COMPUTER-AIDED INDUSTRIAL DESIGN

Dick Finch

COMPUTER-AIDED INDUSTRIAL DESIGN

COMPUTER-AIDED INDUSTRIAL DESIGN

Dick Finch

BIBLIOTEX
Digital Library

Computer-aided Industrial Design
by Dick Finch

www.bibliotex.com

Ebook ISBN: 9781984663931

Published by:
Bibliotex
Canada
Website: www.bibliotex.com

Contents

1

Computer-aided Industrial Design and Electronic Design Automation

Computer-aided industrial design (CAID) is a subset of computer-aided design (CAD) that includes software that directly helps in product development. Within CAID programmes designers have the freedom of creativity, but typically follow a simple design methodology:

- Creating sketches, using a stylus
- Generating curves directly from the sketch
- Generating surfaces directly from the curves

The end result is a 3D model that projects the main design intent the designer had in mind. The model can then be saved in STL format to send it to a rapid prototyping machine to create the real-life model. CAID helps the designer to focus on the technical part of the design methodology rather than taking care of sketching and modeling—then contributing to the selection of a better product proposal

in less time. Later, when the requisites and parameters of the product have been defined by means of using CAID software, the designer can import the result of his work into a CAD programme (typically a Solid Modeler) for adjustments prior to production and generation of blueprints and manufacturing processes. What differentiates CAID from CAD is that the former is far more conceptual and less technical than the latter. Within a CAID programme, the designer can express him/herself without extents, whilst in CAD software there is always the manufacturing factor.

Electronic Design Automation

Electronic design automation (EDA or ECAD) is a category of software tools for designing electronic systems such as printed circuit boards and integrated circuits. The tools work together in a design flow that chip designers use to design and analyze entire semiconductor chips.

HISTORY

EARLY DAYS

Before EDA, integrated circuits were designed by hand, and manually laid out. Some advanced shops used geometric software to generate the tapes for the Gerber photoplotter, but even those copied digital recordings of mechanically-drawn components. The process was fundamentally graphic, with the translation from electronics to graphics done manually. The best known company from this era was Calma, whose GDSII format survives. By the mid-70s, developers started to automate the design, and not just the

drafting. The first placement and routing (Place and route) tools were developed. The proceedings of the Design Automation Conference cover much of this era. The next era began about the time of the publication of "Introduction to VLSI Systems" by Carver Mead and Lynn Conway in 1980. This ground breaking text advocated chip design with programming languages that compiled to silicon.

The immediate result was a considerable increase in the complexity of the chips that could be designed, with improved access to design verification tools that used logic simulation. Often the chips were easier to lay out and more likely to function correctly, since their designs could be simulated more thoroughly prior to construction.

Although the languages and tools have evolved, this general approach of specifying the desired behavior in a textual programming language and letting the tools derive the detailed physical design remains the basis of digital IC design today.

The earliest EDA tools were produced academically. One of the most famous was the "Berkeley VLSI Tools Tarball", a set of UNIX utilities used to design early VLSI systems. Still widely used is the Espresso heuristic logic minimizer and Magic.

Another crucial development was the formation of MOSIS, a consortium of universities and fabricators that developed an inexpensive way to train student chip designers by producing real integrated circuits. The basic concept was to use reliable, low-cost, relatively low-technology IC processes, and pack a large number of projects per wafer, with just a few copies of each projects' chips. Cooperating

fabricators either donated the processed wafers, or sold them at cost, seeing the programme as helpful to their own long-term growth.

BIRTH OF COMMERCIAL EDA

1981 marks the beginning of EDA as an industry. For many years, the larger electronic companies, such as Hewlett Packard, Tektronix, and Intel, had pursued EDA internally. In 1981, managers and developers spun out of these companies to concentrate on EDA as a business. Daisy Systems, Mentor Graphics, and Valid Logic Systems were all founded around this time, and collectively referred to as DMV. Within a few years there were many companies specializing in EDA, each with a slightly different emphasis. The first trade show for EDA was held at the Design Automation Conference in 1984. In 1986, Verilog, a popular high-level design language, was first introduced as a hardware description language by Gateway Design Automation. In 1987, the U.S. Department of Defense funded creation of VHDL as a specification language. Simulators quickly followed these introductions, permitting direct simulation of chip designs: executable specifications. In a few more years, back-ends were developed to perform logic synthesis.

CURRENT STATUS

Current digital flows are extremely modular (see Integrated circuit design, Design closure, and Design flow (EDA)). The front ends produce standardized design descriptions that compile into invocations of “cells,”, without regard to the

cell technology. Cells implement logic or other electronic functions using a particular integrated circuit technology. Fabricators generally provide libraries of components for their production processes, with simulation models that fit standard simulation tools. Analog EDA tools are far less modular, since many more functions are required, they interact more strongly, and the components are (in general) less ideal. EDA for electronics has rapidly increased in importance with the continuous scaling of semiconductor technology. Some users are foundry operators, who operate the semiconductor fabrication facilities, or "fabs", and design-service companies who use EDA software to evaluate an incoming design for manufacturing readiness. EDA tools are also used for programming design functionality into FPGAs.

SOFTWARE FOCUSES

DESIGN

- High-level synthesis(syn. behavioural synthesis, algorithmic synthesis) For digital chips
- Logic synthesis translation of abstract, logical language such as Verilog or VHDL into a discrete netlist of logic-gates
- Schematic Capture For standard cell digital, analog, rf like Capture CIS in Orcad by CADENCE and ISIS in Proteus
- Layout like Layout in Orcad by Cadence, ARES in Proteus

DESIGN FLOWS

Design flows are the explicit combination of electronic design automation tools to accomplish the design of an integrated circuit. Moore's law has driven the entire IC implementation RTL to GDSII design flows from one which uses primarily standalone synthesis, placement, and routing algorithms to an integrated construction and analysis flows for design closure. The challenges of rising interconnect delay led to a new way of thinking about and integrating design closure tools. New scaling challenges such as leakage power, variability, and reliability will keep on challenging the current state of the art in design closure. The RTL to GDSII flow underwent significant changes from 1980 through 2005. The continued scaling of CMOS technologies significantly changed the objectives of the various design steps.

The lack of good predictors for delay has led to significant changes in recent design flows. Challenges like leakage power, variability, and reliability will continue to require significant changes to the design closure process in the future. Many factors describe what drove the design flow from a set of separate design steps to a fully integrated approach, and what further changes are coming to address the latest challenges. In his keynote at the 40th Design Automation Conference entitled The Tides of EDA, Alberto Sangiovanni-Vincentelli distinguished three periods of EDA: The Age of the Gods, The Age of the Heroes, and The Age of the Men. These eras were characterized respectively by senses, imagination, and reason. When we limit ourselves to the RTL to GDSII flow of the CAD area, we can distinguish

three main eras in its development: the Age of Invention, the Age of Implementation, and the Age of Integration.

- The Age of Invention: During the invention era, routing, placement, static timing analysis and logic synthesis were invented.
- The Age of Implementation: In the age of implementation, these steps were drastically improved by designing sophisticated data structures and advanced algorithms. This allowed the tools in each of these design steps to keep pace with the rapidly increasing design sizes. However, due to the lack of good predictive cost functions, it became impossible to execute a design flow by a set of discrete steps, no matter how efficiently each of the steps was implemented.
- The Age of Integration: This led to the age of integration where most of the design steps are performed in an integrated environment, driven by a set of incremental cost analyzers.

SIMULATION

- Transistor simulation – low-level transistor-simulation of a schematic/layout's behavior, accurate at device-level.
- Logic simulation – digital-simulation of an RTL or gate-netlist's digital (boolean 0/1) behavior, accurate at boolean-level.
- Behavioral Simulation – high-level simulation of a design's architectural operation, accurate at cycle-level or interface-level.

- Hardware emulation – Use of special purpose hardware to emulate the logic of a proposed design. Can sometimes be plugged into a system in place of a yet-to-be-built chip; this is called in-circuit emulation.
- Technology CAD simulate and analyze the underlying process technology. Electrical properties of devices are derived directly from device physics.
- Electromagnetic field solvers, or just field solvers, solve Maxwell's equations directly for cases of interest in IC and PCB design. They are known for being slower but more accurate than the layout extraction above.

ELECTRONIC CIRCUIT SIMULATION

Electronic circuit simulation uses mathematical models to replicate the behavior of an actual electronic device or circuit. Simulation software allows for modeling of circuit operation and is an invaluable analysis tool. Due to its highly accurate modeling capability, many Colleges and Universities use this type of software for the teaching of electronics technician and electronics engineering programmes. Electronics simulation software engages the user by integrating them into the learning experience. These kinds of interactions actively engage learners to analyze, synthesize, organize, and evaluate content and result in learners constructing their own knowledge. Simulating a circuit's behavior before actually building it can greatly improve design efficiency by making faulty designs known as such, and providing insight into the behavior of electronics circuit designs.

In particular, for integrated circuits, the tooling (photomasks) is expensive, breadboards are impractical, and probing the behavior of internal signals is extremely difficult. Therefore almost all IC design relies heavily on simulation. The most well known analog simulator is SPICE. Probably the best known digital simulators are those based on Verilog and VHDL. Some electronics simulators integrate a schematic editor, a simulation engine, and on-screen waveforms, and make "what-if" scenarios easy and instant. They also typically contain extensive model and device libraries. These models typically include IC specific transistor models such as BSIM, generic components such as resistors, capacitors, inductors and transformers, user defined models (such as controlled current and voltage sources, or models in Verilog-A or VHDL-AMS). Printed circuit board (PCB) design requires specific models as well, such as transmission lines for the traces and IBIS models for driving and receiving electronics.

Types

While there are strictly analog electronics circuit simulators, popular simulators often include both analog and event-driven digital simulation capabilities, and are known as mixed-mode simulators. This means that any simulation may contain components that are analog, event driven (digital or sampled-data), or a combination of both. An entire mixed signal analysis can be driven from one integrated schematic. All the digital models in mixed-mode simulators provide accurate specification of propagation time and rise/fall time delays.

The event driven algorithm provided by mixed-mode simulators is general purpose and supports non-digital types of data. For example, elements can use real or integer values to simulate DSP functions or sampled data filters. Because the event driven algorithm is faster than the standard SPICE matrix solution, simulation time is greatly reduced for circuits that use event driven models in place of analog models.

Mixed-mode simulation is handled on three levels; (a) with primitive digital elements that use timing models and the built-in 12 or 16 state digital logic simulator, (b) with subcircuit models that use the actual transistor topology of the integrated circuit, and finally, (c) with In-line Boolean logic expressions.

Exact representations are used mainly in the analysis of transmission line and signal integrity problems where a close inspection of an IC's I/O characteristics is needed. Boolean logic expressions are delay-less functions that are used to provide efficient logic signal processing in an analog environment. These two modeling techniques use SPICE to solve a problem while the third method, digital primitives, use mixed mode capability. Each of these methods has its merits and target applications. In fact, many simulations (particularly those which use A/D technology) call for the combination of all three approaches. No one approach alone is sufficient.

Another type of simulation used mainly for power electronics represent piecewise linear algorithms. These algorithms use an analog (linear) simulation until a power

electronic switch changes its state. At this time a new analog model is calculated to be used for the next simulation period. This methodology both enhances simulation speed and stability significantly.

Complexities

Often circuit simulators do not take into account the process variations that occur when the design is fabricated into silicon. These variations can be small, but taken together can change the output of a chip significantly. Process variations occur in the manufacture of circuits in silicon. Temperature variation can also be modeled to simulate the circuit's performance through temperature ranges.

ANALYSIS AND VERIFICATION

- Functional verification
- Clock Domain Crossing Verification (CDC check): Similar to linting, but these checks/tools specialize in detecting and reporting potential issues like data loss, meta-stability due to use of multiple clock domains in the design.
- Formal verification, also model checking: Attempts to prove, by mathematical methods, that the system has certain desired properties, and that certain undesired effects (such as deadlock) cannot occur.
- Equivalence checking: algorithmic comparison between a chip's RTL-description and synthesized gate-netlist, to ensure functional equivalence at the *logical* level.

- Static timing analysis: Analysis of the timing of a circuit in an input-independent manner, hence finding a worst case over all possible inputs.
- Physical verification, PV: checking if a design is physically manufacturable, and that the resulting chips will not have any function-preventing physical defects, and will meet original specifications.

MANUFACTURING PREPARATION

- Mask data preparation, MDP: generation of actual lithography photomask used to physically manufacture the chip.
  - Resolution enhancement techniques, RET – methods of increasing of quality of final photomask.
  - Optical proximity correction, OPC – up-front compensation for diffraction and interference effects occurring later when chip is manufactured using this mask.
  - Mask generation – generation of flat mask image from hierarchical design.
  - Automatic test pattern generation, ATPG – generates pattern-data to systematically exercise as many logic-gates, and other components, as possible.
  - Built-in self-test, or BIST – installs self-contained test-controllers to automatically test a logic (or memory) structure in the design

COMPANIES

For more details on this topic, see List of EDA companies.

TOP COMPANIES

- $3.73 billion - Synopsys
- $2.06 billion - Cadence
- $1.18 billion - Mentor Graphics
- $233 million - Magma Design Automation
- $157 million - Zuken Inc.

Note: Market caps current as of October, 2010. EEsof should likely be on this list, but does not have a market cap as it is the EDA division of Agilent.

ACQUISITIONS

Many of the EDA companies acquire small companies with software or other technology that can be adapted to their core business. Most of the market leaders are rather incestuous amalgamations of many smaller companies. This trend is helped by the tendency of software companies to design tools as accessories that fit naturally into a larger vendor's suite of programmes (on digital circuitry, many new tools incorporate analog design, and mixed systems. This is happening because there is now a trend to place entire electronic systems on a single chip.

Computer Graphics

The development of computer graphics has made computers easier to interact with, and better for understanding and interpreting many types of data. Developments in computer graphics have had a profound impact on many types of media and have revolutionized animation, movies and the video game industry. The term computer graphics has been used in a broad sense to

describe "almost everything on computers that is not text or sound". Typically, the term *computer graphics* refers to several different things:

- the representation and manipulation of image data by a computer
- the various technologies used to create and manipulate images
- the images so produced, and
- the sub-field of computer science which studies methods for digitally synthesizing and manipulating visual content, see study of computer graphics

Today, computers and computer-generated images touch many aspects of daily life. Computer imagery is found on television, in newspapers, for example in weather reports, or for example in all kinds of medical investigation and surgical procedures. A well-constructed graph can present complex statistics in a form that is easier to understand and interpret. In the media "such graphs are used to illustrate papers, reports, thesis", and other presentation material. Many powerful tools have been developed to visualize data. Computer generated imagery can be categorized into several different types: 2D, 3D, 4D, 7D, and animated graphics. As technology has improved, 3D computer graphics have become more common, but 2D computer graphics are still widely used.

Computer graphics has emerged as a sub-field of computer science which studies methods for digitally synthesizing and manipulating visual content. Over the past decade, other specialized fields have been developed

like information visualization, and scientific visualization more concerned with "the visualization of three dimensional phenomena (architectural, meteorological, medical, biological, etc.), where the emphasis is on realistic renderings of volumes, surfaces, illumination sources, and so forth, perhaps with a dynamic (time) component". The advance in computer graphics was to come from Ivan Sutherland. In 1961 Sutherland created another computer drawing programme called Sketchpad. Using a light pen, Sketchpad allowed one to draw simple shapes on the computer screen, save them and even recall them later. The light pen itself had a small photoelectric cell in its tip. This cell emitted an electronic pulse whenever it was placed in front of a computer screen and the screen's electron gun fired directly at it. By simply timing the electronic pulse with the current location of the electron gun, it was easy to pinpoint exactly where the pen was on the screen at any given moment. Once that was determined, the computer could then draw a cursor at that location. Sutherland seemed to find the perfect solution for many of the graphics problems he faced.

Even today, many standards of computer graphics interfaces got their start with this early Sketchpad programme. One example of this is in drawing constraints. If one wants to draw a square for example, s/he doesn't have to worry about drawing four lines perfectly to form the edges of the box. One can simply specify that s/he wants to draw a box, and then specify the location and size of the box. The software will then construct a perfect box, with the right dimensions and at the right location. Another example is that Sutherland's software modeled objects - not

just a picture of objects. In other words, with a model of a car, one could change the size of the tires without affecting the rest of the car. It could stretch the body of the car without deforming the tires. These early computer graphics were Vector graphics, composed of thin lines whereas modern day graphics are Raster based using pixels. The difference between vector graphics and raster graphics can be illustrated with a shipwrecked sailor.

He creates an SOS sign in the sand by arranging rocks in the shape of the letters "SOS." He also has some brightly colored rope, with which he makes a second "SOS" sign by arranging the rope in the shapes of the letters. The rock SOS sign is similar to raster graphics. Every pixel has to be individually accounted for. The rope SOS sign is equivalent to vector graphics. The computer simply sets the starting point and ending point for the line and perhaps bend it a little between the two end points. The disadvantages to vector files are that they cannot represent continuous tone images and they are limited in the number of colors available. Raster formats on the other hand work well for continuous tone images and can reproduce as many colors as needed. Also in 1961 another student at MIT, Steve Russell, created the first video game, Spacewar. Written for the DEC PDP-1, Spacewar was an instant success and copies started flowing to other PDP-1 owners and eventually even DEC got a copy. The engineers at DEC used it as a diagnostic programme on every new PDP-1 before shipping it. The sales force picked up on this quickly enough and when installing new units, would run the world's first video game for their new customers.

E. E. Zajac, a scientist at Bell Telephone Laboratory (BTL), created a film called "Simulation of a two-giro gravity attitude control system" in 1963. In this computer generated film, Zajac showed how the attitude of a satellite could be altered as it orbits the Earth. He created the animation on an IBM 7090 mainframe computer. Also at BTL, Ken Knowlton, Frank Sindon and Michael Noll started working in the computer graphics field. Sindon created a film called Force, Mass and Motion illustrating Newton's laws of motion in operation.

Around the same time, other scientists were creating computer graphics to illustrate their research. At Lawrence Radiation Laboratory, Nelson Max created the films, "Flow of a Viscous Fluid" and "Propagation of Shock Waves in a Solid Form." Boeing Aircraft created a film called "Vibration of an Aircraft." It wasn't long before major corporations started taking an interest in computer graphics. TRW, Lockheed-Georgia, General Electric and Sperry Rand are among the many companies that were getting started in computer graphics by the mid 1960's. IBM was quick to respond to this interest by releasing the IBM 2250 graphics terminal, the first commercially available graphics computer. Ralph Baer, a supervising engineer at Sanders Associates, came up with a home video game in 1966 that was later licensed to Magnavox and called the Odyssey. While very simplistic, and requiring fairly inexpensive electronic parts, it allowed the player to move points of light around on a screen. It was the first consumer computer graphics product.

Also in 1966, Sutherland at MIT invented the first computer controlled head-mounted display (HMD). Called

the Sword of Damocles because of the hardware required for support, it displayed two separate wireframe images, one for each eye. This allowed the viewer to see the computer scene in stereoscopic 3D. After receiving his Ph.D. from MIT, Sutherland became Director of Information Processing at ARPA (Advanced Research Projects Agency), and later became a professor at Harvard. Dave Evans was director of engineering at Bendix Corporation's computer division from 1953 to 1962, after which he worked for the next five years as a visiting professor at Berkeley. There he continued his interest in computers and how they interfaced with people. In 1968 the University of Utah recruited Evans to form a computer science programme, and computer graphics quickly became his primary interest. This new department would become the world's primary research center for computer graphics. In 1967 Sutherland was recruited by Evans to join the computer science programme at the University of Utah. There he perfected his HMD. Twenty years later, NASA would re-discover his techniques in their virtual reality research.

At Utah, Sutherland and Evans were highly sought after consultants by large companies but they were frustrated at the lack of graphics hardware available at the time so they started formulating a plan to start their own company. A student by the name of Edwin Catmull started at the University of Utah in 1970 and signed up for Sutherland's computer graphics class. Catmull had just come from The Boeing Company and had been working on his degree in physics. Growing up on Disney, Catmull loved animation yet quickly discovered that he didn't have the talent for drawing. Now

Catmull (along with many others) saw computers as the natural progression of animation and they wanted to be part of the revolution. The first animation that Catmull saw was his own. He created an animation of his hand opening and closing. It became one of his goals to produce a feature length motion picture using computer graphics. In the same class, Fred Parke created an animation of his wife's face.

Because of Evan's and Sutherland's presence, UU was gaining quite a reputation as the place to be for computer graphics research so Catmull went there to learn 3D animation. As the UU computer graphics laboratory was attracting people from all over, John Warnock was one of those early pioneers; he would later found Adobe Systems and create a revolution in the publishing world with his PostScript page description language. Tom Stockham led the image processing group at UU which worked closely with the computer graphics lab. Jim Clark was also there; he would later found Silicon Graphics, Inc. The first major advance in 3D computer graphics was created at UU by these early pioneers, the hidden-surface algorithm. In order to draw a representation of a 3D object on the screen, the computer must determine which surfaces are "behind" the object from the viewer's perspective, and thus should be "hidden" when the computer creates (or renders) the image.

IMAGE TYPES

2D COMPUTER GRAPHICS

2D computer graphics are the computer-based generation of digital images—mostly from two-dimensional models, such

as 2D geometric models, text, and digital images, and by techniques specific to them. 2D computer graphics are mainly used in applications that were originally developed upon traditional printing and drawing technologies, such as typography, cartography, technical drawing, advertising, etc.. In those applications, the two-dimensional image is not just a representation of a real-world object, but an independent artifact with added semantic value; two-dimensional models are therefore preferred, because they give more direct control of the image than 3D computer graphics, whose approach is more akin to photography than to typography.

Pixel Art

Pixel art is a form of digital art, created through the use of raster graphics software, where images are edited on the pixel level. Graphics in most old (or relatively limited) computer and video games, graphing calculator games, and many mobile phone games are mostly pixel art.

Vector 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 Vector graphics consists in encoding information about shapes and colors that comprise the image, which can allow for more flexibility in rendering. There are instances when working with vector tools and formats is best practice, and instances when working with raster tools and formats is 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.

3D COMPUTER GRAPHICS

3D computer graphics in contrast to 2D computer graphics are graphics that use a three-dimensional representation of geometric data that is stored in the computer for the purposes of performing calculations and rendering 2D images. Such images may be for later display or for real-time viewing. Despite these differences, 3D computer graphics rely on many of the same algorithms as 2D computer vector graphics in the wire frame model and 2D computer raster graphics in the final rendered display. In computer graphics software, the distinction between 2D and 3D is occasionally blurred; 2D applications may use 3D techniques to achieve effects such as lighting, and primarily 3D may use 2D rendering techniques. 3D computer graphics are often referred to as 3D models. Apart from the rendered graphic, the model is contained within the graphical data file. However, there are differences. A 3D model is the mathematical representation of any three-dimensional object. A model is not technically a graphic until it is visually displayed. Due to 3D printing, 3D models are not confined to virtual space. A model can be displayed visually as a two-dimensional image through a process called *3D rendering*, or used in non-graphical computer simulations and calculations. There are some 3D computer graphics software for users to create 3D images.

COMPUTER ANIMATION

Computer animation is the art of creating moving images via the use of computers. It is a subfield of computer graphics and animation. Increasingly it is created by means of 3D computer graphics, though 2D computer graphics are still widely used for stylistic, low bandwidth, and faster real-time rendering needs. Sometimes the target of the animation is the computer itself, but sometimes the target is another medium, such as film. It is also referred to as CGI (Computer-generated imagery or computer-generated imaging), especially when used in films. Virtual entities may contain and be controlled by assorted attributes, such as transform values (location, orientation, and scale) stored in an object's transformation matrix. Animation is the change of an attribute over time. Multiple methods of achieving animation exist; the rudimentary form is based on the creation and editing of keyframes, each storing a value at a given time, per attribute to be animated. The 2D/3D graphics software will interpolate between keyframes, creating an editable curve of a value mapped over time, resulting in animation.

Other methods of animation include procedural and expression-based techniques: the former consolidates related elements of animated entities into sets of attributes, useful for creating particle effects and crowd simulations; the latter allows an evaluated result returned from a user-defined logical expression, coupled with mathematics, to automate animation in a predictable way (convenient for controlling bone behavior beyond what a hierarchy offers in skeletal system set up). To create the illusion of movement, an image is displayed on the computer screen then quickly

replaced by a new image that is similar to the previous image, but shifted slightly. This technique is identical to the illusion of movement in television and motion pictures.

CONCEPTS AND PRINCIPLES

Images are typically produced by optical devices;such as cameras, mirrors, lenses, telescopes, microscopes, etc. and natural objects and phenomena, such as the human eye or water surfaces. A digital image is a representation of a two-dimensional image in binary format as a sequence of ones and zeros. Digital images include both vector images and raster images, but raster images are more commonly used.

PIXEL

In digital imaging, a pixel (or picture element) is a single point in a raster image. Pixels are normally arranged in a regular 2-dimensional grid, and are often represented using dots or squares. Each pixel is a sample of an original image, where more samples typically provide a more accurate representation of the original. The intensity of each pixel is variable; in color systems, each pixel has typically three components such as red, green, and blue.

