Computer Graphics Design and System

Jay Baldwin

COMPUTER GRAPHICS DESIGN AND SYSTEM

COMPUTER GRAPHICS DESIGN AND SYSTEM

Jay Baldwin

BIBLIOTEX
Digital Library

Computer Graphics Design and System
by Jay Baldwin

www.bibliotex.com

Ebook ISBN: 9781984664020

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

Contents

1

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 subsequently 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. Aerodynamics, 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.

2

Applications of Computer Graphics

Computers have become a powerful tool for the rapid and economical production of pictures. Advances in computer technology have made interactive computer graphics a practical tool. Today, computer graphics is used in the areas as science, engineering, medicine, business, industry, government, art, entertainment, advertising, education, and training.

COMPUTER AIDED DESIGN

A major use of computer graphics is in design processes, particularly for engineering and architectural systems. For some design applications; objects are first displayed in a wireframe outline form that shows the overall sham and internal features of objects.

Software packages for CAD applications typically provide the designer with a multi-window environment. Each

window can show enlarged sections or different views of objects. Standard shapes for electrical, electronic, and logic circuits are often supplied by the design package. The connections between the components have been mad automatically.

- Animations are often used in CAD applications.
- Real-time animations using wire frame displays are useful for testing performance of a vehicle.
- Wire frame models allow the designer to see the interior parts of the vehicle during motion.
- When object designs are complete, realistic lighting models and surface rendering are applied.
- Manufacturing process of object can also be controlled through CAD.
- Interactive graphics methods are used to layout the buildings.
- Three-dimensional interior layouts and lighting also provided.
- With virtual-reality systems, the designers can go for a simulated walk inside the building.

Presentation Graphics

- It is used to produce illustrations for reports or to generate slide for with projections.
- Examples of presentation graphics are bar charts, line graphs, surface graphs, pie charts and displays showing relationships between parameters.
- 3-D graphics can provide more attraction to the presentation.

Computer Art

- Computer graphics methods are widely used in both fine are and commercial art applications.
- The artist uses a combination of 3D modelling packages, texture mapping, drawing programmes and CAD software.
- Pen plotter with specially designed software can create "automatic art".
- "Mathematical Art" can be produced using mathematical functions, fractal procedures.
- These methods are also applied in commercial art.
- Photorealistic techniques are used to render images of a product.
- Animations are also used frequently in advertising, and television commercials are produced frame by frame. Film animations require 24 frames for each second in the animation sequence.
- A common graphics method employed in many commercials is morphing, where one object is transformed into another.

Entertainment

- CG methods are now commonly used in making motion pictures, music videos and television shows.
- Many TV series regularly employ computer graphics method.
- Graphics objects can be combined with a live action.

Education and Training

- Computer-generated models of physical, financial and economic systems are often used as educational aids.
- For some training applications, special systems are designed.
  Eg. Training of ship captains, aircraft pilots etc.
- Some simulators have no video screens, but most simulators provide graphics screen for visual operation. Some of them provide only the control panel.

Visualization

- The numerical and scientific data are converted to a visual form for analysis and to study the behaviour called visualization.
- Producing graphical representation for scientific data sets are calls scientific visualization.
- And business visualization is used to represent the data sets related to commerce and industry.
- The visualization can be either 2D or 3D.

Image Processing

- Computer graphics is used to create a picture.
- Image processing applies techniques to modify or interpret existing pictures.
- To apply image processing methods, the image must be digitized first.
- Medical applications also make extensive use of image processing techniques for picture enhancements, simulations of operations, etc.

Graphical User Interface

- Nowadays software packages provide graphics user interface (GUI) for the user to work easily.
- A major component in GUI is a window.
- Multiple windows can be opened at a time.
- To activate any one of the window, the user needs just to check on that window.
- Menus and icons are used for fast selection of processing operations.
- Icons are used as shortcut to perform functions. The advantages of icons are which takes less screen space.
- And some other interfaces like text box, buttons, and list are also used.

NEW TECHNIQUE IMPROVES RENDERING OF SMOKE, DUST AND PARTICIPATING MEDIA

Computer graphic artists often struggle to render smoke and dust in a way that makes a scene look realistic, but researchers at Disney Research, Zürich, Karlsruhe Technical Institute in Germany, and the University of Montreal in Canada have developed a new and efficient way to simulate how light is absorbed and scattered in such scenes.

LED LIGHTING

"Our technique could be used to simulate anything from vast cloudscapes, to everyday 'solid' objects such as a glass

of orange juice, a piece of fruit or virtually any organic substance," said Dr. Wojciech Jarosz of Disney Research Zürich, who led the research team.

The team's new virtual ray lights technique will be presented Aug. 7 in the "Light Rays" session at this year's international SIGGRAPH conference, which focuses on the latest and greatest advances in Computer Graphics and Interactive Techniques, at the Los Angeles Convention Center.

Beneath the surface, this new approach leverages another Disney Research technology — photon beams, also developed by a team lead by Jarosz — which challenged the traditional views on how to simulate light in scenes with smoke, dust or other "participating media." Normally, realistic rendering techniques simulate light using a set of particles (virtual "photons") that bounce off of walls and objects, depositing tiny bits of light-energy along their trip. It's this light-energy that's collected to form the final simulated image. But Disney researchers have found that it's much more efficient to use long and thin beams of light, instead of tiny photon particles, as a building block for generating images.

This photon beams approach was presented at last year's SIGGRAPH conference in Vancouver and was also used to create magical wispy effects for Disney's Tangled. Since then Disney researchers have been hard at work to make the technique even better. This latest work in the photon beams family looks at how photon beams contribute to so-called secondary lighting events in participating media:

namely, when light enters a smoky or dusty room, the light particles actually hit and bounce off of the smoke or dust. This special game of ping-pong happens in reality at the speed of light, and the newly proposed technique investigates how entire beams of light are ping-ponged around the dust and smoke clouds in a room, or the pulp inside a glass of orange juice. With this new technique computer graphics experts can simulate more realistic participating media effects, which occur more commonly than one would expect in the real-world: participating media effects account for the way we observe clouds, the appearance of fruit juices and milk, the haze in smoggy cities, and even the subtle dimming of distant objects on an otherwise clear day. The amount to which these effects can contribute to a final rendered image varies from tiny shifts to major rifts, but even the most subtle of changes to the appearance of a virtual scene, when executed correctly, can help convince otherwise oblivious audience members that what they are seeing is "real."

