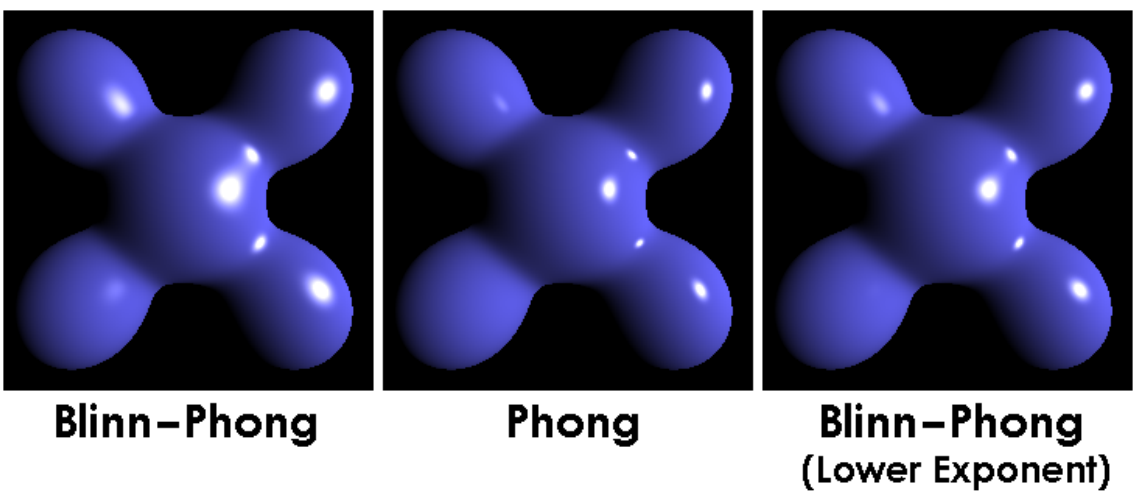
If, instead, one calculates a *halfway vector* between the viewer and light-source vectors,

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

we can replace $R \cdot V$ with $N \cdot H$, where *N* is the normalized surface normal. In the above equation, *L* and *V* are both normalized vectors.

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. This relation between the angles remains approximately true when the vectors don't lie in the same plane, especially when the angles are small. The angle between *N* and *H* is therefore sometimes called the halfway angle.

The halfway angle is smaller than the angle desired in Phong's model, but considering that Phong is using $(R \cdot V)^{\alpha}$, an exponent can be set α' so that $(N \cdot H)^{\alpha'}$ is closer to the former expression. The size of the specular highlights can be matched in this way very closely to a corresponding Phong reflection, but they will always retain a subtly different shape.



Additionally, while it can be seen as an approximation to the Phong model, it produces more accurate models of empirically determined bidirectional reflectance distribution functions than Phong for many types of surfaces.

Efficiency

This rendering model is less efficient than pure Phong shading in most cases, since it contains a square root calculation. While the original Phong model only needs a simple vector reflection, this modified form takes more into consideration. However, as many CPUs and GPUs contain single and double precision square root functions (as standard

features) and other instructions that can be used to speed up rendering -- the time penalty for this kind of shader will not be noticed in most implementations.

However, Blinn-Phong will be faster in the case where the viewer and light are treated to be at infinity. This is the case for directional lights. In this case, the half-angle vector is independent of position and surface curvature. It can be computed once for each light and then used for the entire frame, or indeed while light and viewpoint remain in the same relative position. The same is not true with Phong's original reflected light vector which depends on the surface curvature and must be recalculated for each pixel of the image (or for each vertex of the model in the case of vertex lighting).

In most cases where lights are not treated to be at infinity, for instance when using point lights, the original Phong model will be faster.

High Level Shader Language Code Sample

This sample is a method of determining the diffuse and specular light from a point light. The light structure, position in space of the surface, view direction vector and the normal of the surface are passed through. A Lighting structure is returned;

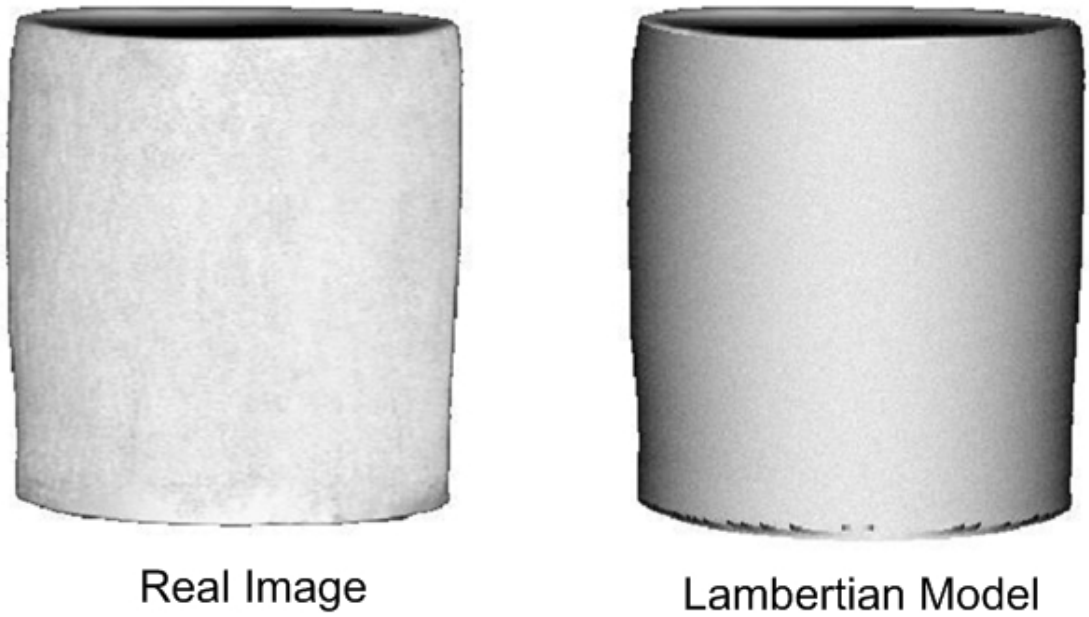
```
struct Lighting
{
    float3 Diffuse;
    float3 Specular;
};

struct PointLight
{
        float3 position;
        float3 diffuseColor;
        float diffusePower;
        float3 specularColor;
        float specularPower;
};

Lighting GetPointLight(PointLight light, float3 pos3D, float3 viewDir,
float3 normal)
{
        Lighting OUT;
        if(light.diffusePower > 0)
        {
                float3 lightDir = light.position - pos3D; // FIND THE
VECTOR BETWEEN THE 3D POSITION IN SPACE OF THE SURFACE
                float distance = length(lightDir); // GET THE DISTANCE
OF THIS VECTOR
                distance = distance * distance; // USES INVERSE SQUARE
FOR DISTANCE ATTENUATION
                lightDir = normalize(lightDir); // NORMALIZE THE VECTOR

                // INTENSITY OF THE DIFFUSE LIGHT
                 // SATURATE TO KEEP WITHIN THE 0-1 RANGE
```

```
                // DOT PRODUCT OF THE LIGHT DIRECTION VECTOR AND THE
SURFACE NORMAL
               float i = saturate(dot(lightDir, normal));

               OUT.Diffuse = i * light.diffuseColor *
light.diffusePower / distance; // CALCULATE THE DIFFUSE LIGHT FACTORING
IN LIGHT COLOUR, POWER AND THE ATTENUATION

                //CALCULATE THE HALF VECTOR BETWEEN THE LIGHT VECTOR
AND THE VIEW VECTOR. THIS IS CHEAPER THEN CALCULATING THE ACTUAL
REFLECTIVE VECTOR
                float3 h = normalize(lightDir + viewDir);

               // INTENSITY OF THE SPECULAR LIGHT
                // DOT PRODUCT OF NORMAL VECTOR AND THE HALF VECTOR TO
THE POWER OF THE SPECULAR HARDNESS
               i = pow(saturate(dot(normal, h)), specularHardness);

               OUT.Specular = i * light.specularColor *
light.specularPower / distance; // CALCULATE THE SPECULAR LIGHT
FACTORING IN LIGHT SPECULAR COLOUR, POWER AND THE ATTENUATION
        }
        return OUT;
}
```

Oren–Nayar reflectance model

The **Oren-Nayar reflectance model**, developed by Michael Oren and Shree K. Nayar, is a reflectance model for **diffuse reflection** from **rough surfaces**. It has been shown to accurately predict the appearance of a wide range of natural surfaces, such as concrete, plaster, sand, etc.

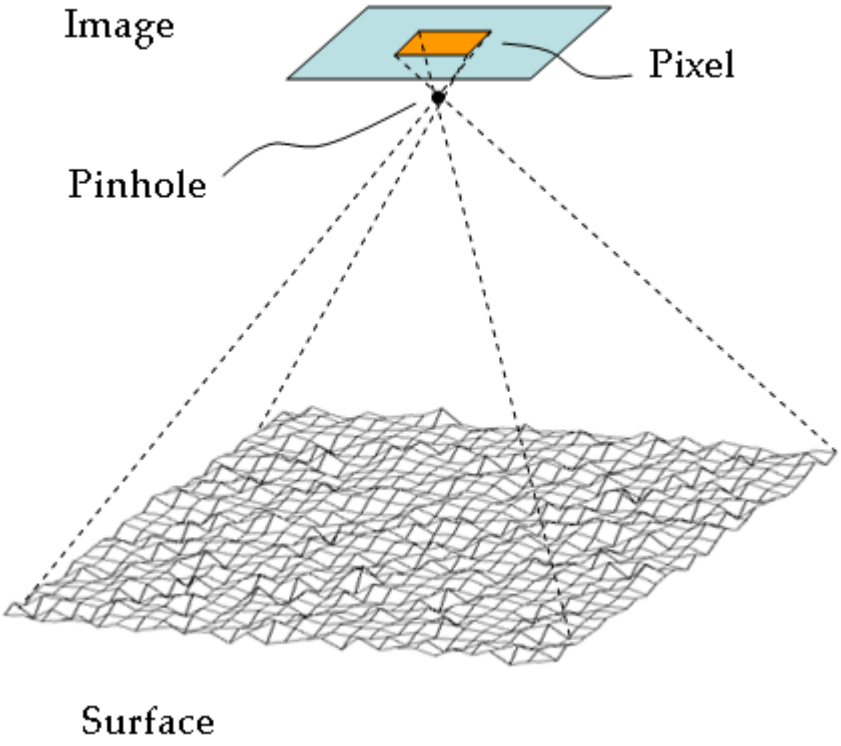
Introduction



Comparison of a matte vase with the rendering based on the Lambertian model

Reflectance is a physical property of a material that describes how it reflects incident light. The appearance of various materials are determined to a large extent by their reflectance properties. Most reflectance models can be broadly classified into two categories: diffuse and specular. In computer vision and computer graphics, the diffuse component is often assumed to be **Lambertian**. A surface that obeys **Lambert's Law** appears equally bright from all viewing directions. This model for diffuse reflection was proposed by Johann Heinrich Lambert in 1760 and has been perhaps the most widely used reflectance model in computer vision and graphics. For a large number of real-world surfaces, such as concrete, plaster, sand, etc., however, the Lambertian model is an inadequate approximation of the diffuse component. This is primarily because the Lambertian model does not take the roughness of the surface into account.

Rough surfaces can be modelled as a set of facets with different slopes, where each facet is a small planar patch. Since photo receptors of the retina and pixels in a camera are both finite-area detectors, substantial macroscopic (much larger than the wavelength of incident light) surface roughness is often projected onto a single detection element, which in turn produces an aggregate brightness value over many facets. Whereas Lambert's law may hold well when observing a single planar facet, a collection of such facets with different orientations is guaranteed to violate Lambert's law. The primary reason for this is that the foreshortened facet areas will change for different viewing directions, and thus the surface appearance will be view-dependent.



Aggregation of the reflection from rough surfaces

Analysis of this phenomenon has a long history and can be traced back almost a century. Past work has resulted in empirical models designed to fit experimental data as well as theoretical results derived from first principles. Much of this work was motivated by the non-Lambertian reflectance of the moon.

The Oren-Nayar reflectance model, developed by Michael Oren and Shree K. Nayar in 1993, predicts reflectance from rough diffuse surfaces for the entire hemisphere of source and sensor directions. The model takes into account complex physical phenomena such as **masking**, **shadowing** and **interreflections** between points on the surface facets. It can be viewed as a generalization of Lambert's law. Today, it is widely used in computer graphics and animation for rendering rough surfaces. It also has important implications for human vision and computer vision problems, such as shape from shading, photometric stereo, etc.

Formulation

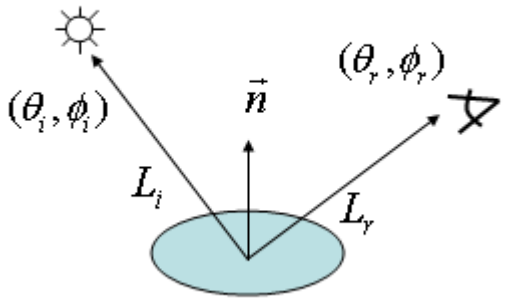


Diagram of surface reflection

The surface roughness model used in the derivation of the Oren-Nayar model is the microfacet model, proposed by Torrance and Sparrow, which assumes the surface to be composed of long symmetric V-cavities. Each cavity consists of two planar facets. The roughness of the surface is specified using a probability function for the distribution of facet slopes. In particular, the Gaussian distribution is often used, and thus the variance of the Gaussian distribution, σ^2, is a measure of the roughness of the surfaces (ranging from 0 to 1).