GRAPHICS

Graphics are visual presentations on some surface, such as a wall, canvas, computer screen, paper, or stone to brand, inform, illustrate, or entertain. Examples are photographs, drawings, line art, graphs, diagrams, typography, numbers, symbols, geometric designs, maps,

engineering drawings, or other images. Graphics often combine text, illustration, and color. Graphic design may consist of the deliberate selection, creation, or arrangement of typography alone, as in a brochure, flier, poster, web site, or book without any other element. Clarity or effective communication may be the objective, association with other cultural elements may be sought, or merely, the creation of a distinctive style.

RENDERING

Rendering is the process of generating an image from a model (or models in what collectively could be called a *scene* file), by means of computer programmes. A scene file contains objects in a strictly defined language or data structure; it would contain geometry, viewpoint, texture, lighting, and shading information as a description of the virtual scene. The data contained in the scene file is then passed to a rendering programme to be processed and output to a digital image or raster graphics image file. The rendering programme is usually built into the computer graphics software, though others are available as plug-ins or entirely separate programmes. The term "rendering" may be by analogy with an "artist's rendering" of a scene. Though the technical details of rendering methods vary, the general challenges to overcome in producing a 2D image from a 3D representation stored in a scene file are outlined as the graphics pipeline along a rendering device, such as a GPU. A GPU is a purpose-built device able to assist a CPU in performing complex rendering calculations. If a scene is to look relatively realistic and predictable under virtual lighting,

the rendering software should solve the rendering equation. The rendering equation doesn't account for all lighting phenomena, but is a general lighting model for computer-generated imagery. 'Rendering' is also used to describe the process of calculating effects in a video editing file to produce final video output.

3D Projection

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

Ray Tracing

Ray tracing is a technique for generating an image by tracing the path of light through pixels in an image plane. The technique is capable of producing a very high degree of photorealism; usually higher than that of typical scanline rendering methods, but at a greater computational cost.

Shading

Shading refers to depicting depth in 3D models or illustrations by varying levels of darkness. It 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. The term has been recently generalized to mean that shaders are applied.

Texture Mapping

Texture mapping is a method for adding detail, surface texture, or colour to a computer-generated graphic or 3D model. Its application to 3D graphics was pioneered by Dr Edwin Catmull in 1974. A texture map is applied (mapped) to the surface of a shape, or polygon. This process is akin to applying patterned paper to a plain white box. Multitexturing is the use of more than one texture at a time on a polygon. Procedural textures (created from adjusting parameters of an underlying algorithm that produces an output texture), and bitmap textures (created in an image editing application) are, generally speaking, common methods of implementing texture definition from a 3D animation programme, while intended placement of textures onto a model's surface often requires a technique known as UV mapping.

Anti-aliasing

Rendering resolution-independent entities (such as 3D models) for viewing on a raster (pixel-based) device such as a LCD display or CRT television inevitably causes aliasing artifacts mostly along geometric edges and the boundaries of texture details; these artifacts are informally called "jaggies". Anti-aliasing methods rectify such problems,

resulting in imagery more pleasing to the viewer, but can be somewhat computationally expensive. Various anti-aliasing algorithms (such as supersampling) are able to be employed, then customized for the most efficient rendering performance versus quality of the resultant imagery; a graphics artist should consider this trade-off if anti-aliasing methods are to be used. A pre-anti-aliased bitmap texture being displayed on a screen (or screen location) at a resolution different than the resolution of the texture itself (such as a textured model in the distance from the virtual camera) will exhibit aliasing artifacts, while any procedurally-defined texture will always show aliasing artifacts as they are resolution-independent; techniques such as mipmapping and texture filtering help to solve texture-related aliasing problems.

VOLUME RENDERING

Volume rendering is a technique used to display a 2D projection of a 3D discretely sampled data set. A typical 3D data set is a group of 2D slice images acquired by a CT or MRI scanner. Usually these are acquired in a regular pattern (e.g., one slice every millimeter) and usually have a regular number of image pixels in a regular pattern. This is an example of a regular volumetric grid, with each volume element, or voxel represented by a single value that is obtained by sampling the immediate area surrounding the voxel.

3D MODELING

3D modeling is the process of developing a mathematical, wireframe representation of any three-dimensional object,

called a "3D model", via specialized software. Models may be created automatically or manually; the manual modeling process of preparing geometric data for 3D computer graphics is similar to plastic arts such as sculpting. 3D models may be created using multiple approaches: use of NURBS curves to generate accurate and smooth surface patches, polygonal mesh modeling (manipulation of faceted geometry), or polygonal mesh subdivision (advanced tessellation of polygons, resulting in smooth surfaces similar to NURBS models). A 3D model can be displayed as a two-dimensional image through a process called *3D rendering*, used in a computer simulation of physical phenomena, or animated directly for other purposes. The model can also be physically created using 3D Printing devices.

PIONEERS IN GRAPHIC DESIGN

CHARLES CSURI

Charles Csuri is a pioneer in computer animation and digital fine art and created the first computer art in 1964. Csuri was recognized by *Smithsonian* as the father of digital art and computer animation, and as a pioneer of computer animation by the Museum of Modern Art (MoMA) and Association for Computing Machinery-SIGGRAPH.

DONALD P. GREENBERG

Donald P. Greenberg is a leading innovator in computer graphics. Greenberg has authored hundreds of articles and served as a teacher and mentor to many prominent computer graphic artists, animators, and researchers such as Robert

L. Cook, Marc Levoy, and Wayne Lytle. Many of his former students have won Academy Awards for technical achievements and several have won the SIGGRAPH Achievement Award. Greenberg was the founding director of the NSF Center for Computer Graphics and Scientific Visualization.

A. MICHAEL NOLL

Noll was one of the first researchers to use a digital computer to create artistic patterns and to formalize the use of random processes in the creation of visual arts. He began creating digital computer art in 1962, making him one of the earliest digital computer artists. In 1965, Noll along with Frieder Nake and Georg Nees were the first to publicly exhibit their computer art. During April 1965, the Howard Wise Gallery exhibited Noll's computer art along with random-dot patterns by Bela Julesz.

OTHER PIONEERS

- Jim Blinn
- Arambilet
- Benoît B. Mandelbrot
- Henri Gouraud
- Bui Tuong Phong
- Pierre Bézier
- Paul de Casteljau
- Daniel J. Sandin
- Alvy Ray Smith
- Ton Roosendaal
- Ivan Sutherland
- Steve Russell

COMPUTER GRAPHICS STUDY

The study of computer graphics is a sub-field of computer science which studies methods for digitally synthesizing and manipulating visual content. Although the term often refers to three-dimensional computer graphics, it also encompasses two-dimensional graphics and image processing. As an academic discipline, computer graphics studies the manipulation of visual and geometric information using computational techniques. It focuses on the *mathematical* and *computational* foundations of image generation and processing rather than purely aesthetic issues. Computer graphics is often differentiated from the field of visualization, although the two fields have many similarities.

2

Computer-aided Design

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

engineering drawings, the output of CAD must convey information, such as materials, processes, dimensions, and tolerances, according to application-specific conventions. CAD may be used to design curves and figures in two-dimensional (2D) space; or curves, surfaces, and solids in three-dimensional (3D) objects. CAD is an important industrial art extensively used in many applications, including automotive, shipbuilding, and aerospace industries, industrial and architectural design, prosthetics, and many more. CAD is also widely used to produce computer animation for special effects in movies, advertising and technical manuals. The modern ubiquity and power of computers means that even perfume bottles and shampoo dispensers are designed using techniques unheard of by engineers of the 1960s. Because of its enormous economic importance, CAD has been a major driving force for research in computational geometry, computer graphics (both hardware and software), and discrete differential geometry. The design of geometric models for object shapes, in particular, is occasionally called *computer-aided geometric design* (*CAGD*).

Beginning in the 1980s Computer-Aided Design programme s reduced the need of draftsmen significantly, especially in small to mid-sized companies. Their affordability and ability to run on personal computers also allowed engineers to do their own drafting work, eliminating the need for entire departments. In today's world most, if not all, students in universities do not learn drafting techniques because they are not required to do so. The days of hand drawing for final drawings are almost obsolete. Universities no longer require the use of protractors and compasses to

create drawings, instead there are several classes that focus on the use of CAD software such as Pro Engineer or IEAS-MS. Current computer-aided design software packages range from 2D vector-based drafting systems to 3D solid and surface modellers. Modern CAD packages can also frequently allow rotations in three dimensions, allowing viewing of a designed object from any desired angle, even from the inside looking out. Some CAD software is capable of dynamic mathematic modeling, in which case it may be marketed as CADD — *computer-aided design and drafting*.

CAD is used in the design of tools and machinery and in the drafting and design of all types of buildings, from small residential types (houses) to the largest commercial and industrial structures (hospitals and factories). CAD is mainly used for detailed engineering of 3D models and/or 2D drawings of physical components, but it is also used throughout the engineering process from conceptual design and layout of products, through strength and dynamic analysis of assemblies to definition of manufacturing methods of components. It can also be used to design objects. CAD has become an especially important technology within the scope of computer-aided technologies, with benefits such as lower product development costs and a greatly shortened design cycle. CAD enables designers to lay out and develop work on screen, print it out and save it for future editing, saving time on their drawings.

TYPES

There are several different types of CAD. Each of these different types of CAD systems require the operator to think

differently about how he or she will use them and he or she must design their virtual components in a different manner for each. There are many producers of the lower-end 2D systems, including a number of free and open source programmes. These provide an approach to the drawing process without all the fuss over scale and placement on the drawing sheet that accompanied hand drafting, since these can be adjusted as required during the creation of the final draft.

3D wireframe is basically an extension of 2D drafting (not often used today). Each line has to be manually inserted into the drawing. The final product has no mass properties associated with it and cannot have features directly added to it, such as holes. The operator approaches these in a similar fashion to the 2D systems, although many 3D systems allow using the wireframe model to make the final engineering drawing views. *3D "dumb" solids* are created in a way analogous to manipulations of real world objects (not often used today). Basic three-dimensional geometric forms (prisms, cylinders, spheres, and so on) have solid volumes added or subtracted from them, as if assembling or cutting real-world objects. Two-dimensional projected views can easily be generated from the models. Basic 3D solids don't usually include tools to easily allow motion of components, set limits to their motion, or identify interference between components. *3D parametric solid modeling* require the operator to use what is referred to as "design intent". The objects and features created are adjustable. Any future modifications will be simple, difficult, or nearly impossible, depending on how the original part was created. One must

think of this as being a "perfect world" representation of the component. If a feature was intended to be located from the center of the part, the operator needs to locate it from the center of the model, not, perhaps, from a more convenient edge or an arbitrary point, as he could when using "dumb" solids. Parametric solids require the operator to consider the consequences of his actions carefully.

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

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

MAIN USES

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

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

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

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

TECHNOLOGY

Originally software for Computer-Aided Design systems was developed with computer languages such as Fortran, but with the advancement of object-oriented programming methods this has radically changed. Typical modern parametric feature based modeler and freeform surface

systems are built around a number of key C modules with their own APIs. A CAD system can be seen as built up from the interaction of a graphical user interface (GUI) with NURBS geometry and/or boundary representation (B-rep) data via a geometric modeling kernel. A geometry constraint engine may also be employed to manage the associative relationships between geometry, such as wireframe geometry in a sketch or components in an assembly. Unexpected capabilities of these associative relationships have led to a new form of prototyping called digital prototyping.

In contrast to physical prototypes, which entail manufacturing time in the design. Today, CAD systems exist for all the major platforms (Windows, Linux, UNIX and Mac OS X); some packages even support multiple platforms. Right now, no special hardware is required for most CAD software. However, some CAD systems can do graphically and computationally expensive tasks, so a good graphics card, high speed (and possibly multiple) CPUs and large amounts of RAM are recommended. The human-machine interface is generally via a computer mouse but can also be via a pen and digitizing graphics tablet. Manipulation of the view of the model on the screen is also sometimes done with the use of a Spacemouse/SpaceBall. Some systems also support stereoscopic glasses for viewing the 3D model.

HISTORY

Designers have long used computers for their calculations. Initial developments were carried out in the 1960s within the aircraft and automotive industries in the area of 3D

surface construction and NC programming, most of it independent of one another and often not publicly published until much later. Some of the mathematical description work on curves was developed in the early 1940s by Robert Issac Newton from Pawtucket, Rhode Island. Robert A. Heinlein in his 1957 novel *The Door into Summer* suggested the possibility of a robotic *Drafting Dan*. However, probably the most important work on polynomial curves and sculptured surface was done by Pierre Bezier (Renault), Paul de Casteljau (Citroen), Steven Anson Coons (MIT, Ford), James Ferguson (Boeing), Carl de Boor (GM), Birkhoff (GM) and Garibedian (GM) in the 1960s and W. Gordon (GM) and R. Riesenfeld in the 1970s. It is argued that a turning point was the development of the SKETCHPAD system at MIT in 1963 by Ivan Sutherland (who later created a graphics technology company with Dr. David Evans).

The distinctive feature of SKETCHPAD was that it allowed the designer to interact with his computer graphically: the design can be fed into the computer by drawing on a CRT monitor with a light pen. Effectively, it was a prototype of graphical user interface, an indispensable feature of modern CAD. The first commercial applications of CAD were in large companies in the automotive and aerospace industries, as well as in electronics. Only large corporations could afford the computers capable of performing the calculations. Notable company projects were at GM (Dr. Patrick J.Hanratty) with DAC-1 (Design Augmented by Computer) 1964; Lockheed projects; Bell GRAPHIC 1 and at Renault (Bezier) – UNISURF 1971 car body design and tooling. One of the most influential events in the development of CAD was the

founding of MCS (Manufacturing and Consulting Services Inc.) in 1971 by Dr. P.J. Hanratty, who wrote the system ADAM (Automated Drafting And Machining) but more importantly supplied code to companies such as McDonnell Douglas (Unigraphics), Computervision (CADDS), Calma, Gerber, Autotrol and Control Data. As computers became more affordable, the application areas have gradually expanded.

The development of CAD software for personal desktop computers was the impetus for almost universal application in all areas of construction. Other key points in the 1960s and 1970s would be the foundation of CAD systems United Computing, Intergraph, IBM, Intergraph IGDS in 1974 (which led to Bentley Systems MicroStation in 1984)

CAD implementations have evolved dramatically since then. Initially, with 3D in the 1970s, it was typically limited to producing drawings similar to hand-drafted drawings. Advances in programming and computer hardware, notably solid modeling in the 1980s, have allowed more versatile applications of computers in design activities. Key products for 1981 were the solid modelling packages -Romulus (ShapeData) and Uni-Solid (Unigraphics) based on PADL-2 and the release of the surface modeler CATIA (Dassault Systemes). Autodesk was founded 1982 by John Walker, which led to the 2D system AutoCAD.

The next milestone was the release of Pro/ENGINEER in 1988, which heralded greater usage of feature-based modeling methods and parametric linking of the parameters of features. Also of importance to the development of CAD was the development of the B-rep solid modeling kernels

(engines for manipulating geometrically and topologically consistent 3D objects) Parasolid (ShapeData) and ACIS (Spatial Technology Inc.) at the end of the 1980s and beginning of the 1990s, both inspired by the work of Ian Braid. This led to the release of mid-range packages such as SolidWorks in 1995, Solid Edge (then Intergraph) in 1996 and Autodesk Inventor in 1999.

3

Computer: Drawing, Painting and Design

Drawing and painting software is available on most platforms at the University. However, there are many differences between software intended primarily for drawing and that intended for painting. Drawing software will provide the user with a set of 'entities' used to construct the drawing (an entity is a drawing element such as a line, circle, or text string). Drawing entities can range from simple lines, points and curves in 2D to their equivalents in 3D and may include 3D surfaces. Advanced versions of drawing packages used for design are referred to as Computer Aided Design (CAD) systems. Painting software tends to work on a conceptually lower layer. Whilst it may provide some entities for constructing geometric shapes (these tend to be 2D geometric shapes), a painting package will also provide control over individual pixels in the image, *i.e.* it provides direct control

over the bitmap. It is worth remembering that opening any image in a painting package causes it to become pixelated.

The following packages are available on the ISS NT Cluster Desktop:

- Paint Very basic painting programme. Can create simple pictures and edit bitmaps. Only possible to read in and save files in a BMP format.
- Picture Publisher Painting package used to edit and create pictures. Can read in and save files in a number of different formats.
- Paint Shop Pro Recommended as the main painting package on the desktop. Used to edit and create pictures. Can read in and save files in a number of different formats.
- CorelDraw Recommended as the main drawing package on the desktop. Useful for editing vector graphics.Can read in and save files in both vector and bitmap formats.
- Micrografx Designer Drawing package used for technical drawing.

The following drawing and painting software is available on the Suns:

- Island Paint Painting programme that provides tools for creating and editing images formed by monochrome and colour bitmaps. Several painting tools can be used to create geometric and freehand shapes. Scanned images and clip art can also be imported.
- Island Draw 2D drawing package.
- Island Paint General purpose CAD system in use in

engineering, and allows 3D solid modelling as well as 2D/3D draughting. An extension, AEC, for architectural and construction applications, is also available.

COMPUTER AIDED DESIGN AND DRAWING

CAD systems provide drawing entities with powerful construction, editing and database techniques. CAD data can also be output and read in by other applications software for analysing the CAD model. For example, a CAD system could be used to generate a 3D model which could then be read into a finite element analysis package. A common requirement in engineering design is to produce a drawing which is a schematic layout of components, and which accurately reflects the relative sizes and relationships of these parts. Engineering drawing and draughting is a specialist area with its own set of procedures and practices which have become de facto standards in the engineering industry. Manual methods are now being replaced by computer-assisted methods, and the software that is used to enable these drawings to be produced embodies the functions and capabilities that are required.

CAD applications are very powerful tools that can be used by a designer. The speed and ease with which a drawing can be prepared and modified using a computer have a tremendous advantage over hand-based drawing techniques. CAD-based drawings can be created very easily using the drawing primitives made available by the software (2D/3D lines, arcs, curves, 3D surfaces, text etc.). The drawing can be shared by a number of designers over a computer network who could all be specialists in particular design areas and

located at different sites. CAD also allows drawings to be rapidly edited and modified, any number of times.

Drawings can also be linked into databases that could hold material specifications, material costs etc., thereby providing a comprehensive surveillance from design through to manufacturing. In engineering applications, CAD system specifications can be passed through to numerically controlled (NC) machines to manufacture parts directly.

For creating three-dimensional objects, most CAD systems will provide 3D primitives (such as boundary representations of spheres, cubes, surfaces of revolution and surface patches). They may also provide a solid modelling facility through Constructive Solid Geometry (CSG). Using CSG, basic 3D solids (usually cubes, spheres, wedges, cones, cylinders and tori) more complex composite solids can be created using three basic operations: joining (union) solids, removing (subtraction) solids and finding the common volume (intersection) of solids. With solid modelling, mass properties of solids (*e.g.* moments of inertia, principal moments etc.) can be quickly calculated.

There is virtually no limit to the kind of drawings and models that can be prepared using a CAD system: if it can be created by hand, a CAD system will allow it to be drawn and modelled. Some of the applications where CAD is used are: architectural and interior design, almost all engineering disciplines (*e.g.* electronic, chemical, civil, mechanical, automotive and aerospace), presentation drawings, topographic maps, musical scores, technical illustration, company logos and line drawing for fine art.

Most CAD models can be enhanced for further understanding and presentation by the use of advanced rendering animation techniques (by adding material specifications, light sources and camera motion paths to the model) to produce realistic images and interactive motion through the model. AutoCad is the primary general purpose CAD system in use in engineering, and allows 3D solid modelling as well as 2D/3D draughting. An extension, AEC, for architectural and construction applications, is also available.

SCIENTIFIC VISUALISATION

Scientific Visualisation is concerned with exploring data and information graphically - as a means of gaining insight into and understanding the data. By displaying multi-dimensional data in an easily-understandable form on a 2D screen, it enables insights into 3D and higher dimensional data and data sets that were not formerly possible. The difference between scientific visualisation and presentation graphics is that the latter is primarily concerned with the communication of information and results that are already understood. In scientific visualisation we are seeking to understand the data.

The recent upsurge of interest in scientific visualisation has been brought about principally by the provision of powerful and high-level tools coupled with the availability of powerful workstations, excellent colour graphics, and access to supercomputers if required. This symbiosis provides a powerful and flexible environment for visualising all kinds and quantities of data.

This was once regarded as the exclusive domain of expert system and application programmers who could write the large programmes required, incorporate the algorithms for the graphics, get rid of the bugs in the resulting programme (a non-trivial and time-consuming task), and then process the data. Most of this now comes already available 'off the shelf' - all the users have to do is activate it and plug in their data sets.

Visualisation tools range from lower-level presentation packages, through turnkey graphics packages and libraries, to higher-level application builders. The former are used for simple and modest requirements on small to medium sized data sets and are often used on PCs. The second take larger and more complex data sets and have a variety of facilities for analysis and presentation of the data in two and three dimensions. The latter enable users to specify their requirements in terms of their application and 'build' a customised system out of pre-defined components supplied by the software. This can usually be done visually on the screen and then the data can be read in, processed and viewed. You can interact with it by changing parameters or altering values.

Presentation Packages

Many spreadsheet packages for the PC have the facilities for doing elementary 2D graphics, *i.e.* to take a table of X, Y data and show it in visual form on X, Y axes. This enables us to see the overall form of the data much more easily than looking at the table of numbers.

It also enables us to identify any kinks or unusual features and even missing or incorrect data. These facilities

are also available in PC graphics packages such as Origin - this is menu-driven and allows users to read in data and select the options required without any programming knowledge.

Turnkey Graphics Packages and Libraries

Turnkey graphics packages include the Uniras interactive modules Unigraph, Unimap and Gsharp. Unigraph is used for scientific graphing and charting in two and three dimensions. Unimap is used for mapping, contouring and surface drawing. Gsharp is used for both. All these programmes contain advanced facilities for processing data and for the selection of curve and surface requirements. No programming knowledge or experience is required; the user interacts with the modules via menus on the screen.

Application Builders

These are large systems which contain a wide variety of pre-defined functions and facilities. Building an application consists of visually selecting the iconised functions on the screen, connecting them together by 'pipes' and then activating the network to read in the data and feed it through the interconnected modules. Many state-of-the-art functions for graphics, imaging, rendering, interfacing and displaying are contained in the system. Users can extend the functions available by writing their own modules and adding them to the system.

Examples of visualisation application builders are AVS/Express and IRIS Explorer. AVS/Express is an advanced interactive visualisation environment for scientists and engineers. AVS/Express supports geometric, image and

volume datasets. Modules can be dynamically added, connected and deleted. Modules have control panels for interactive control of input parameters in the form of on-screen sliders, file browsers, dials and buttons. AVS/Express has a wide range of data input, filter, mapper and renderer modules. Examples of mappers include isosurfaces of a 3D field, 2D slices of a 3D data volume and 3D meshes from 2D elevation datasets. Multiple visualisation techniques can be selected to suit the problem being studied. User-written programmes or subroutines in FORTRAN or C can be easily converted into AVS/Express modules which can then be integrated into networks using the network editor.

IRIS Explorer provides similar visualisation and analysis functionality. With IRIS Explorer, users view data and create applications by visually connecting software modules into flow chart configurations called module maps. Modules, the building blocks of IRIS Explorer, perform specific programme functions such as data reading, data analysis, image processing, geometric and volume rendering and many other tasks.

DESKTOP MAPPING AND GIS

Graphs which are maps, or have a cartographic component, are a special case of a 2D graph which requires some special techniques. Many people who are not geographers require this form of graph. Mapping and GIS are two areas that benefit greatly from computer processing of images. It has been estimated that 85% of all the information used by private and public sector organisations contains some sort of geographic element such as street

addresses, cities, states, postcodes or even telephone numbers with area codes. Any of these geographic components can be used to help visualise and summarise the data on a map display, enabling you to see patterns and relationships in the data quickly and easily.

MapInfo Professional is a comprehensive desktop mapping tool, available on the PC network, that enables you to create maps, create thematic maps, integrate tabular data onto maps, as well as perform complex geographic analysis such as redistricting and buffering, linking to your remote data, dragging and dropping map objects into your applications, and much more. A GIS (Geographical Information System) is a system for sorting, manipulating, analysing and displaying information with a significant spatial (map-related) content. ArcView and ArcInfo are the two packages available in this category. ArcView is a leading software package for GIS and mapping. It gives you the power to visualise, explore, query and analyse data geographically. ArcView also has three add-on packages - Spatial, Network and 3D Analyst - for more complicated queries. ArcView is available on the NT Cluster Desktop and on the Sun workstations.

ArcInfo is an advanced GIS that gives users of geographic data one of the best geoprocessing systems available at present. It integrates the modern principles of software engineering, database management and cartographic theory. Users are advised that this is a very comprehensive GIS package and requires familiarity with and understanding of GIS concepts. ArcInfo is available on the Sun workstations.