The virtual ray lights technique was also designed to be flexible, and the researchers hope it will find its way into many different areas of the computer animation and special effects industry. Virtual ray lights were designed to be progressive, meaning that they can very quickly generate a preview-quality result while converging to a final result as time goes by.

On the one hand, this allows skilled technical artists at feature-film studios to quickly get feedback about their lighting setups and designs, as opposed to making

changes and grabbing a coffee before being able to tell if their design changes are helpful or not.

This rapid-feedback property reduces iteration time, allowing artists to focus on the end-goal instead of wrestling with the lighting tool. Another benefit of the progressive nature of the new technique is that users can choose between quality and performance, which is ideal for game developers who don't care so much about being 100 percent realistic but instead want their scenes to "just look great." Finally, the virtual ray lights project is a shining example of Disney's commitment to bringing together the brightest experts from across industrial and academic research settings in order to push the limits of the state-of-the-art.

This project was undertaken by four researchers in three countries across two continents: Jan Novák, an intern at Disney Research in Zürich and full-time PhD student at Karlsruhe Institute of Technology, worked alongside Wojciech Jarosz, a research scientist and head of the Rendering Group at Disney Research Zürich, Derek Nowrouzezahrai, a Disney Research post-doc and now assistant professor at the University of Montreal, and Carsten Dachsbacher, the head of the Computer Graphics Group at Karlsruhe Institute of Technology.

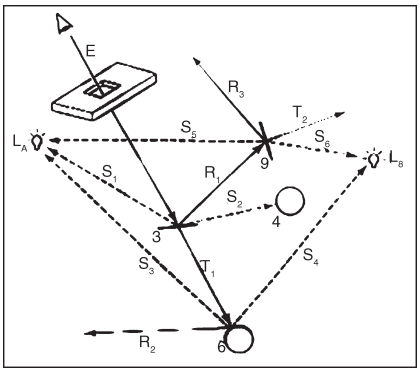
RAY TRACING

Ray Tracing is a global illumination based rendering method. It traces rays of light from the eye back through the image plane into the scene. Then the rays are tested

against all objects in the scene to determine if they intersect any objects. If the ray misses all objects, then that pixel is shaded the background colour. Ray tracing handles shadows, multiple specular reflections, and texture mapping in a very easy straight-forward manner. Note that ray tracing, like scan-line graphics, is a point sampling algorithm. We sample a continuous image in world coordinates by shooting one or more rays through each pixel. Like all point sampling algorithms, this leads to the potential problem of aliasing, which is manifested in computer graphics by jagged edges or other nasty visual artifacts. In ray tracing, a ray of light is traced in a backwards direction. That is, we start from the eye or camera and trace the ray through a pixel in the image plane into the scene and determine what it hits. The pixel is then set to the colour values returned by the ray.

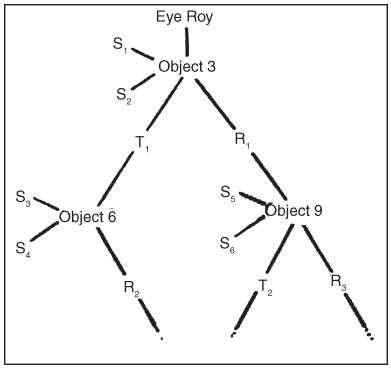
SIMPLE GLOBAL ILLUMINATION MODEL FOR RAY TRACING

A primary ray is shot through each pixel and tested for intersection against all objects in the scene. If there is an intersection with an object then several other rays are generated. Shadow rays are sent towards all light sources to determine if any objects occlude the intersection spot. The shadow rays are labelled S_i and are sent towards the two light sources L_A and L_B. If the surface is reflective then a reflected ray, R_i, is generated. If the surface is not opaque, then a transmitted ray, T_i, is generated. Each of the secondary rays is tested against all the objects in the scene.



The reflective and/or transmitted rays are continually generated until the ray leaves the scene without hitting any object or a preset recursion level has been reached. This then generates a ray tree, as shown below.

The appropriate local illumination model is applied at each level and the resultant intensity is passed up through the tree, until the primary ray is reached. Thus we can modify the local illumination model by (at each tree node).



I = Ilocal + Kr * R + Kt * T where R is the intensity of light from the reflected ray and T is the intensity of light from the transmitted ray. Kr and Kt are the reflection and transmission coefficients.

For a very specular surface, such as plastic, we sometimes do not compute a local intensity, Ilocal, but only use the reflected/transmitted intensity values.

RAY OBJECT INTERSECTIONS

The general idea behind ray-object intersections is to put the mathematical equation for the ray into the equation for the object and determine if there is a real solution. If there is a real solution then there is an intersection (hit) and we must return the closest point of intersection and the normal (N) at the intersection point. For a shadow ray we must return whether any ray-object intersection is closer than the ray-light intersection. For a ray tested against a boundary volume, we just return a simple hit or no hit. For texture mapping we need the intersection point relative to some reference frame for the surface.

We define a ray as:

R0 = [x0, y0, z0] - origin of ray

Rd = [xd, yd, zd] - direction of ray

then define a set of points on the ray:

R(t) = R0 + Rd * t with t > 0.0

If Rd is normalized, then t equals the distance of the ray from origin in World Coordinates, else it is just a multiple of Rd, so we want to normalize Rd. Note that many of the intersection computations require the solution of the quadratic equation.

PRACTICAL CONSIDERATIONS IN WRITING A RAY TRACER

Process: For each pixel a primary ray will be generated and then tested against all objects in the scene.

Create Model

The first step is to create the model of the image.