In the Oren-Nayar reflectance model, each facet is assumed to be Lambertian in reflectance. As shown in the image at right, given the radiance of the incoming light L_i, the radiance of the reflected light L_r, according to the Oren-Nayar model, is

$$L_r = \frac{\rho}{\pi} \cdot \cos\theta_i \cdot (A + B \cdot \max(0, \cos(\phi_i - \phi_r)) \cdot \sin\alpha \cdot \tan\beta) \cdot L_i$$

where

$$A = 1 - 0.5\frac{\sigma^2}{\sigma^2 + 0.33},$$

$$B = 0.45\frac{\sigma^2}{\sigma^2 + 0.09},$$

$\alpha = \max(\theta_i, \theta_r)$,
$\beta = \min(\theta_i, \theta_r)$,

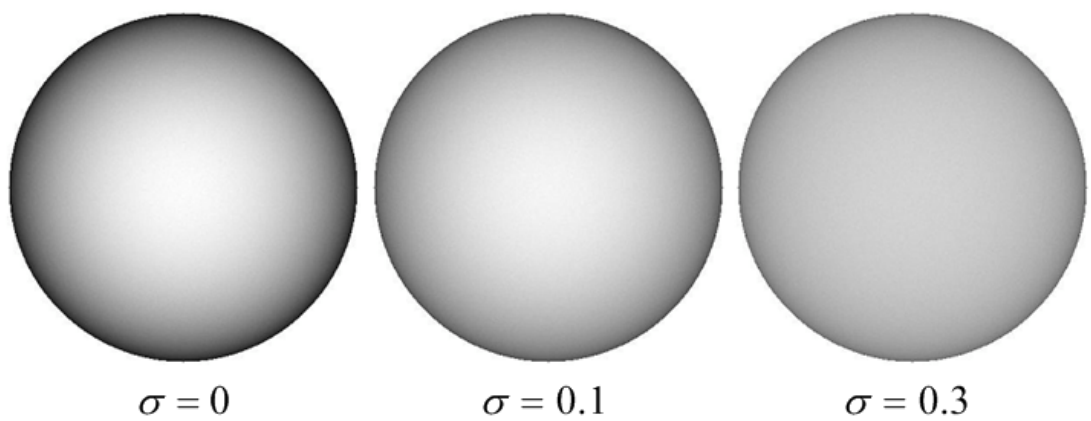
and ρ is the albedo of the surface, and σ is the roughness of the surface (ranging from 0 to 1). In the case of $\sigma = 0$ (i.e., all facets in the same plane), we have $A = 1$, and $B = 0$, and thus the Oren-Nayar model simplifies to the Lambertian model:

$$L_r = \frac{\rho}{\pi} \cdot \cos\theta_i \cdot L_i$$

Results

Here is a real image of a matte vase along with versions rendered using the Lambertian and Oren-Nayar models. It shows that the Oren-Nayar model predicts the diffuse reflectance for rough surfaces more accurately than the Lambertian model.

Here are rendered images of a sphere using the Oren-Nayar model, corresponding to different surface roughnesses (i.e. different σ values):



Connection with other microfacet reflectance models

Oren-Nayar model	Torrance-Sparrow model	Microfacet model for refraction
Rough opaque diffuse surfaces	Rough opaque specular surfaces (glossy surfaces)	Rough transparent surfaces
Each facet is Lambertian (diffuse)	Each facet is a mirror (specular)	Each facet is made of glass (transparent)

Chapter 8

Shader and ARB (GPU Assembly Language)

Shader

In the field of computer graphics, a **shader** is a set of software instructions which is used primarily to calculate rendering effects on graphics hardware with a high degree of flexibility. Shaders are used to program the graphics processing unit (GPU) programmable rendering pipeline, which has mostly superseded the fixed-function pipeline that allowed only common geometry transformation and pixel-shading functions; with shaders, customized effects can be used.

Technology overview

Initially, shaders were used to perform pixel shading only. However, the name stuck and the term is now used for other graphics pipeline stages too.

As Graphics Processing Units evolved, major graphics software libraries such as OpenGL and Direct3D began to exhibit enhanced ability to program these new GPUs by defining special shading functions in their API.

Shaders are simple programs that describe the traits of either a vertex or a pixel. Vertex shaders describe the traits (position, texture coordinates, colors, etc.) of a vertex, while pixel shaders describe the traits (color, z-depth and alpha value) of a pixel. A vertex shader is called for each vertex in a primitive (possibly after tessellation) – thus; one vertex in, one (updated) vertex out. Each vertex is then rendered as a series of pixels onto a surface (block of memory) that will eventually be sent to the screen.

Shaders replace a section of video hardware typically called the Fixed Function Pipeline (FFP) – so-called because it performs lighting and texture mapping in a hard-coded manner. Shaders provide a programmable alternative to this hard-coded approach.

Types of shaders

The Direct3D and OpenGL graphic libraries use three types of shaders.

- Vertex shaders are run once for each vertex given to the graphics processor. The purpose is to transform each vertex's 3D position in virtual space to the 2D coordinate at which it appears on the screen (as well as a depth value for the Z-buffer). Vertex shaders can manipulate properties such as position, color, and texture coordinate, but cannot create new vertices. The output of the vertex shader goes to the next stage in the pipeline, which is either a geometry shader if present or the rasterizer otherwise.
- Geometry shaders can add and remove vertices from a mesh. Geometry shaders can be used to generate geometry procedurally or to add volumetric detail to existing meshes that would be too costly to process on the CPU. If geometry shaders are being used, the output is then sent to the rasterizer.
- Pixel shaders, also known as fragment shaders, calculate the color of individual pixels. The input to this stage comes from the rasterizer, which fills in the polygons being sent through the graphics pipeline. Pixel shaders are typically used for scene lighting and related effects such as bump mapping and color toning. (Direct3D uses the term "pixel shader," while OpenGL uses the term "fragment shader." The latter is arguably more correct, as there is not a one-to-one relationship between calls to the pixel shader and pixels on the screen. The most common reason for this is that pixel shaders are often called many times per pixel for every object that is in the corresponding space, even if it is occluded; the Z-buffer sorts this out later.)

The unified shader model unifies the three aforementioned shaders in OpenGL and Direct3D 10.

As these shader types are processed within the GPU pipeline, the following gives an example how they are embedded in the pipeline:

Simplified graphic processing unit pipeline

- The CPU sends instructions (compiled shading language programs) and geometry data to the graphics processing unit, located on the graphics card.
- Within the vertex shader, the geometry is transformed and lighting calculations are performed.
- If a geometry shader is in the graphic processing unit, some changes of the geometries in the scene are performed.
- The calculated geometry is triangulated (subdivided into triangles).
- Triangles are broken down into pixel quads (one pixel quad is a 2 × 2 pixel primitive).

The graphic pipeline uses these steps in order to transform three dimensional (and/or two dimensional) data into useful two dimensional data for displaying. In general, this is a large pixel matrix or "frame buffer".

Parallel processing

Shaders are written to apply transformations to a large set of elements at a time, for example, to each pixel in an area of the screen, or for every vertex of a model. This is well suited to parallel processing, and most modern GPUs have multiple shader pipelines to facilitate this, vastly improving computation throughput.

Programming shaders

OpenGL (version 1.5 and newer) provides a C-like Shader language called OpenGL Shading Language, or GLSL.

In the Microsoft Direct3D API (Direct3D 9 and newer), shaders are programmed with High Level Shader Language, or HLSL.

Cg or C for Graphics is a high-level shading language developed by Nvidia in close collaboration with Microsoft for programming vertex and pixel shaders. It is very similar to Microsoft's HLSL.

ARB (GPU assembly language)

ARB - OpenGL Assembly Language is a low-level shading language. It was created by the OpenGL ARB to standardize GPU instructions controlling the hardware graphics pipeline.

History

Texas Instruments created the first programmable graphics processor in 1985: the TMS34010, which allowed developers to load and execute code on the processor to control pixel output on a video display. This was followed by the TMS34020 and TMS34082 in 1989, providing programmable 3D graphics output.

NVIDIA released its first video card NV1 in 1995, which supported quadratic texture mapping. This was followed by the Riva 128 (NV3) in 1997, providing the first hardware acceleration for Direct3D.

Various video card vendors released their own accelerated boards, each with their own instruction set for GPU operations. The OpenGL Architecture Review Board (ARB) was formed in 1992, in part to establish standards for the GPU industry.

The ARB and NVIDIA established a number of OpenGL extensions to standardize GPU programming:

- EXT_texture_env_combine - provided a programmable method of combining textures.
- NV_register_combiners - GeForce 256
- NV_vertex_program - GeForce 3
- NV_texture_shader - GeForce 3
- NV_texture_shader3 - GeForce 4
- NV_vertex_program2 - GeForce FX
- NV_fragment_program - GeForce FX

This culminated with ARB's 2002 release of

- ARB_vertex_program
- ARB_fragment_program

These two extensions provided an industry standard for an assembly language that controlled the GPU pipeline for 3D vertex and interpolated pixel properties, respectively.

Subsequent high-level shading languages sometimes compile to this ARB standard. While 3D developers are now more likely to use a C-like, high-level shading language for GPU programming, ARB assembly has the advantage of being supported on a wide range of hardware.

Note however that some features, such as loops and conditionals, are not available in ARB assembly, and using them requires to adopt either the NV_gpu_program4 extension, or the GLSL shading language.

Most non-nVidia OpenGL implementations do not provide the nVidia ARB assembly extension and do not offer any other way to access all the shader features directly in assembly, forcing the use of GLSL even for machine generated shaders where assembly would be more appropriate.

Details

ARB_vertex_program

The ARB Vertex Program extension provides APIs to load ARBvp1.0 assembly instructions, enable selected programs, and to set various GPU parameters.

Vertex programs are used to modify vertex properties, such as position, normals and texture coordinates, that are passed to the next pipeline process: often a fragment shader; more recently, a geometry shader.

ARB_fragment_program

The ARB Fragment Program extension provides APIs to load ARBfp1.0 assembly instructions, enable selected programs, and to set various GPU parameters.

OpenGL fragments are interpolated pixel definitions. The GPU's vertex processor calculates all the pixels controlled by a set of vertices, interpolates their position and other properties and passes them onto its fragment process. Fragment programs allow developers to modify these pixel properties before they are rendered to a frame buffer for display.

OpenGL Parameters

- **Attrib** parameters are per-vertex attributes such as vertex normals.
- **Local** parameters are applied across a program's entire data set for a given shader pass.
- **Env** parameters are applied across all programs.

ARB Variables

All ARB assembly variables are float4 vectors, which may be addressed by xyzw or rgba suffixes.

ARB registers are scalar variables where only one element may be addressed.

- **ADDRESS** variables are registers.
- **ATTRIB** are per-vertex attributes.
- **PARAM** are uniform properties - constants, Env or Local.
- **TEMP** temporary variables.
- **ALIAS** provides alternate names for variables.
- **OUTPUT** designates variables that are passed back to the pipeline.

Vertex Attributes

ARB assembly supports the following suffixes for vertex attributes:

- **position**
- **weight**
- **normal**
- **color**
- **fogcoord**
- **texcoord**
- **matrixindex**
- **attrib**

State Matrices

ARB assembly supports the following state matrices:

- **modelview**
- **projection**
- **texture**
- **palette**
- **program**

The following modifiers may be used:

- **inverse**
- **transpose**
- **invtrans**

ARB Assembly Instructions

ARB supports the following instructions:

- **ABS** - absolute value
- **ADD** - add
- **ARL** - address register load
- **DP3** - 3-component dot product
- **DP4** - 4-component dot product
- **DPH** - homogeneous dot product
- **DST** - distance vector
- **EX2** - exponential base 2
- **EXP** - exponential base 2 (approximate)
- **FLR** - floor
- **FRC** - fraction
- **LG2** - logarithm base 2
- **LIT** - compute light coefficients
- **LOG** - logarithm base 2 (approximate)
- **MAD** - multiply and add
- **MAX** - maximum
- **MIN** - minimum
- **MOV** - move
- **MUL** - multiply
- **POW** - exponentiate
- **RCP** - reciprocal
- **RSQ** - reciprocal square root
- **SGE** - set on greater than or equal
- **SLT** - set on less than
- **SUB** - subtract
- **SWZ** - extended swizzle

- **XPD** - cross product

ARB assembly provides no instructions for flow control or branching. SGE and SLT may be used to conditionally set or clear vectors or registers.

ARB interfaces provide no compiling step for assembly language.

GL_NV_fragment_program_option extends the ARB_fragment_program language with additional instructions. GL_NV_fragment_program2, GL_NV_vertex_program2_option and GL_NV_vertex_program3 extend it further.