SUBROUTINE LIBRARIES FOR GRAPHICS

Uniras and OpenGL are subroutine libraries which are available at Leeds. The former is available on both the Sun and the Silicon Graphics workstations whilst the latter is only available on the Silicon Graphics workstations. Both libraries have at least FORTRAN and C bindings. This means that users have to embed their graphics requirements into their own application programmes and write their own programme code to do this. In contrast, the interactive modules of Uniras (*e.g.* Gsharp or Unigraph) work entirely off data sets - you do not need to write a programme. If you have a pre-existing applications programme for which you require graphical output, it may be easier just to produce a data file from the execution of this programme and then read this data file into a software package. It only becomes necessary to write your own programme (or extend your existing programme to include calls to graphics library routines) if you have to embed your graphics requirements to make them an integral part of your application environment, or (in the case of Uniras) you need the more advanced library functions which are not available in the interactive modules.

Multimedia

There is joint provision for networked colour printing, graphics, slides and video by Information Systems Services and the Print & Copy Bureau.

On-line Services: Printers, Slide Makers and Scanners

A4 monochrome (black and white) and colour postscript printers are available on the network. Users can send

electronic picture and text information for direct output on to paper or OHP foil. Additional printing facilities are provided by Media Services where users can also discuss converting draft electronic information into pre-designed images with design staff.

Computer-Based Video Production

Data can be displayed or animated in real-time on a high-powered workstation. However, the audience is clearly limited to those who can sit at the workstation. For research seminars, conference presentations, and grant proposals it is often more useful to be able to record the real-time image sequences on video tape and present them to the audience via a video player or video projector. To ensure such presentations are effective, they have to be at a professional standard of presentation. All of us have become unconsciously accustomed to a high quality of presentation from watching programmes on television. Anything less than this immediately looks inferior and can often reflect on the content of what is being presented.

Graphic Design

Graphic design is a creative process – most often involving a client and a designer and usually completed in conjunction with producers of form (i.e., printers, programmers, signmakers, etc.) – undertaken in order to convey a specific message (or messages) to a targeted audience. The term "graphic design" can also refer to a number of artistic and professional disciplines that focus on visual communication and presentation. The field as a whole is also often referred to as *Visual Communication* or *Communication Design*.

Various methods are used to create and combine words, symbols, and images to create a visual representation of ideas and messages. A graphic designer may use typography, visual arts and page layout techniques to produce the final result. Graphic design often refers to both the process (designing) by which the communication is created and the products (designs) which are generated. Common uses of graphic design include identity (logos and branding), web sites, publications (magazines, newspapers, and books), advertisements and product packaging. For example, a product package might include a logo or other artwork, organized text and pure design elements such as shapes and color which unify the piece.

Composition is one of the most important features of graphic design, especially when using pre-existing materials or diverse elements. While Graphic Design as a discipline has a relatively recent history, with the name 'graphic design" first coined by William Addison Dwiggins in 1922, graphic design-like activities span the history of humankind: from the caves of Lascaux, to Rome's Trajan's Column to the illuminated manuscripts of the Middle Ages, to the dazzling neons of Ginza.

In both this lengthy history and in the relatively recent explosion of visual communication in the 20th and 21st centuries, there is sometimes a blurring distinction and over-lapping of advertising art, graphic design and fine art. After all, they share many of the same elements, theories, principles, practices and languages, and sometimes the same benefactor or client. In advertising art the ultimate objective is the sale of goods and services. In graphic design,

"the essence is to give order to information, form to ideas, expression and feeling to artifacts that document human experience."

ADVENT OF PRINTING

During the Tang Dynasty (618–907) between the 4th and 7th century AD, wood blocks were cut to print on textiles and later to reproduce Buddhist texts. A Buddhist scripture printed in 868 is the earliest known printed book. Beginning in the 11th century, longer scrolls and books were produced using movable type printing making books widely available during the Song dynasty (960–1279). Sometime around 1450, Johann Gutenberg's printing press made books widely available in Europe. The book design of Aldus Manutius developed the book structure which would become the foundation of western publication design. This era of graphic design is called Humanist or Old Style.

EMERGENCE OF THE DESIGN INDUSTRY

In late 19th century Europe, especially in the United Kingdom, the movement began to separate graphic design from fine art. In 1849, Henry Cole became one of the major forces in design education in Great Britain, informing the government of the importance of design in his *Journal of Design and Manufactures*. He organized the Great Exhibition as a celebration of modern industrial technology and Victorian design. From 1891 to 1896, William Morris' Kelmscott Press published books that are some of the most significant of the graphic design products of the Arts and Crafts movement, and made a very lucrative business of

creating books of great stylistic refinement and selling them to the wealthy for a premium. Morris proved that a market existed for works of graphic design in their own right and helped pioneer the separation of design from production and from fine art. The work of the Kelmscott Press is characterized by its obsession with historical styles. This historicism was, however, important as it amounted to the first significant reaction to the stale state of nineteenth-century graphic design. Morris' work, along with the rest of the Private Press movement, directly influenced Art Nouveau and is indirectly responsible for developments in early twentieth century graphic design in general.

TWENTIETH CENTURY DESIGN

The name "Graphic Design" first appeared in print in the 1922 essay "New Kind of Printing Calls for New Design" by William Addison Dwiggins, an American book designer in the early 20th century. Raffe's *Graphic Design*, published in 1927, is considered to be the first book to use "Graphic Design" in its title. The signage in the London Underground is a classic design example of the modern era and used a font designed by Edward Johnston in 1916. In the 1920s, Soviet constructivism applied 'intellectual production' in different spheres of production. The movement saw individualistic art as useless in revolutionary Russia and thus moved towards creating objects for utilitarian purposes. They designed buildings, theater sets, posters, fabrics, clothing, furniture, logos, menus, etc.

Jan Tschichold codified the principles of modern typography in his 1928 book, *New Typography*. He later

repudiated the philosophy he espoused in this book as being fascistic, but it remained very influential. Tschichold, Bauhaus typographers such as Herbert Bayer and Laszlo Moholy-Nagy, and El Lissitzky have greatly influenced graphic design as we know it today. They pioneered production techniques and stylistic devices used throughout the twentieth century. The following years saw graphic design in the modern style gain widespread acceptance and application. A booming post-World War II American economy established a greater need for graphic design, mainly advertising and packaging. The emigration of the German Bauhaus school of design to Chicago in 1937 brought a "mass-produced" minimalism to America; sparking a wild fire of "modern" architecture and design. Notable names in mid-century modern design include Adrian Frutiger, designer of the typefaces Univers and Frutiger; Paul Rand, who, from the late 1930s until his death in 1996, took the principles of the Bauhaus and applied them to popular advertising and logo design, helping to create a uniquely American approach to European minimalism while becoming one of the principal pioneers of the subset of graphic design known as corporate identity; and Josef Müller-Brockmann, who designed posters in a severe yet accessible manner typical of the 1950s and 1970s era.

The growth of the graphic design industry has grown in parallel with the rise of consumerism. This has raised some concerns and criticisms, notably from within the graphic design community with the First Things First manifesto. First launched by Ken Garland in 1964, it was re-published as the First Things First 2000 manifesto in 1999 in the

magazine *Emigre* 51 stating "We propose a reversal of priorities in favor of more useful, lasting and democratic forms of communication - a mindshift away from product marketing and toward the exploration and production of a new kind of meaning.

The scope of debate is shrinking; it must expand. Consumerism is running uncontested; it must be challenged by other perspectives expressed, in part, through the visual languages and resources of design." Both editions attracted signatures from respected design practitioners and thinkers, for example; Rudy VanderLans, Erik Spiekermann, Ellen Lupton and Rick Poynor. The 2000 manifesto was also notably published in Adbusters, known for its strong critiques of visual culture.

APPLICATIONS

From road signs to technical schematics, from interoffice memorandums to reference manuals, graphic design enhances transfer of knowledge. Readability is enhanced by improving the visual presentation of text. Design can also aid in selling a product or idea through effective visual communication. It is applied to products and elements of company identity like logos, colors, packaging, and text. Together these are defined as branding. Branding has increasingly become important in the range of services offered by many graphic designers, alongside corporate identity. Whilst the terms are often used interchangeably, branding is more strictly related to the identifying mark or trade name for a product or service, whereas corporate

identity can have a broader meaning relating to the structure and ethos of a company, as well as to the company's external image.

Graphic designers will often form part of a team working on corporate identity and branding projects. Other members of that team can include marketing professionals, communications consultants and commercial writers. Textbooks are designed to present subjects such as geography, science, and math. These publications have layouts which illustrate theories and diagrams. A common example of graphics in use to educate is diagrams of human anatomy. Graphic design is also applied to layout and formatting of educational material to make the information more accessible and more readily understandable. Graphic design is applied in the entertainment industry in decoration, scenery, and visual story telling.

Other examples of design for entertainment purposes include novels, comic books, DVD covers, opening credits and closing credits in film, and programmes and props on stage. This could also include artwork used for t-shirts and other items screenprinted for sale. From scientific journals to news reporting, the presentation of opinion and facts is often improved with graphics and thoughtful compositions of visual information - known as information design. Newspapers, magazines, blogs, television and film documentaries may use graphic design to inform and entertain. With the advent of the web, information designers with experience in interactive tools such as Adobe Flash are increasingly being used to illustrate the background to news stories.

SKILLS

A graphic design project may involve the stylization and presentation of existing text and either preexisting imagery or images developed by the graphic designer. For example, a newspaper story begins with the journalists and photojournalists and then becomes the graphic designer's job to organize the page into a reasonable layout and determine if any other graphic elements should be required. In a magazine article or advertisement, often the graphic designer or art director will commission photographers or illustrators to create original pieces just to be incorporated into the design layout. Or the designer may utilize stock imagery or photography. Contemporary design practice has been extended to the modern computer, for example in the use of WYSIWYG user interfaces, often referred to as interactive design, or multimedia design.

VISUAL ARTS

Before any graphic elements may be applied to a design, the graphic elements must be originated by means of visual art skills. These graphics are often (but not always) developed by a graphic designer. Visual arts include works which are primarily visual in nature using anything from traditional media, to photography or computer generated art. Graphic design principles may be applied to each graphic art element individually as well as to the final composition.

TYPOGRAPHY

Typography is the art, craft and techniques of type design, modifying type glyphs, and arranging type. Type glyphs

(characters) are created and modified using a variety of illustration techniques. The arrangement of type is the selection of typefaces, point size, line length, leading (line spacing) and letter spacing. Typography is performed by typesetters, compositors, typographers, graphic artists, art directors, and clerical workers. Until the Digital Age, typography was a specialized occupation. Digitization opened up typography to new generations of visual designers and lay users.

PAGE LAYOUT

The page layout aspect of graphic design deals with the arrangement of elements (content) on a page, such as image placement, and text layout and style. Beginning from early illuminated pages in hand-copied books of the Middle Ages and proceeding down to intricate modern magazine and catalogue layouts, structured page design has long been a consideration in printed material. With print media, elements usually consist of type (text), images (pictures), and occasionally place-holder graphics for elements that are not printed with ink such as die/laser cutting, foil stamping or blind embossing.

INTERFACE DESIGN

Since the advent of the World Wide Web and computer software development, many graphic designers have become involved in interface design. This has included web design and software design, when end user interactivity is a design consideration of the layout or interface. Combining visual communication skills with the interactive communication

skills of user interaction and online branding, graphic designers often work with software developers and web developers to create both the look and feel of a web site or software application and enhance the interactive experience of the user or web site visitor. An important aspect of interface design is icon design.

PRINTMAKING

Printmaking is the process of making artworks by printing on paper and other materials or surfaces. Except in the case of monotyping, the process is capable of producing multiples of the same piece, which is called a print. Each piece is not a copy but an original since it is not a reproduction of another work of art and is technically known as an impression. Painting or drawing, on the other hand, create a unique original piece of artwork. Prints are created from a single original surface, known technically as a matrix. Common types of matrices include: plates of metal, usually copper or zinc for engraving or etching; stone, used for lithography; blocks of wood for woodcuts, linoleum for linocuts and fabric plates for screen-printing. But there are many other kinds, discussed below. Works printed from a single plate create an edition, in modern times usually each signed and numbered to form a limited edition. Prints may also be published in book form, as artist's books. A single print could be the product of one or multiple techniques.

TOOLS

The mind may be the most important graphic design tool. Aside from technology, graphic design requires judgment

and creativity. Critical, observational, quantitative and analytic thinking are required for design layouts and rendering. If the executor is merely following a solution (e.g. sketch, script or instructions) provided by another designer (such as an art director), then the executor is not usually considered the designer. The method of presentation (e.g. arrangement, style, medium) may be equally important to the design. The layout is produced using external traditional or digital image editing tools. The appropriate development and presentation tools can substantially change how an audience perceives a project. In the mid 1980s, the arrival of desktop publishing and graphic art software applications introduced a generation of designers to computer image manipulation and creation that had previously been manually executed. Computer graphic design enabled designers to instantly see the effects of layout or typographic changes, and to simulate the effects of traditional media without requiring a lot of space. However, traditional tools such as pencils or markers are useful even when computers are used for finalization; a designer or art director may hand sketch numerous concepts as part of the creative process.

Some of these sketches may even be shown to a client for early stage approval, before the designer develops the idea further using a computer and graphic design software tools. Computers are considered an indispensable tool in the graphic design industry. Computers and software applications are generally seen by creative professionals as more effective production tools than traditional methods. However, some designers continue to use manual and traditional tools for production, such as Milton Glaser. New

ideas can come by way of experimenting with tools and methods. Some designers explore ideas using pencil and paper. Others use many different mark-making tools and resources from computers to sculpture as a means of inspiring creativity. One of the key features of graphic design is that it makes a tool out of appropriate image selection in order to possibly convey meaning.ArtsComputers and the Creative Process

There is some debate whether computers enhance the creative process of graphic design. Rapid production from the computer allows many designers to explore multiple ideas quickly with more detail than what could be achieved by traditional hand-rendering or paste-up on paper, moving the designer through the creative process more quickly. However, being faced with limitless choices does not help isolate the best design solution and can lead to endless iterations with no clear design outcome. A graphic designer may use sketches to explore multiple or complex ideas quickly without the distractions and complications of software.

Hand-rendered comps are often used to get approval for an idea execution before a design invests time to produce finished visuals on a computer or in paste-up. The same thumbnail sketches or rough drafts on paper may be used to rapidly refine and produce the idea on the computer in a hybrid process. This hybrid process is especially useful in logo design where a software learning curve may detract from a creative thought process. The traditional-design/ computer-production hybrid process may be used for freeing

one's creativity in page layout or image development as well. In the early days of computer publishing, many 'traditional' graphic designers relied on computer-savvy production artists to produce their ideas from sketches, without needing to learn the computer skills themselves. However, this practice has been increasingly less common since the advent of desktop publishing over 30 years ago. The use of computers and graphics software is now taught in most graphic design courses.

OCCUPATIONS

Graphic design career paths cover all ends of the creative spectrum and often overlap. The main job responsibility of a Graphic Designer is the arrangement of visual elements in some type of media. The main job titles include graphic designer, art director, creative director, and the entry level production artist. Depending on the industry served, the responsibilities may have different titles such as "DTP Associate" or "Graphic Artist", but despite changes in title, graphic design principles remain consistent. The responsibilities may come from or lead to specialized skills such as illustration, photography or interactive design. Today's graduating graphic design students are normally exposed to all of these areas of graphic design and urged to become familiar with all of them as well in order to be competitive. Graphic designers can work in a variety of environments. Whilst many will work within companies devoted specifically to the industry, such as design consultancies or branding agencies, others may work within

publishing, marketing or other communications companies. Increasingly, especially since the introduction of personal computers to the industry, many graphic designers have found themselves working within non-design oriented organizations, as in-house designers. Graphic designers may also work as free-lance designers, working on their own terms, prices, ideas, etc.

A graphic designer reports to the art director, creative director or senior media creative. As a designer becomes more senior, they may spend less time designing media and more time leading and directing other designers on broader creative activities, such as brand development and corporate identity development. As graphic designers become more senior, they are often expected to interact more directly with clients.

SOFTWARE THAT CREATES GRAPHIC ORGANIZERS

You probably have a lot of software on programmes on the computer that you use that can create Graphic Organizers.

These include the Office Productivity Suite applications (Word Processing, Spreadsheet, and Presentation Programs). If you use Microsoft(TM) Windows, you probably have a low end drawing programme called, "Paint." All these programmes can create Graphics Organizers.

If you do not have this Office Suite, we have included an Open Source (Free) Office Suite called "Open Office." This programme is free to use and to share with others. Open

Office applications also can save your Graphic Organizer files in the PDF file format. If you save Graphic Organizer files in the PDF format, you can share them with everyone, and the file will print exactly as you created it.

OPEN OFFICE (OPEN SOURCE)

The catch with sharing Graphic Organizers that are saved in the PDF file format is that you cannot make changes to them without expensive software. However, the viewer programme that opens and prints the files is free and most people who connect to the Internet have the Acrobat Reader programme. We have included the latest version to save you from having to download it from the Internet.

SOFTWARE THAT IS A GRAPHIC ORGANIZER

There are a lot of software products on the market that are Graphic Organizers.

The majority of these products call themselves, "Mind Mapping" software.

The competition in this market is very strong, so all vendors seem to offer free trials of their products. It is possible that a teacher could use a different trial version of these products each month, and never purchase a copy.

The only catch is that the formats of the various products are proprietary. This means that you cannot open the files you create with another company's product. Inspiration(TM) and Kidspiration(TM) are products that fall into this category, and these products are often available in school districts. Inspiration and Kidspiration are easy to use, but low-end products.

GRAPHICS PIPELINE PERFORMANCE

Over the past few years, the hardware-accelerated rendering pipeline has rapidly increased in complexity, bringing with it increasingly intricate and potentially confusing performance characteristics.

Improving performance used to mean simply reducing the CPU cycles of the inner loops in your renderer; now it has become a cycle of determining bottlenecks and systematically attacking them.

This loop of *identification* and *optimization* is fundamental to tuning a heterogeneous multiprocessor system; the driving idea is that a pipeline, by definition, is only as fast as its slowest stage. Thus, while premature and unfocused optimization in a single-processor system can lead to only minimal performance gains, in a multiprocessor system such optimization very often leads to *zero* gains.

Working hard on graphics optimization and seeing zero performance improvement is no fun. The goal of this chapter is to keep you from doing exactly that.

THE PIPELINE

The pipeline, at the very highest level, can be broken into two parts: the CPU and the GPU. Although CPU optimization is a critical part of optimizing your application, it will not be the focus of this chapter, because much of this optimization has little to do with the graphics pipeline.

The GPU, there are a number of functional units operating in parallel, which essentially act as separate

special-purpose processors, and a number of spots where a bottleneck can occur. These include vertex and index fetching, vertex shading (transform and lighting, or T&L), fragment shading, and raster operations (ROP).

Methodology

Optimization without proper bottleneck identification is the cause of much wasted development effort, and so we formalize the process into the following fundamental identification and optimization loop:

1. Identify the bottleneck. For each stage in the pipeline, vary either its workload or its computational ability (that is, clock speed). If performance varies, you've found a bottleneck.
2. Optimize. Given the bottlenecked stage, reduce its workload until performance stops improving or until you achieve your desired level of performance.
3. Repeat. Do steps 1 and 2 again until the desired performance level is reached.

LOCATING THE BOTTLENECK

Locating the bottleneck is half the battle in optimization, because it enables you to make intelligent decisions about focusing your actual optimization efforts. A flow chart depicting the series of steps required to locate the precise bottleneck in your application. Note that we start at the back end of the pipeline, with the frame-buffer operations (also called raster operations) and end at the CPU. Note also that while any single primitive (usually a triangle), by definition, has a single bottleneck, over the course of a frame the

bottleneck most likely changes. Thus, modifying the workload on more than one stage in the pipeline often influences performance. For example, a low-polygon skybox is often bound by fragment shading or frame-buffer access; a skinned mesh that maps to only a few pixels on screen is often bound by CPU or vertex processing. For this reason, it frequently helps to vary workloads on an object-by-object, or material-by-material, basis.

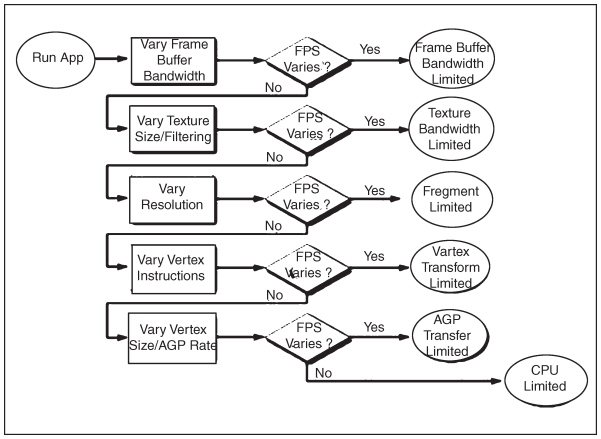

Fig. Bottleneck Flowchart

For each pipeline stage, we also mention the GPU clock to which it's tied (that is, core or memory). This information is useful in conjunction with tools such as PowerStrip (EnTech Taiwan 2003), which allows you to reduce the relevant clock speed and observe performance changes in your application.

Raster Operations

The very back end of the pipeline, raster operations (often called the ROP), is responsible for reading and writing depth and stencil, doing the depth and stencil comparisons, reading and writing colour, and doing alpha blending and testing.

As you can see, much of the ROP workload taxes the available frame-buffer bandwidth. The best way to test if your application is frame-buffer-bandwidth bound is to vary the bit depths of the colour or the depth buffers, or both. If reducing your bit depth from 32-bit to 16-bit significantly improves your performance, then you are definitely frame-buffer-bandwidth bound.

Frame-buffer bandwidth is a function of GPU memory clock, so modifying memory clocks is another technique for helping to identify this bottleneck.

Texture Bandwidth

Texture bandwidth is consumed any time a texture fetch request goes out to memory. Although modern GPUs have texture caches designed to minimize extraneous memory requests, they obviously still occur and consume a fair amount of memory bandwidth.

Modifying texture formats can be trickier than modifying frame-buffer formats as we did when inspecting the ROP; instead, we recommend changing the effective texture size by using a large amount of positive mipmap level-of-detail (LOD) bias. This makes texture fetches access very coarse levels of the mipmap pyramid, which effectively reduces the texture size. If this modification causes performance to improve significantly, you are bound by texture bandwidth.

Texture bandwidth is also a function of GPU memory clock.

Fragment Shading

Fragment shading refers to the actual cost of generating a fragment, with associated colour and depth values. This

is the cost of running the "pixel shader" or "fragment shader." Note that fragment shading and frame-buffer bandwidth are often lumped together under the heading *fill rate*, because both are a function of screen resolution. However, they are two distinct stages in the pipeline, and being able to tell the difference between the two is critical to effective optimization.

Before the advent of highly programmable fragment-processing GPUs, it was rare to be bound by fragment shading. It was often frame-buffer bandwidth that caused the inevitable correlation between screen resolution and performance. This pendulum is now starting to swing towards fragment shading, however, as the newfound flexibility enables developers to spend oodles of cycles making fancy pixels.

The first step in determining if fragment shading is the bottleneck is simply to change the resolution. Because we've already ruled out frame-buffer bandwidth by trying different frame-buffer bit depths, if adjusting resolution causes performance to change, the culprit is most likely fragment shading. A supplementary approach would be to modify the length of your fragment programmes and see if this influences performance. But be careful not to add instructions that can easily be optimized away by a clever device driver.

Fragment-shading speed is a function of the GPU core clock.

Vertex Processing

The vertex transformation stage of the rendering pipeline is responsible for taking an input set of vertex attributes

(such as model-space positions, vertex normals, texture coordinates, and so on) and producing a set of attributes suitable for clipping and rasterization (such as homogeneous clip-space position, vertex lighting results, texture coordinates, and more). Naturally, performance in this stage is a function of the work done per vertex, along with the number of vertices being processed.

With programmable transformations, determining if vertex processing is your bottleneck is a simple matter of changing the length of your vertex programme. If performance changes, you are vertex-processing bound.

If you're adding instructions, be careful to add ones that actually do meaningful work; otherwise, the instructions may be optimized away by the compiler or the driver. For example, no-ops that refer to constant registers (such as adding a constant register that has a value of zero) often cannot be optimized away because the driver usually doesn't know the value of a constant at programme-compile time.

If you're using fixed-function transformations, it's a little trickier. Try modifying the load by changing vertex work such as specular lighting or texture-coordinate generation state. Vertex processing speed is a function of the GPU core clock.

Vertex and Index Transfer

Vertices and indices are fetched by the GPU as the first step in the GPU part of the pipeline. The performance of vertex and index fetching can vary depending on where the actual vertices and indices are placed. They are usually either in system memory—which means they will be transferred to the GPU over a bus such as AGP or PCI Express—or in local frame-buffer memory. Often, on PC platforms especially,

this decision is left up to the device driver instead of the application, although modern graphics APIs allow applications to provide usage hints to help the driver choose the correct memory type.

Determining if vertex or index fetching is a bottleneck in your application entails modifying the vertex format size.

Vertex and index fetching performance is a function of the AGP/PCI Express rate if the data is placed in system memory; it's a function of the memory clock if data is placed in local frame-buffer memory.