One should not hardcode objects into the programme, but instead use an input file. Here is a sample Input file:

Generate Primary Rays and Test for Object-Ray Intersections

For each pixel we must generate a primary ray and test for intersection with all of the objects in the scene. If there is more than one ray-object intersection then we must choose the closest intersection (the smallest positive value of t). To ensure that there are no objects intersected in front of the image plane (this is called near plane clipping), we keep the distance of the primary ray to the screen and test all intersections against this distance. If the t value is less than this distance, then we ignore the object.

A sample calculation of forming a ray and testing it for intersection with a sphere. If there is an intersection then we must compute the shadow rays and the reflection rays.

Shadow Ray

The shadow ray is a ray from the point of intersection to the light source. Its purpose is to determine if the intersection point is in the shadow of a particular light. There should be one shadow ray for each light source. The origin of the shadow ray is the intersection point and the direction vector is the normalized vector between the intersection point and the position of the light source. Note that this is the same as the light vector (L) that is used to compute the local illumination.

Compute the Local Illumination at each point, carry it back to the next level of the ray tree so that the intensity

I = Ilocal + Kr * R + Kt * T. Note that Kr can be taken as the same as Ks. For each colour (R, G, B) I is in the range 0.0 <= I <= 1.0. This must be converted to an integer value of 0 <= I <= 255. The result is then written to the output file.

Output File

The output file will consist of three intensity values (Red, Green, and Blue) for each pixel. For a system with a 24-bit framebuffer this file could be directly displayed. However, for a system with an 8-bit framebuffer, the 24-bit image must be converted to an 8 bit image, which can then be displayed. A suggested format for the output file is the Microsoft Windows 24-bit BMP image file format.

ACCELERATING RAY TRACING

Ray Tracing is so time-consuming because of the intersection calculations. Since each ray must be checked against all objects, for a naive raytracer (with no speedup techniques) the time is proportional to the number of rays X the number of objects in the scene. Each intersection requires from a few (5-7) to many (15-20) floating point (fp) operations. Thus for a scene with 100 objects and computed with a spatial resolution of 512 × 512, assuming 10 fp operations per object test there are about 250,000 × 100 × 10 = 250,000,000 fps. This is just for the primary rays (from the eye through the image plane) with no anti-aliasing. Clearly there are computational problems with this.

There are several approaches to speeding up computations:

- Use faster machines
- Use specialized hardware, especially parallel processors.
- Speed up computations by using more efficient algorithms
- Reduce the number of ray - object computations.

REDUCING RAY-OBJECT INTERSECTIONS

Adaptive Depth Control

This means that we stop generating reflected/transmitted rays when the computed intensity becomes less than a certain threshold. You must always set a certain maximum depth or else the programme would generate an infinite number of rays. But it is not always necessary to go to the maximum depth if the surfaces are not highly reflective. To test for this the ray tracer must compute and keep the product of the global and reflection coefficients as the rays are traced.

Example: let Kr = 0.5 for a set of surfaces. Then from the first surface the maximum contribution is 0.5, for the reflection from the second: 0.5 * 0.5 = 0.25, the third: 0.25 * 0.5 = 0.125, the fourth: 0.125 * 0.5 = 0.0625, the fifth: 0.0625 * 0.5 = 0.03125, etc. In addition we might implement a distance attenuation factor such as 1/D2, which would also decrease the intensity contribution.

For a transmitted ray we could do something similar but in that case the distance traveled through the object would cause even faster intensity decrease. As an example of this, Hall & Greenbergfound that even for a very reflective scene, using this with a maximum depth of 15 resulted in an average ray tree depth of 1.7.

Bounding Volumes

We enclose groups of objects in sets of hierarchical bounding volumes and first test for intersection with the bounding volume, and then only if there is an intersection, against the objects enclosed by the volume.

Bounding volumes should be easy to test for intersection, for example a sphere or box (slab). The best bounding volume will be determined by the shape of the underlying object or objects.

For example, if the objects are long and thin then a sphere will enclose mainly empty space and a box is much better. Boxes are also easier for hierarchical bounding volumes.

Note that using a herarchical system like this (assuming it is done carefully) changes the intersection computational time from a linear dependence on the number of objects to something between linear and a logorithmic dependence. This is because, for a perfect case, each interesction test would divide the possibilities by two, and we would have a binary tree type structure. Spatial subdivision methods, discussed below, try to achieve this.

Kay & Kajiya give a list of properties for hierarchical bounding volumes:

- Subtrees should contain objects that are near each other and the further down the tree the closer should be the objects.
- The volume of each node should be minimal.
- The sum of the volumes of all bounding volumes should be minimal.

- Greater attention should be placed on the nodes near the root since pruning a branch near the root will remove more potential objects than one farther down the tree.
- The time spent constructing the hierarchy should be much less than the time saved by using it.

First-Hit Speedup

On adaptive depth control, by using that technique the average depth of the ray tree may be less than two. This means that a large percentage of the work is performed in finding the first intersection. Weghorst has suggested using a modified Z-buffer algorithm to determine the first hit. The scene would be pre-processed, with the resultant z-buffer storing pointers to the objects intersected. Then the ray tracing would proceed from that point.

He showed that incorporating the above three techniques (adaptive depth control, hierarchical bounding volumes, and first-hit speedup) approximately halved the intersection computational time for complex scenes. Note that he use spheres as the bounding volumes. In general the computational improvement was inversely dependent on scene complexity.

Weghorst showed that incorporating the above three techniques (adaptive depth control, hierarchical bounding volumes, and first-hit speedup) approximately halved the intersection computational time for complex scenes. Note that he used spheres as the bounding volumes. In general the computational improvement was inversely dependent on scene complexity.

DETAILED DESCRIPTION OF RAY TRACING COMPUTER ALGORITHM AND ITS GENESIS

What Happens in Nature

In nature, a light source emits a ray of light which travels, eventually, to a surface that interrupts its progress. One can think of this "ray" as a stream of photons traveling along the same path. In a perfect vacuum this ray will be a straight line (ignoring relativistic effects). In reality, any combination of four things might happen with this light ray: absorption, reflection, refraction and fluorescence. A surface may absorb part of the light ray, resulting in a loss of intensity of the reflected and/or refracted light. It might also reflect all or part of the light ray, in one or more directions.