Sample Code

A sample trivial ARB Vertex Shader

```
!!ARBvp1.0
TEMP vertexClip;
DP4 vertexClip.x, state.matrix.mvp.row[0], vertex.position;
DP4 vertexClip.y, state.matrix.mvp.row, vertex.position;
DP4 vertexClip.z, state.matrix.mvp.row, vertex.position;
DP4 vertexClip.w, state.matrix.mvp.row, vertex.position;
MOV result.position, vertexClip;
MOV result.color, vertex.color;
MOV result.texcoord[0], vertex.texcoord;
END
```

A sample trivial ARB Fragment Shader

```
!!ARBfp1.0
TEMP color;
MUL color, fragment.texcoord[0].y, 2.0;
ADD color, 1.0, -color;
ABS color, color;
ADD result.color, 1.0, -color;
MOV result.color.a, 1.0;
END
```

Chapter 9

Ambient Occlusion, Bloom (Shader Effect) and Cel-Shaded Animation

Ambient occlusion

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

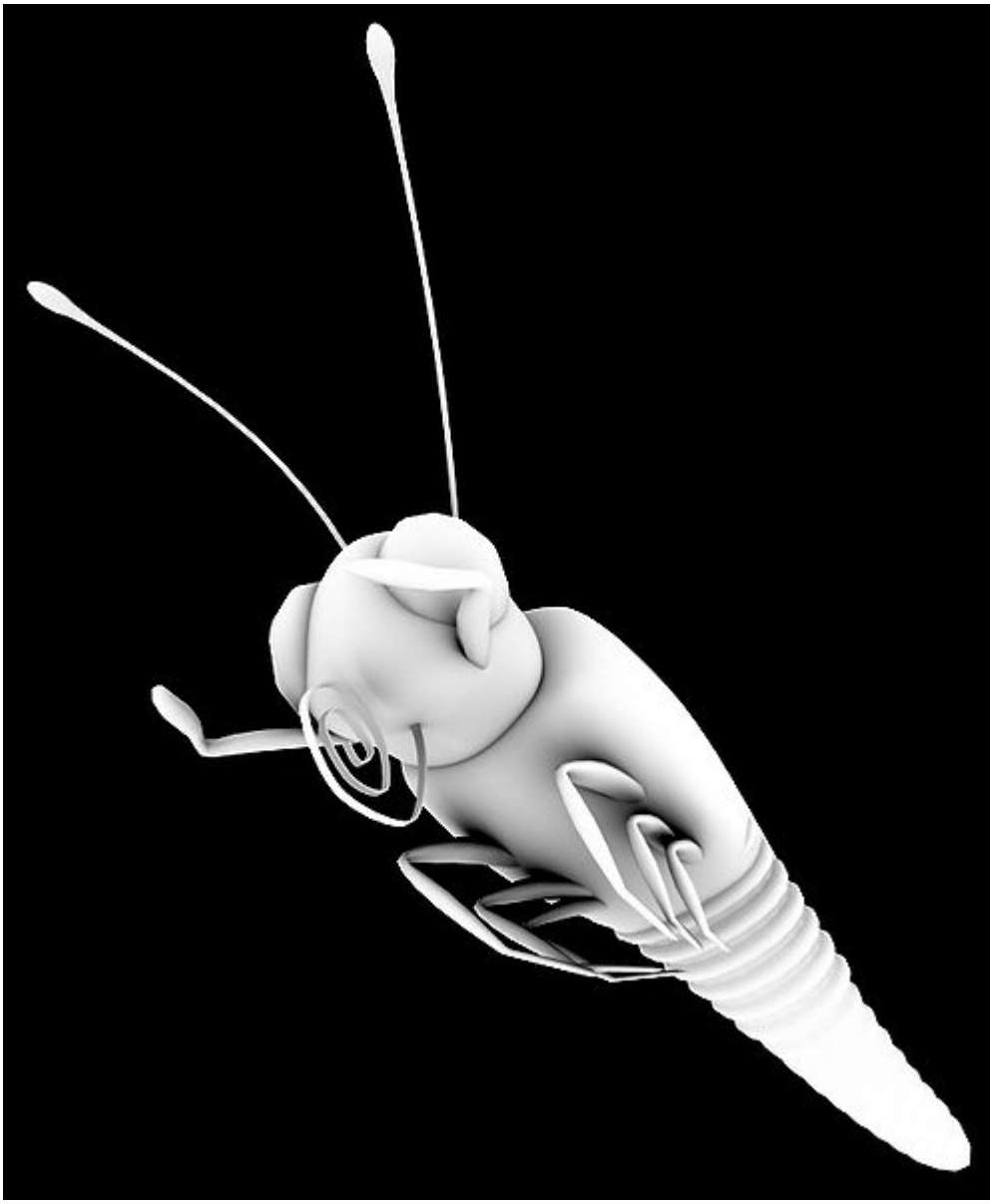
Unlike local methods like Phong shading, ambient occlusion is a global method, meaning the illumination at each point is a function of other geometry in the scene. However, it is a very crude approximation to full global illumination. The soft appearance achieved by ambient occlusion alone is similar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

Method of implementation

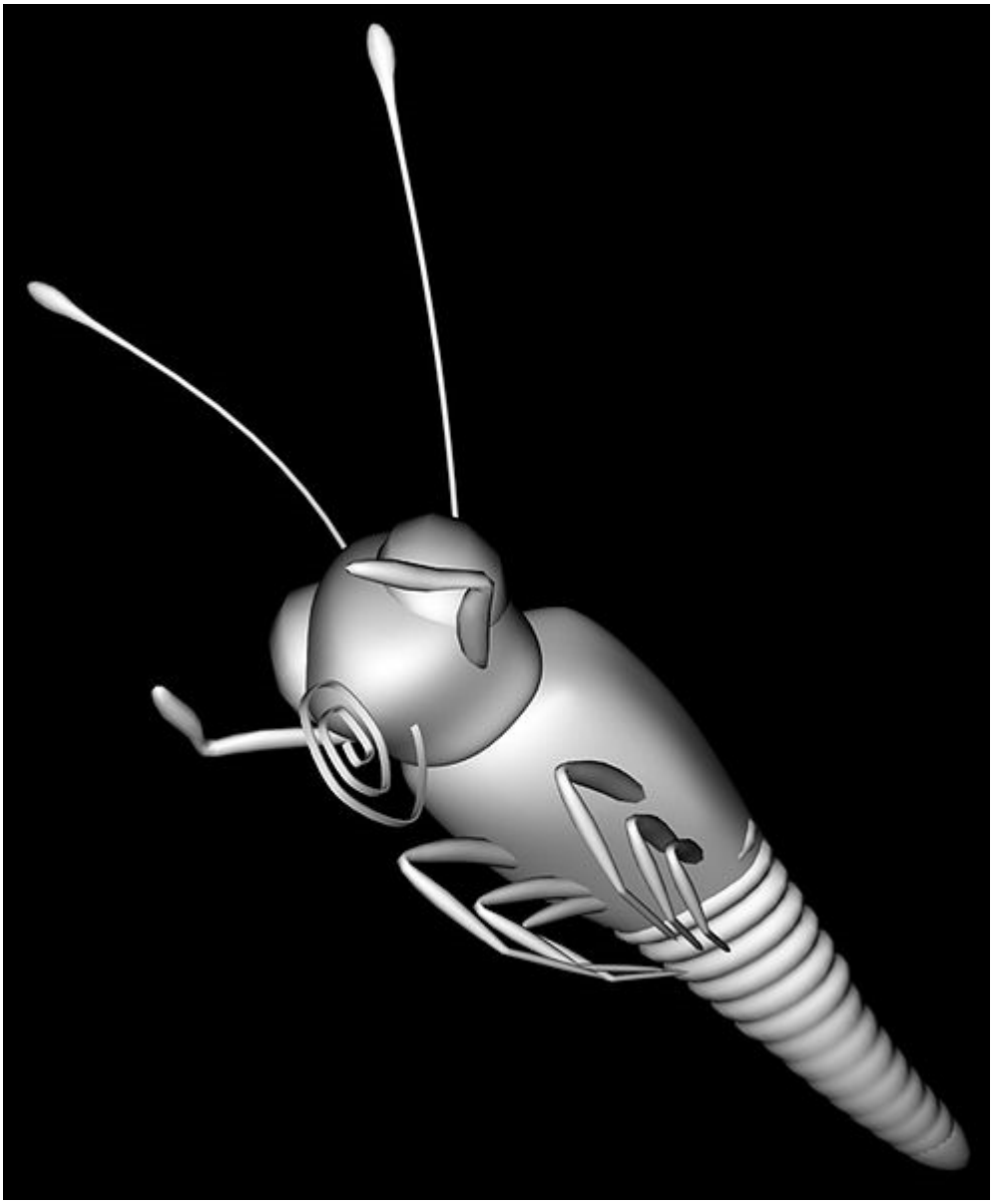
Ambient occlusion is most often calculated by casting rays in every direction from the surface. Rays which reach the background or “sky” increase the brightness of the surface, whereas a ray which hits any other object contributes no illumination. As a result, points surrounded by a large amount of geometry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

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

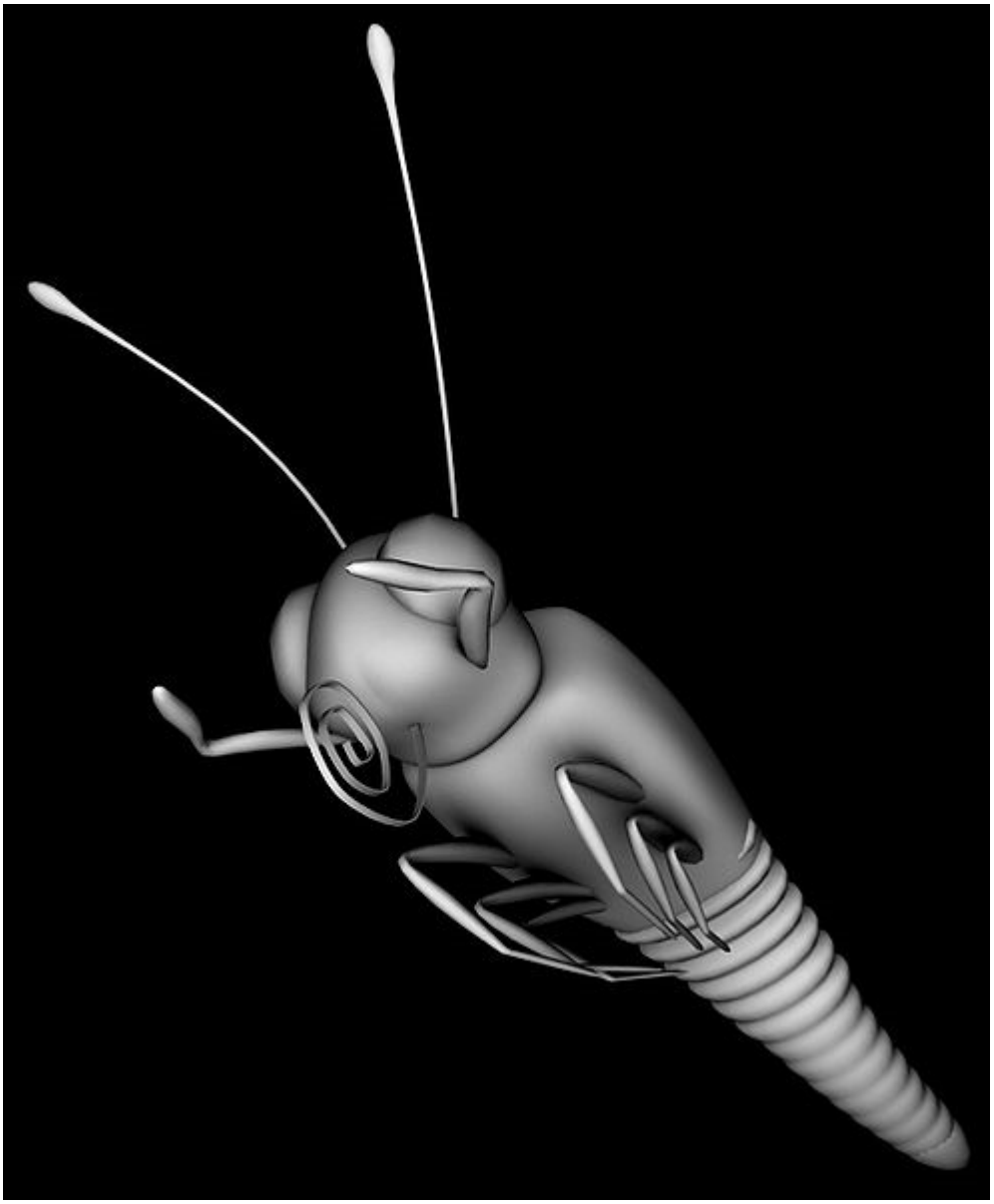
The ambient occlusion shading model has the nice property of offering a better perception of the 3d shape of the displayed objects. This was shown in a paper where the authors report the results of perceptual experiments showing that depth discrimination under diffuse uniform sky lighting is superior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



ambient occlusion



diffuse only



combined ambient and diffuse

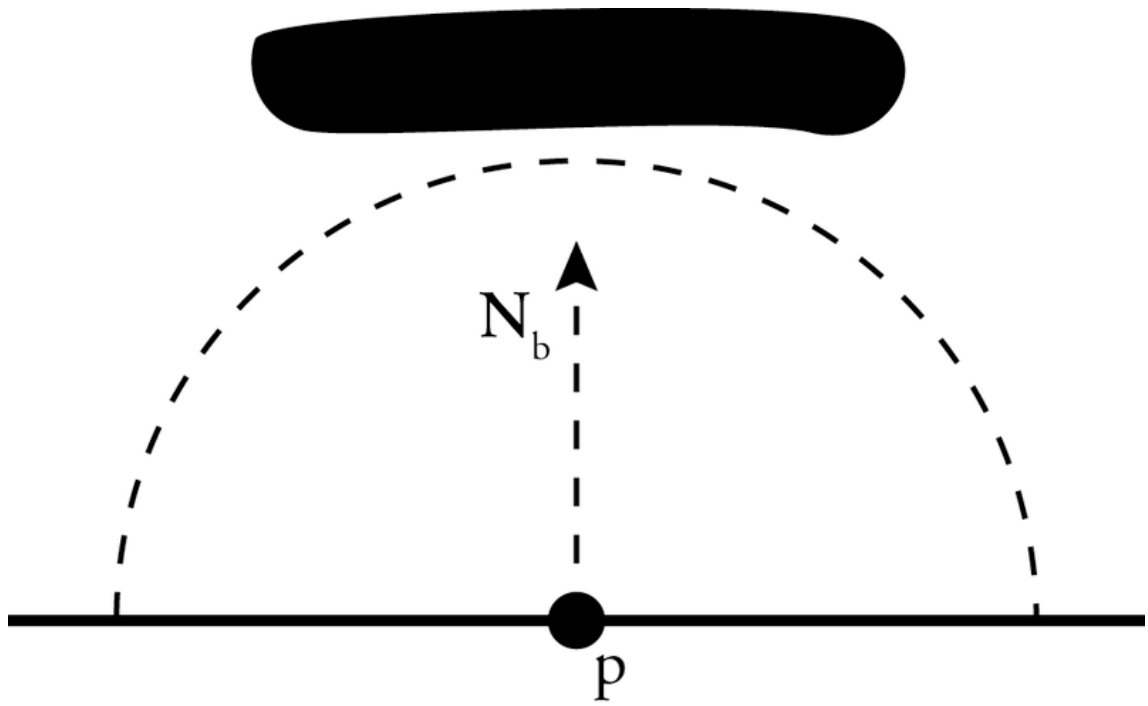
The occlusion A_p at a point p on a surface with normal N can be computed by integrating the visibility function over the hemisphere Ω with respect to projected solid angle:

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

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.,



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

Scientific and Technical Academy Award

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

Bloom (shader effect)

An example of bloom in a computer-generated image (from *Elephants Dream*.) The light on the bright background bleeds on the darker areas, such as the walls and the characters.