If none of these tests influences your performance significantly, you are primarily CPU bound. You may verify this fact by underclocking your CPU: if performance varies proportionally, you are CPU bound.

OPTIMIZATION

Now that we have identified the bottleneck, we must optimize that particular stage to improve application performance. The following tips are categorized by offending stage.

Optimizing on the CPU

Many applications are CPU bound—sometimes for good reason, such as complex physics or AI, and sometimes because of poor batching or resource management. If you've found that your application is CPU bound, try the following suggestions to reduce CPU work in the rendering pipeline.

Reduce Resource Locking

Anytime you perform a synchronous operation that demands access to a GPU resource, there is the potential to

massively stall the GPU pipeline, which costs both CPU and GPU cycles. CPU cycles are wasted because the CPU must sit and spin in a loop, waiting for the (very deep) GPU pipeline to idle and return the requested resource. GPU cycles are then wasted as the pipeline sits idle and has to refill.

This locking can occur anytime you

- Lock or read from a surface you were previously rendering to
- Write to a surface the GPU is reading from, such as a texture or a vertex buffer.

In general, you should avoid accessing a resource the GPU is using during rendering.

Maximize Batch Size

We can also call this tip "Minimize the Number of Batches." A *batch* is a group of primitives rendered with a single API rendering call (for example, DrawIndexedPrimitive in DirectX 9). The *size* of a batch is the number of primitives it contains.

As a wise man once said, "Batch, Batch, Batch!". Every API function call to draw geometry has an associated CPU cost, so maximizing the number of triangles submitted with every draw call will minimize the CPU work done for a given number of triangles rendered.

Some tips to maximize the size of your batches:

- If using triangle strips, use degenerate triangles to stitch together disjoint strips. This will enable you to send multiple strips, provided that they share material, in a single draw call.
- Use texture pages. Batches are frequently broken when different objects use different textures. By

arranging many textures into a single 2D texture and setting your texture coordinates appropriately, you can send geometry that uses multiple textures in a single draw call. Note that this technique can have issues with mipmapping and antialiasing. One technique that sidesteps many of these issues is to pack individual 2D textures into each face of a cube map.

- Use GPU shader branching to increase batch size. Modern GPUs have flexible vertex- and fragment-processing pipelines that allow for branching inside the shader. For example, if two batches are separate because one requires a four-bone skinning vertex shader and the other requires a two-bone skinning vertex shader, you could instead write a vertex shader that loops over the number of bones required, accumulating blending weights, and then breaks out of the loop when the weights sum to one. This way, the two batches could be combined into one. On architectures that don't support shader branching, similar functionality can be implemented, at the cost of shader cycles, by using a four-bone vertex shader on everything and simply zeroing out the bone weights on vertices that have fewer than four bone influences.
- Use the vertex shader constant memory as a lookup table of matrices. Often batches get broken when many small objects share all material properties but differ only in matrix state (for example, a forest of similar trees, or a particle system). In these cases, you can load *n* of the differing matrices into the vertex

shader constant memory and store indices into the constant memory in the vertex format for each object. Then you would use this index to look up into the constant memory in the vertex shader and use the correct transformation matrix, thus rendering *n* objects at once.

- Defer decisions as far down in the pipeline as possible. It's faster to use the alpha channel of your texture as a gloss factor, rather than break the batch to set a pixel shader constant for glossiness. Similarly, putting shading data in your textures and vertices can allow for larger batch submissions.

Reducing the Cost of Vertex Transfer

Vertex transfer is rarely the bottleneck in an application, but it's certainly not impossible for it to happen.

If the transfer of vertices or, less likely, indices is the bottleneck in your application, try the following:

- Use the fewest possible bytes in your vertex format. Don't use floats for everything if bytes would suffice (for colours, for example).
- Generate potentially derivable vertex attributes inside the vertex programme instead of storing them inside the input vertex format. For example, there's often no need to store a tangent, binormal, and normal: given any two, the third can be derived using a simple cross product in the vertex programme. This technique trades vertex-processing speed for vertex transfer rate.
- Use 16-bit indices instead of 32-bit indices. 16-bit indices are cheaper to fetch, are cheaper to move around, and take less memory.

- Access vertex data in a relatively sequential manner. Modern GPUs cache memory accesses when fetching vertices. As in any memory hierarchy, spatial locality of reference helps maximize hits in the cache, thus reducing bandwidth requirements.

Optimizing Vertex Processing

Vertex processing is rarely the bottleneck on modern GPUs, but it may occur, depending on your usage patterns and target hardware.

Try these suggestions if you're finding that vertex processing is the bottleneck in your application:

- Optimize for the post-T&L vertex cache. Modern GPUs have a small first-in, first-out (FIFO) cache that stores the result of the most recently transformed vertices; a hit in this cache saves all transform and lighting work, along with all work done earlier in the pipeline. To take advantage of this cache, you must use indexed primitives, and you must order your vertices to maximize locality of reference over the mesh. There are tools available—including D3DX and NVTriStrip (NVIDIA 2003)—that can help you with this task.
- Reduce the number of vertices processed. This is rarely the fundamental issue, but using a simple level-of-detail scheme, such as a set of static LODs, certainly helps reduce vertex-processing load.
- Use vertex-processing LOD. Along with using LODs for the number of vertices processed, try LODing the vertex computations themselves. For example, it is likely unnecessary to do full four-bone skinning on

distant characters, and you can probably get away with cheaper approximations for the lighting. If your material is multipassed, reducing the number of passes for lower LODs in the distance will also reduce vertex-processing cost.

- Pull out per-object computations onto the CPU. Often, a calculation that changes once per object or per frame is done in the vertex shader for convenience. For example, transforming a directional light vector to eye space is sometimes done in the vertex shader, although the result of the computation changes only once per frame.
- Use the correct coordinate space. Frequently, choice of coordinate space affects the number of instructions required to compute a value in the vertex programme. For example, when doing vertex lighting, if your vertex normals are stored in object space and the light vector is stored in eye space, then you will have to transform one of the two vectors in the vertex shader. If the light vector was instead transformed into object space once per object on the CPU, no per-vertex transformation would be necessary, saving GPU vertex instructions.
- Use vertex branching to "early-out" of computations. If you are looping over a number of lights in the vertex shader and doing normal, low-dynamic-range, [0..1] lighting, you can check for saturation to 1—or if you're facing away from the light—and then break out of further computations. A similar optimization can occur with skinning, where you can break when your weights

sum to 1 (and therefore all subsequent weights would be 0). Note that this depends on how the GPU implements vertex branching, and it isn't guaranteed to improve performance on all architectures.

Speeding Up Fragment Shading

If you're using long and complex fragment shaders, it is often likely that you're fragment-shading bound. If so, try these suggestions:

- Render depth first. Rendering a depth-only (no-colour) pass before rendering your primary shading passes can dramatically boost performance, especially in scenes with high depth complexity, by reducing the amount of fragment shading and frame-buffer memory access that needs to be performed. To get the full benefits of a depth-only pass, it's not sufficient to just disable colour writes to the frame buffer; you should also disable all shading on fragments, even shading that affects depth as well as colour (such as alpha test).
- Help early-z optimizations throw away fragment processing. Modern GPUs have silicon designed to avoid shading occluded fragments, but these optimizations rely on knowledge of the scene up to the current point; they can be improved dramatically by rendering in a roughly front-to-back order. Also, laying down depth first in a separate pass can help substantially speed up subsequent passes (where all the expensive shading is done) by effectively reducing their shaded-depth complexity to 1.

- Store complex functions in textures. Textures can be enormously useful as lookup tables, and their results are filtered for free. The canonical example here is a normalization cube map, which allows you to normalize an arbitrary vector at high precision for the cost of a single texture lookup.
- Move per-fragment work to the vertex shader. Just as per-object work in the vertex shader should be moved to the CPU instead, per-vertex computations (along with computations that can be correctly linearly interpolated in screen space) should be moved to the vertex shader. Common examples include computing vectors and transforming vectors between coordinate systems.
- Use the lowest precision necessary. APIs such as DirectX 9 allow you to specify precision hints in fragment shader code for quantities or calculations that can work with reduced precision. Many GPUs can take advantage of these hints to reduce internal precision and improve performance.
- Avoid excessive normalization. A common mistake is to get "normalization-happy": normalizing every single vector every step of the way when performing a calculation. Recognize which transformations preserve length (such as transformations by an orthonourmal basis) and which computations do not depend on vector length (such as cube-map lookups).
- Consider using fragment shader level of detail. Although it offers less bang for the buck than vertex LOD (simply because objects in the distance naturally

LOD themselves with respect to pixel processing, due to perspective), reducing the complexity of the shaders in the distance, and decreasing the number of passes over a surface, can lessen the fragment-processing workload.

- Disable trilinear filtering where unnecessary. Trilinear filtering, even when not consuming extra texture bandwidth, costs extra cycles to compute in the fragment shader on most modern GPU architectures. On textures where mip-level transitions are not readily discernible, turn trilinear filtering off to save fill rate.
- Use the simplest shader type possible. In both Direct3D and OpenGL, there are a number of different ways to shade fragments. For example, in Direct3D 9, you can specify fragment shading using, in order of increasing complexity and power, texture-stage states, pixel shaders version 1.x (ps.1.1 – ps.1.4), pixel shaders version 2.x., or pixel shaders version 3.0. In general, you should use the simplest shader type that allows you to create the intended effect. The simpler shader types offer a number of implicit assumptions that often allow them to be compiled to faster native pixel-processing code by the GPU driver. A nice side effect is that these shaders would then work on a broader range of hardware.

Reducing Texture Bandwidth

If you've found that you're memory-bandwidth bound, but mostly when fetching from textures, consider these optimizations:

- Reduce the size of your textures. Consider your target resolution and texture coordinates. Do your users ever get to see your highest mip level? If not, consider scaling back the size of your textures. This can be especially helpful if overloaded frame-buffer memory has forced texturing to occur from nonlocal memory (such as system memory, over the AGP or PCI Express bus). The NVPerfHUD tool (NVIDIA 2003) can help diagnose this problem, as it shows the amount of memory allocated by the driver in various heaps.
- Compress all colour textures. All textures that are used just as decals or detail textures should be compressed, using DXT1, DXT3, orDXT5, depending on the specific texture's alpha needs. This step will reduce memory usage, reduce texture bandwidth requirements, and improve texture cache efficiency.
- Avoid expensive texture formats if not necessary. Large texture formats, such as 64-bit or 128-bit floating-point formats, obviously cost much more bandwidth to fetch from. Use these only as necessary.
- Always use mipmapping on any surface that may be minified. In addition to improving quality by reducing texture aliasing, mipmapping improves texture cache utilization by localizing texture-memory access patterns for minified textures. If you find that mipmapping on certain surfaces makes them look blurry, avoid the temptation to disable mipmapping or add a large negative LOD bias. Prefer anisotropic filtering instead and adjust the level of anisotropy per batch as appropriate.

Optimizing Frame-Buffer Bandwidth

The final stage in the pipeline, ROP, interfaces directly with the frame-buffer memory and is the single largest consumer of frame-buffer bandwidth. For this reason, if bandwidth is an issue in your application, it can often be traced to the ROP.

Here's how to optimize for frame-buffer bandwidth:

- Render depth first. This step reduces not only fragment-shading cost, but also frame-buffer bandwidth cost.
- Reduce alpha blending. Note that alpha blending, with a destination-blending factor set to anything other than 0, requires both a read and a write to the frame buffer, thus potentially consuming double the bandwidth. Reserve alpha blending for only those situations that require it, and be wary of high levels of alpha-blended depth complexity.
- Turn off depth writes when possible. Writing depth is an additional consumer of bandwidth, and it should be disabled in multipass rendering (where the final depth is already in the depth buffer); when rendering alpha-blended effects, such as particles; and when rendering objects into shadow maps (in fact, for rendering into colour-based shadow maps, you can turn off depth reads as well).
- Avoid extraneous colour-buffer clears. If every pixel is guaranteed to be overwritten in the frame buffer by your application, then avoid clearing colour, because it costs precious bandwidth. Note, however, that you should clear the depth and stencil buffers whenever you can,

because many early-z optimizations rely on the deterministic contents of a cleared depth buffer.

- Render roughly front to back. In addition to the fragment-shading advantages mention, there are similar benefits for frame-buffer bandwidth. Early-z hardware optimizations can discard extraneous frame-buffer reads and writes. In fact, even older hardware, which lacks these optimizations, will benefit from this step, because more fragments will fail the depth test, resulting in fewer colour and depth writes to the frame buffer.
- Optimize skybox rendering. Skyboxes are often frame-buffer-bandwidth bound, but you must decide how to optimize them: (1) render them last, reading (but *not* writing) depth, and allow the early-z optimizations along with regular depth buffering to save bandwidth; or (2) render the skybox first, and disable all depth reads and writes. Which option will save you more bandwidth is a function of the target hardware and how much of the skybox is visible in the final frame. If a large portion of the skybox is obscured, the first technique will likely be better; otherwise, the second one may save more bandwidth.
- Use floating-point frame buffers only when necessary. These formats obviously consume much more bandwidth than smaller, integer formats. The same applies for multiple render targets.
- Use a 16-bit depth buffer when possible. Depth transactions are a huge consumer of bandwidth, so using 16-bit instead of 32-bit can be a giant win, and

16-bit is often enough for small-scale, indoor scenes that don't require stencil. A 16-bit depth buffer is also often enough for render-to-texture effects that require depth, such as dynamic cube maps.

- Use 16-bit colour when possible. This advice is especially applicable to render-to-texture effects, because many of these, such as dynamic cube maps and projected-colour shadow maps, work just fine in 16-bit colour.

As power and programmability increase in modern GPUs, so does the complexity of extracting every bit of performance out of the machine. Whether your goal is to improve the performance of a slow application or to look for areas where you can improve image quality "for free," a deep understanding of the inner workings of the graphics pipeline is required. As the GPU pipeline continues to evolve, the fundamental ideas of optimization will still apply: first identify the bottleneck, by varying the load or the computational power of each unit; then systematically attack those bottlenecks, using your understanding of how each pipeline unit behaves.

4

Computer Programming Language

Computer programming (often shortened to programming or coding) is the process of designing, writing, testing, debugging / troubleshooting, and maintaining the source code of computer programmes. This source code is written in a programming language. The purpose of programming is to create a programme that exhibits a certain desired behaviour. The process of writing source code often requires expertise in many different subjects, including knowledge of the application domain, specialized algorithms and formal logic.

DEFINITION

Hoc and Nguyen-Xuan define computer programming as "the process of transforming a mental plan in familiar terms into one compatible with the computer." Said another way, programming is the craft of transforming requirements into something that a computer can execute.

OVERVIEW

Within software engineering, programming (the *implementation*) is regarded as one phase in a software development process. There is an ongoing debate on the extent to which the writing of programmes is an art, a craft or an engineering discipline. In general, good programming is considered to be the measured application of all three, with the goal of producing an efficient and evolvable software solution (the criteria for "efficient" and "evolvable" vary considerably). The discipline differs from many other technical professions in that programmers, in general, do not need to be licensed or pass any standardized (or governmentally regulated) certification tests in order to call themselves "programmers" or even "software engineers."

However, representing oneself as a "Professional Software Engineer" without a license from an accredited institution is illegal in many parts of the world. However, because the discipline covers many areas, which may or may not include critical applications, it is debatable whether licensing is required for the profession as a whole. In most cases, the discipline is self-governed by the entities which require the programming, and sometimes very strict environments are defined (e.g. United States Air Force use of AdaCore and security clearance). Another ongoing debate is the extent to which the programming language used in writing computer programmes affects the form that the final programme takes. This debate is analogous to that surrounding the Sapir–Whorf hypothesis in linguistics, which postulates that a particular spoken language's nature influences the habitual

thought of its speakers. Different language patterns yield different patterns of thought. This idea challenges the possibility of representing the world perfectly with language, because it acknowledges that the mechanisms of any language condition the thoughts of its speaker community.

HISTORY

The Antikythera mechanism from ancient Greece was a calculator utilizing gears of various sizes and configuration to determine its operation, which tracked the metonic cycle still used in lunar-to-solar calendars, and which is consistent for calculating the dates of the Olympiads. Al-Jazari built programmable Automata in 1206. One system employed in these devices was the use of pegs and cams placed into a wooden drum at specific locations. which would sequentially trigger levers that in turn operated percussion instruments. The output of this device was a small drummer playing various rhythms and drum patterns. The Jacquard Loom, which Joseph Marie Jacquard developed in 1801, uses a series of pasteboard cards with holes punched in them. The hole pattern represented the pattern that the loom had to follow in weaving cloth. The loom could produce entirely different weaves using different sets of cards. Charles Babbage adopted the use of punched cards around 1830 to control his Analytical Engine. The synthesis of numerical calculation, predetermined operation and output, along with a way to organize and input instructions in a manner relatively easy for humans to conceive and produce, led to the modern development of computer programming. Development of computer programming accelerated through

the Industrial Revolution. In the late 1880s, Herman Hollerith invented the recording of data on a medium that could then be read by a machine. Prior uses of machine readable media, above, had been for control, not data. "After some initial trials with paper tape, he settled on punched cards..."

To process these punched cards, first known as "Hollerith cards" he invented the tabulator, and the keypunch machines. These three inventions were the foundation of the modern information processing industry. In 1896 he founded the *Tabulating Machine Company* (which later became the core of IBM). The addition of a control panel (plugboard) to his 1906 Type I Tabulator allowed it to do different jobs without having to be physically rebuilt. By the late 1940s, there were a variety of plug-board programmable machines, called unit record equipment, to perform data-processing tasks (card reading). Early computer programmers used plug-boards for the variety of complex calculations requested of the newly invented machines. The invention of the von Neumann architecture allowed computer programmes to be stored in computer memory.

Early programmes had to be painstakingly crafted using the instructions (elementary operations) of the particular machine, often in binary notation. Every model of computer would likely use different instructions (machine language) to do the same task. Later, assembly languages were developed that let the programmer specify each instruction in a text format, entering abbreviations for each operation code instead of a number and specifying addresses in symbolic form (e.g., ADD X, TOTAL). Entering a programme in assembly language is usually more convenient, faster,

and less prone to human error than using machine language, but because an assembly language is little more than a different notation for a machine language, any two machines with different instruction sets also have different assembly languages. In 1954, FORTRAN was invented; it was the first high level programming language to have a functional implementation, as opposed to just a design on paper. (A high-level language is, in very general terms, any programming language that allows the programmer to write programmes in terms that are more abstract than assembly language instructions, i.e. at a level of abstraction "higher" than that of an assembly language.) It allowed programmers to specify calculations by entering a formula directly (e.g. Y = X*2 + 5*X + 9). The programme text, or *source*, is converted into machine instructions using a special programme called a compiler, which translates the FORTRAN programme into machine language. In fact, the name FORTRAN stands for "Formula Translation". Many other languages were developed, including some for commercial programming, such as COBOL. Programmes were mostly still entered using punched cards or paper tape. By the late 1960s, data storage devices and computer terminals became inexpensive enough that programmes could be created by typing directly into the computers. Text editors were developed that allowed changes and corrections to be made much more easily than with punched cards. (Usually, an error in punching a card meant that the card had to be discarded and a new one punched to replace it.)

As time has progressed, computers have made giant leaps in the area of processing power. This has brought

about newer programming languages that are more abstracted from the underlying hardware. Although these high-level languages usually incur greater overhead, the increase in speed of modern computers has made the use of these languages much more practical than in the past. These increasingly abstracted languages typically are easier to learn and allow the programmer to develop applications much more efficiently and with less source code.

However, high-level languages are still impractical for a few programmes, such as those where low-level hardware control is necessary or where maximum processing speed is vital. Throughout the second half of the twentieth century, programming was an attractive career in most developed countries. Some forms of programming have been increasingly subject to offshore outsourcing (importing software and services from other countries, usually at a lower wage), making programming career decisions in developed countries more complicated, while increasing economic opportunities in less developed areas. It is unclear how far this trend will continue and how deeply it will impact programmer wages and opportunities.

MODERN PROGRAMMING

QUALITY REQUIREMENTS

Whatever the approach to software development may be, the final programme must satisfy some fundamental properties. The following properties are among the most relevant:

- Efficiency/performance: the amount of system resources a programme consumes (processor time, memory space, slow devices such as disks, network bandwidth and to some extent even user interaction): the less, the better. This also includes correct disposal of some resources, such as cleaning up temporary files and lack of memory leaks.
- Reliability: how often the results of a programme are correct. This depends on conceptual correctness of algorithms, and minimization of programming mistakes, such as mistakes in resource management (e.g., buffer overflows and race conditions) and logic errors (such as division by zero or off-by-one errors).
- Robustness: how well a programme anticipates problems not due to programmer error. This includes situations such as incorrect, inappropriate or corrupt data, unavailability of needed resources such as memory, operating system services and network connections, and user error.
- Usability: the ergonomics of a program: the ease with which a person can use the programme for its intended purpose, or in some cases even unanticipated purposes. Such issues can make or break its success even regardless of other issues. This involves a wide range of textual, graphical and sometimes hardware elements that improve the clarity, intuitiveness, cohesiveness and completeness of a program's user interface.
- Portability: the range of computer hardware and operating system platforms on which the source code

of a programme can be compiled/interpreted and run. This depends on differences in the programming facilities provided by the different platforms, including hardware and operating system resources, expected behaviour of the hardware and operating system, and availability of platform specific compilers (and sometimes libraries) for the language of the source code.

- Maintainability: the ease with which a programme can be modified by its present or future developers in order to make improvements or customizations, fix bugs and security holes, or adapt it to new environments. Good practices during initial development make the difference in this regard. This quality may not be directly apparent to the end user but it can significantly affect the fate of a programme over the long term.

READABILITY OF SOURCE CODE

In computer programming, readability refers to the ease with which a human reader can comprehend the purpose, control flow, and operation of source code. It affects the aspects of quality above, including portability, usability and most importantly maintainability. Readability is important because programmers spend the majority of their time reading, trying to understand and modifying existing source code, rather than writing new source code.

Unreadable code often leads to bugs, inefficiencies, and duplicated code. A study found that a few simple readability transformations made code shorter and drastically reduced

the time to understand it. Following a consistent programming style often helps readability.

However, readability is more than just programming style. Many factors, having little or nothing to do with the ability of the computer to efficiently compile and execute the code, contribute to readability. Some of these factors include:

- Different indentation styles (whitespace)
- Comments
- Decomposition
- Naming conventions for objects (such as variables, classes, procedures, etc)

ALGORITHMIC COMPLEXITY

The academic field and the engineering practice of computer programming are both largely concerned with discovering and implementing the most efficient algorithms for a given class of problem. For this purpose, algorithms are classified into *orders* using so-called Big O notation, *O(n)*, which expresses resource use, such as execution time or memory consumption, in terms of the size of an input. Expert programmers are familiar with a variety of well-established algorithms and their respective complexities and use this knowledge to choose algorithms that are best suited to the circumstances.

METHODOLOGIES

The first step in most formal software development projects is requirements analysis, followed by testing to determine value modeling, implementation, and failure elimination (debugging). There exist a lot of differing

approaches for each of those tasks. One approach popular for requirements analysis is Use Case analysis.

Nowadays many programmers use forms of Agile software development where the various stages of formal software development are more integrated together into short cycles that take a few weeks rather than years. There are many approaches to the Software development process. Popular modeling techniques include Object-Oriented Analysis and Design (OOAD) and Model-Driven Architecture (MDA). The Unified Modeling Language (UML) is a notation used for both the OOAD and MDA. A similar technique used for database design is Entity-Relationship Modeling (ER Modeling). Implementation techniques include imperative languages (object-oriented or procedural), functional languages, and logic languages.

MEASURING LANGUAGE USAGE

It is very difficult to determine what are the most popular of modern programming languages. Some languages are very popular for particular kinds of applications (e.g., COBOL is still strong in the corporate data center, often on large mainframes, FORTRAN in engineering applications, scripting languages in web development, and C in embedded applications), while some languages are regularly used to write many different kinds of applications. Also many applications use a mix of several languages in their construction and use.

Methods of measuring programming language popularity include: counting the number of job advertisements that

mention the language, the number of books teaching the language that are sold (this overestimates the importance of newer languages), and estimates of the number of existing lines of code written in the language (this underestimates the number of users of business languages such as COBOL).

DEBUGGING

Debugging is a very important task in the software development process, because an incorrect programme can have significant consequences for its users. Some languages are more prone to some kinds of faults because their specification does not require compilers to perform as much checking as other languages. Use of a static analysis tool can help detect some possible problems.

Debugging is often done with IDEs like Eclipse, Kdevelop, NetBeans, Code::Blocks, and Visual Studio. Standalone debuggers like gdb are also used, and these often provide less of a visual environment, usually using a command line.