If the surface has any transparent or translucent properties, it refracts a portion of the light beam into itself in a different direction while absorbing some (or all) of the spectrum (and possibly altering the colour). Less commonly, a surface may absorb some portion of the light and fluorescently re-emit the light at a longer wavelength colour in a random direction, though this is rare enough that it can be discounted from most rendering applications. Between absorption, reflection, refraction and fluorescence, all of the incoming light must be accounted for, and no more. A surface cannot, for instance, reflect 66% of an incoming light ray, and refract 50%, since the two would add up to be 116%. From here, the reflected and/or refracted rays may strike other surfaces, where their absorptive, refractive, reflective and fluorescent properties again affect the progress of the incoming rays. Some of

these rays travel in such a way that they hit our eye, causing us to see the scene and so contribute to the final rendered image.

Ray Casting Algorithm

The first ray casting (versus ray tracing) algorithm used for rendering was presented by Arthur Appel in 1968. The idea behind ray casting is to shoot rays from the eye, one per pixel, and find the closest object blocking the path of that ray – think of an image as a screen-door, with each square in the screen being a pixel. This is then the object the eye normally sees through that pixel. Using the material properties and the effect of the lights in the scene, this algorithm can determine the shading of this object. The simplifying assumption is made that if a surface faces a light, the light will reach that surface and not be blocked or in shadow.

The shading of the surface is computed using traditional 3D computer graphics shading models. One important advantage ray casting offered over older scanline algorithms is its ability to easily deal with non-planar surfaces and solids, such as cones and spheres. If a mathematical surface can be intersected by a ray, it can be rendered using ray casting. Elaborate objects can be created by using solid modelling techniques and easily rendered.

Ray Tracing Algorithm

The next important research breakthrough came from Turner Whitted in 1979. Previous algorithms cast rays from the eye into the scene until they hit an object, but the rays were traced no further.

Fig. Ray Tracing can Achieve a Very High Degree of Visual Realism.

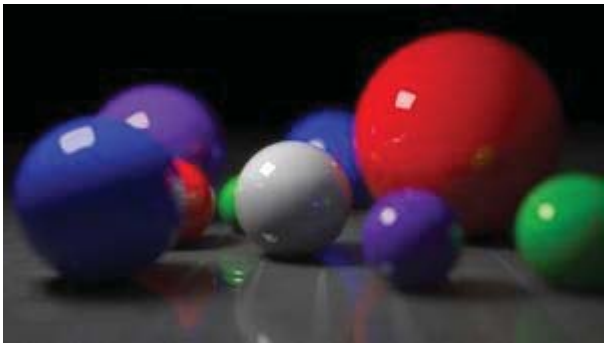


Fig. In Addition to the High Degree of Realism, Ray Tracing can Simulate the Effects of a Camera due to Depth of Field and Aperture Shape.

Fig. The Number of Reflections a "Ray" can Take and how it is Affected each Time it Encounters a Surface is all Controlled via Software Settings during Ray Tracing. Here, Each Ray was Allowed to Reflect up to 16 Times. Multiple "Reflections of Reflections" can thus be Seen.

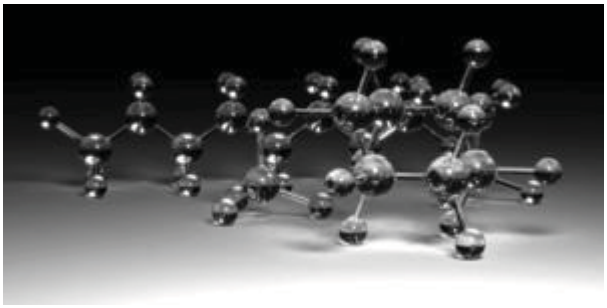


Fig. The Number of Refractions a "Ray" can take and how it is Affected each Time it Encounters a Surface is all Controlled via Software Settings during Ray Tracing.

Whitted continued the process. When a ray hits a surface, it could generate up to three new types of rays: reflection, refraction, and shadow. A reflected ray continues on in the mirror-reflection direction from a shiny surface.

It is then intersected with objects in the scene; the closest object it intersects is what will be seen in the reflection. Refraction rays traveling through transparent material work similarly, with the addition that a refractive ray could be entering or exiting a material. To further avoid tracing all rays in a scene, a shadow ray is used to test if a surface is visible to a light.

A ray hits a surface at some point. If the surface at this point faces a light, a ray (to the computer, a line segment) is traced between this intersection point and the light. If any opaque object is found in between the surface and the light, the surface is in shadow and so the light does not contribute to its shade. This new layer of ray calculation added more realism to ray traced images.

Advantages over other Rendering Methods

Ray tracing's popularity stems from its basis in a realistic simulation of lighting over other rendering methods (such

as scanline rendering or ray casting). Effects such as reflections and shadows, which are difficult to simulate using other algorithms, are a natural result of the ray tracing algorithm. Relatively simple to implement yet yielding impressive visual results, ray tracing often represents a first foray into graphics programming. The computational independence of each ray makes ray tracing amenable to parallelization.

Disadvantages

A serious disadvantage of ray tracing is performance. Scanline algorithms and other algorithms use data coherence to share computations between pixels, while ray tracing normally starts the process anew, treating each eye ray separately. However, this separation offers other advantages, such as the ability to shoot more rays as needed to perform spatial anti-aliasing and improve image quality where needed. Although it does handle interreflection and optical effects such as refraction accurately, traditional ray tracing is also not necessarily photorealistic. True photorealism occurs when the rendering equation is closely approximated or fully implemented. Implementing the rendering equation gives true photorealism, as the equation describes every physical effect of light flow. However, this is usually infeasible given the computing resources required. The realism of all rendering methods, then, must be evaluated as an approximation to the equation, and in the case of ray tracing, it is not necessarily the most realistic. Other

methods, including photon mapping, are based upon ray tracing for certain parts of the algorithm, yet give far better results.