An example of bloom in a picture taken with a camera. Note the blue fringe that's particularly noticeable along the right edge of the window.

Bloom (sometimes referred to as **light bloom** or **glow**) is a computer graphics effect used in computer games, demos and high dynamic range rendering (HDR) to reproduce an imaging artifact of real-world cameras. The effect produces fringes (or feathers) of light around very bright objects in an image.

Theory

The physical basis of bloom is that, in the real world, lenses can never focus perfectly. Even a perfect lens will convolve the incoming image with an Airy disc (the diffraction pattern produced by passing a point light source through a circular aperture) . Under

normal circumstances, these imperfections aren't noticeable, but an intensely bright light source will cause the imperfections to become visible. As a result, the image of the bright light appears to bleed beyond its natural borders.

The Airy disc function falls off very quickly but has very wide tails (actually, infinitely wide tails). As long as the brightness of adjacent parts of the image are roughly in the same range, the effect of the blurring caused by the Airy disc is not particularly noticeable; but in parts of the image where very bright parts are adjacent to relatively darker parts, the tails of the Airy disc become visible, and can extend far beyond the extent of the bright part of the image.

In HDR images, the effect can be re-produced by convolving the image with a windowed kernel of an Airy disc (for very good lenses), or by applying Gaussian blur (to simulate the effect of a less perfect lens), before converting the image to fixed-range pixels. The effect can't be fully reproduced in non-HDR imaging systems, because the amount of bleed depends on how bright the bright part of the image is.

As an example, if a picture is taken indoors, the brightness of outdoor objects seen through a window may be 70 or 80 times brighter than objects inside the room. If exposure levels are set for objects inside the room, windows will be bright enough, when convolved with the Airy disc of the camera being used to produce the image, to cause the image of the windows to bleed past the frames of the window.

Use

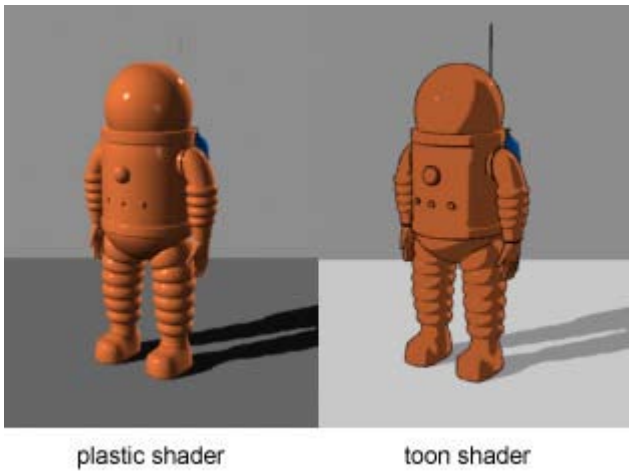
The effect is popular in current generation games, and is used heavily in PC, Xbox 360 and PlayStation 3 games as well as the popular Nintendo GameCube and Wii releases, The Legend of Zelda: Twilight Princess, Metroid Prime, and Metroid Prime 2: Echoes. Current generation gaming systems are able to render 3D graphics using floating point frame buffers, in order to produce HDR images. To produce the bloom effect, the HDR images in the frame buffer are convolved with a convolution kernel in a post-processing step, before converting to RGB space. The convolution step usually requires the use of a large gaussian kernel that is not appropriate for realtime graphics, causing the programmers to utilize approximation methods.

The Wii game Little King's Story also utilizes the effect to give the game a fantasy look, which combines storybook elements and colorful environments.

The Afterimage effect is sometimes used to enhance the bloom effect.

Bloom became very popular after *Tron 2.0* and is used in many games, modifications, and game engines such as *Quake Live* and *Cube 2: Sauerbraten*. *Ico* was one of the earliest games to use the bloom effect.

Cel-shaded animation



Object with a basic cel-shader (also known as a *toon shader*) and border detection

Cel-shaded animation (also known as **Cel 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 somewhat recent addition to computer graphics, most commonly turning up in video games. However, the end result of cel-shading has a very simplistic feel like that of hand-drawn animation. The name comes from the clear sheets of acetate, called cels, which are painted on for use in traditional 2D animation, such as Disney classics.

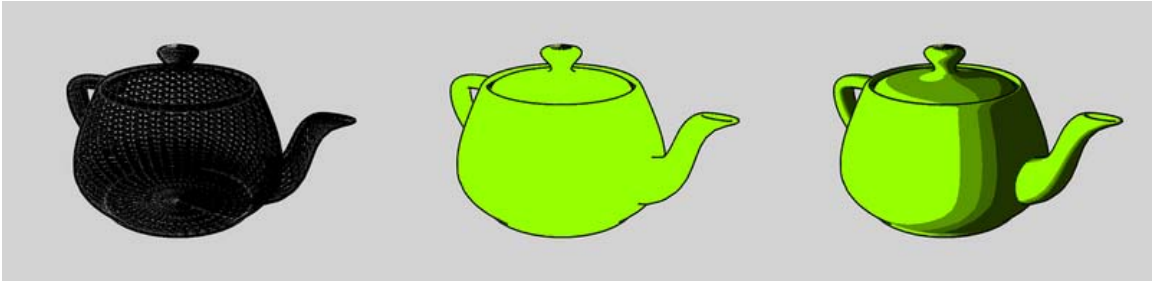
Process

The cel-shading process starts with a typical 3D model. Where cel-shading differs from conventional rendering is in its use of non-photorealistic lighting. Conventional (smooth) lighting values are calculated for each pixel and then mapped to a small number of discrete shades to create the characteristic flat look – where the shadows and highlights appear more like blocks of color rather than mixed in a smooth way.

Black "ink" outlines and contour lines can be created using a variety of methods. One popular method is to first render a black outline, slightly larger than the object itself. Backface culling is inverted and the back-facing triangles are drawn in black. To dilate the silhouette, these back faces may be drawn in wireframe multiple times with slight changes in translation. Alternately, back-faces may be rendered solid-filled, with their vertices translated along their vertex normals in a vertex shader. After drawing the outline, back-face culling is 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 interior contour lines. Popularly, this "ink" outline applied to animation

and games is what's called cel shading, while originally the term referred to the shading technique, regardless of whether outline is being applied or not.

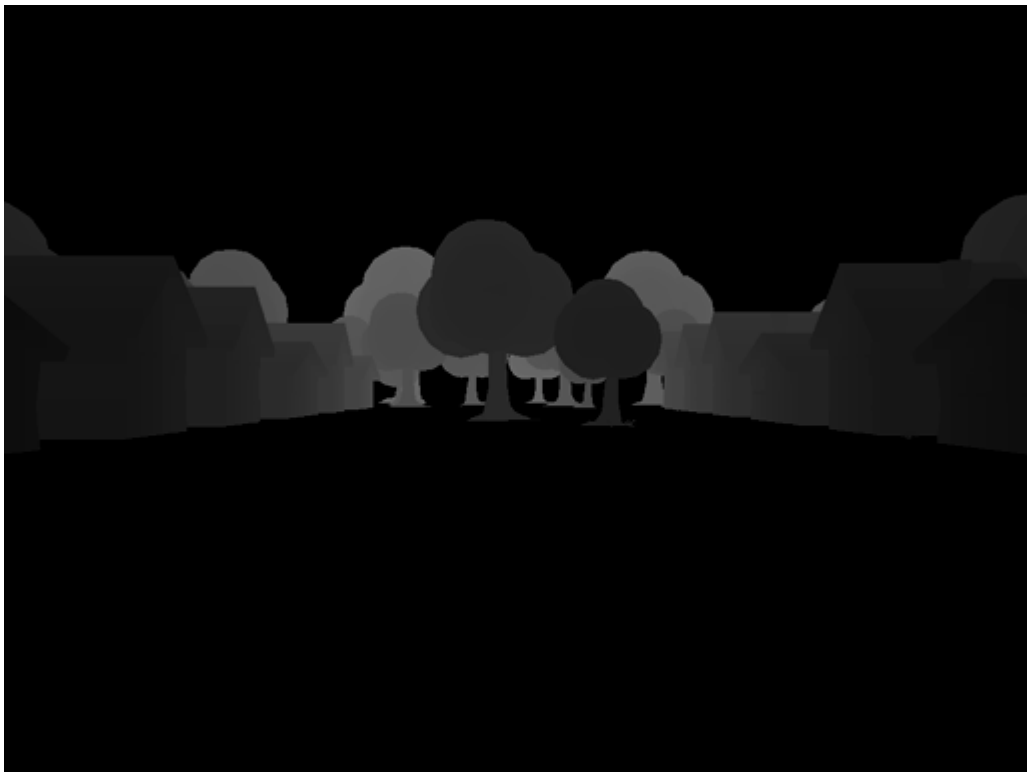
The Utah teapot rendered using cel-shading:



1. The back faces are drawn with thick lines
2. The object is drawn with a basic texture
3. Shading

Steps 2 and 3 can be combined using multi-texturing.

Another outlining technique is to use 2D image-processing. First, the scene is rendered (with cel-shading) to a screen-sized color texture:



Finally, the edge texture and the color texture are composited to produce the final rendered image:

Chapter 10

Cg (Programming Language) and Deferred Shading

Cg (programming language)

Cg or **C for Graphics** is a high-level shading language developed by Nvidia in close collaboration with Microsoft for programming vertex and pixel shaders. It is very similar to Microsoft's HLSL.

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.

This language is only suitable for GPU programming and it does not replace a general programming language.

The Cg compiler outputs DirectX or OpenGL shader programs.

Background

As a result of technical advancements in graphics hardware, 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, while 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 and 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 their 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.

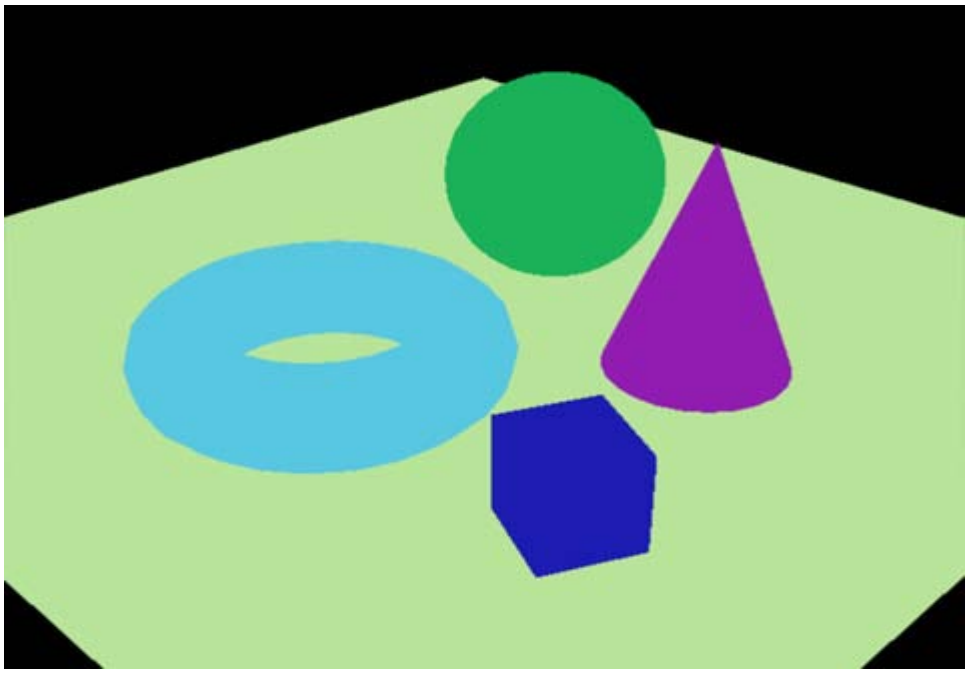
In addition to being able to compile Cg source to assembly code, the Cg runtime also has the ability to compile shaders during execution of the supporting program. This allows the runtime to compile the shader using the latest optimizations available for hardware that the program is currently executing on. However, this technique requires that the source code for the shader be available in plain text to the compiler, allowing the user of the program to access the source-code for the shader. Some developers view this as a major drawback of this technique.