PROGRAMMING LANGUAGES

Different programming languages support different styles of programming (called *programming paradigms*). The choice of language used is subject to many considerations, such as company policy, suitability to task, availability of third-party packages, or individual preference. Ideally, the programming language best suited for the task at hand will be selected. Trade-offs from this ideal involve finding enough programmers who know the language to build a team, the availability of compilers for that language, and the efficiency

with which programmes written in a given language execute. Languages form an approximate spectrum from "low-level" to "high-level"; "low-level" languages are typically more machine-oriented and faster to execute, whereas "high-level" languages are more abstract and easier to use but execute less quickly. It is usually easier to code in "high-level" languages than in "low-level" ones. Allen Downey, in his book *How To Think Like A Computer Scientist*, writes:

The details look different in different languages, but a few basic instructions appear in just about every language:

- input: Get data from the keyboard, a file, or some other device.
- output: Display data on the screen or send data to a file or other device.
- arithmetic: Perform basic arithmetical operations like addition and multiplication.
- conditional execution: Check for certain conditions and execute the appropriate sequence of statements.
- repetition: Perform some action repeatedly, usually with some variation.

Many computer languages provide a mechanism to call functions provided by libraries such as in .dlls. Provided the functions in a library follow the appropriate run time conventions (e.g., method of passing arguments), then these functions may be written in any other language.

PROGRAMMERS

Computer programmers are those who write computer software. Their jobs usually involve:

- Coding
- Compilation
- Debugging
- Documentation
- Integration
- Maintenance
- Requirements analysis
- Software architecture
- Software testing
- Specification

Programming Language

A programming language is an artificial language designed to express computations that can be performed by a machine, particularly a computer. Programming languages can be used to create programmes that control the behavior of a machine, to express algorithms precisely, or as a mode of human communication. The earliest programming languages predate the invention of the computer, and were used to direct the behavior of machines such as Jacquard looms and player pianos. Thousands of different programming languages have been created, mainly in the computer field, with many more being created every year.

Most programming languages describe computation in an imperative style, i.e., as a sequence of commands, although some languages, such as those that support functional programming or logic programming, use alternative forms of description. A programming language is usually split into the two components of syntax (form) and semantics (meaning) and many programming languages

have some kind of written specification of their syntax and/or semantics. Some languages are defined by a specification document, for example, the C programming language is specified by an ISO Standard, while other languages, such as Perl, have a dominant implementation that is used as a reference.

DEFINITIONS

A programming language is a notation for writing programmes, which are specifications of a computation or algorithm. Some, but not all, authors restrict the term "programming language" to those languages that can express *all* possible algorithms. Traits often considered important for what constitutes a programming language include:

- *Function and target:* A *computer programming language* is a language used to write computer programmes, which involve a computer performing some kind of computation or algorithm and possibly control external devices such as printers, disk drives, robots, and so on. For example PostScript programmes are frequently created by another programme to control a computer printer or display. More generally, a programming language may describe computation on some, possibly abstract, machine. It is generally accepted that a complete specification for a programming language includes a description, possibly idealized, of a machine or processor for that language. In most practical contexts, a programming language involves a computer; consequently programming

languages are usually defined and studied this way. Programming languages differ from natural languages in that natural languages are only used for interaction between people, while programming languages also allow humans to communicate instructions to machines.

- *Abstractions:* Programming languages usually contain abstractions for defining and manipulating data structures or controlling the flow of execution. The practical necessity that a programming language support adequate abstractions is expressed by the abstraction principle; this principle is sometimes formulated as recommendation to the programmer to make proper use of such abstractions.
- *Expressive power:* The theory of computation classifies languages by the computations they are capable of expressing. All Turing complete languages can implement the same set of algorithms. ANSI/ISO SQL and Charity are examples of languages that are not Turing complete, yet often called programming languages.

Markup languages like XML, HTML or troff, which define structured data, are not generally considered programming languages. Programming languages may, however, share the syntax with markup languages if a computational semantics is defined. XSLT, for example, is a Turing complete XML dialect. Moreover, LaTeX, which is mostly used for structuring documents, also contains a Turing complete subset.

The term *computer language* is sometimes used interchangeably with programming language. However, the usage of both terms varies among authors, including the exact scope of each. One usage describes programming languages as a subset of computer languages. In this vein, languages used in computing that have a different goal than expressing computer programmes are generically designated computer languages. For instance, markup languages are sometimes referred to as computer languages to emphasize that they are not meant to be used for programming. Another usage regards programming languages as theoretical constructs for programming abstract machines, and computer languages as the subset thereof that runs on physical computers, which have finite hardware resources. John C. Reynolds emphasizes that formal specification languages are just as much programming languages as are the languages intended for execution. He also argues that textual and even graphical input formats that affect the behavior of a computer are programming languages, despite the fact they are commonly not Turing-complete, and remarks that ignorance of programming language concepts is the reason for many flaws in input formats.

ELEMENTS

All programming languages have some primitive building blocks for the description of data and the processes or transformations applied to them (like the addition of two numbers or the selection of an item from a collection). These primitives are defined by syntactic and semantic rules which describe their structure and meaning respectively.

SYNTAX

A programming language's surface form is known as its syntax. Most programming languages are purely textual; they use sequences of text including words, numbers, and punctuation, much like written natural languages. On the other hand, there are some programming languages which are more graphical in nature, using visual relationships between symbols to specify a programme.

The syntax of a language describes the possible combinations of symbols that form a syntactically correct programme. The meaning given to a combination of symbols is handled by semantics (either formal or hard-coded in a reference implementation). Since most languages are textual, this article discusses textual syntax.

Programming language syntax is usually defined using a combination of regular expressions (for lexical structure) and Backus–Naur Form (for grammatical structure). Below is a simple grammar, based on Lisp:

```
expression ::= atom | list
atom ::= number | symbol
number ::= [+-]?['0'-'9']+
symbol ::= ['A'-'Z''a'-'z'].*
list ::= '(' expression* ')'
```

This grammar specifies the following:

- an *expression* is either an *atom* or a *list*;
- an *atom* is either a *number* or a *symbol*;
- a *number* is an unbroken sequence of one or more decimal digits, optionally preceded by a plus or minus sign;

- a *symbol* is a letter followed by zero or more of any characters (excluding whitespace); and
- a *list* is a matched pair of parentheses, with zero or more *expressions* inside it.

The following are examples of well-formed token sequences in this grammar: '12345', '()', '(a b c232 (1))'

Not all syntactically correct programmes are semantically correct. Many syntactically correct programmes are nonetheless ill-formed, per the language's rules; and may (depending on the language specification and the soundness of the implementation) result in an error on translation or execution. In some cases, such programmes may exhibit undefined behavior. Even when a programme is well-defined within a language, it may still have a meaning that is not intended by the person who wrote it.

Using natural language as an example, it may not be possible to assign a meaning to a grammatically correct sentence or the sentence may be false:

- "Colorless green ideas sleep furiously." is grammatically well-formed but has no generally accepted meaning.
- "John is a married bachelor." is grammatically well-formed but expresses a meaning that cannot be true.

The following C language fragment is syntactically correct, but performs an operation that is not semantically defined (because p is a null pointer, the operations p->real and p->im have no meaning):

```
complex *p = NULL;
complex abs_p = sqrt (p->real * p->real + p->im * p->im);
```

If the type declaration on the first line were omitted, the programme would trigger an error on compilation, as the variable "p" would not be defined. But the programme would still be syntactically correct, since type declarations provide only semantic information.

The grammar needed to specify a programming language can be classified by its position in the Chomsky hierarchy. The syntax of most programming languages can be specified using a Type-2 grammar, i.e., they are context-free grammars. Some languages, including Perl and Lisp, contain constructs that allow execution during the parsing phase. Languages that have constructs that allow the programmer to alter the behavior of the parser make syntax analysis an undecidable problem, and generally blur the distinction between parsing and execution. In contrast to Lisp's macro system and Perl's BEGIN blocks, which may contain general computations, C macros are merely string replacements, and do not require code execution.

SEMANTICS

The term *semantics* refers to the meaning of languages, as opposed to their form (syntax).

Static Semantics

The static semantics defines restrictions on the structure of valid texts that are hard or impossible to express in standard syntactic formalisms. For compiled languages, static semantics essentially include those semantic rules that can be checked at compile time. Examples include checking that every identifier is declared before it is used

(in languages that require such declarations) or that the labels on the arms of a case statement are distinct. Many important restrictions of this type, like checking that identifiers are used in the appropriate context (e.g. not adding a integer to a function name), or that subroutine calls have the appropriate number and type of arguments can be enforced by defining them as rules in a logic called a type system. Other forms of static analyses like data flow analysis may also be part of static semantics. Newer programming languages like Java and C# have definite assignment analysis, a form of data flow analysis, as part of their static semantics.

Dynamic Semantics

Once data has been specified, the machine must be instructed to perform operations on the data. For example, the semantics may define the strategy by which expressions are evaluated to values, or the manner in which control structures conditionally execute statements. The *dynamic semantics* (also known as *execution semantics*) of a language defines how and when the various constructs of a language should produce a programme behavior. There are many ways of defining execution semantics. Natural language is often used to specify the execution semantics of languages commonly used in practice. A significant amount of academic research went into formal semantics of programming languages, which allow execution semantics to be specified in a formal manner. Results from this field of research have seen limited application to programming language design and implementation outside academia.

Type System

A type system defines how a programming language classifies values and expressions into *types*, how it can manipulate those types and how they interact. The goal of a type system is to verify and usually enforce a certain level of correctness in programmes written in that language by detecting certain incorrect operations. Any decidable type system involves a trade-off: while it rejects many incorrect programmes, it can also prohibit some correct, albeit unusual programmes. In order to bypass this downside, a number of languages have *type loopholes*, usually unchecked casts that may be used by the programmer to explicitly allow a normally disallowed operation between different types. In most typed languages, the type system is used only to type check programmes, but a number of languages, usually functional ones, infer types, relieving the programmer from the need to write type annotations. The formal design and study of type systems is known as *type theory*.

Typed versus untyped languages: A language is *typed* if the specification of every operation defines types of data to which the operation is applicable, with the implication that it is not applicable to other types. For example, the data represented by "this text between the quotes" is a string. In most programming languages, dividing a number by a string has no meaning. Most modern programming languages will therefore reject any programme attempting to perform such an operation. In some languages, the meaningless operation will be detected when the programme is compiled ("static" type checking), and rejected by the compiler, while in others, it will be detected when the programme is run

("dynamic" type checking), resulting in a runtime exception. A special case of typed languages are the *single-type* languages. These are often scripting or markup languages, such as REXX or SGML, and have only one data type—most commonly character strings which are used for both symbolic and numeric data. In contrast, an *untyped language*, such as most assembly languages, allows any operation to be performed on any data, which are generally considered to be sequences of bits of various lengths. High-level languages which are untyped include BCPL and some varieties of Forth. In practice, while few languages are considered typed from the point of view of type theory (verifying or rejecting *all* operations), most modern languages offer a degree of typing. Many production languages provide means to bypass or subvert the type system.

Static versus dynamic typing: In *static typing* all expressions have their types determined prior to the programme being run (typically at compile-time). For example, 1 and (2+2) are integer expressions; they cannot be passed to a function that expects a string, or stored in a variable that is defined to hold dates.

Statically typed languages can be either *manifestly typed* or *type-inferred*. In the first case, the programmer must explicitly write types at certain textual positions (for example, at variable declarations). In the second case, the compiler *infers* the types of expressions and declarations based on context. Most mainstream statically typed languages, such as C++, C# and Java, are manifestly typed. Complete type inference has traditionally been associated with less mainstream languages, such as Haskell and ML. However,

many manifestly typed languages support partial type inference; for example, Java and C# both infer types in certain limited cases.

Dynamic typing, also called *latent typing*, determines the type-safety of operations at runtime; in other words, types are associated with *runtime values* rather than *textual expressions*. As with type-inferred languages, dynamically typed languages do not require the programmer to write explicit type annotations on expressions. Among other things, this may permit a single variable to refer to values of different types at different points in the programme execution. However, type errors cannot be automatically detected until a piece of code is actually executed, potentially making debugging more difficult. Ruby, Lisp, JavaScript, and Python are dynamically typed.

Weak and strong typing: Weak typing allows a value of one type to be treated as another, for example treating a string as a number. This can occasionally be useful, but it can also allow some kinds of programme faults to go undetected at compile time and even at runtime.

Strong typing prevents the above. An attempt to perform an operation on the wrong type of value raises an error. Strongly typed languages are often termed *type-safe* or *safe*.

An alternative definition for "weakly typed" refers to languages, such as Perl and JavaScript, which permit a large number of implicit type conversions. In JavaScript, for example, the expression 2 * x implicitly converts x to a number, and this conversion succeeds even if x is null, undefined, an Array, or a string of letters. Such implicit

conversions are often useful, but they can mask programming errors. *Strong* and *static* are now generally considered orthogonal concepts, but usage in the literature differs. Some use the term *strongly typed* to mean *strongly, statically typed*, or, even more confusingly, to mean simply *statically typed*. Thus C has been called both strongly typed and weakly, statically typed.

Standard Library and Run-time System

Most programming languages have an associated core library (sometimes known as the 'standard library', especially if it is included as part of the published language standard), which is conventionally made available by all implementations of the language. Core libraries typically include definitions for commonly used algorithms, data structures, and mechanisms for input and output.

A language's core library is often treated as part of the language by its users, although the designers may have treated it as a separate entity. Many language specifications define a core that must be made available in all implementations, and in the case of standardized languages this core library may be required. The line between a language and its core library therefore differs from language to language. Indeed, some languages are designed so that the meanings of certain syntactic constructs cannot even be described without referring to the core library. For example, in Java, a string literal is defined as an instance of the java.lang.String class; similarly, in Smalltalk, an anonymous function expression (a "block") constructs an instance of the library's BlockContext class. Conversely, Scheme

contains multiple coherent subsets that suffice to construct the rest of the language as library macros, and so the language designers do not even bother to say which portions of the language must be implemented as language constructs, and which must be implemented as parts of a library.

DESIGN AND IMPLEMENTATION

Programming languages share properties with natural languages related to their purpose as vehicles for communication, having a syntactic form separate from its semantics, and showing *language families* of related languages branching one from another. But as artificial constructs, they also differ in fundamental ways from languages that have evolved through usage. A significant difference is that a programming language can be fully described and studied in its entirety, since it has a precise and finite definition. By contrast, natural languages have changing meanings given by their users in different communities. While constructed languages are also artificial languages designed from the ground up with a specific purpose, they lack the precise and complete semantic definition that a programming language has.

Many languages have been designed from scratch, altered to meet new needs, combined with other languages, and eventually fallen into disuse. Although there have been attempts to design one "universal" programming language that serves all purposes, all of them have failed to be generally accepted as filling this role. The need for diverse programming languages arises from the diversity of contexts in which languages are used:

- Programmes range from tiny scripts written by individual hobbyists to huge systems written by hundreds of programmers.
- Programmers range in expertise from novices who need simplicity above all else, to experts who may be comfortable with considerable complexity.
- Programmes must balance speed, size, and simplicity on systems ranging from microcontrollers to supercomputers.
- Programmes may be written once and not change for generations, or they may undergo continual modification.
- Finally, programmers may simply differ in their tastes: they may be accustomed to discussing problems and expressing them in a particular language.

One common trend in the development of programming languages has been to add more ability to solve problems using a higher level of abstraction. The earliest programming languages were tied very closely to the underlying hardware of the computer. As new programming languages have developed, features have been added that let programmers express ideas that are more remote from simple translation into underlying hardware instructions. Because programmers are less tied to the complexity of the computer, their programmes can do more computing with less effort from the programmer. This lets them write more functionality per time unit. Natural language processors have been proposed as a way to eliminate the need for a specialized language for programming. However, this goal remains distant and its benefits are open to debate. Edsger W.

Dijkstra took the position that the use of a formal language is essential to prevent the introduction of meaningless constructs, and dismissed natural language programming as "foolish".

Alan Perlis was similarly dismissive of the idea. Hybrid approaches have been taken in Structured English and SQL. A language's designers and users must construct a number of artifacts that govern and enable the practice of programming. The most important of these artifacts are the language *specification* and *implementation*.

SPECIFICATION

The specification of a programming language is intended to provide a definition that the language users and the implementors can use to determine whether the behavior of a programme is correct, given its source code. A programming language specification can take several forms, including the following:

- An explicit definition of the syntax, static semantics, and execution semantics of the language. While syntax is commonly specified using a formal grammar, semantic definitions may be written in natural language (e.g., as in the C language), or a formal semantics (e.g., as in Standard ML and Scheme specifications).
- A description of the behavior of a translator for the language (e.g., the C++ and Fortran specifications). The syntax and semantics of the language have to be inferred from this description, which may be written in natural or a formal language.

- A *reference* or *model* implementation, sometimes written in the language being specified (e.g., Prolog or ANSI REXX). The syntax and semantics of the language are explicit in the behavior of the reference implementation.

IMPLEMENTATION

An implementation of a programming language provides a way to execute that programme on one or more configurations of hardware and software. There are, broadly, two approaches to programming language implementation: *compilation* and *interpretation*. It is generally possible to implement a language using either technique. The output of a compiler may be executed by hardware or a programme called an interpreter. In some implementations that make use of the interpreter approach there is no distinct boundary between compiling and interpreting. For instance, some implementations of BASIC compile and then execute the source a line at a time. Programmes that are executed directly on the hardware usually run several orders of magnitude faster than those that are interpreted in software. One technique for improving the performance of interpreted programmes is just-in-time compilation. Here the virtual machine, just before execution, translates the blocks of bytecode which are going to be used to machine code, for direct execution on the hardware.

USAGE

Thousands of different programming languages have been created, mainly in the computing field. Programming

languages differ from most other forms of human expression in that they require a greater degree of precision and completeness. When using a natural language to communicate with other people, human authors and speakers can be ambiguous and make small errors, and still expect their intent to be understood. However, figuratively speaking, computers "do exactly what they are told to do", and cannot "understand" what code the programmer intended to write. The combination of the language definition, a programme, and the program's inputs must fully specify the external behavior that occurs when the programme is executed, within the domain of control of that programme. A programming language provides a structured mechanism for defining pieces of data, and the operations or transformations that may be carried out automatically on that data. A programmer uses the abstractions present in the language to represent the concepts involved in a computation.

These concepts are represented as a collection of the simplest elements available (called primitives). *Programming* is the process by which programmers combine these primitives to compose new programmes, or adapt existing ones to new uses or a changing environment. Programmes for a computer might be executed in a batch process without human interaction, or a user might type commands in an interactive session of an interpreter. In this case the "commands" are simply programmes, whose execution is chained together. When a language is used to give commands to a software application (such as a shell) it is called a scripting language.

MEASURING LANGUAGE USAGE

It is difficult to determine which programming languages are most widely used, and what usage means varies by context. One language may occupy the greater number of programmer hours, a different one have more lines of code, and a third utilize the most CPU time. Some languages are very popular for particular kinds of applications. For example, COBOL is still strong in the corporate data center, often on large mainframes; FORTRAN in scientific and engineering applications; C in embedded applications and operating systems; and other languages are regularly used to write many different kinds of applications.

Various methods of measuring language popularity, each subject to a different bias over what is measured, have been proposed:

- counting the number of job advertisements that mention the language
- the number of books sold that teach or describe the language
- estimates of the number of existing lines of code written in the language—which may underestimate languages not often found in public searches
- counts of language references (i.e., to the name of the language) found using a web search engine.

Combining and averaging information from various internet sites, langpop.com claims that in 2008 the 10 most cited programming languages are (in alphabetical order): C, C++, C#, Java, JavaScript, Perl, PHP, Python, Ruby, and SQL.

TAXONOMIES

There is no overarching classification scheme for programming languages. A given programming language does not usually have a single ancestor language. Languages commonly arise by combining the elements of several predecessor languages with new ideas in circulation at the time. Ideas that originate in one language will diffuse throughout a family of related languages, and then leap suddenly across familial gaps to appear in an entirely different family. The task is further complicated by the fact that languages can be classified along multiple axes.

For example, Java is both an object-oriented language (because it encourages object-oriented organization) and a concurrent language (because it contains built-in constructs for running multiple threads in parallel). Python is an object-oriented scripting language. In broad strokes, programming languages divide into *programming paradigms* and a classification by *intended domain of use*. Traditionally, programming languages have been regarded as describing computation in terms of imperative sentences, i.e. issuing commands. These are generally called imperative programming languages. A great deal of research in programming languages has been aimed at blurring the distinction between a programme as a set of instructions and a programme as an assertion about the desired answer, which is the main feature of declarative programming. More refined paradigms include procedural programming, object-oriented programming, functional programming, and logic programming; some languages are hybrids of paradigms or

multi-paradigmatic. An assembly language is not so much a paradigm as a direct model of an underlying machine architecture. By purpose, programming languages might be considered general purpose, system programming languages, scripting languages, domain-specific languages, or concurrent/distributed languages (or a combination of these). Some general purpose languages were designed largely with educational goals. A programming language may also be classified by factors unrelated to programming paradigm. For instance, most programming languages use English language keywords, while a minority do not. Other languages may be classified as being esoteric or not.

HISTORY

EARLY DEVELOPMENTS

The first programming languages predate the modern computer. The 19th century had "programmable" looms and player piano scrolls which implemented what are today recognized as examples of domain-specific languages. By the beginning of the twentieth century, punch cards encoded data and directed mechanical processing. In the 1930s and 1940s, the formalisms of Alonzo Church's lambda calculus and Alan Turing's Turing machines provided mathematical abstractions for expressing algorithms; the lambda calculus remains influential in language design. In the 1940s, the first electrically powered digital computers were created.

The first high-level programming language to be designed for a computer was Plankalkül, developed for the German Z3 by Konrad Zuse between 1943 and 1945. However, it was

not implemented until 1998 and 2000. Programmers of early 1950s computers, notably UNIVAC I and IBM 701, used machine language programmes, that is, the first generation language (1GL). 1GL programming was quickly superseded by similarly machine-specific, but mnemonic, second generation languages (2GL) known as assembly languages or "assembler".

Later in the 1950s, assembly language programming, which had evolved to include the use of macro instructions, was followed by the development of "third generation" programming languages (3GL), such as FORTRAN, LISP, and COBOL. 3GLs are more abstract and are "portable", or at least implemented similarly on computers that do not support the same native machine code. Updated versions of all of these 3GLs are still in general use, and each has strongly influenced the development of later languages. At the end of the 1950s, the language formalized as ALGOL 60 was introduced, and most later programming languages are, in many respects, descendants of Algol. The format and use of the early programming languages was heavily influenced by the constraints of the interface.

REFINEMENT

The period from the 1960s to the late 1970s brought the development of the major language paradigms now in use, though many aspects were refinements of ideas in the very first Third-generation programming languages:

- APL introduced *array programming* and influenced functional programming.

- PL/I (NPL) was designed in the early 1960s to incorporate the best ideas from FORTRAN and COBOL.
- In the 1960s, Simula was the first language designed to support *object-oriented programming*; in the mid-1970s, Smalltalk followed with the first "purely" object-oriented language.
- C was developed between 1969 and 1973 as a *system programming* language, and remains popular.
- Prolog, designed in 1972, was the first *logic programming* language.
- In 1978, ML built a polymorphic type system on top of Lisp, pioneering *statically typed functional programming* languages.

Each of these languages spawned an entire family of descendants, and most modern languages count at least one of them in their ancestry.

The 1960s and 1970s also saw considerable debate over the merits of *structured programming*, and whether programming languages should be designed to support it. Edsger Dijkstra, in a famous 1968 letter published in the Communications of the ACM, argued that GOTO statements should be eliminated from all "higher level" programming languages.

The 1960s and 1970s also saw expansion of techniques that reduced the footprint of a programme as well as improved productivity of the programmer and user. The card deck for an early 4GL was a lot smaller for the same functionality expressed in a 3GL deck.

CONSOLIDATION AND GROWTH

The 1980s were years of relative consolidation. C++ combined object-oriented and systems programming. The United States government standardized Ada, a systems programming language derived from Pascal and intended for use by defense contractors. In Japan and elsewhere, vast sums were spent investigating so-called "fifth generation" languages that incorporated logic programming constructs. The functional languages community moved to standardize ML and Lisp. Rather than inventing new paradigms, all of these movements elaborated upon the ideas invented in the previous decade. One important trend in language design for programming large-scale systems during the 1980s was an increased focus on the use of *modules*, or large-scale organizational units of code. Modula-2, Ada, and ML all developed notable module systems in the 1980s, although other languages, such as PL/I, already had extensive support for modular programming. Module systems were often wedded to generic programming constructs.

The rapid growth of the Internet in the mid-1990s created opportunities for new languages. Perl, originally a Unix scripting tool first released in 1987, became common in dynamic websites. Java came to be used for server-side programming, and bytecode virtual machines became popular again in commercial settings with their promise of "Write once, run anywhere" (UCSD Pascal had been popular for a time in the early 1980s). These developments were not fundamentally novel, rather they were refinements to existing languages and paradigms, and largely based on the C family of programming languages.

Programming language evolution continues, in both industry and research. Current directions include security and reliability verification, new kinds of modularity (mixins, delegates, aspects), and database integration such as Microsoft's LINQ. The 4GLs are examples of languages which are domain-specific, such as SQL, which manipulates and returns sets of data rather than the scalar values which are canonical to most programming languages. Perl, for example, with its 'here document' can hold multiple 4GL programmes, as well as multiple JavaScript programmes, in part of its own perl code and use variable interpolation in the 'here document' to support multi-language programming.