Reversed Direction of Traversal of Scene by the Rays

The process of shooting rays from the eye to the light source to render an image is sometimes called *backwards ray tracing*, since it is the opposite direction photons actually travel. However, there is confusion with this terminology. Early ray tracing was always done from the eye, and early researchers such as James Arvo used the term *backwards ray tracing* to mean shooting rays from the lights and gathering the results. Therefore it is clearer to distinguish *eye-based* versus *light-based* ray tracing.

While the direct illumination is generally best sampled using eye-based ray tracing, certain indirect effects can benefit from rays generated from the lights. Caustics are bright patterns caused by the focusing of light off a wide reflective region onto a narrow area of (near-)diffuse surface. An algorithm that casts rays directly from lights onto reflective objects, tracing their paths to the eye, will better sample this phenomenon. This integration of eye-based and light-based rays is often expressed as bidirectional path tracing, in which paths are traced from both the eye and lights, and the paths subsequently joined by a connecting ray after some length.

Photon mapping is another method that uses both light-based and eye-based ray tracing; in an initial pass,

energetic photons are traced along rays from the light source so as to compute an estimate of radiant flux as a function of 3-dimensional space (the eponymous photon map itself). In a subsequent pass, rays are traced from the eye into the scene to determine the visible surfaces, and the photon map is used to estimate the illumination at the visible surface points. The advantage of photon mapping versus bidirectional path tracing is the ability to achieve significant reuse of photons, reducing computation, at the cost of statistical bias.

An additional problem occurs when light must pass through a very narrow aperture to illuminate the scene (consider a darkened room, with a door slightly ajar leading to a brightly lit room), or a scene in which most points do not have direct line-of-sight to any light source (such as with ceiling-directed light fixtures or torchieres).

In such cases, only a very small subset of paths will transport energy; Metropolis light transport is a method which begins with a random search of the path space, and when energetic paths are found, reuses this information by exploring the nearby space of rays.

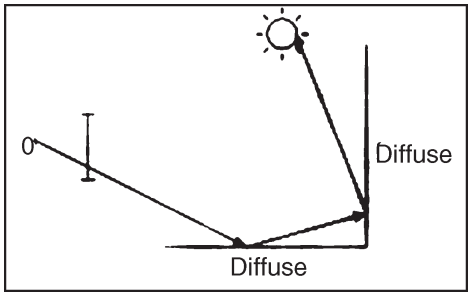


Fig. To the Right is an Image Showing a Simple Example of a Path of Rays Recursively Generated from the Camera (or Eye) to the Light Source using the above Algorithm. A Diffuse Surface Reflects Light in all Directions.

First, a ray is created at an eyepoint and traced through a pixel and into the scene, where it hits a diffuse surface. From that surface the algorithm recursively generates a reflection ray, which is traced through the scene, where it hits another diffuse surface.

Finally, another reflection ray is generated and traced through the scene, where it hits the light source and is absorbed.

The colour of the pixel now depends on the colours of the first and second diffuse surface and the colour of the light emitted from the light source. For example if the light source emitted white light and the two diffuse surfaces were blue, then the resulting colour of the pixel is blue.

GRAPHICAL I/O DEVICES

Computer graphics gives us added dimensions for communication between the user and the machine. Complex organizations and relationships can be conveyed clearly to the user. But communication should be a two-way process. It is desirable to allow the user to respond to this information. The most common form of computer is a string of characters printed on the page or on the surface of a CRT terminal. The corresponding form of input is also a stream of characters coming from a keyboard. So to perform such I/O operations, there is a need of I/O devices. The following are the I/O devices for graphic implementation.

INPUT DEVICES

Various hardware devices have been developed to enable the user to interact in the more natural manner. These devices can be separated into two classes. They are Locators and Selectors.

Locators: Locators are the devices which give position information. The computer receives from a Locater the coordinates for a point. Using a locator we can indicate a position on the screen. The different locators are as follows:

Thumbwheels: A pair of Thumbwheels such as is found on the Tektronix 4010 graphics terminal. These are two potentiometers mounted on the keyboard, which the user can adjust. One potentiometer is used for x direction and the other for the y direction. Analog-to-digital converters change the potentiometer setting into a digital value which the computer can read. The potentiometer settings may be read whenever desired. The two potentiometer readings together form the coordinates of a point.

To be useful, this scheme must also present user with information as to which point the thumbwheels are specifying. Some feedback mechanism is needed. This may be in the form of a special screen cursor, that is, a special marker placed on the screen at the point which is being indicated. It might also be done by a pair of cross hairs which cross at the indicated point. As a thumbwheel is turned, the marker or cross hair moves across the screen to show the set which position is being read.

Joystick: A Joystick has two potentiometers, just as a pair of thumbwheels. They have been attached to a single

lever. Moving the lever forward or back changes the setting on one potentiometer. Moving it left or right changes the setting on the other potentiometer. Thus with a joystick both x and y coordinate positions can be simultaneously altered by the motion of a single lever.

The potentiometer settings are processed in the same manner as they are for thumbwheels. Some joysticks may return to their zero position when released, whereas thumbwheels remain at their last position until changed joysticks are inexpensive and are quite common on displays where only rough positioning is needed.

Mouse: A Mouse is palm-sized box with a ball on the bottom connected to wheels for the x and y directions. These locator devices use switches attached to wheels instead of potentiometers. As the wheels are turned, the switches produce pulses which may be counted. The count indicates how much a wheel has rotated. As the mouse is pushed across a surface, the wheels turned, proving distance and direction information. This can then be used to alter the position of a cursor on the screen a mouse may also come with one or more buttons which may be sensed. There are also mice which use photocells rather than wheels and switches to sense position. Photocells in the bottom of the mouse sense the movement across the grid and produce pulses to report the motion.