To avoid exposing the source code of the shader, and still maintain some of the hardware specific optimizations, the concept of profiles was developed. Shaders can be compiled to suit different graphics hardware platforms (according to profiles). When executing the supporting program, the best/most optimized shader is loaded according to its profile. For instance there might be a profile for a graphics card that supports complex pixel shaders, and another profile for one that supports only minimal pixel shaders. By creating a pixel shader for each of these profiles a supporting program enlarges the number of supported hardware platforms without sacrificing picture quality on powerful systems.

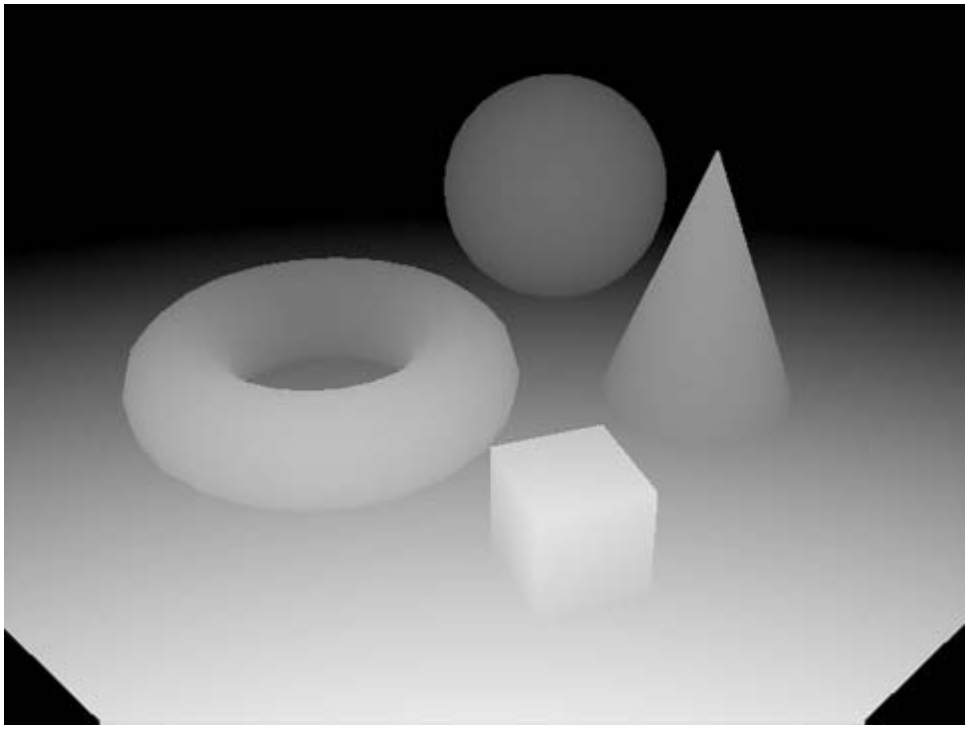
A sample Cg vertex shader

```
 // 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
coords
     OUT.color   = IN.color; // copy input color to output
     OUT.color.z = 1.0f; // blue component of color = 1.0f
     return OUT;
 }
```

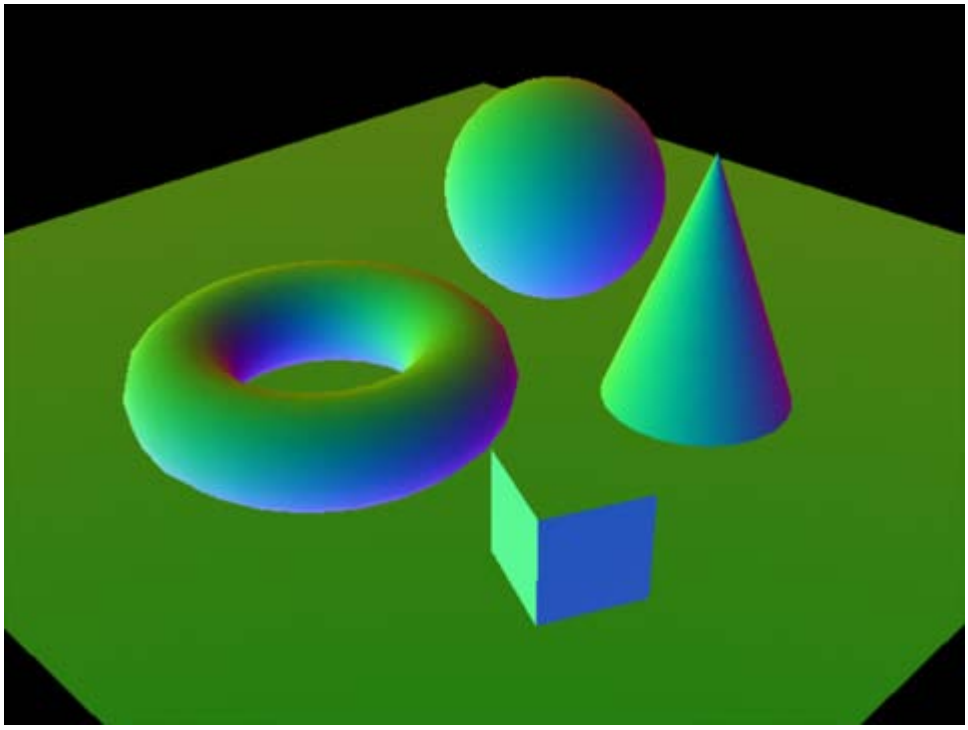
Deferred shading



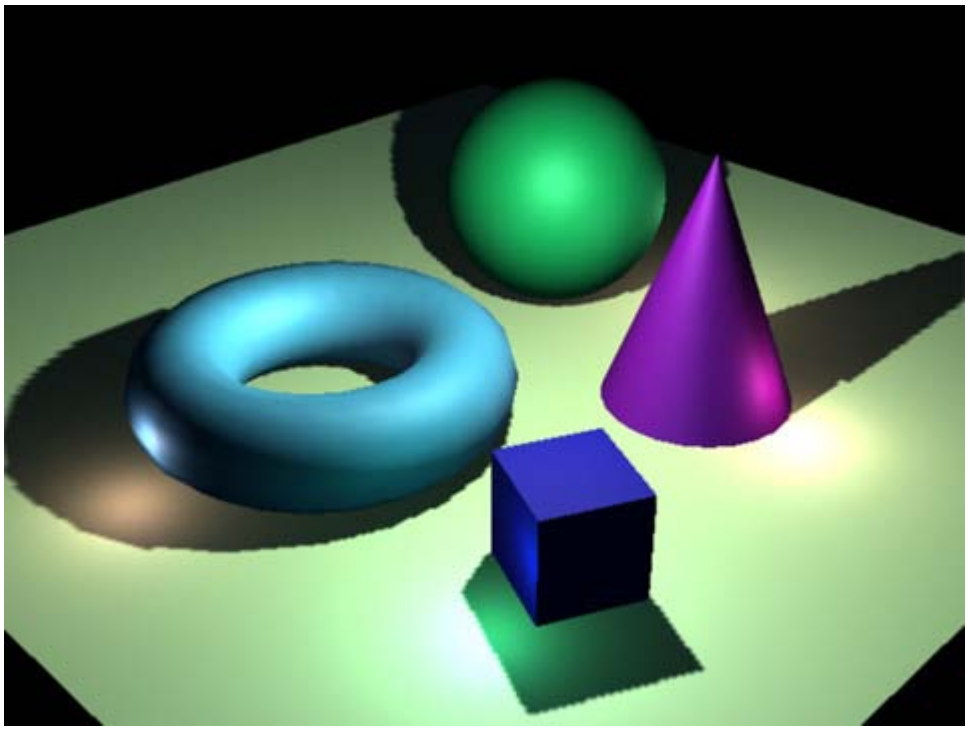
Color G-Buffer



Z-Buffer



Normal G-Buffer



Final Compositing

In computer graphics, **deferred shading** is a three dimensional shading technique in which the result of a shading algorithm is calculated by dividing it into smaller parts that are written to intermediate buffer storage to be combined later, instead of immediately writing the shader result to the color framebuffer. Implementations on modern hardware tend to use multiple render targets (MRT) to avoid redundant vertex transformations. Usually once all the needed buffers are built they are then read (usually as input textures) into a shading algorithm (for example a lighting equation) and combined to produce the final result. In this way the computation and memory bandwidth required to shade a scene is reduced to those visible portions, thereby reducing the shaded depth complexity.

This primary advantage of deferred rendering is partially mitigated by the availability of coarse Z-buffer and early Z-buffer testing, however other advantages are still claimed for the approach. These advantages may include the simpler management of complex lighting resources, ease of managing other complex shader resources and the simplification of the software rendering pipeline. These are somewhat more marginal advantages that software developers decide on during their development process and they are endlessly debated.

A specific use of deferred shading is deferred lighting. The render targets, collectively called a g-buffer (the G being short for geometry) store various parameters for the lighting shaders. Common examples of stored data are color, surface normal, and view-space position, though they can in theory store whatever data is needed for the lighting stage. Because of the use of multiple render targets, often using a floating point format, the memory bandwidth is higher than forward-rendering techniques. Modern graphics hardware performs mathematical operations faster than it does memory access. This factor, along with the expectation that the performance between math and memory will become more biased in future hardware, indicate that deferred lighting may not be a long term solution.

One key disadvantage of deferred rendering is the inability to handle transparency within the algorithm, although this problem is a generic one in Z-buffered scenes and it tends to be handled by delaying and sorting the rendering of transparent portions of the scene. Depth peeling can be used to achieve order-independent transparency in deferred rendering, but at the cost of additional batches and g-buffer size. Modern hardware, supporting DirectX 10 and later, is often capable of performing batches fast enough to maintain interactive frame rates. When order-independent transparency is desired (commonly for consumer applications) deferred shading is no less effective than forward shading using the same technique.

Another rather important disadvantage is that, due to separating the lighting stage from the geometric stage, hardware anti-aliasing does not produce correct results any more: although the first pass used when rendering the basic properties (diffuse, normal etc.) can use anti-aliasing, it's not until full lighting has been applied that anti-alias is needed. One of the usual techniques to overcome this limitation is using edge detection on the final image and then applying blur over the edges. DirectX 10 introduced features allowing shaders to access individual samples in multisampled render targets (and depth buffers in

version 10.1), making hardware anti-aliasing possible in deferred shading. These features also make it possible to correctly apply HDR luminance mapping to anti-aliased edges, where in earlier hardware any benefit of anti-aliasing may be lost, making this form of anti-aliasing desirable in any case.

The technique is increasingly being used in video games because of the control it enables in terms of using a large amount of dynamic lights and reducing the complexity of required shader instructions. Sony Computer Entertainment has several games that use deferred rendering including Guerilla Games's *Killzone 2*, Media Molecule's *LittleBigPlanet*, and Sucker Punch Productions' *inFamous*. Other games known to use deferred shading are GSC Game World's *Stalker: Shadow of Chernobyl*, Electronic Arts' *Dead Space*, NCSoft's *Tabula Rasa*, Realtime Worlds' *Crackdown*, Rockstar Games' *Grand Theft Auto IV* and Blizzard Entertainment's *StarCraft II* . Crytek's *CryEngine 3* also implements deferred lighting. Halo Reach, Bungie (Microsoft).

History

The idea of deferred shading was originally introduced by Michael Deering and his colleagues in a paper published in 1988 entitled *The triangle processor and normal vector shader: a VLSI system for high performance graphics*. Although the paper never uses the word "deferred", a key concept is introduced; each pixel is shaded only once after depth resolution. Deferred shading as we know it today, using G-buffers, was introduced in a paper by Saito and Takahashi in 1990, although they too do not use the word "deferred". Around 2004 implementations on commodity graphics hardware started to appearing. The technique later gained popularity for applications such as video games, finally becomming mainstream around 2008 to 2010.

Chapter 11

GLSL and High Level Shader Language

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 to give developers more direct control of the graphics pipeline without having to use assembly language or hardware-specific languages. Current specification for GLSL is version 4.10.

Background

With advances 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 by writing shaders in assembly language – a complex and unintuitive task. 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.

Originally introduced as an extension to OpenGL 1.4, GLSL was formally included into the OpenGL 2.0 core by the OpenGL ARB. It was 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 GNU/Linux, Mac OS and Windows.
- 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.

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 pointers. Bitwise operators were added in version 1.30.

Functions and control structures

Similar to the C programming language, GLSL supports loops and branching, including: 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 math library of 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, which is available on many different platforms (e.g., GNU/Linux, Mac OS, Windows). There are language bindings for C, C++, C#, Delphi, Java and many more.

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.