5

Building Information Modeling

Building Information Modeling (BIM) is the process of generating and managing building data during its life cycle. Typically it uses three-dimensional, real-time, dynamic building modeling software to increase productivity in building design and construction. The process produces the Building Information Model (also abbreviated BIM), which encompasses building geometry, spatial relationships, geographic information, and quantities and properties of building components.

ORIGINS OF BIM

Charles M. Eastman at Georgia Tech coined the term BIM,. This theory is based on a view that the term BIM "Building Information Model" is basically the same as "Building Product Model", which Eastman has used extensively in his book and papers since the late 1970s.

('Product model' means 'data model' or 'information model' in engineering.). Architect and Autodesk building industry strategist Phil Bernstein, FAIA, first used the actual term BIM "building information modeling." Jerry Laiserin then helped popularize and standardize it as a common name for the digital representation of the building process as then offered primarily by Graphisoft, Bentley Systems, and Autodesk to facilitate exchange and interoperability of information in digital format. According to him and others, the first implementation of BIM was under the Virtual Building concept by Graphisoft's ArchiCAD, in its debut in 1987.

DEFINITION

Building information modeling covers geometry, spatial relationships, light analysis, geographic information, quantities and properties of building components (for example manufacturers' details). BIM can be used to demonstrate the entire building life cycle, including the processes of construction and facility operation. Quantities and shared properties of materials can be extracted easily. Scopes of work can be isolated and defined. Systems, assemblies and sequences can be shown in a relative scale with the entire facility or group of facilities. Dynamic information of the building, such as sensor measurements and control signals from the building systems, can also be incorporated within BIM to support analysis of building operation and maintenance. Under the guidance of a Virtual Design to Construction Project Manager (VDC) BIM can be seen as a companion to PLM as in the Product Lifecycle

Management, since it goes beyond geometry and addresses issues such as Cost Management, Project Management and provides a way to work concurrently on most aspects of building life cycle processes.

BIM goes far beyond switching to a new software. It requires changes to the definition of traditional architectural phases and more data sharing than most architects and engineers are used to. BIM is able to achieve such improvements by modeling representations of the actual parts and pieces being used to build a building. This is a substantial shift from the traditional computer aided drafting method of drawing with vector file-based lines that combine to represent objects.

The interoperability requirements of construction documents include the drawings, procurement details, environmental conditions, submittal processes and other specifications for building quality. It is anticipated by proponents that VDC utilizing BIM can bridge the information loss associated with handing a project from design team, to construction team and to building owner/operator, by allowing each group to add to and reference back to all information they acquire during their period of contribution to the BIM model.

For example, a building owner may find evidence of a leak in his building. Rather than exploring the physical building, he may turn to his BIM and see that a water valve is located in the suspect location. He could also have in the model the specific valve size, manufacturer, part number, and any other information ever researched in the past, pending adequate computing power. Such problems were

initially addressed by Leite et al. when developing a vulnerability representation of facility contents and threats for supporting the identification of vulnerabilities in building emergencies There have been attempts at creating a BIM for older, pre-existing facilities.

They generally reference key metrics such as the Facility Condition Index (FCI). The validity of these models will need to be monitored over time, because trying to model a building constructed in, say 1927, requires numerous assumptions about design standards, building codes, construction methods, materials, etc., and therefore is far more complex than building a BIM at time of initial design. The American Institute of Architects has further defined BIM as "a model-based technology linked with a database of project information", and this reflects the general reliance on database technology as the foundation. In the future, structured text documents such as specifications may be able to be searched and linked to regional, national, and international standards.

MANAGING THE BIM MODEL GUIDELINES

"The production of a Building Information Model (BIM) for the construction of a project involves the use of an integrated multi-disciplinary performance model to encompass the building geometry, spatial relationships, geographic information, along with quantities and properties of the building components.

The Virtual Design to Construction Project Manager (VDC - also known as VDCPM) is a professional in the field of

project management and delivery. The VDC is retained by a design build team on the clients' behalf from the pre-design phase through certificate of occupancy in order to develop and to track the object oriented BIM against predicted and measured performance objectives. The VDC manages the project delivery through multi-disciplinary building information models that drive analysis, schedules, take-off, and logistics.

The VDC is skilled in the use of BIM as a tool to manage and assess the technology, staff, and procedural needs of a project. In short the VDC is a contemporary project managing architect who is equipped to deal with the current evolution of project delivery. The VDC acts as a conduit to bridge time tested construction knowledge to digital analysis and representation."

BIM IN THE UK

In the UK, CPIC, responsible for providing best practice guidance on construction production information and formed by representatives of the major UK industry institutions, has proposed a definition of Building Information Modelling for adoption throughout the UK construction industry and has invited all UK industry parties to discuss it in order to ensure an agreed starting point. The proliferation of interpretations of the term currently hampers the adoption of a working method that will drastically improve the construction industry and the quality and sustainability of the deliveries from the design and construction team to clients.

BIM IN THE USA

CONTRACTORS

The Associated General Contractors and contracting firms also have developed a variety of working definitions of BIM that describe it generally as "an object-oriented building development tool that utilizes 5-D modeling concepts, information technology and software interoperability to design, construct and operate a building project, as well as communicate its details." 5-D modeling concepts involve modeling not only the 3 primary spatial dimensions of X, Y, and Z; but also time as the 4th dimension and cost as the 5th.

Although the concept of BIM and relevant processes are being explored by contractors, architects and developers alike, the term itself is under debate, and it is yet to be seen whether it will win over alternatives, which include:

- Virtual Building Environment (VBE)
- Virtual Design to Construction Project Manager (VDC)

BIM is seen to be closely related to Integrated Project Delivery (IPD) where the primary motive is to bring the teams together early on the project. A full implementation of BIM also requires the project teams to collaborate from the inception stage and formulate model sharing and ownership contract documents. BIM is often associated with IFCs (Industry Foundation Classes) and aecXML, which are data structures for representing information used in BIM. IFCs is developed by buildingSMART (International Alliance for Interoperability). Other data structures are proprietary, and many have been developed by CAD firms

that are now incorporating BIM into their software. One of the earliest examples of a nationally approved BIM standard is the AISC (American Institute of Steel Construction)-approved CIS/2 standard, a non proprietary standard with its roots in the UK.

Proponents claim that BIM offers:

1. Improved visualization
2. Improved productivity due to easy retrieval of information
3. Increased coordination of construction documents
4. Embedding and linking of vital information such as vendors for specific materials, location of details and quantities required for estimation and tendering
5. Increased speed of delivery
6. Reduced costs

In August 2004 the US National Institute of Standards and Technology (NIST) issued a report entitled "Cost Analysis of Inadequate Interoperability in the U.S. Capital Facilities Industry" (NIST GCR 04-867 (PDF), which came to the conclusion that, as a conservative estimate, $15.8 billion is lost annually by the U.S. capital facilities industry resulting from inadequate interoperability due to "the highly fragmented nature of the industry, the industry's continued paperbased business practices, a lack of standardization, and inconsistent technology adoption among stakeholders".

BIM IN FRANCE

In France, several bodies are pushing for a more integrated adoption of BIM standards, in order to improve software

interoperability and cooperation among actors of the building industry. Examples are the FFB (Fédération française du bâtiment), or the French arm of buildingSMART International who are supporting IFCs.

On the other hand, software editing companies such as Vizelia were early adopters of IFCs and can now benefit from the full potential of BIM in the Green Building fast-emerging business.

ADDITIONAL RESOURCES

BOOKS

BIG BIM little bim
Published October 2007
Written by Finith Jernigan, AIA
ISBN 978-0-9795699-0-6

Building Information Modeling: A Strategic Implementation Guide for Architects, Engineers, Constructors, and Real Estate Asset Managers
Published April 2009
Written by Dana K. Smith and Michael Tardif
ISBN 978-0-470-250003-7

Building Information Modeling: Planning and Managing Construction Projects with 4D CAD and Simulations
Published April 2008
Written by Willem Kymmell
ISBN 978-0-07-149453-3

BIM Handbook: A Guide to Building Information Modeling for Owners, Managers, Designers
Published March 2008
Written by Chuck Eastman, Paul Teicholz, Rafael Sacks, and Kathleen Liston
ISBN 978-0-470-18528-5

Interoperable Methodologies and Techniques in CAD. Chapter 4.
Written by Semiha Kiziltas, Fernanda Leite, Burcu Akinci, Robert Lipman
In: CAD and GIS Integration

Published December 2009
Edited by Hassan Karimi, Burcu Akinci
ISBN 978-1-4200-6805-4

Green BIM: Successful Sustainable Design with Building Information Modeling
Published April 2008
Written by Eddy Krygiel, Brad Nies; foreword by Steve McDowell, FAIA, BNIM
ISBN 978-0-470-23960-5

BIM and Construction Management: Proven Tools, Methods and Workflows
Published May 2009
Written by Brad Hardin; foreword by Eddy Krygiel
ISBN 978-0-470-40235-1

Handbook of Research on Building Information Modeling and Construction Informatics: Concepts and Technologies
Published December 2009
Written by Jason Underwood, Umit Isikdag; foreword by Dana K. Smith
ISBN 978-1-60566-928-1

RESEARCH REPORTS

McGraw-Hill Construction SmartMarket Report on BIM.
Published December 2008
Written by Stephen A Jones
Research with hundreds of current BIM users on implementation and ROI. Includes 4-page special section “Introduction to BIM”.

The Business Value of BIM - McGraw-Hill Construction SmartMarket Report
Published December 2009
Written by Stephen A Jones
Study of BIM adoption in North America and the ways in which users are experiencing business value and generating ROI

Green BIM: How BIM is Contributing to Green Design and Construction - McGraw-Hill Construction SmartMarket Report
Published August 2010
Written by Stephen A Jones
Study of how the tools and processes of BIM are contributing to higher performing buildings, more environmentally responsible construction practices and achievement of green objectives during operations and maintenance.

The Business Value of BIM in Europe - McGraw-Hill Construction SmartMarket Report

Published October 2010
Written by Stephen A Jones
Study of how BIM is being adopted and implemented in 3 major European economies: France, Germany and UK. Includes comparisons to North American data collected as part of 2009 SmartMarket Report on Business value of BIM in North America.

VIDEOS

Thoughts on BIM by John Stebbins, CEO, Digital Vision Automation http://www.digitalvis.com/bim/

ANTICIPATED FUTURE POTENTIAL

BIM is a relatively new technology in an industry typically slow to adopt change. Yet many early adopters are confident that BIM will grow to play an even more crucial role in building documentation. BIM provides the potential for a virtual information model to be handed from Design Team (architects, surveyors, consulting engineers, and others) to Contractor and Subcontractors and then to the Owner, each adding their own additional discipline-specific knowledge and tracking of changes to the single model.

The result greatly reduces the information loss that occurs when a new team takes "ownership" of the project as well as in delivering extensive information to owners of complex structures. It also prevents errors made by design team members as well as the construction team (Contractors and Subcontractors) by allowing the use of conflict detection where the computer actually informs team members about parts of the building in conflict or clashing, and through detailed computer visualization of each part in relation to the total building. As computers and software become more capable of handling more building information, this will

become even more pronounced than it is in current design and construction projects. This error reduction is a great part of cost savings realized by all members of a project. Reduction in time required to complete construction directly contributes to the cost savings numbers as well. It's important to realize that this decrease can only be accomplished if the models are sufficiently developed in the Design Development phase. The Industry Foundation Classes (IFC/ifcXML) are an open specification for Building Information Modeling and are used to share and exchange BIM in a neutral format among various software applications.

Green Building XML (gbXML) is an emerging schema, a subset of the Building Information Modeling efforts, focused on green building design and operation. gbXML is used as input in several energy simulation engines. But with the development of modern computer technology, a large number of building energy simulation tools are available on the market. When choosing which simulation tool to use in a project, the user must consider the tool's accuracy and reliability, considering the building information they have at hand, which will serve as input for the tool. Yezioro, Dong and Leite developed an artificial intelligence approach towards assessing building performance simulation results and found that more detailed simulation tools have the best simulation performance in terms of heating and cooling electricity consumption within 3% of mean absolute error.

Whole Building Design Guide

The Whole Building Design Guide or WBDG is the most used online resource for building information in the world,

with over 500,000 distinct users per month and over 3 million document downloads according to data as of January 2011. The online portal covers a wide range of topics, from performance and sustainability to security and resilience. WBDG is based on the premise that to create a successful high-performance building, one must apply an integrated design and team approach in all phases of a project, including planning, design, construction, operations and maintenance. The WBDG is managed by the National Institute of Building Sciences.

HISTORY

The WBDG was initially designed to serve U.S. Department of Defense (DOD) construction programmes. A 2003 DOD memorandum named WBDG the "sole portal to design and construction criteria produced by the U.S. Army Corps of Engineers (USACE), Naval Facilities Engineering Command (NAVFAC), and U.S. Air Force." Since then, WBDG has expanded to give all building industry professionals free, wide access to federal and other design, construction and performance criteria. The WBDG is public and freely available to anyone. The majority of users are from the private sector. The WBDG draws information from the Construction Criteria Base and a privately-owned database run by Information Handling Services.

A significant amount of the Whole Building Design Guide content is organized by three categories: Design Guidance, Project Management, and Operations and Maintenance. It is structured to provide WBDG visitors first a broad

understanding then increasingly specific information more targeted towards building industry professionals. WBDG is the resource that federal agencies look to for policy and technical guidance on Federal High Performance and Sustainable Buildings In addition, the WBDG contains online tools, the original Construction Criteria Base, Building Information Modeling guides and libraries, a database of select case studies, federal mandates and other resources. The WBDG also provides online continuing education courses for architects and other building professionals, free of charge.

DEVELOPMENT

Development of the WBDG is a collaborative effort among federal agencies, private sector companies, non-profit organizations and educational institutions. Its success depends on industry and government experts contributing their knowledge and experience to better serve the building community. The WBDG web site is offered as an assistant to the building community by the National Institute of Building Sciences through funding support from the DOD, the NAVFAC Engineering Innovation and Criteria Office, U.S. Army Corps of Engineers, the U.S. Air Force, the U.S. General Services Administration (GSA), the U.S. Department of Veterans Affairs, the National Aeronautics and Space Administration (NASA), and the U.S. Department of Energy (DOE), and the assistance of the Sustainable Buildings Industry Council (SBIC). A Board of Direction and an Advisory Committee consisting of representatives from over 25 participating federal agencies guide the development of the WBDG.

Virtual Design and Construction

Virtual Design to Construction (VDC) is the management of integrated multi-disciplinary performance models of design-construction projects, including the Product (i.e., facilities), Work Processes and Organization of the design - construction - operation team in order to support explicit and public business objectives.

The theoretical basis of VDC includes:

- Engineering modeling methods: product, organization, process
- Analysis methods (model-based): including schedule, cost, 4D interactions and process risks, these are termed BIM tools
- Visualization methods
- Business metrics and focus on strategic management
- Economic Impact analysis (i.e., models of both the cost and value of capital investments)

VDC PROJECT MANAGER

"The production of a Building Information Model (BIM) for the construction of a project involves the use of an integrated multi-disciplinary performance model to encompass the building geometry, spatial relationships, geographic information, along with quantities and properties of the building components. The Virtual Design to Construction Project Manager (VDC - also known as VDCPM) is a professional in the field of project management and delivery.

The VDC is retained by a design build team on the clients' behalf from the pre-design phase through certificate

of occupancy in order to develop and to track the object oriented BIM against predicted and measured performance objectives. The VDC manages the project delivery through multi-disciplinary building information models that drive analysis, schedules, take-off, and logistics. The VDC is skilled in the use of BIM as a tool to manage and assess the technology, staff, and procedural needs of a project. In short the VDC is a contemporary project managing architect who is equipped to deal with the current evolution of project delivery.

The VDC acts as a conduit to bridge time tested construction knowledge to digital analysis and representation. VDC position avoids the well intentioned failures created by competent managers who lack the knowledge to implement the technology for which they are entrusted. Recent economic conditions have placed a spot light on industry wide deficiency in the organization of architectural staff, the lack of interoperability of project generated information, and the amount of non-beneficial redundancy which eventually finds its way to the client through an inferior project with increased cost.

The VDC fulfills a critical role in contemporary project delivery in part due to the single platform integration of sketch tools, massing, solid modeling, analysis, & rendering organized within a singular object change engine. Available technology removes the need for digital redundancies and file conversions at each stage of design. Information can be tracked and managed from inception to project delivery with the use of a qualified VDC who secures the clients return on investment by tracking stated project performance

objectives. The development of virtual design tools from 1957 to 2007 has created a digital landfill of applications, many whose continued use has hindered progress all the while accelerating Architect, Engineering, Contractor costs without increased accuracy, efficiency, or integration of disciplines."

VDC MANAGED BIM PROJECT MODEL

"Virtual Design to Construction BIM models are virtual because they show computer-based descriptions of the project. The BIM project model emphasizes those aspects of the project that can be designed and managed, i.e., the *product* (typically a building or plant), the *organization* that will define, design, construct and operate it, and the *process* that the organization teams will follow, or POP. These models are logically integrated in the sense that they all can access shared data, and if a user highlights or changes an aspect of one, the integrated models can highlight or change the dependent aspects of related models. The models are multi-disciplinary in the sense that they represent the Architect, Engineering, contractor (AEC) and Owner of the project, as well as relevant sub disciplines. The models are performance models in the sense that they predict some aspects of project performance, track many that are relevant, and can show predicted and measured performance in relationship to stated project performance objectives. Some companies now practice the first steps of BIM modeling, and they consistently find that they improve business performance by doing so."

CONSTRUCTION INDUSTRY BIM TOOLS AND METHODOLOGIES UTILIZED BY VDC

BIM SOFTWARE TOOLS

- ArchiCAD from Graphisoft
- Building Explorer
- Autodesk Navisworks JetStream 4D
- Autodesk Revit
- Autodesk AutoCAD Civil 3D
- Tekla Structures from Tekla Corporation
- Advance Concrete
- Advance Steel
- Microstation

VDC RELATED METHODOLOGIES

- Semantic integration
- Learning-by-doing
- Deductive-nomological model
- Scientific evidence
- Hypothesis
- Qualitative research
- Quantitative research
- Case-based reasoning
- Action research
- Power of a method
- Upper ontology within the Ontology domain
- Schema representation
- Work breakdown structure
- Object-oriented programming

3D Floor Plan

A 3D floor plan, or 3D floorplan, is a virtual model of a building floor plan, depicted from a birds eye view, utilized within the building industry to better convey archtitectural plans. Usually built to scale, a 3D floor plan must include walls and a floor and typically includes exterior wall fenestrations, windows, and doorways. It does not include a ceiling so as not to obstruct the view. Other common attributes may be added, but are not required, such as cabinets, flooring, bathroom fixtures, paint color, wall tile, and other interior finishes. Furniture may be added to assist in communicating proper home staging and interior design.

PURPOSE

3D floor plans assist real estate marketers and architects in explaining floor plans to clients. Their simplicity allows individuals unfamiliar with conventional floor plans to understand difficult architectural concepts. This allows architects and homeowners to literally see design elements prior to construction and alter design elements during the design phase. 3D floorplans are often commissioned by architects, builders, hotels, universities, real estate agents, and property owners to assist in relating their floor plans to clients.

CONSTRUCTION

A 3d floor plan is built utilizing advanced 3d rendering software, the same type of software used to create major

animated motion pictures. Through complex lighting, staging, camera, and rendering techniques 3D floorplans appear to be real photographs rather than digital representations of the buildings they are modeled after.

6

CAD Standards Design

CAD Standards have been created to improve productivity and interchange of Computer-aided design documents between different offices and CAD programmes, especially in architecture and engineering.

AEC (ARCHITECTURE ENGINEERING AND CONSTRUCTION) STANDARDS

CAD LAYER STANDARDS

Most common:

- *BS 1192*, which relies heavily on the Code of Procedure for the Construction Industry
- *AIA Cad Layer Guidelines*, 2nd edition (1997), has a great usage in the USA;
- *ISO 13567-1/3*, International standard, common in Northern Europe;

- *AEC (UK)*, an adaptation of BS1192 based on Uniclass.
- *A/E/C CADD Standard, CADD/GIS standards adopted by U.S. Government's CADD/GIS Technology Center for Facilities, Infrastructure, and Environment..]*
- *SIA 2014* (1996), Swiss standard for engineers and architects, based on ISO 13567.

Samples of standardised layers:

A-B374—E- (ISO13567: agent Architect, element Roof window in SfB, presentation graphic element);

A-37420-T2N01B113B23pro (ISO13567: agent Architect, element Roof Window in SfB, presentation Text#2, New part, floor 01, block B1, phase 1, projection 3D, scale 1:5(B), work package 23 and user definition "pro");

A-G25—D-R (ISO13567: agent Architect, element wall in Uniclass, presentation dimensions, status Existing to be removed);

A-G251-G-WallExtl-Fwd (AEC(UK): agent Architect, element External Wall in Uniclass, presentation graphic element, user definition "WallExtl" and view Forward);

A210_M_ExtWall (BS1192: agent Architect, element External Wall in SfB, presentation model, user definition "ExtWall");

A-WALL-FULL (AIA: agent Architect, element Wall, Full height).

LINE THICKNESS

Thickness for pens and plot: *0,13 mm Gray, 0,18 mm Red, 0,25 mm White, 0,35 mm Yellow, 0,50 mm Magenta, 0,70 mm Blue, 1,00 mm Green*. In AutoCAD usually parts to be printed in black are drawn in 1 to 7 basic colors. Color layer: Green-Center, Magenta-Measure of length and Blue-Hidden.

Description	***Line thickness in mm***	***Color Codes***
Out Line	0.20 or 0.25	White, Cyan, Yellow,Blue
Hidden Line	0.00 or 0.05	Blue, Gray, 241
Center Line	0.10 or 0.15	Green, Red
Note	0.18 or 0.20	Cyan, Green, 41
Thin Line	0.00 or 0.05	Gray, 08, 111

Reference Line	0.000	Magenta, Gray
Hatch Line	0.000	Magenta, Green, Gray
Color-9 to 256	0.000	
Dimension line		
Leader Line with Arrows	0.000	Gray Color-9,or 8, Red
Text	0.18 or 0.20	Cyan, Green,

TEXT AND DIMENSION

Heights: 1,8 mm, 2,5 mm, 3,5 mm, 5,0 mm, 7,0 mm (relative thickness are 1/10 of the heights). Font styles: "Romans", "ISO CP". Exceptional use of true type fonts (arial, etc.).

SCALES

1:1, 1:10, 1:100 .... 1:2, 1:20, 1:200 .... 1:5, 1:50, 1:500 ....

FILE NAMING STANDARDS

- BS 1192:
  Discipline (1 char), Element (2 char, using SfB Table 1 or Uniclass), Drawing type (1 char, P=preliminary, X=special/xref, L=layout, C=component, S=schedules, A=assembly drawings, K=co-ordination drawing), Unique number (3 char), Revision (1 char, A=emission, B,C,D...= revisions). Samples: A22P012G.dwg (architect, internal walls in SfB, preliminary design, sheet 012, revision G).
- AIA
- AEC

Samples: ZE1G-124.dwg, XE1G-100.dwg

- AEC (UK)

Project (unlimited char), Discipline (2 char max, recommended mandatory), Zone (optional), View (1 char, rec.mand.), Level (2 char, rec.mand.), Content (rec.mand.), Sequential number (up to 3 char); Samples: 1234-A-Off-P-M1-Furn-11c.dwg (project #1234, architect, office zone, plan, mezzanine 1, furnitures, version 1.1 revision c), A-P-01-Part (architect, plan, 1st floor, partitions), 1234-A-S-055.dgn (project #1234, architect, section, sheet 055), A-S-xx-AA.dwg (architect, section, full building, section AA), A-P-x-Grid.dgn (architect, plan, all floor, grid), 1838-S-C-P-03 (project 1838, structures, building C, plan, 3rd floor).

- Other local:
  Job number (4 char), Agent (1 char), Section (4 char), phase (1 char), sheet number (2 char), revision (2 char) Samples: 0512-A-00A_-1-01-02.dwg Job number (3 char), View (2 char), section (2char), phase (1 char), revision (1 char) Samples: 123p0s2d0.dwg (job 1239, plan, 2nd floor, definitive drawing, emission), 459s0BBD0.dgn (job 123, section B, definitive drawing, emission).

MCAD (MECHANICAL) STANDARDS

GEOMETRIC DIMENSIONING AND TOLERANCING

Industry standards for defining Product Manufacturing Information (PMI), a 3D extension of Geometric Dimensioning and Tolerancing (GD&T), include ASME Y14.41-2003 Product

Data Definition and ISO 1101 Representation of specifications in the form of a 3D model.

GEOMETRY QUALITY

VDA 4955

PRODUCT DATA QUALITY

PDQ is a field of PLM relating to the quality of product data, particularly the geometrical and organizational quality of CAD data. Checkers, software that analyze CAD data formats, are often employed before and after data translation. The checkers can check the organization and quality of the data against internal company standards and international or industry standards. These checkers can be built into specific CAD packages or work on a number of CAD file formats.