Tablet: A Tablet composed of a flat surface and a pen like stylus or window like tablet cursor. The tablet is able to sense the position of the stylus or tablet cursor on the surface. A number of different physical principles have

been employed for the sensing of the stylus. Most do not require actual contact between the stylus and the tablet surface, so that a drawing or blueprint might be placed upon the surface and the stylus used to trace it. A feedback mechanism on the screen is not as necessary for a graphics tablet as it is for a joystick because the user can look at the tablet to see what position he is indicating. If tablet entries are to be coordinated with items already on the screen, then some form of feedback, such as a screen cursor, is useful.

Selector Device: Selector devices are used to select a particular graphical object. A selector may pick a particular item but provide no information about that item is located on the screen. The different selector devices are as follows.

Light Pen: A light pen is composed of a photocell mounted in a penlike case. This pen may be pointed at the screen on a refresh display. The pen will send a pulse whenever the phosphor below it is illuminated. While the image on a refresh display may appear to be stable, it is in fact blinking on and off faster than the eye can detect. This blinking is not too fast for the light pen. The light pen can easily determine the time at which phosphor is illuminated. Since, there is only one electron beam on the refresh display, only one line segment can be drawn at a time and no two segments are drawn simultaneously.

When the light pen senses the phosphor beneath it being illuminated, it can interrupt the display processor's interpreting of the display file. The processor's instruction register tells which display file instruction is currently being

drawn. Once this information is extracted, the processor is allowed to continue its display. Thus the light pen tells us which display file instruction was being executed in order to illuminate the phosphor at which it was pointing. By determining which part of the picture contained the instruction that triggered the light pen, the machine can discover which object the user is indicating. It is often possible to turn the interrupt mechanism on or off during the display process and thereby select or deselect objects on the display for sensing by the light pen.

Keyboards: An alphanumeric keyboard on a graphics system is used primarily as a device for entering text strings. The keyboard is an efficient device for inputting such non--graphic data. Cursor control keys and function keys are common features on general purpose keyboards. Function keys allows user to enter frequently used operations in a single keystroke and cursor control keys can be used to select displayed objects or co-ordinate positions by positioning the screen cursor. Additional a numeric keypad is often included on the keyboard for fast entry of numeric data. The latest keyboards are coming with a facility to perform all the operations related to multimedia and internet browsing *etc.*

Trackball and Space Ball: A track ball is a ball that can be rotated with the fingers or palm of the hand to produce screen-cursor movement. Potentiometers, attached to the ball, measure the amount and direction of rotation. It is a two dimensional positioning device.

A space ball provides six degrees of freedom. Unlike the track ball, a space ball does not actually move. Strain

gauges measure the amount of pressure applied to the space ball to provide input for spatial positioning and orientation as the ball is pushed or pulled in various directions. Space balls are used for three dimensional positioning and selection operations in virtual reality systems, modelling, animation, CAD and other applications.

Data Glove: A data glove that can be used to grasp a virtual object. The glove is constructed with a series of sensors that detect hand and finger motions. Electromagnetic coupling between transmitting antennas and receiving antennas is used to provide information about the position and orientation of the hand. The transmitting and receiving antennas can each be structured as a set of three mutually perpendicular coils, forming a three dimensional co-ordinate system. Input from the glove can be used position or manipulate objects in a virtual scene. A two-dimensional projection of the scene can be viewed on a video monitor, or a three-dimensional projection can be viewed with a headset.

Digitizers: A common device for drawing, painting or interactively selecting co-ordinate positions on an object is a digitizer. These devices can be used to input co-ordinate values in either a 2D or 3D space. Digitizer is used to scan over a drawing or object and to input a set of discrete co-ordinate positions, which can be joined with straight line segments to approximate the curve or surface shapes. 3D digitizers use sonic or electromagnetic transmissions to record positions. One electromagnetic transmission method is similar to that used in the data glove: a coupling between

the transmitter and receiver is used to compute the location of a stylus as it moves over the surface of an object.

Image Scanners: Drawings, graphs, colour and black and white photos or text can be stored for computer processing with an image scanner by passing an optical scanning mechanism over the information to be stored. The gradations of gray scale or colour are then recorded and stored in an array. Once we have the internal representation of a picture, we can apply transformations to rotate, scale or crop the picture to a particular screen area. We can also apply various image processing methods to modify the array representation of the picture. For scanned text input, various editing operations can be performed on the stored documents. Some scanners are able to scan either graphical representations or text and they come in a variety of sizes and capabilities.

Touch Panels: Touch panels allow displayed objects or screen positions to be selected with the touch of a finger. A typical application of touch panels is for the selection of processing options that are represented with graphical icons. Some systems such as plasma panels are designed with touch screens.

Other systems can be adapted for touch input by fitting a transparent device with a touch sensing mechanism over the video monitor screen. Touch input can be recorded using three methods. They are

- Optical touch panels.
- Electrical touch panels.
- Acoustical touch panels.

Optical Touch Panels: They employ a line of infrared light emitting diodes (LEDs) along one vertical edge and along one horizontal edge of the frame. The opposite vertical and horizontal edges contain light detectors. These detectors are used to record which beams are interrupted when the panel is touched.

The two crossing beam that are interrupted identify the horizontal and vertical coordinates of the screen position selected. Positions can be selected with an accuracy of about ¼ inch.

Electrical Touch Panels: It is constructed with two transparent plates separated by a small distance. One of the plates is coated with a conducting material and the other plate is coated with a resistive material. When the outer plate is touched, it is forced into contact with the inner plate. This contact creates a voltage drop across the resistive plate that is converted to the coordinate values of the selected screen position.

Acoustical Touch Panels: In these high frequency sound waves are generated in the horizontal and vertical directions across a glass plate. Touching the screen causes part of each wave to be reflected from the finger to the emitters. The screen position at the point of contact is calculated from a measurement of the time interval between the transmission of each wave and its reflection to the emitter.

Voice Systems

Speech recognizers are used in some graphics workstations as input devices to accept voice command.

The voice system input can be used to initiate graphics operations or to enter data. These systems operate by matching an input against a predefined dictionary of words and phrases.