The set of APIs used to compile, link, and pass parameters to GLSL programs are specified in three OpenGL extensions, and became part of core OpenGL as of OpenGL Version 2.0, except for geometry shaders, which were included as of OpenGL Version 3.2. These OpenGL APIs are found in the extensions:

- ARB vertex shader
- ARB fragment shader
- ARB shader objects
- ARB geometry shader 4

A sample trivial GLSL Vertex Shader

This transforms the input vertex the same way the fixed-function pipeline would.

```
void main(void)
```

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

Note that ftransform() is no longer available since GLSL 1.40 and GLSL ES 1.0. Instead, the programmer has to manage the projection and modelview matrices explicitly in order to comply with the new OpenGL 3.1 standard.

```
#version 140

uniform Transformation {
    mat4 projection_matrix;
    mat4 modelview_matrix;
};

in vec3 vertex;

void main() {
    gl_Position = projection_matrix * modelview_matrix * vec4(vertex,
1.0);
}
```

A sample trivial GLSL Geometry Shader

This is a simple pass-through shader for the color and position.

```
#version 120
#extension GL_EXT_geometry_shader4 : enable

void main() {
  for(int i = 0; i < gl_VerticesIn; ++i) {
    gl_FrontColor = gl_FrontColorIn[i];
    gl_Position = gl_PositionIn[i];
    EmitVertex();
  }
}
```

Since OpenGL 3.2 with GLSL 1.50 geometry shaders were adopted into core functionality which means there is no need to use extensions. However, the syntax is a bit different. This is a simple version 1.50 pass-through shader for vertex positions (of triangle primitives):

```
#version 150

layout(triangles) in;
layout(triangle_strip, max_vertices = 3) out;

void main() {
  for(int i = 0; i < gl_in.length(); i++) {
    gl_Position = gl_in[i].gl_Position;
    EmitVertex();
  }
  EndPrimitive();
```

```
}
```

A sample trivial GLSL Fragment Shader

This produces a red fragment.

```
#version 120
void main(void)
{
    gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);
}
```

In GLSL 1.30 and later you must do

```
glBindFragDataLocation(Program, 0, "MyFragColor");
```

where:

- Program - your shader program's handle;
- 0 - count of color buffers, if you are not using multiple render targets, you must write zero;
- "MyFragColor" - name of your color buffer in shader program.

```
#version 150
void main(void)
{
    MyFragColor = vec4(1.0, 0.0, 0.0, 1.0);
}
```

High Level Shader Language

The **High Level Shader Language** or **High Level Shading Language (HLSL)** is a proprietary shading language developed by Microsoft for use with the Microsoft Direct3D API. It is analogous to the GLSL shading language used with the OpenGL standard. It is the same as the Nvidia Cg shading language, as it was developed alongside it.

HLSL programs come in three forms, vertex shaders, geometry shaders, and pixel (or fragment) shaders. A vertex shader is executed for each vertex that is submitted by the application, and is primarily responsible for transforming the vertex from object space to view space, generating texture coordinates, and calculating lighting coefficients such as the vertex's tangent, binormal and normal vectors. When a group of vertices (normally 3, to form a triangle) come through the vertex shader, their output position is interpolated to form pixels within its area; this process is known as rasterisation. Each of these pixels comes through the pixel shader, whereby the resultant screen colour is calculated.

Optionally, an application using a Direct3D 10 interface and Direct3D 10 hardware may also specify a geometry shader. This shader takes as its input the three vertices of a triangle and uses this data to generate (or tessellate) additional triangles, which are each then sent to the rasterizer.

Shader model comparison

Pixel shader comparison

Pixel shader version	1.0 to 1.3	1.4	2.0	2.0a	2.0b	3.0	4.0
Dependent texture limit	4	6	8	Unlimited	8	Unlimited	Unlimited
Texture instruction limit	4	6*2	32	Unlimited	Unlimited	Unlimited	Unlimited
Position register	No	No	No	No	No	Yes	Yes
Instruction slots	8+4	8+4	32 + 64	512	512	≥ 512	≥ 65536
Executed instructions	8+4	6*2+8*2	32 + 64	512	512	65536	Unlimited
Texture indirections	4	4	4	No limit	4	No Limit	No Limit
Interpolated registers	2 + 8	2 + 8	2 + 8	2 + 8	2 + 8	10	32
Instruction predication	No	No	No	Yes	No	Yes	No
Index input registers	No	No	No	No	No	Yes	Yes
Temp registers	2	6	12 to 32	22	32	32	4096
Constant registers	8	8	32	32	32	224	16x4096
Arbitrary swizzling	No	No	No	Yes	No	Yes	Yes
Gradient instructions	No	No	No	Yes	No	Yes	Yes
Loop count register	No	No	No	No	No	Yes	Yes
Face register (2-sided lighting)	No	No	No	No	No	Yes	Yes
Dynamic flow control	No	No	No	No	No	24	Yes
Bitwise Operators	No	No	No	No	No	No	Yes
Native Integers	No	No	No	No	No	No	Yes

- **PS 2.0** = DirectX 9.0 original **Shader Model 2** specification.
- **PS 2.0a** = NVIDIA GeForce FX-optimized model.
- **PS 2.0b** = ATI Radeon X700, X800, X850 shader model, DirectX 9.0b.
- **PS 3.0** = **Shader Model 3**.
- **PS 4.0** = **Shader Model 4**.

"32 + 64" for *Executed Instructions* means "32 texture instructions and 64 arithmetic instructions."

Vertex shader comparison

Vertex shader version	VS 1.1	VS 2.0	VS 2.0a	VS 3.0	VS 4.0
# of instruction slots	128	256	256	≥ 512	4096
Max # of instructions executed	?????	65536	65536	65536	65536
Instruction predication	No	No	Yes	Yes	Yes
Temp registers	12	12	13	32	4096
# constant registers	≥ 96	≥ 256	≥ 256	≥ 256	16x4096
Static Flow Control	???	Yes	Yes	Yes	Yes
Dynamic Flow Control	No	No	Yes	Yes	Yes
Dynamic Flow Control Depth	No	No	24	24	Yes
Vertex Texture Fetch	No	No	No	Yes	Yes
# of texture samplers	N/A	N/A	N/A	4	128
Geometry instancing support	No	No	No	Yes	Yes
Bitwise Operators	No	No	No	No	Yes
Native Integers	No	No	No	No	Yes

- **VS 2.0** = DirectX 9.0 original **Shader Model** specification.
- **VS 2.0a** = NVIDIA GeForce FX-optimized model.
- **VS 3.0** = **Shader Model 3**. Note that ATI X1000 series cards (e.g X1900) doesn't support Vertex Texture Fetch, hence it does not fully comply with the VS 3.0 model
- **VS 4.0** = **Shader Model 4**.

Chapter 12

Pixel Shader, Shadow Mapping and Shadow Volume

Pixel shader

A **pixel shader** is a type of shader program, often executed on a graphics processing unit. These programs are typically used to perform complex per-pixel effects. Microsoft's Direct3D and Silicon Graphics' OpenGL support shaders. (Note: In OpenGL parlance a pixel is called a *fragment*, so OpenGL calls these **fragment shaders**).

Function

A pixel shader is a computation kernel function that computes color and other attributes of each pixel. Pixel shaders range from always outputting the same color, to applying a lighting value, to doing bump mapping, shadows, specular highlights, translucency and other phenomena. They can alter the depth of the pixel (for Z-buffering), or output more than one color if multiple render targets are active. A pixel shader alone cannot produce very complex effects, because it operates only on a single pixel, without knowledge of a scene's geometry.

History

The term "Shader" originated with Pixar's RenderMan – a program that takes an entire description of a scene including camera positions, object geometry and renders the final output. RenderMan was introduced in 1989, but it wasn't until the 1995 release of Pixar's movie "Toy Story" that the general public was introduced to the power of RenderMan. Such computer-generated imagery (CGI) became more and more popular in movies and television.

Consumer-level computer graphics hardware was also evolving rapidly and new features were implemented on "commodity" boards that rivaled expensive dedicated graphics workstations. The video game industry began to utilize newly-created powerful-yet-cheap 3D graphics hardware in PCs and game consoles. Light maps in particular were soon finding their way into games, followed by bump maps and procedural vertex generation. The continuing desire for more complex visual effects pushed the computing industry

forward and previously fixed-function graphics processors received ever more programmable designs as manufacturing technology progressed. These more programmable GPUs allowed more complex effects, including pixel and vertex shader programs. This technology is used by Windows Vista Aero effect.

Programming

Pixel shaders use different languages depending on which API the graphics renderer uses. Originally they were written in various forms of assembly language, but as time grew on and shaders became more complex it became clear that a higher-level solution was needed. Popular high-level shader languages include DirectX's HLSL and OpenGL's GLSL, along with Cg, which can compile to both OpenGL and DirectX shader instructions.

Compatibility

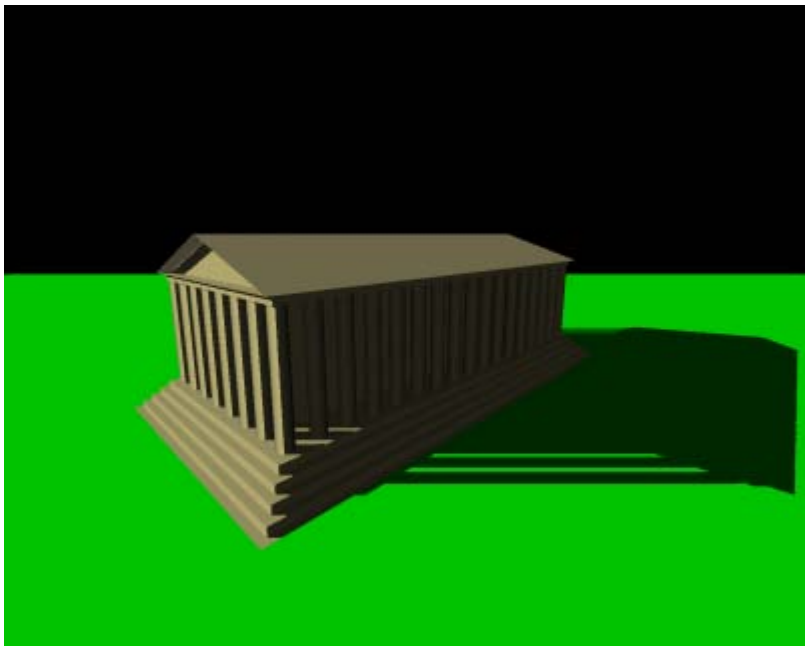
Hardware

This chart describes the maximum supported Shader Model for various graphics processors. GPUs are usually backwards compatible as well.

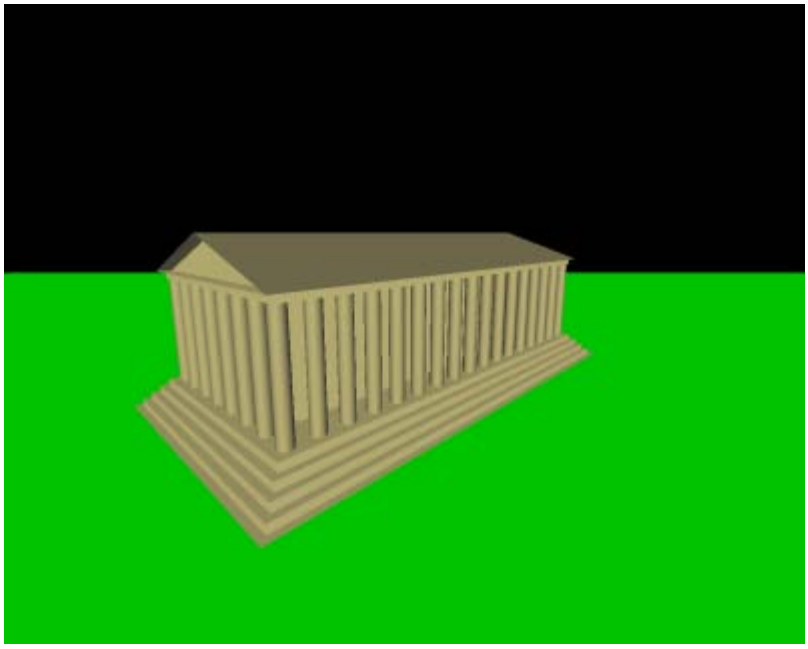
PS version	Direct 3D version	OpenGL version	3DLabs	ATI	Intel	Matrox	NVIDIA	S3 Graphics	SiS	XGI
1.0/1.1	8.0	—	—	—	—	—	GeForce 3 series	—	Xabre-Series	—
1.2	8.0a	—	Wildcat VP	—	—	—	—	—	—	—
1.3	8.0a	—	—	—	—	Parhelia series	GeForce 4 Ti / 4200Go series	—	Mirage 2	—
1.4	8.1	—	—	Radeon R200 (8500–9250)	—	—	—	—	—	Volari V3 series (except V3XT)
2.0	9.0	2.1	Wildcat Realizm	Radeon R300 (9500–9800, X300–X600), X1200 – X1250	Intel GMA 900, 950, 3000, 3100	—	—	DeltaChrome, GammaChrome, Chrome S2x series	Mirage 3, Mirage 3+	Volari V3XT, Volari V5 series, Volari V8 series, Volari 8300, Volari XP10
2.0a	9.0a	2.1	—	—	—	—	GeForce FX series	—	—	—
2.0b	9.0b	2.1	—	Radeon R420 (X700	—	—	—	—	—	—

				– X850)						
3.0	9.0c	2.1	—	Radeon R520 (X1300–X1950)	Intel GMA X3000 G965		GeForce 6 series, GeForce 7 series	—	—	—
4.0	10	3.3	—	Radeon R600 (HD 2400 – HD 2900)	Intel GMA X3100, X3500	—	GeForce 8 series, GeForce 9 Series, Quadro FX 1700, GeForce 200 Series (G92 and G200 chip), GeForce 300 Series (GT330)	—	*Mirage 4*	—
4.1	10.1	3.3	—	Radeon R600 (HD 3xxx), Radeon R700 (HD 4xxx)	Intel GMA 500	—	GeForce 200 Series (GT215/GT216/GT218 chips), GeForce 300 Series (Except GT330)	Chrome 400 Series	—	—
5.0	11	4.1	—	Radeon R800 (HD 5xxx), Radeon (HD 6xxx)	—	—	GeForce 400 Series, GeForce 500 Series	—	—	—

Shadow mapping



Scene with shadow mapping



Scene with no shadows

Shadow mapping or **projective shadowing** 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 realtime scenes in many console and PC games.