In 2006/2007 Part 59 of STEP ISO 10303-59 *Product data representation and exchange: Integrated generic resource: Quality of product shape data* is under development. It defines how to represent quality criteria together with measurement requirements and representation of inspection results.

Digital Architecture

Digital architecture uses computer modeling, programming, simulation and imaging to create both virtual forms and physical structures. The terminology has also been used to refer to other aspects of architecture that feature digital technologies. The emergent field is not clearly

delineated to this point, and the terminology is also used to apply to digital skins that can be streamed images and have their appearance altered. A headquarters building design for Boston television and radio station WGBH by Polshek Partnership has been discussed as an example of digital architecture and includes a digital skin. Architecture created digitally might not involve the use of actual materials (brick, stone, glass, steel, wood).

It relies on "sets of numbers stored in electromagnetic format" used to create representations and simulations that correspond to material performance and to map out built artifacts. Digital architecture does not just represent "ideated space" it also creates places for human interaction that do not resemble physical architectural spaces. Examples of these places in the "Internet Universe" and cyberspace include websites, multi-user dungeons, MOOs, and web chatrooms.

Digital architecture allows complex calculations that delimit architects and allow a diverse range of complex forms to be created with great ease using computer algorithms. The new genre of "scripted, iterative, and indexical architecture" produces a proliferation of formal outcomes, leaving the designer the role of selection and increasing the possibilities in architectural design. This has "re-initiated a debate regarding curvilinearity, expressionism and role of technology in society" leading to new forms of non-standard architecture by architects such as Zaha Hadid and UN Studio. A conference held in London in 2009 named "Digital Architecture London" introduced the latest development in digital design practice. The Far Eastern

International Digital Design Award (The Feidad Award) has been in existence since 2000 and honours "innovative design created with the aid of digital media."

In 2005 a jury with members including a representative from Quantum Film, Greg Lynn from Greg Lynn FORM, Jacob van Rijs of MVRDV, Gerhard Schmitt, Birger Sevaldson (Ocean North), chose among submissions "exploring digital concepts such as computing, information, electronic media, hyper-, virtual-, and cyberspace in order to help define and discuss future space and architecture in the digital age."

Molecular Design Software

Molecular design software is a software for molecular modeling, distinctive property of which is the presence of the special support for developing the molecular models. In contrast to the usual molecular modeling programmes such as the molecular dynamics and quantum chemistry programmes, such software directly supports the aspects related to the construction of molecular models:

- Molecular graphics
- interactive molecular drawing and conformational editing
- building of polymeric molecules, crystals and solvated systems
- partial charges development
- geometry optimization
- support for the different aspects of Force Field development
- *etc.*

Molecular Modelling

Molecular modelling encompasses all theoretical methods and computational techniques used to model or mimic the behaviour of molecules. The techniques are used in the fields of computational chemistry, computational biology and materials science for studying molecular systems ranging from small chemical systems to large biological molecules and material assemblies. The simplest calculations can be performed by hand, but inevitably computers are required to perform molecular modelling of any reasonably sized system. The common feature of molecular modelling techniques is the atomistic level description of the molecular systems; the lowest level of information is individual atoms (or a small group of atoms). This is in contrast to quantum chemistry (also known as electronic structure calculations) where electrons are considered explicitly. The benefit of molecular modelling is that it reduces the complexity of the system, allowing many more particles (atoms) to be considered during simulations.

MOLECULAR MECHANICS

Molecular mechanics is one aspect of molecular modelling, as it refers to the use of classical mechanics/Newtonian mechanics to describe the physical basis behind the models. Molecular models typically describe atoms (nucleus and electrons collectively) as point charges with an associated mass. The interactions between neighbouring atoms are described by spring-like interactions (representing chemical bonds) and van der Waals forces. The Lennard-Jones potential is commonly used to describe van der Waals forces.

The electrostatic interactions are computed based on Coulomb's law. Atoms are assigned coordinates in Cartesian space or in internal coordinates, and can also be assigned velocities in dynamical simulations. The atomic velocities are related to the temperature of the system, a macroscopic quantity.

The collective mathematical expression is known as a potential function and is related to the system internal energy (U), a thermodynamic quantity equal to the sum of potential and kinetic energies. Methods which minimize the potential energy are known as energy minimization techniques (e.g., steepest descent and conjugate gradient), while methods that model the behaviour of the system with propagation of time are known as molecular dynamics. This function, referred to as a potential function, computes the molecular potential energy as a sum of energy terms that describe the deviation of bond lengths, bond angles and torsion angles away from equilibrium values, plus terms for non-bonded pairs of atoms describing van der Waals and electrostatic interactions.

The set of parameters consisting of equilibrium bond lengths, bond angles, partial charge values, force constants and van der Waals parameters are collectively known as a force field. Different implementations of molecular mechanics use different mathematical expressions and different parameters for the potential function. The common force fields in use today have been developed by using high level quantum calculations and/or fitting to experimental data. The technique known as energy minimization is used to find positions of zero gradient for all atoms, in other words, a

local energy minimum. Lower energy states are more stable and are commonly investigated because of their role in chemical and biological processes.

A molecular dynamics simulation, on the other hand, computes the behaviour of a system as a function of time. It involves solving Newton's laws of motion, principally the second law. Integration of Newton's laws of motion, using different integration algorithms, leads to atomic trajectories in space and time. The force on an atom is defined as the negative gradient of the potential energy function. The energy minimization technique is useful for obtaining a static picture for comparing between states of similar systems, while molecular dynamics provides information about the dynamic processes with the intrinsic inclusion of temperature effects.

VARIABLES

Molecules can be modelled either in vacuum or in the presence of a solvent such as water. Simulations of systems in vacuum are referred to as *gas-phase* simulations, while those that include the presence of solvent molecules are referred to as *explicit solvent* simulations. In another type of simulation, the effect of solvent is estimated using an empirical mathematical expression; these are known as *implicit solvation* simulations.

APPLICATIONS

Molecular modelling methods are now routinely used to investigate the structure, dynamics, surface properties and thermodynamics of inorganic, biological and polymeric

systems. The types of biological activity that have been investigated using molecular modelling include protein folding, enzyme catalysis, protein stability, conformational changes associated with biomolecular function, and molecular recognition of proteins, DNA, and membrane complexes.

POPULAR SOFTWARE FOR MOLECULAR MODELLING

- Abalone
- ADF
- AMBER
- Ascalaph Designer
- AutoDock,
- AutoDock Vina,
- BALLView
- Biskit
- BOSS
- Cerius2
- CHARMM
- Chimera
- Coot
- COSMOS (software)
- CP2K
- CPMD
- Culgi
- Discovery Studio
- DOCK
- Firefly

- FoldX
- GAMESS (UK)
- GAMESS (US)
- GAUSSIAN
- Ghemical
- Gorgon
- GROMACS
- GROMOS
- InsightII
- LAMMPS
- LigandScout
- MacroModel
- MADAMM.
- MarvinSpace
- Materials and Processes Simulations
- Materials Studio
- MDynaMix
- MMTK
- Molecular Docking Server
- Molecular Operating Environment (MOE)
- MolIDE
- Molsoft ICM
- MOPAC
- NAMD
- NOCH
- Oscail X
- PyMOL
- Q-Chem
- ReaxFF
- ROSETTA

- SCWRL
- Sirius
- Spartan (software)
- StruMM3D (STR3DI32)
- Sybyl (software)
- MCCCS Towhee
- TURBOMOLE
- VMD
- WHAT IF
- xeo
- YASARA
- Zodiac (software)

Molecule Editor

A molecule editor is a computer programme for creating and modifying representations of chemical structures. Molecule editors can manipulate chemical structure representations in either two- or three-dimensions. Two-Dimensional editors generate output used as illustrations or for querying chemical databases. Three-dimensional molecule editors are used to build molecular models, usually as part of molecular modelling software packages. Database molecular editors such as Leatherface, RECAP and Molecule Slicer allow large numbers of molecules to be modified automatically according to rules such as 'deprotonate carboxylic acids' or 'break exocyclic bonds' that can be specified by the user. Molecule editors typically support reading and writing at least one file format or line notation. Examples of each include Molfile and SMILES, respectively. Files generated by molecule editors can be displayed by molecular graphics tools.

ONLINE EDITORS

- ChemDoodle Web Components HTML5 chemistry web components including viewers, animations, interactive components and editors by iChemLabs. Pure Javascript code using Canvas and WebGL graphics. Free and open source under the GPL v3.0 license.
- ChemWriter by Metamolecular. Written in pure JavaScript. Runs on Internet Explorer 6-9 and modern standards-compliant browsers. Touch interface supported on iPad.
- jsMolEditor, the world's first molecule structure editor in Javascript. Runs in most web browsers, no plugin or virtual machine is required. Free and open source under the LGPL v3.0 license.
- Marvin molecule editor and viewer: proprietary software from ChemAxon. Supports all major formats and structure/query features. This Java implementation also includes unlimited structure based predictions for a range of properties (pKa, logD, name<>structure, etc.).
- Molinspiration WebME molecule editor: proprietary software, based on Ajax technology which does not require Java.
- PubChem online molecule editor, supports SMILES, SMARTS and InChI as well as all common chemical file formats.
- Molecular Editor and Image Sharer Molecular editor based on JChemPaint. Allows to store generated images on the server.

MOBILE EDITORS

- ChemJuice: iPhone app from IDBS.
- Mobile Molecular DataSheet: BlackBerry app from Molecular Materials Informatics.

7

Computer Design and Analysis Technologies

The Design and Analysis Technologies critical technology area includes technologies or processes that are pervasive within the aerospace and defence sector. The technology elements within the Design Technologies critical technology area are depicted in the figure below and described in subsequent paragraphs:

Multidisciplinary Design and Optimization

Multidisciplinary design and optimization is as the name implies, the process of combining a full set of computational design tools to create an optimum design. The process is necessarily iterative in nature and all of the disciplines normally utilized in an aircraft design are computationally intensive. An MDO approach for an aircraft could include aerodynamics, structures, and systems Computer Aided

Engineering (CAE) tools. Initial design assumptions would be input to each CAE toolset and the constraints and parameters to be optimized defined. Each CAE suite would then compute design parameters that would be utilized by the other CAE tools as a subset of their required inputs.

The ultimate design would theoretically be structurally sounder, lighter and more cost effective to fabricate. The design timeframe would be also very much shortened. The challenges to this process are in the exchange of data between the CAE applications and the tuning of the entire process to achieve convergence on the final solution set in an efficient manner.

Structural Analysis

The optimization of analytical design tools is a process that will lead to shortened design time frames, lighter and more efficient designs, with reduced production and life cycle costs of the final design.

The many analytical tools now available have been typically developed for specific applications and are often not readily applicable outside of their original design target arena. An example lies in the structural analysis field where tools developed for metallics will be much different from those developed for composite materials where material properties may vary according to axis.

The ability to rapidly define an optimized aircraft structure having light weight, and improved fatigue and damage tolerance capabilities, is a critical technology to maintain competitive leadership in the development and supply of future new aircraft. This will be achieved by the

extensive use of computerized methods for structural analysis and design optimization, and the analysis of failure and fracture mechanics. The methods must be integrated with the in-house design and manufacturing data bases, the 3-D CAD/CAMsystems, and also be easy to use. Suppliers and partners will have access to the resulting design information via Technical Data Interchange (TDI). This will ensure consistency with an up-to-date knowledge of the requirements for loads, interfaces and the space envelopes available for their products. The immediate dissemination to suppliers of information on design changes will help diminish subsequent redesign activity and the time and cost penalties incurred for rework.

The preliminary structural design will often use detailed Finite Element Methods (FEM) for analysis, coupled with constrained optimization, and the process must be highly automated for rapid creation of FEM meshing for models. In order to achieve shortened design cycle time, the loads and dynamics stiffness requirements must become available much sooner than at present. This will require early development of MDO models for overall aerodynamic and structural optimization that will define the static and dynamic loads for flight and ground operations. Trade-off studies must rapidly search for the best designs and arrive at realistic structural sizes, providing space envelopes and accurate weights to minimize subsequent redesign.

Structural Design, Analysis and Optimization

Shortened design cycle times are necessary for achieving market advantage in the aerospace and defence sector. Improvements in the structural analysis, design and

optimization of gas turbine engines is necessary to achieve these goals while also meeting the overall objectives of increased durability and efficiency at lower costs.

A Multi-disciplinary Design Optimization (MDO) approach that combines finite element analysis and aerodynamic design techniques is employed. MDO is necessary to rapidly determine the structure of the engine and identify critical areas requiring further or more detailed analysis.

Many of the structural and aerodynamic codes developed by companies are proprietary in nature and the integration and refinement of these codes is an on-going challenge.

COMPUTATIONAL FLUID DYNAMICS

Computational Development and Validation

Computational Fluid Dynamics (CFD) has had the greatest effect on both aircraft and engine design of any single design tool over the past twenty-five years. Computational power and cost have enabled widespread application and development of CFD techniques. Computational fluid dynamics is basically the use of computers to numerically model flows of interest. Nodes in the flowpath are identified and equations of motion solved at these locations to identify flow parameters.

In essence a grid or mesh is defined over the surface of the object that extends outwards into the flowfield containing the object. Flow equations are then calculated at each node in the grid, and iteratively re-calculated until all results for each node are within an acceptable variance. The equations used are either Euler based which do not include viscous

effects (boundary layers) directly, or Navier-Stokes equations which include viscous effects and which produce more accurate but computationally more demanding solutions. Such methods can be used for external flows about an aircraft or for internal flows in a gas turbine including combustion. The Euler based analyses are typically less computationally demanding but are less precise for modeling separated flows on wings and bodies, or for internal reversed flows. It should be noted that Navier first developed his equations in 1823 and that Stokes refined them in 1845. The development of solutions to these equations was not feasible until the latter part of this century. Today much R&D effort on NS methods is expended on improving modeling of the turbulent flow terms for specific problems.

Numerous forms of Euler and Navier-Stokes solutions have been developed to address particular design problems. Solutions to these equations are dependent on experimentation for both coefficients and for validation.

Mesh selection and node placement is critical to the solution of the flowfield. The automated generation of meshes is now in wide spread use and can often be linked to Computer Aided Engineering and Design tools. The form of the equation used, the density of the mesh or grid and convergence requirements determine computational demands. Complete aircraft solutions require huge computer resources and much R&D is aimed at improving the speed of the solution.

Computational Fluid Dynamics - Gas Turbines

CFD is perhaps the single most critical technology for gas turbine engines. Gas turbineCFD needs have typically posed

the greatest challenges to engine designers, and computational power and code developers. While CFD is of utmost importance to the engine designer it is a very specific disciplinary design requirement and competence is held by a very small number of engine design firms worldwide.

Computation techniques for gas turbine engines also tend to be very module specific — compressor, transition duct, combustor, turbine and exhaust duct/military afterburner are examples. Computational techniques are often also specific to engine size class and thus Canada, focusing on small gas turbines, has a specific set of technology requirements.

Advanced 3D CFD codes have been used to generate the following design improvements:

- In the compressor to develop advanced swept airfoils capable of high compression ratios that in turn yield higher efficiency at less weight and with a smaller parts count (significant life cycle cost factor);
- In the combustor for higher intensity (smaller volumes with much higher energy density) combustors that approach stoichiometric conditions to yield higher efficiency with lower weight; and
- In the turbine to produce higher stage loading with reduced turbine cooling air requirements that again reduces weight and cost while reducing fuel burn.

Combustion Systems Computation

The combustor of a gas turbine engine is that part of the engine that receives the compressed air from the compressor. Energy is added to the airflow in the combustor in the form

of chemical energy derived from fuel. The combustor discharge air is expanded across a turbine or turbines where energy is extracted to drive the compressor and gearbox of a turboshaft/turboprop engine, or to provide jet thrust via a turbofan and core nozzle in a thrust engine.

Small gas turbines, of the size that have typically been designed and built in Canada pose significant design challenges because of their size. Pratt and Whitney Canada combustors are the highest intensity combustors in the world, where intensity can be thought of as the amount of energy converted per unit volume within the combustor. The design objectives for gas turbine engines, including small ones, are to increase both overall pressure ratios and cycle temperatures, which lead to increased efficiency and smaller size and weight, while simultaneously producing reduced noise and noxious emissions levels.

Combustor technology development challenges for Canadian engine manufacturers include.

Computational fluid dynamics: CFD analyses are complicated by the reverse flow designs typically selected to maintain short combustors within small volumes. Cooling flow and chemical additions to the CFD design further complicate the process as the temperatures of gases at the core of the flows are well above the melting temperatures of the combustor materials. Pressure losses and cooling flow requirements must be minimized to improve performance.

Materials: Increasing compressor ratios result in increased compressor discharge temperatures and decreased cooling capability. These increased temperatures also push for higher fuel to air ratios and higher temperatures within

the combustor. Stoichiometric ratio is that ratio when all oxygen is consumed in the combustion process leaving less air for cooling. Materials challenges in this environment are the most demanding. Fuel injection and mixing: CFD and injector specific techniques are required.

Emissions: While not legislated and not contributing significantly in absolute terms, there is a drive for lower emissions that drives designs often in the opposite direction to those factors identified above.

AERODYNAMICS AND FLIGHT MECHANICS

Aerodynamics is the study of forces on wing bodies and controls due to air pressure and viscous (drag) effects. Flight mechanics is the study of the resulting motion of objects through the air and includes the stability and control Behaviour. The laws of motion and aerodynamics are combined to ensure that an aircraft flies in the intended manner. Much of the aerodynamics and flight mechanics work that is pursued for the purposes of aircraft designed and built in Canada will pertain to such issues as the design of improved wings, the integration of various components onto an aircraft or issues such as flight in adverse conditions where the handling qualities of an aircraft will be adversely influenced by the build up of ice on the surface of the wing. Advanced technology development in this field will be directed towards supersonic transports and eventually hypersonic flight. There are considerable differences between fixed wing and rotary wing aircraft aerodynamics and flight mechanics and both areas are of considerable interest to the Canadian aerospace and defence industry.

Technologies relevant to Aerodynamics and Flight Mechanics are described below:

Advanced Aerodynamics and Handling

Included here are technologies that will enable the Canadian Aerospace industry to contribute to the design of advanced concept aircraft technologies or components or be the lead design integrator.

These enabling technologies should be pursued dependent on their links to, and pre-positioning for potential application to specific aircraft platforms or types as follows:

- *Future Transport Aircraft*: Future transport aircraft will have to demonstrate increased speed and load carrying capabilities over greatly extended ranges. Specific targets have been set by the U.S. for next generation transport aircraft although no new advanced concept transport aircraft are currently well advanced. Wing loading factors will double over that of existing aircraft with the development of materials new to the transport aircraft envelope. For shorter-range aircraft, a key enabling technology will be that of high efficiency turboprop engines with cruise speeds above the M.72 range. Propulsion technology and propulsion integration issues, aircraft design optimization, CFD, and materials technology development and insertion will be key to the success of the future transport aircraft.
- *Hypersonic Aircraft*: Hypersonic aircraft are in exploratory or advanced development model stage at this time and will be used initially for low cost space launch and delivery platforms and subse-quently for

commercial transport. Propulsion technologies are significant to hypersonic vehicle feasibility and are now the limiting factor. Variable cycle engines, advanced materials, endothermic fuels and fuel control technologies are key aeropropulsion technology elements where significantR&D remains unsatisfied. Numerous controls and materials research topics require further investment as well, although less uncertainty remains in these areas due to advances made through the shuttle Programmes.

- *Advanced Rotorcraft*: Future rotorcraft will demonstrate increased cruise speeds of 200 kts or greater with tiltrotor speeds approaching 450 kts. These cruise speeds will be possible at significantly reduced vibration levels and with greatly increased range/fuel economy. Many of the design concepts for attaining these performance improvements are already in development, however much work remains undone.
- *Advanced Rotorcraft Flight Mechanics*: For both conventional helicopter and tiltrotor blades, the wings and propulsion system operate in a very complex aeromechanical environment. Aerodyn-amics, structures, vibration and acoustics parameters are inseparable and typically drive the design of the entire air vehicle. In trimmed forward flight the advancing blade tip will be moving at near sonic velocities whilst the retreating blade is often in near stall conditions.

Advanced Design and Development

General aviation aircraft pose specific design challenges in all aspects of their design and fabrication. Increasing

availability of low cost and high performance avionics, advanced composite designs and powerplant integration all offer opportunities for general aviation aircraft designers and builders.

Many of the technologies being furthered for use in military unmanned aerial vehicles will be of pertinence to general aviation aircraft.

Low cost gas turbine technologies and composite structures development and certification issues will likely be the technologies of greatest interest.

The development of technologies for military purposes will underwrite some of the costs of introduction of those design concepts into general aviation use.

Experimental Assessment and Performance

Analytical design and analysis techniques are a prerequisite to reductions in design cycle time, design and production costs, and improved safety and environmental impact. The development of these analytical or numerical design techniques will remain heavily dependent on experimental validation of design codes and performance targets for another 10-15 years. Whereas in the past, experimental resources such as wind tunnels were used primarily for design development and refinement, in the future they may increasingly be used for the validation of computational design tools.

Notwithstanding the foregoing, there will continue to be a requirement for national facilities including wind tunnels, engine test facilities, flight test resources, and specialized resources including icing tunnels and rig test facilities for some time to come.

Experimental design and performance validation technology investment will be required in the following areas to support the aerospace industry in Canada:

- *Data Capture and Analysis Automation*: Automated methods for intelligent data capture and analysis will be required to reduce large facility run times and meet the challenges of design tool validation. This will require investment both in sensors and in computational tools;
- *Experimental Code Development*: Increased data capture rates and fidelity will be required and will necessitate the development of specific codes for experimental design and performance validation. Facilities and infrastructure will have to be maintained or enhanced to achieve these goals; and
- *Infrastructure Support*: The maintenance of critical national facilities will have to be supported in concert with other government departments and industry. The objective will not necessarily be to create new facilities but rather to improve the functionality of existing resources to meet the needs of new technology developments.

Aeropropulsion Performance Assessment

Test cells utilized for Canadian aero-engine Programmes, and also those developed for sale, have typically been sea-level static facilities offering little or no altitude, forward flight velocity or temperature pressure simulation. Some limited flying test bed capability exists in Canada for the testing of engines.

That being said, the National Research Council has participated in numerous international projects in the process ensuring that a world leading test cell capability exists both for engine qualification testing, performance testing and for the development of performance assessment techniques.

Engine test cells take a number of forms. Sea level test facilities are used for Engine Qualification Testing that involves the monitoring of a relatively small number of parameters over long periods where in-service usage is evaluated in a time compressed manner. Qualification testing also involves the ingestion of ice or water to ensure that unacceptable engine degradation does not occur in those instances. The NRC Institute for Aerospace Research has developed world recognized icing testing competencies and icing test facilities that are used by Canadian and off-shore engine manufacturers for qualification testing.

Altitude test cells are used to qualify engines over a full flight envelope as opposed to the endurance type testing previously described.

The National Research Council in colLabouration with Pratt and Whitney Canada have developed and operated one small altitude test cell at NRC for some time. An initiative that began in 2000 will see the development and commissioning of a somewhat larger and more capable altitude facility, again as a colLabourative effort between NRC and P&WC.

Test cells can also be used for the analysis of problems or validation of problem resolution. In these cases the test cells often require enhanced instrumentation suites and a

much more careful design to ensure that performance parameters are correctly measured. World interest in advanced test cell technologies has been directed at those required to support hypersonic vehicles for military uses or for space launch vehicles.

This type of test cell is very resource intensive and highly specialized and will likely be of little interest or utility to any but a limited number of Canadian firms. The Short Take Off and Vertical Landing (STOVL) version of the F35 Joint Strike Fighter has recently posed new challenges in the world of aeropropulsion testing. For this testing, in-flow preparation, exhaust treatment, fan drive systems, and 6 axis thrust measurement in the vertical axis will all pose significant new challenges to the performance assessment community.

ADVANCED CONCEPTS OF DESIGN

Analysis and Design Integration

Advanced aerodynamics profile development in Canada will be primarily directed at wing design for subsonic aircraft carrying less than 120 passengers. The objective of work done on advanced aerodynamic profiles will be to increase efficiency and cruise speeds through reduced drag while improving structural and control characteristics. Wing profile, control surface effectiveness, airframe and engine interface effects with the wing and wing tip designs are areas of research and development interest. Also, developments improving wing-flap high lift performance are important areas for minimizing wing size required and hence costs.

Laminar flow control is a term that deserves discussion. Airflow over wings begins as a laminar or ordered flowfield and will transition to a higher drag producing turbulent flow based on flow characteristics such as speed and wing influences including wing shape, surface roughness. It has been estimated that if laminar flow could be maintained on the wings of a large aircraft, fuel savings of up to 25% could be achieved.

Wing and flight characteristics of small aircraft are such that laminar flow can be relatively easily maintained over much of the flight envelope. A variety of methods can be used to increase laminar flow regions on aircraft of larger size and having higher Reynolds numbers and sweep angles.

Computational fluid dynamics will be the most important technology relevant to the development of advanced aerodynamic profiles. A number of areas require R&D activity and support for aircraft design particular to Canadian aerospace interests. Large-scaleCFD code refinement and validation is one area requiring work to improve accuracy and reduce computational times for MDO by more rapid design convergence. These CFD codes will also require validation in Labouratories and in wind tunnels.