A dictionary is set up for a particular operator by having the operator speak the command words to be used into the system. Each word is spoken several times, and the system analyses the word and establishes a frequency pattern for that word in the dictionary alone with the corresponding function to be performed.

When a voice command is given, the system searches the dictionary for a frequency pattern match. Voice input is typically spoken into a microphone mounted on a headset. If a different operator is to use the system, the dictionary must be reestablished with that operator's voice patterns.

OUTPUT DEVICES

Printers

Printers produce output by either impact or non--impact methods. Impact printers press formed character faces against an inked ribbon onto the paper. A line printer is an example of an impact device, with the typefaces mounted on bands, chains, drums or wheels. Non--impact printers and plotters use laser techniques, ink-jet sprays, xerographic processes, electrostatic methods and electro thermal methods to get images onto the paper.

Character impact printers often have a dot-matrix print head containing a rectangular array of protruding wire pins,

with the number of pins depending on the quality of the printer. Individual characters or graphics patterns are obtained by retracting certain pins so that the remaining pins form the pattern to be printed.

In a laser device, a laser beam creates a change distribution on a rotating drum coated with a photoelectric material. Toner is applied to the drum and then transferred to paper. Ink-jet methods produce output by squirting ink in horizontal rows across a roll of a paper wrapped on a drum. The electrically charged ink stream is deflected by an electric field to produce dot-matrix patterns.

An electrostatic device places a negative charge on the paper, one complete row at a time along the length of the paper. Then the paper is exposed to a toner. The toner is positively charged and so is attracted to the negatively charged areas, where it adheres to produce the specified output.

We can get limited coloured ribbons. Non- impact devices use various techniques to combine three colour pigments to produce a range of colour patterns. Laser and xerographic devices deposit the three pigments on separate passes; ink-jet methods shoot the three colours simultaneously on a single pass along each print line on the paper.

Plotters

Drafting layouts and other drawings are typically generated with ink-jet or pen plotters. A pen plotter has one or more pens mounted on a carriage, or crossbar that spans a sheet of paper.

Pens with varying colours and widths are used to produce a variety of shadings and line styles. Wet-ink, ball point and felt tip pens are all possible choices for use with a pen plotter. Plotter paper can lie flat or be rolled onto a drum or belt. Crossbars can be either moveable or stationary, while the pen moves back and forth along the bar. Either clamps, a vacuum, or an electrostatic charge hold the paper in position.

Display Devices

In most applications of computer graphics the quality of the displayed image is very important. A great deal of effort has been directed towards the development of high quality computer display devices. The CRT was the only available device capable of converting the computer's electrical signals into visible images at high speeds. CRT technology has produced a range of extremely effective computer display devices. At the same time the CRT's peculiar characteristics have had a significant influence on the development of interactive computer graphics.

The CRT

The basic arrangement of CRT. At the narrow end of a sealed conical glass tube is an electron gun that emits a high velocity, finely focused beam of electrons. The other end, the face of the CRT, is more or less flat and is coated on the inside with phosphor, which glows when the electron beam strikes it. The energy of the beam can be controlled so as to vary the intensity of light output and when necessary to cut off the light altogether. A yoke or system of

electromagnetic coils is mounted on the outside of the tube at the base of the neck; it deflects the electron beam to different parts of the tube face when currents pass through the coils. The light output of the CRT's phosphor falls off rapidly after the electron beam has passed by and a steady picture is maintained by tracing it out rapidly and repeatedly; generally this refresh process is performed at least 30 times a second.

Electron Gun

Electron gun makes use of electrostatic fields to focus and accelerate the electron beam. A field is generated when two surfaces are raised to different potentials; electrons within the field tend to travel towards the surface with the more positive potential. The force attracting the electron is directly proportional to the field potential.

The purpose of the electron gun in the CRT is to produce an electron beam with the following properties:

- It must be accurately focused so that it produces a sharp spot of light where it strikes the phosphor.
- It must have high velocity, since, the brightness of the image depends on the velocity of the electron beam.
- Means must be provided to control the flow of electrons so that the intensity of the trace of the beam can be controlled.

Electrons are generated by a cathode heated by an electric filament. Surrounding the cathode is a cylindrical metal control grid, with a hole at one end that allows electrons to escape. The control grid is kept at a lower

potential than the cathode, creating an electrostatic field that directs the electrons through a point source; this simplifies the subsequent focusing process. By altering the control grid potential, we can modify the rate of flow of electrons, or beam current and can thus control the brightness of the image; we can even cut off the flow of electrons altogether. Focusing is achieved by a focusing structure, used to focus finely and highly concentrated at the precise moment at which it strikes the phosphor. An accelerating structure is generally combined with the focusing structure. It consists of two metal plates mounted perpendicular to the beam axis with holes at their centres through which the beam can pass. The two plates are maintained at a sufficiently high relative potential to accelerate the beam to the necessary velocity; accelerating potentials of several thousand volts are not uncommon. The resulting electron gun structure has the advantage that it can be built as a single physical unit and mounted inside the CRT envelope. Other types of gun exist, whose focusing is performed by a coil mounted outside the tube; this is called electromagnetic focusing.

The Deflection System

A set of coils or yoke, mounted at the neck of the tube, forms part of the deflection system responsible for addressing in the CRT. Two pairs of coils are used, one to control horizontal deflection and the other for vertical. A primary requirement of the deflection system is that it deflects rapidly, since, speed of deflection determines how much information can be displayed without flicker. To

achieve fast deflection, we must use large amplitude currents in the yoke. An important part of the deflection system is therefore the set of amplifiers that convert the small voltages received from the display controller into currents of the appropriate magnitude.

The voltages used for deflection are generated by the display controller from digital values provided by the computer. These values normally represent coordinates that are converted into voltages by digital to analog conversion. To draw a vector a pair of gradually changing voltages must be generated for the horizontal and vertical deflection coils.