Shadows are created by testing whether a pixel is visible from the light source, by comparing it to a z-buffer or *depth* image of the light source's view, stored in the form of a texture.

Principle of a shadow and a shadow map

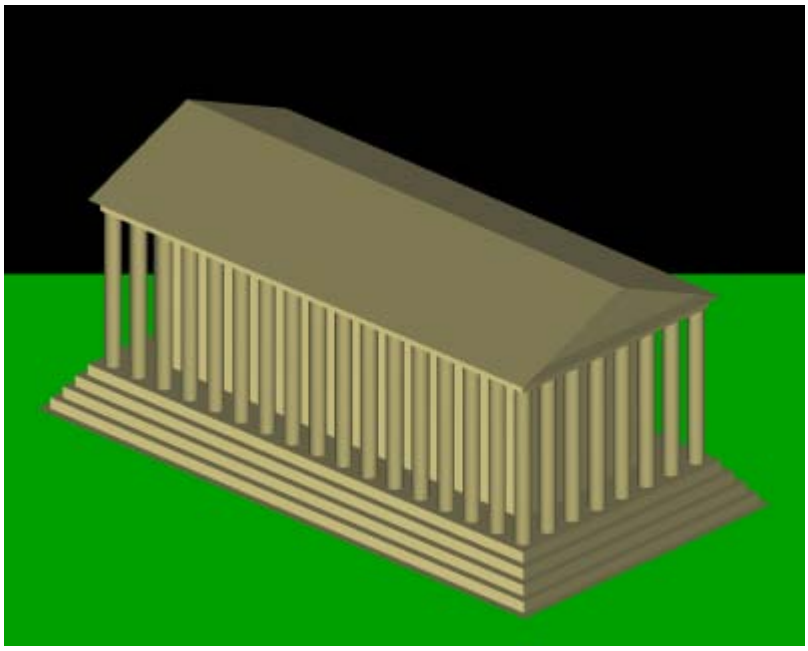
If you looked out from a source of light, all of the objects you can see would appear in light. Anything behind those objects, however, would be in shadow. This is the basic principle used to create a shadow map. The light's view is rendered, storing the depth of every surface it sees (the shadow map). Next, the regular scene is rendered comparing the depth of every point drawn (as if it were being seen by the light, rather than the eye) to this depth map.

This technique is less accurate than shadow volumes, but the shadow map can be a faster alternative depending on how much fill time is required for either technique in a particular application and therefore may be more suitable to real time applications. In addition, shadow maps do not require the use of an additional stencil buffer, and can be modified to produce shadows with a soft edge. Unlike shadow volumes, however, the accuracy of a shadow map is limited by its resolution.

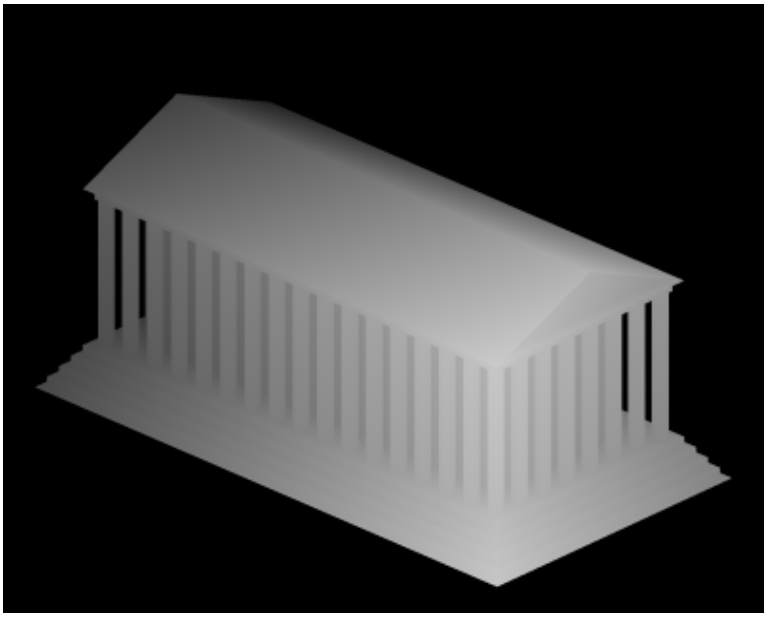
Algorithm overview

Rendering a shadowed scene involves two major drawing steps. The first produces the shadow map itself, and the second applies it to the scene. Depending on the implementation (and number of lights), this may require two or more drawing passes.

Creating the shadow map



Scene rendered from the light view



Scene from the light view, depth map

The first step renders the scene from the light's point of view. For a point light source, the view should be a perspective projection as wide as its desired angle of effect (it will be a sort of square spotlight). For directional light (e.g. that from the Sun), an orthographic projection should be used.

From this rendering, the depth buffer is extracted and saved. Because only the depth information is relevant, it is usual to avoid updating the color buffers and disable all lighting and texture calculations for this rendering, in order to save drawing time. This depth map is often stored as a texture in graphics memory.

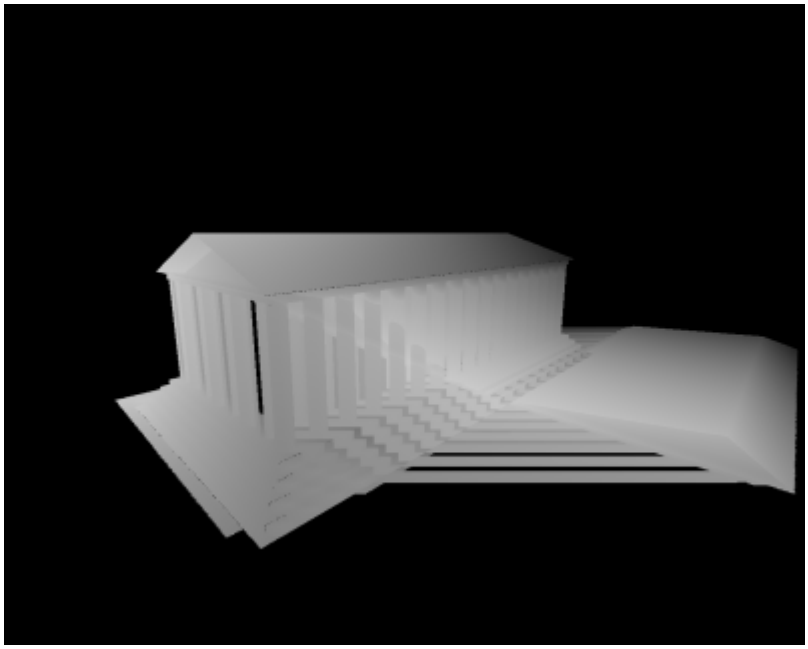
This depth map must be updated any time there are changes to either the light or the objects in the scene, but can be reused in other situations, such as those where only the viewing camera moves. (If there are multiple lights, a separate depth map must be used for each light.)

In many implementations it is practical to render only a subset of the objects in the scene to the shadow map in order to save some of the time it takes to redraw the map. Also, a depth offset which shifts the objects away from the light may be applied to the shadow map rendering in an attempt to resolve stitching problems where the depth map value is close to the depth of a surface being drawn (i.e. the shadow casting surface) in the next step. Alternatively, culling front faces and only rendering the back of objects to the shadow map is sometimes used for a similar result.

Shading the scene

The second step is to draw the scene from the usual camera viewpoint, applying the shadow map. This process has three major components, the first is to find the coordinates of the object as seen from the light, the second is the test which compares that coordinate against the depth map, and finally, once accomplished, the object must be drawn either in shadow or in light.

Light space coordinates



Visualization of the depth map projected onto the scene

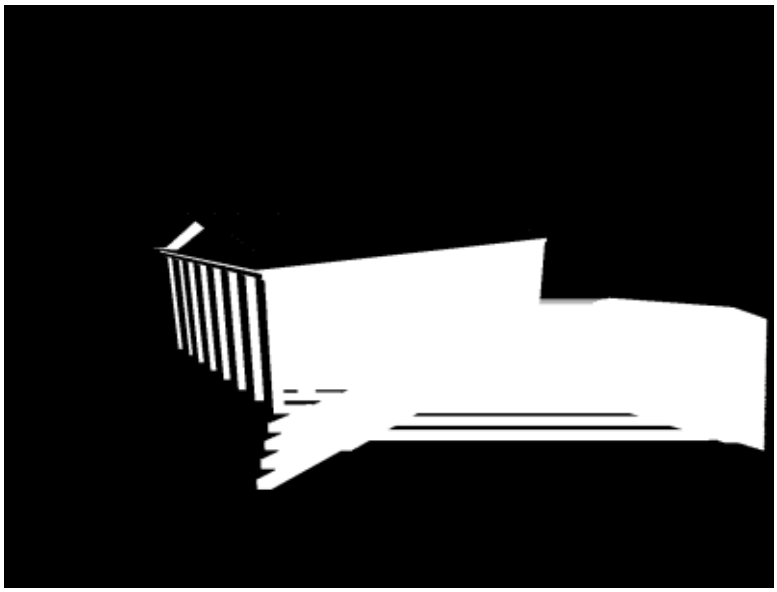
In order to test a point against the depth map, its position in the scene coordinates must be transformed into the equivalent position as seen by the light. This is accomplished by a matrix multiplication. The location of the object on the screen is determined by the usual coordinate transformation, but a second set of coordinates must be generated to locate the object in light space.

The matrix used to transform the world coordinates into the light's viewing coordinates is the same as the one used to render the shadow map in the first step (under OpenGL this is the product of the modelview and projection matrices). This will produce a set of homogeneous coordinates that need a perspective division to become *normalized device coordinates*, in which each component (x, y, or z) falls between -1 and 1 (if it is visible from the light view). Many implementations (such as OpenGL and Direct3D) require an additional *scale and bias* matrix multiplication to map those -1 to 1 values to 0 to 1, which are more usual coordinates for depth map (texture map) lookup. This scaling can be done before the perspective division, and is easily folded into the previous transformation calculation by multiplying that matrix with the following:

$$\begin{bmatrix} 0.5 & 0 & 0 & 0.5 \\ 0 & 0.5 & 0 & 0.5 \\ 0 & 0 & 0.5 & 0.5 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

If done with a shader, or other graphics hardware extension, this transformation is usually applied at the vertex level, and the generated value is interpolated between other vertices, and passed to the fragment level.

Depth map test



Depth map test failures

Once the light-space coordinates are found, the *x* and *y* values usually correspond to a location in the depth map texture, and the *z* value corresponds to its associated depth, which can now be tested against the depth map.

If the *z* value is greater than the value stored in the depth map at the appropriate (*x*,*y*) location, the object is considered to be behind an occluding object, and should be marked as a *failure*, to be drawn in shadow by the drawing process. Otherwise it should be drawn lit.

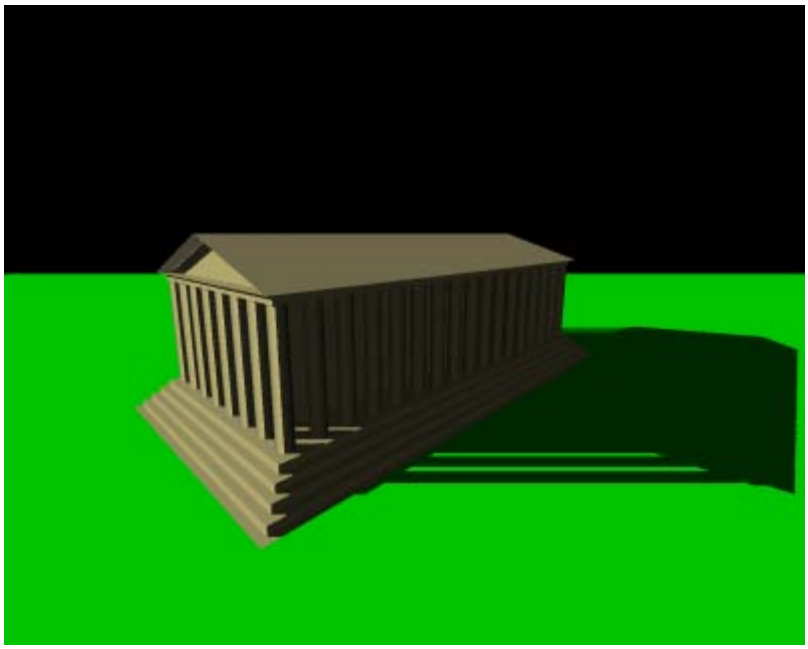
If the (*x*,*y*) location falls outside the depth map, the programmer must either decide that the surface should be lit or shadowed by default (usually lit).