All-Electric Aircraft Concept Development

The all-electric aircraft will utilize electronic actuators to replace equivalent hydraulic system components. The intent is to save weight and increase reliability. For example, electrical generators would provide power to electric actuators for flight control surface movement rather than equivalent hydraulic powered components. Electric power

cables are lighter and less prone to damage or service induced degradation such as fitting vibration that results in leakage in hydraulic systems. Alternate power supply redundancy is an additional advantage of this concept. Challenges associated with this type of technology insertion would be related to electromagnetic interference (EMI), and rapid load fluctuations imposed on the power generation engines.

Fly-by-Light Concept Development

Fly-by-Light (FBL) technology involves the replacement of electronic data transmission, mechanical control linkages, and electronic sensors with optical components and subsystems. Benefits include lower initial acquisition and life cycle costs, reduced weight, and increased aircraft performance and reliability.

Fibre-optic cables are essentially immune to electromagnetic interference and therefore not affected by fields generated by other lines or electrical devices in close proximity, nor are they affected by lightning strikes. For flight controls, hydraulic or electric actuators are still employed but receive their command inputs via fibre-optic cables. Weight reductions are significant as the fibre-optic cables need only be protected from physical damage, whereas electric cables must be insulated and shielded increasing weight significantly. Also with a FBL connection multiple routes can be readily provided that are well separated to provide control redundancy.

There are a number of enabling technologies that must be developed in order to enable photonics technology insertion. Fibre-optic connectors for in-line and end connections must

be developed that are durable and insensitive to in-service maintenance activities. Fibre-optic sensors development will also be necessary to allow the achievement of the full range of benefits that can be obtained in fly-by-light aircraft. This technology is usually associated with smart structures concepts such as smart skins where fibre-optic cabling can be readily embedded in a composite lay-up to achieve dispersed damage, stress, temperature or vibration sensing capability.

Detection Management and Control Systems

Regional airliners and helicopters operating in lower level airspace are increasingly exposed to hazardous icing conditions. This has increased the need for technologies for proactive and reactive ice detection and protection. Reactive technologies are those related to the detection of runback icing and attempt to monitor real-time or infer likely aerodynamic performance degradation.

Proactive systems forecast the potential for icing conditions and provide on-board avoidance advisory information. Reactive systems provide reasonable protection of the aircraft within the regulated flight envelope but are essentially go/no-go decision aids. Aircraft on Search and Rescue Missions and most civil transport aircraft often do not have the option of avoiding hazardous icing conditions and should have pro-active pilot advisors and ice removal systems.

Reactive ice detection devices include: embedded sensors that are mounted on the wing surface in a critical location and monitor ice build-up; and aerodynamic performance sensors that typically monitor pressure within the boundary layer of the wing to determine lift performance degradation.

Proactive systems require the remote measurement of Liquid Water Content (LWC), Outside Air Temperature (OAT) and Mean Volume Diameter (MVD) of the liquid water. Knowledge of these three parameters is required to predict hazardous icing conditions. Additional R&D work on MVD measurement is required.

Ice control and removal systems may use heated air from the engines or electrical heat elements to remove ice from airfoil surfaces. Coatings that are termed"iceophobic" may also be applied to minimize ice build-up. CFD tools are needed to Analyse ice-buildup characteristics, assess aerodynamic degradation, and improve ice removal air supply performance. This technology area is of particular interest because of the types of aircraft produced in Canada and because of climatic conditions.

Design Techniques

A previously stated objective for noise reduction is in the order of 6 EPNdB (Effective Perceived Noise in dB). This objective can be achieved through the utilization of larger by-pass ratio fans, innovative design concepts for turbo fans and sound conscious designs in the combustor and exhaust nozzles/liners. Generally speaking, noise improvements and fuel efficiency must be improved to meet future regulatory requirements without sacrifice of overall engine efficiency. Of special interest will be advanced ducted propulsors (ADF) that offer both noise attenuation and increased efficiency potential. This technology area will be heavily dependent on computational design techniques and multidisciplinary design optimization.

The reduction in aircraft emissions is also a regulated requirement. While small aircraft engines contribute an insignificant amount of pollution they are still the targets of increased environmental scrutiny. Regulatory requirements are targeted at Nitrous Oxides (NOx), Carbon Monoxide (CO) and visible particulate emissions. CFD analysis techniques specific to combustion processes will be the major tool used to lower aeropropulsion emissions.

8

Software Configuration Management

In software engineering, software configuration management (SCM) is the task of tracking and controlling changes in the software. Configuration management practices include revision control and the establishment of baselines. SCM concerns itself with answering the question "Somebody did something, how can one reproduce it?" Often the problem involves not reproducing "it" identically, but with controlled, incremental changes. Answering the question thus becomes a matter of comparing different results and of analysing their differences. Traditional configuration management typically focused on controlled creation of relatively simple products. Now, implementers of SCM face the challenge of dealing with relatively minor increments under their own control, in the context of the complex system being developed.

TERMINOLOGY

The history and terminology of SCM (which often varies) has given rise to controversy. Roger Pressman, in his book *Software Engineering: A Practitioner's Approach*, states that SCM is a "set of activities designed to control change by identifying the work products that are likely to change, establishing relationships among them, defining' for managing different versions of these work products, controlling the changes imposed, and auditing and reporting on the changes made."

Source configuration management is a related practice often used to indicate that a variety of artifacts may be managed and versioned, including software code, hardware, documents, design models, and even the directory structure itself. Atria (later Rational Software, now a part of IBM), used "SCM" to mean "software configuration management". Gartner uses the term *software change and configuration management*.

PURPOSES

The goals of SCM are generally:

- Configuration identification - Identifying configurations, configuration items and baselines.
- Configuration control - Implementing a controlled change process. This is usually achieved by setting up a change control board whose primary function is to approve or reject all change requests that are sent against any baseline.

- Configuration status accounting - Recording and reporting all the necessary information on the status of the development process.
- Configuration auditing - Ensuring that configurations contain all their intended parts and are sound with respect to their specifying documents, including requirements, architectural specifications and user manuals.
- Build management - Managing the process and tools used for builds.
- Process management - Ensuring adherence to the organization's development process.
- Environment management - Managing the software and hardware that host the system.
- Teamwork - Facilitate team interactions related to the process.
- Defect tracking - Making sure every defect has traceability back to the source.

Software Documentation

Software documentation or source code documentation is written text that accompanies computer software. It either explains how it operates or how to use it, and may mean different things to people in different roles.

INVOLVEMENT OF PEOPLE IN SOFTWARE LIFE

Documentation is an important part of software engineering. Types of documentation include:

1. Requirements - Statements that identify attributes, capabilities, characteristics, or qualities of a system. This is the foundation for what shall be or has been implemented.
2. Architecture/Design - Overview of softwares. Includes relations to an environment and construction principles to be used in design of software components.
3. Technical - Documentation of code, algorithms, interfaces, and APIs.
4. End User - Manuals for the end-user, system administrators and support staff.
5. Marketing - How to market the product and analysis of the market demand.

REQUIREMENTS DOCUMENTATION

Requirements documentation is the description of what a particular software does or shall do. It is used throughout development to communicate what the software does or shall do. It is also used as an agreement or as the foundation for agreement on what the software shall do. Requirements are produced and consumed by everyone involved in the production of software: end users, customers, product managers, project managers, sales, marketing, software architects, usability engineers, interaction designers, developers, and testers, to name a few. Thus, requirements documentation has many different purposes.

Requirements come in a variety of styles, notations and formality. Requirements can be goal-like (e.g., *distributed work environment*), close to design (e.g., *builds can be started*

by right-clicking a configuration file and select the 'build' function), and anything in between. They can be specified as statements in natural language, as drawn figures, as detailed mathematical formulas, and as a combination of them all. The variation and complexity of requirements documentation makes it a proven challenge.

Requirements may be implicit and hard to uncover. It is difficult to know exactly how much and what kind of documentation is needed and how much can be left to the architecture and design documentation, and it is difficult to know how to document requirements considering the variety of people that shall read and use the documentation. Thus, requirements documentation is often incomplete (or non-existent). Without proper requirements documentation, software changes become more difficult—and therefore more error prone (decreased software quality) and time-consuming (expensive). The need for requirements documentation is typically related to the complexity of the product, the impact of the product, and the life expectancy of the software. If the software is very complex or developed by many people (e.g., mobile phone software), requirements can help to better communicate what to achieve. If the software is safety-critical and can have negative impact on human life (e.g., nuclear power systems, medical equipment), more formal requirements documentation is often required.

If the software is expected to live for only a month or two (e.g., very small mobile phone applications developed specifically for a certain campaign) very little requirements documentation may be needed. If the software is a first release that is later built upon, requirements documentation

is very helpful when managing the change of the software and verifying that nothing has been broken in the software when it is modified. Traditionally, requirements are specified in requirements documents (e.g. using word processing applications and spreadsheet applications). To manage the increased complexity and changing nature of requirements documentation (and software documentation in general), database-centric systems and special-purpose requirements management tools are advocated.

ARCHITECTURE/DESIGN DOCUMENTATION

Architecture documentation is a special breed of design document. In a way, architecture documents are third derivative from the code (design document being second derivative, and code documents being first). Very little in the architecture documents is specific to the code itself. These documents do not describe how to programme a particular routine, or even why that particular routine exists in the form that it does, but instead merely lays out the general requirements that would motivate the existence of such a routine. A good architecture document is short on details but thick on explanation. It may suggest approaches for lower level design, but leave the actual exploration trade studies to other documents. Another breed of design docs is the comparison document, or trade study.

This would often take the form of a *whitepaper*. It focuses on one specific aspect of the system and suggests alternate approaches. It could be at the user interface, code, design, or even architectural level. It will outline what the situation is, describe one or more alternatives, and enumerate the

pros and cons of each. A good trade study document is heavy on research, expresses its idea clearly (without relying heavily on obtuse jargon to dazzle the reader), and most importantly is impartial. It should honestly and clearly explain the costs of whatever solution it offers as best. The objective of a trade study is to devise the best solution, rather than to push a particular point of view. It is perfectly acceptable to state no conclusion, or to conclude that none of the alternatives are sufficiently better than the baseline to warrant a change. It should be approached as a scientific endeavor, not as a marketing technique.

A very important part of the design document in enterprise software development is the Database Design Document (DDD). It contains Conceptual, Logical, and Physical Design Elements. The DDD includes the formal information that the people who interact with the database need. The purpose of preparing it is to create a common source to be used by all players within the scene. The potential users are:

- Database Designer
- Database Developer
- Database Administrator
- Application Designer
- Application Developer

When talking about Relational Database Systems, the document should include following parts:

- Entity - Relationship Schema, including following information and their clear definitions:
  - Entity Sets and their attributes
  - Relationships and their attributes

  - Candidate keys for each entity set
  - Attribute and Tuple based constraints
- Relational Schema, including following information:
  - Tables, Attributes, and their properties
  - Views
  - Constraints such as primary keys, foreign keys,
  - Cardinality of referential constraints
  - Cascading Policy for referential constraints
  - Primary keys

It is very important to include all information that is to be used by all actors in the scene. It is also very important to update the documents as any change occurs in the database as well.

TECHNICAL DOCUMENTATION

This is what most programmers mean when using the term *software documentation*. When creating software, code alone is insufficient. There must be some text along with it to describe various aspects of its intended operation. It is important for the code documents to be thorough, but not so verbose that it becomes difficult to maintain them. Several How-to and overview documentation are found specific to the software application or software product being documented by API Writers. This documentation may be used by developers, testers and also the end customers or clients using this software application. Today, we see lot of high end applications in the field of power, energy, transportation, networks, aerospace, safety, security, industry automation and a variety of other domains. Technical documentation has become important within such

organizations as the basic and advanced level of information may change over a period of time with architecture changes. Hence, technical documentation has gained lot of importance in recent times, especially in the software field. Often, tools such as Doxygen, NDoc, javadoc, EiffelStudio, Sandcastle, ROBODoc, POD, TwinText, or Universal Report can be used to auto-generate the code documents—that is, they extract the comments and software contracts, where available, from the source code and create reference manuals in such forms as text or HTML files.

Code documents are often organized into a *reference guide* style, allowing a programmer to quickly look up an arbitrary function or class. The idea of auto-generating documentation is attractive to programmers for various reasons. For example, because it is extracted from the source code itself (for example, through comments), the programmer can write it while referring to the code, and use the same tools used to create the source code to make the documentation. This makes it much easier to keep the documentation up-to-date. Of course, a downside is that only programmers can edit this kind of documentation, and it depends on them to refresh the output (for example, by running a cron job to update the documents nightly).

Some would characterize this as a pro rather than a con. Donald Knuth has insisted on the fact that documentation can be a very difficult afterthought process and has advocated literate programming, writing at the same time and location as the source code and extracted by automatic means. Elucidative Programming is the result of practical applications of Literate Programming in real programming

contexts. The Elucidative paradigm proposes that source code and documentation be stored separately. This paradigm was inspired by the same experimental findings that produced Kelp. Often, software developers need to be able to create and access information that is not going to be part of the source file itself. Such annotations are usually part of several software development activities, such as code walks and porting, where third party source code is analysed in a functional way. Annotations can therefore help the developer during any stage of software development where a formal documentation system would hinder progress. Kelp stores annotations in separate files, linking the information to the source code dynamically.

USER DOCUMENTATION

Unlike code documents, user documents are usually far more diverse with respect to the source code of the programme, and instead simply describe how it is used. In the case of a software library, the code documents and user documents could be effectively equivalent and are worth conjoining, but for a general application this is not often true. Typically, the user documentation describes each feature of the programme, and assists the user in realizing these features.

A good user document can also go so far as to provide thorough troubleshooting assistance. It is very important for user documents to not be confusing, and for them to be up to date. User documents need not be organized in any particular way, but it is very important for them to have a thorough index. Consistency and simplicity are also very

valuable. User documentation is considered to constitute a contract specifying what the software will do. API Writers are very well accomplished towards writing good user documents as they would be well aware of the software architecture and programming techniques used. There are three broad ways in which user documentation can be organized.

1. Tutorial: A tutorial approach is considered the most useful for a new user, in which they are guided through each step of accomplishing particular tasks.
2. Thematic: A thematic approach, where chapters or sections concentrate on one particular area of interest, is of more general use to an intermediate user. Some authors prefer to convey their ideas through a knowledge based article to facilitating the user needs. This approach is usually practiced by a dynamic industry, such as Information technology, where the user population is largely correlated with the troubleshooting demands.
3. List or Reference: The final type of organizing principle is one in which commands or tasks are simply listed alphabetically or logically grouped, often via cross-referenced indexes. This latter approach is of greater use to advanced users who know exactly what sort of information they are looking for.

A common complaint among users regarding software documentation is that only one of these three approaches was taken to the near-exclusion of the other two. It is common to limit provided software documentation for personal computers to online help that give only reference

information on commands or menu items. The job of tutoring new users or helping more experienced users get the most out of a programme is left to private publishers, who are often given significant assistance by the software developer.

MARKETING DOCUMENTATION

For many applications it is necessary to have some promotional materials to encourage casual observers to spend more time learning about the product. This form of documentation has three purposes:-

1. To excite the potential user about the product and instill in them a desire for becoming more involved with it.
2. To inform them about what exactly the product does, so that their expectations are in line with what they will be receiving.
3. To explain the position of this product with respect to other alternatives.

One good marketing technique is to provide clear and memorable *catch phrases* that exemplify the point we wish to convey, and also emphasize the interoperability of the programme with anything else provided by the manufacturer.

Software Quality Assurance

Software quality assurance (SQA) consists of a means of monitoring the software engineering processes and methods used to ensure quality. The methods by which this is accomplished are many and varied, and may include ensuring conformance to one or more standards, such as ISO 9000 or a model such as CMMI. SQA encompasses the

entire software development process, which includes processes such as requirements definition, software design, coding, source code control, code reviews, change management, configuration management, testing, release management, and product integration.

SQA is organized into goals, commitments, abilities, activities, measurements, and verifications. The American Society for Quality offers a Certified Software Quality Engineer (CSQE) certification with exams held a minimum of twice a year.

Software Project Management

Software project management is the art and science of planning and leading software projects. It is a sub-discipline of project management in which software projects are planned, monitored and controlled.

The history of software project management is closely related to the history of software. Software was developed for dedicated purposes for dedicated machines until the concept of object-oriented programming began to become popular in the 1960's, making *repeatable solutions* possible for the software industry. Dedicated systems could be adapted to other uses thanks to component-based software engineering. Companies quickly understood the relative ease of use that software programming had over hardware circuitry, and the software industry grew very quickly in the 1970's and 1980's. To manage new development efforts, companies applied proven project management methods, but project schedules slipped during test runs, especially when confusion occurred in the gray zone between the user

specifications and the delivered software. To be able to avoid these problems, software project management methods focused on matching user requirements to delivered products, in a method known now as the waterfall model. Since then, analysis of software project management failures has shown that the following are the most common causes:

1. Unrealistic or unarticulated project goals
2. Inaccurate estimates of needed resources
3. Badly defined system requirements
4. Poor reporting of the project's status
5. Unmanaged risks
6. Poor communication among customers, developers, and users
7. Use of immature technology
8. Inability to handle the project's complexity
9. Sloppy development practices
10. Poor project management
11. Stakeholder politics
12. Commercial pressures

The first three items in the list above show the difficulties articulating the needs of the client in such a way that proper resources can deliver the proper project goals. Specific software project management tools are useful and often necessary, but the true art in software project management is applying the correct method and then using tools to support the method.

Without a method, tools are worthless. Since the 1960's, several proprietary software project management methods have been developed by software manufacturers for their own use, while computer consulting firms have also

developed similar methods for their clients. Today software project management methods are still evolving, but the current trend leads away from the waterfall model to a more cyclic project delivery model that imitates a Software release life cycle.

SOFTWARE DEVELOPMENT PROCESS

A software development process is concerned primarily with the production aspect of software development, as opposed to the technical aspect, such as software tools. These processes exist primarily for supporting the management of software development, and are generally skewed toward addressing business concerns. Many software development processes can be run in a similar way to general project management processes. Examples are:

- Risk management is the process of measuring or assessing risk and then developing strategies to manage the risk. In general, the strategies employed include transferring the risk to another party, avoiding the risk, reducing the negative effect of the risk, and accepting some or all of the consequences of a particular risk. Risk management in software project management begins with the business case for starting the project, which includes a cost-benefit analysis as well as a list of fallback options for project failure, called a contingency plan.
    - o A subset of risk management that is gaining more and more attention is “Opportunity Management”, which means the same thing, except that the

potential risk outcome will have a positive, rather than a negative impact. Though theoretically handled in the same way, using the term "opportunity" rather than the somewhat negative term "risk" helps to keep a team focussed on possible positive outcomes of any given risk register in their projects, such as spin-off projects, windfalls, and free extra resources.

- Requirements management is the process of identifying, eliciting, documenting, analyzing, tracing, prioritizing and agreeing on requirements and then controlling change and communicating to relevant stakeholders. New or altered computer system Requirements management, which includes Requirements analysis, is an important part of the software engineering process; whereby business analysts or software developers identify the needs or requirements of a client; having identified these requirements they are then in a position to design a solution.
- Change management is the process of identifying, documenting, analyzing, prioritizing and agreeing on changes to scope (project management) and then controlling changes and communicating to relevant stakeholders. Change impact analysis of new or altered scope, which includes Requirements analysis at the change level, is an important part of the software engineering process; whereby business analysts or software developers identify the altered needs or requirements of a client; having identified

these requirements they are then in a position to re-design or modify a solution. Theoretically, each change can impact the timeline and budget of a software project, and therefore by definition must include risk-benefit analysis before approval.

- Software configuration management is the process of identifying, and documenting the scope itself, which is the software product underway, including all sub-products and changes and enabling communication of these to relevant stakeholders. In general, the processes employed include version control, naming convention (programming), and software archival agreements.
- Release management is the process of identifying, documenting, prioritizing and agreeing on releases of software and then controlling the release schedule and communicating to relevant stakeholders. Most software projects have access to three software environments to which software can be released; Development, Test, and Production. In very large projects, where distributed teams need to integrate their work before release to users, there will often be more environments for testing, called unit testing, system testing, or integration testing, before release to User acceptance testing (UAT).
    - A subset of release management that is gaining more and more attention is Data Management, as obviously the users can only test based on data that they know, and "real" data is only in the software environment called "production". In

order to test their work, programmers must therefore also often create "dummy data" or "data stubs". Traditionally, older versions of a production system were once used for this purpose, but as companies rely more and more on outside contributors for software development, company data may not be released to development teams. In complex environments, datasets may be created that are then migrated across test environments according to a test release schedule, much like the overall software release schedule.

PROJECT PLANNING, MONITORING AND CONTROL

The purpose of project planning is to identify the scope of the project, estimate the work involved, and create a project schedule. Project planning begins with requirements that define the software to be developed. The project plan is then developed to describe the tasks that will lead to completion.

The purpose of project monitoring and control is to keep the team and management up to date on the project's progress. If the project deviates from the plan, then the project manager can take action to correct the problem. Project monitoring and control involves status meetings to gather status from the team. When changes need to be made, change control is used to keep the products up to date.

ISSUE

In computing, the term issue is a unit of work to accomplish an improvement in a system. An issue could be a bug, a requested feature, task, missing documentation, and so forth. The word "issue" is popularly misused in lieu of "problem." This usage is probably related. For example, OpenOffice.org used to call their modified version of BugZilla IssueZilla. As of September 2010, they call their system Issue Tracker. Problems occur from time to time and fixing them in a timely fashion is essential to achieve correctness of a system and avoid delayed deliveries of products.

SEVERITY LEVELS

Issues are often categorized in terms of severity levels. Different companies have different definitions of severities, but some of the most common ones are:

- Critical
- High - The bug or issue affects a crucial part of a system, and must be fixed in order for it to resume normal operation.
- Medium - The bug or issue affects a minor part of a system, but has some impact on its operation. This severity level is assigned when a non-central requirement of a system is affected.
- Low - The bug or issue affects a minor part of a system, and has very little impact on its operation. This severity level is assigned when a non-central requirement of a system (and with lower importance) is affected.

- Cosmetic - The system works correctly, but the appearance does not match the expected one. For example: wrong colors, too much or too little spacing between contents, incorrect font sizes, typos, etc. This is the lowest priority issue.

In many software companies, issues are often investigated by Quality Assurance Analysts when they verify a system for correctness, and then assigned to the developer(s) that are responsible for resolving them. They can also be assigned by system users during the User Acceptance Testing (UAT) phase.

Issues are commonly communicated using Issue or Defect Tracking Systems. In some other cases, emails or instant messengers are used.

PHILOSOPHY

As a subdiscipline of project management, some regard the management of software development akin to the management of manufacturing, which can be performed by someone with management skills, but no programming skills. John C. Reynolds rebuts this view, and argues that software development is entirely design work, and compares a manager who cannot programme to the managing editor of a newspaper who cannot write.

User experience design

User eXperience Design (UXD) is a subset of the field of experience design that pertains to the creation of the architecture and interaction models that affect user

experience of a device or system. The scope of the field is directed at affecting "all aspects of the user's interaction with the product: how it is perceived, learned, and used."

THE DESIGNERS

This field has its roots in human factors and ergonomics, a field that since the late 1940s has been focusing on the interaction between human users, machines and the contextual environments to design systems that address the user's experience. The term also has a more recent connection to user-centered design principles and also incorporates elements from similar user-centered design fields.

As with the fields mentioned above, user experience design is a highly multi-disciplinary field, incorporating aspects of psychology, anthropology, sociology, computer science, graphic design, industrial design and cognitive science. Depending on the purpose of the product, UX may also involve content design disciplines such as communication design, instructional design, or game design. The subject matter of the content may also warrant collaboration with a Subject Matter Expert (SME) on planning the UX from various backgrounds in business, government, or private groups.

THE DESIGN

User experience design incorporates most or all of the above disciplines to positively impact the overall experience a person has with a particular interactive system, and its

provider. User experience design most frequently defines a sequence of interactions between a user (individual person) and a system, virtual or physical, designed to meet or support user needs and goals, primarily, while also satisfying systems requirements and organizational objectives.

Typical outputs include:

- Site Audit (usability study of existing assets)
- Flows and Navigation Maps
- User stories or Scenarios
- Persona (Fictitious users to act out the scenarios)
- Site Maps and Content Inventory
- Wireframes (screen blueprints or storyboards)
- Prototypes (For interactive or in-the-mind simulation)
- Written specifications (describing the behavior or design)
- Graphic mockups (Precise visual of the expected end result)

BENEFITS

User experience design is integrated into software development and other forms of application development to inform feature requirements and interaction plans based upon the user's goals. New introduction of software must keep in mind the dynamic pace of technology advancement and the need for change. The benefits associated with integration of these design principles include:

- Avoiding unnecessary product features
- Simplifying design documentation and customer-facing technical publications

- Improving the usability of the system and therefore its acceptance by customers
- Expediting design and development through detailed and properly conceived guidelines
- Incorporating business and marketing goals while catering to the user.