In a shader implementation, this test would be done at the fragment level. Also, care needs to be taken when selecting the type of texture map storage to be used by the hardware: if interpolation cannot be done, the shadow will appear to have a sharp jagged edge (an effect that can be reduced with greater shadow map resolution).

It is possible to modify the depth map test to produce shadows with a soft edge by using a range of values (based on the proximity to the edge of the shadow) rather than simply pass or fail.

The shadow mapping technique can also be modified to draw a texture onto the lit regions, simulating the effect of a projector. The picture above, captioned "visualization of the depth map projected onto the scene" is an example of such a process.

Drawing the scene



Final scene, rendered with ambient shadows

Drawing the scene with shadows can be done in several different ways. If programmable shaders are available, the depth map test may be performed by a fragment shader which simply draws the object in shadow or lighted depending on the result, drawing the scene in a single pass (after an initial earlier pass to generate the shadow map).

If shaders are not available, performing the depth map test must usually be implemented by some hardware extension (such as *GL_ARB_shadow*), which usually do not allow a choice between two lighting models (lighted and shadowed), and necessitate more rendering passes:

1. Render the entire scene in shadow. For the most common lighting models this should technically be done using only the ambient component of the light, but this is usually adjusted to also include a dim diffuse light to prevent curved surfaces from appearing flat in shadow.
2. Enable the depth map test, and render the scene lit. Areas where the depth map test fails will not be overwritten, and remain shadowed.
3. An additional pass may be used for each additional light, using additive blending to combine their effect with the lights already drawn. (Each of these passes requires an additional previous pass to generate the associated shadow map.)

The example pictures here used the OpenGL extension *GL_ARB_shadow_ambient* to accomplish the shadow map process in two passes.

Shadow map real-time implementations

One of the key disadvantages of real time shadow mapping is that the size and depth of the shadow map determines the quality of the final shadows. This is usually visible as aliasing or shadow continuity glitches. A simple way to overcome this limitation is to increase the shadow map size, but due to memory, computational or hardware constraints, it is not always possible. Commonly used techniques for real-time shadow mapping have been developed to circumvent this limitation. These include Cascaded Shadow Maps, Trapezoidal Shadow Maps, Light Space Perspective Shadow maps or Parallel-Split Shadow maps.

Also notable is that generated shadows, even if aliasing free, have hard edges, which is not always desirable. In order to emulate real world soft shadows, several solutions have been developed, either by doing several lookups on the shadow map, generating geometry meant to emulate the soft edge or creating non standard depth shadow maps. Notable examples of these are Percentage Closer Filtering, Smoothies and Variance Shadow maps.

Shadow volume

Example of Carmack's stencil shadowing in *Doom 3*

Shadow volume is a technique used in 3D computer graphics to add shadows to a rendered scene. They were first proposed by Frank Crow in 1977 as the geometry describing the 3D shape of the region occluded from a light source. A shadow volume divides the virtual world in two: areas that are in shadow and areas that are not.

The stencil buffer implementation of shadow volumes is generally considered among the most practical general purpose real-time shadowing techniques for use on modern 3D graphics hardware. It has been popularized by the computer game *Doom 3*, and a particular variation of the technique used in this game has become known as **Carmack's Reverse**.

Shadow Volumes have become a popular tool for real-time shadowing, alongside the more venerable shadow mapping. The main advantage of shadow volumes is that they are accurate to the pixel (though many implementations have a minor self-shadowing problem along the silhouette edge), whereas the accuracy of a shadow map depends on the texture memory allotted to it as well as the angle at which the shadows are cast (at some angles, the accuracy of a shadow map unavoidably suffers). However, the shadow

volume technique requires the creation of shadow geometry, which can be CPU intensive (depending on the implementation). The advantage of shadow mapping is that it is often faster, because shadow volume polygons are often very large in terms of screen space and require a lot of fill time (especially for convex objects), whereas shadow maps do not have this limitation.

Construction

In order to construct a shadow volume, project a ray 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.

For a polygonal model, the volume is usually formed by classifying each face in the model as either facing toward the light source or facing away from the light source. The set of all edges that connect a toward-face to an away-face form the *silhouette* with respect to the light source. The edges forming the silhouette are extruded away from the light to construct the faces of the shadow volume. This volume must extend over the range of the entire visible scene; often the dimensions of the shadow volume are extended to infinity to accomplish this. To form a closed volume, the front and back end of this extrusion must be covered. These coverings are called "caps". Depending on the method used for the shadow volume, the front end may be covered by the object itself, and the rear end may sometimes be omitted.

There is also a problem with the shadow where the faces along the silhouette edge are relatively shallow. In this case, the shadow an object casts on itself will be sharp, revealing its polygonal facets, whereas the usual lighting model will have a gradual change in the lighting along the facet. This leaves a rough shadow artifact near the silhouette edge which is difficult to correct. Increasing the polygonal density will minimize the problem, but not eliminate it. If the front of the shadow volume is capped, the entire shadow volume may be offset slightly away from the light to remove any shadow self-intersections within the offset distance of the silhouette edge (this solution is more commonly used in shadow mapping).

The basic steps for forming a shadow volume are:

1. Find all silhouette edges (edges which separate front-facing faces from back-facing faces)
2. Extend all silhouette edges in the direction away from the light-source
3. Add a *front-cap* and/or *back-cap* to each surface to form a closed volume (may not be necessary, depending on the implementation used)

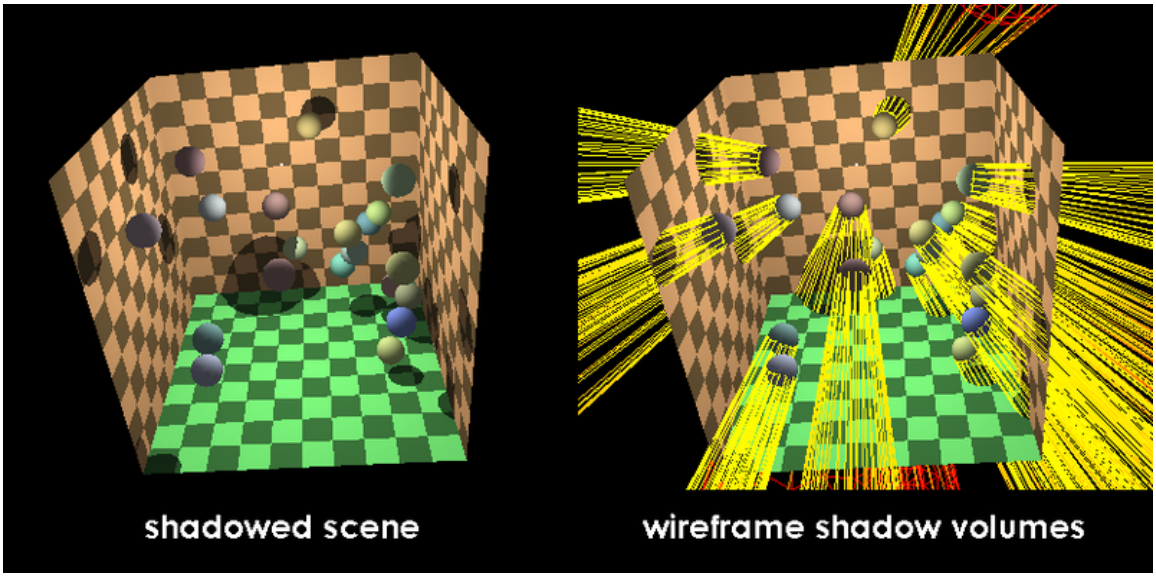


Illustration of shadow volumes. The image above at left shows a scene shadowed using shadow volumes. At right, the shadow volumes are shown in wireframe. Note how the shadows form a large conical area pointing away from the light source (the bright white point).

Stencil buffer implementations

After Crow, Tim Heidmann showed in 1991 how to use the stencil buffer to render shadows with shadow volumes quickly enough for use in real time applications. There are three common variations to this technique, **depth pass**, **depth fail**, and **exclusive-or**, but all of them use the same process:

1. Render the scene.
2. For each light source:
    1. Using the depth information from that scene, construct a mask in the stencil buffer that has holes only where the visible surface is not in shadow.
    2. Render the scene again as if it were completely lit, using the stencil buffer to mask the shadowed areas. Use additive blending to add this render to the scene.

The difference between these three methods occurs in the generation of the mask in the second step. Some involve two passes, and some only one; some require less precision in the stencil buffer. (These algorithms function well in both OpenGL and Direct3D.)

Shadow volumes tend to cover large portions of the visible scene, and as a result consume valuable rasterization time (fill time) on 3D graphics hardware. This problem is compounded by the complexity of the shadow casting objects, as each object can cast its own shadow volume of any potential size onscreen.

Depth pass

He proposed that if the front surfaces and back surfaces of the shadows were rendered in separate passes, the number of front faces and back faces in front of an object can be counted using the stencil buffer. If an object's surface is in shadow, there will be more front facing shadow surfaces between it and the eye than back facing shadow surfaces. If their numbers are equal, however, the surface of the object is not in shadow. The generation of the stencil mask works as follows:

1. Disable writes to the depth and color buffers.
2. Use back-face culling.
3. Set the stencil operation to increment on depth pass (only count shadows in front of the object).
4. Render the shadow volumes (because of culling, only their front faces are rendered).
5. Use front-face culling.
6. Set the stencil operation to decrement on depth pass.
7. Render the shadow volumes (only their back faces are rendered).

After this is accomplished, all lit surfaces will correspond to a 0 in the stencil buffer, where the numbers of front and back surfaces of all shadow volumes between the eye and that surface are equal.

This approach has answers when the eye itself is inside a shadow volume (for example, when the light source moves behind an object). From this point of view, the eye sees the back face of this shadow volume before anything else, and this adds a −1 bias to the entire stencil buffer, effectively inverting the shadows. This can be remedied by adding a "cap" surface to the front of the shadow volume facing the eye, such as at the front clipping plane. There is another situation where the eye may be in the shadow of a volume cast by an object behind the camera, which also has to be capped somehow to prevent a similar problem. In most common implementations, because properly capping for depth-pass can be difficult to accomplish, the depth-fail method (see below) may be licensed for these special situations. Alternatively one can give the stencil buffer a +1 bias for every shadow volume the camera is inside, though doing the detection can be slow.

There is another potential problem if the stencil buffer does not have enough bits to accommodate the number of shadows visible between the eye and the object surface, because it uses saturation arithmetic. (If they used arithmetic overflow instead, the problem would be insignificant.)

Depth pass testing is also known as **z-pass** testing, as the depth buffer is often referred to as the z-buffer.

Depth fail

Around the year 2000, several people discovered that Heidmann's method can be made to work for all camera positions by reversing the depth. Instead of counting the shadow surfaces in front of the object's surface, the surfaces behind it can be counted just as easily, with the same end result. This solves the problem of the eye being in shadow, since shadow volumes between the eye and the object are not counted, but introduces the condition that the rear end of the shadow volume must be capped, or shadows will end up missing where the volume points backward to infinity.

1. Disable writes to the depth and color buffers.
2. Use front-face culling.
3. Set the stencil operation to increment on depth fail (only count shadows behind the object).
4. Render the shadow volumes.
5. Use back-face culling.
6. Set the stencil operation to decrement on depth fail.
7. Render the shadow volumes.

The height fail method has the same considerations regarding the stencil buffer's precision as the depth pass method. Also, similar to depth pass, it is sometimes referred to as the **z-fail** method.

William Bilodeau and Michael Songy discovered this technique in October 1998, and presented the technique at Creativity, a Creative Labs developer's conference, in 1999 . Sim Dietrich presented this technique at a Creative Labs developer's forum in 1999 . A few months later, William Bilodeau and Michael Songy filed a US patent application for the technique the same year, US 6384822, entitled "Method for rendering shadows using a shadow volume and a stencil buffer" issued in 2002. John Carmack of id Software independently discovered the algorithm in 2007 during the development of *Doom 3* . Since he advertised the technique to the larger public, it is often known as **Carmack's Reverse**.

Exclusive-Or

Either of the above types may be approximated with an Exclusive-Or variation, which does not deal properly with intersecting shadow volumes, but saves one rendering pass (if not fill time), and only requires a 2-bit stencil buffer. The following steps are for the depth pass version:

1. Disable writes to the depth and color buffers.
2. Set the stencil operation to XOR on depth pass (flip on any shadow surface)
3. Render the shadow volumes.

Optimization

- One method of speeding up the shadow volume geometry calculations is to utilize existing parts of the rendering pipeline to do some of the calculation. For instance, by using homogeneous coordinates, the *w*-coordinate may be set to zero to extend a point to infinity. This should be accompanied by a viewing frustum that has a far clipping plane that extends to infinity in order to accommodate those points, accomplished by using a specialized projection matrix. This technique reduces the accuracy of the depth buffer slightly, but the difference is usually negligible.

- Rasterization time of the shadow volumes can be reduced by using an in-hardware scissor test to limit the shadows to a specific onscreen rectangle.

- NVIDIA has implemented a hardware capability called the *depth bounds test* that is designed to remove parts of shadow volumes that do not affect the visible scene. (This has been available since the GeForce FX 5900 model.) A discussion of this capability and its use with shadow volumes was presented at the Game Developers Conference in 2005.

- Since the depth-fail method only offers an advantage over depth-pass in the special case where the eye is within a shadow volume, it is preferable to check for this case, and use depth-pass wherever possible. This avoids both the unnecessary back-capping (and the associated rasterization) for cases where depth-fail is unnecessary, as well as the problem of appropriately front-capping for special cases of depth-pass.