Phosphors

The phosphors used in a graphic display are normally chosen for their colour characteristics and persistence. Ideally the persistence, measured as the time for the brightness to drop to one tenth of its initial value, should last about 100 milliseconds or less allowing refresh at 30Hz rates without noticeable smearing as the image moves. Colour should preferably be white, particularly for applications where dark information appears on a light background.

The phosphor should also possess a number of other attributes: small grain size for added resolution, high efficiency in terms of electric energy converted to light and resistance to burning under prolonged excitation.

In attempts to improve performance in one or another of these respects, many different phosphors have been produced, using various compounds of calcium, cadmium

and zinc together with traces of rare earth elements. These phosphors are identified by a numbering system like P1, P4, P7 *etc.*

Raster-scan Displays

The most common type of graphics monitor employing a CRT is the raster scan display. In a raster-scan system, the electron beam is swept across the screen, one row at a time from top to bottom. As the electron beam moves across each row, the beam intensity is turned on and off to create a pattern of illuminated spots. Picture definition is stored in a memory area called the refresh buffer or frame buffer. This memory area holds the set of intensity values for all the screen points. Stored intensity values are then retrieved from the refresh buffer and painted on the screen one row at a time.

Each screen point is referred to as a pixel or pel (picture element). The capability of a raster-scan system to store intensity information for each screen point makes it well suited for the realistic display of scenes. Home televisions and printers are examples of other systems using raster-scan methods.

Intensity range for pixel positions depends on the capability of the raster system. In a simple black and white system, each screen point is either on or off, so only one bit per pixel is needed to control the intensity of screen positions. Here 1 indicates that the electron beam is to be turned on at that position, and value 0 indicates that the electron beam intensity is to be off. Additional bits are needed when colour and intensity variations can be

displayed. Up to 24 bits per pixel are included in high quality systems, which can require several megabytes of storage for the frame buffer, depending on the resolution of 1024 by 1024 requires 3 megabytes of storage for the frame buffer. On a black and white system with one bit per pixel, the frame buffer is commonly called a bitmap. For systems with multiple bits per pixel, the frame buffer is often referred to as a pixmap.

Refreshing on raster-scan displays is carried out at the rate of 60 to 80 frames per second. At the end of each scan line, the electron beam returns to the left side of the screen to begin displaying the next scan line. The return to the left of the screen, after refreshing each scan line, is called the horizontal retrace of the electron beam and at the end of each frame the electron beam returns to the left corner of the screen to begin the next frame. On some raster-scan systems, each frame is displayed in two passes using an interlaced refresh procedure. In the first pass, the beam sweeps across every other scan line from top to bottom. Then after the vertical retrace, the beam sweeps out the remaining scan lines.

Random-scan Displays

When operated as a random-scan display unit, a CRT has the electron beam directed only to the parts of the screen where a picture is to be drawn. Random-scan monitors draw a picture one line at a time and for this reason are also referred to as vector displays. A pen plotter operates in a similar way and is an example of a random-scan, hard copy device. Refresh rate on a random-scan

system depends on the number of lines to be displayed. Picture definition is now stored as a set of line drawing commands in an area of memory referred to as the refresh display file. Sometime the refresh display file is called the display list or display Programme or refresh buffer. To display a specified picture, the system cycles through the set of commands in the display file, drawing each component line in turn. After all line drawing commands have been processed, the system cycles back to the first line command in the list.

Colour CRT Monitors

A CRT monitor displays colour pictures by using a combination of phosphors that emit different-coloured light. By combining the emitted light from the different phosphors, a range of colours can be generated. The two basic techniques for producing colour displays with a CRT are the *beam-penetration method* and the *shadow-mask method*.

The beam-penetration method for displaying colour pictures has been used with random-scan monitors. Two layers of phosphor, usually red and green are coated onto the inside of the CRT screen, and the displayed colour depends on how far the electron beam penetrates into the phosphor layers. A beam of slow electrons excites only the outer red layer. A beam of very fast electrons penetrates through the red layer and excites the inner green layer. At intermediate beam speeds, combinations of red and green light are emitted to show two additional colours, orange and yellow. The speed of the electrons and hence the screen

colour at any point is controlled by the beam-acceleration voltage. Four colours are possible, and the quality of pictures is not as good as with other methods.

Shadow-mask methods are commonly used in raster-scan systems because they produce a much wider range of colours than the beam-penetration method. A shadow-mask CRT has three phosphor colour dots at each pixel position. One phosphor dot emits a red light, another emits a green light, and the third emits a blue light. This type of CRT has three electron guns, one for each colour dot and a shadow-mask grid just behind the phosphor-coated screen. We obtain colour variations in a shadow-mask CRT by varying the intensity levels of the three electron beams. By turning off the red and green guns, we get only the colour coming from the blue phosphor. A white area is the result of activating all three dots with equal intensity.

Direct-View StorageTubes

An alternative method for maintaining a screen image is to store the picture information inside the CRT instead of refreshing the screen. A direct-view storage tube (DVST) stores the picture information as a charge distribution just behind the phosphor-coated screen. Two electron guns are used in a DVST.

One, the primary gun, is used to store the picture pattern; the second, the flood gun, maintains the picture display. A DVST monitor has both disadvantages and advantages compared to the refresh CRT. Because no refreshing is needed, very complex pictures can be displayed at very high resolutions without flicker.

The disadvantages of DVST systems are that they ordinarily do not display colour and that selected parts of a picture cannot be erased. The entire screen must be erased and the modified picture redrawn. The erasing and redrawing process can take several seconds for a complex picture.

Flat-Panel Displays

The term flat-panel display refers to a class of video devices that have reduced volume, weight and power requirements compared to a CRT. Flat panel displays into two categories: emissive displays and noon-emissive displays. The emissive displays are devices that convert electrical energy into light. Plasma panels, thin-film electroluminescent displays, and light emitting diodes are examples of emissive displays. Non--emissive displays use optical effects to convert sunlight or light from some other source into graphics patterns. The most important example of a non--emissive flat-panel display is a liquid crystal device.

Plasma panels also called gas-discharge displays are constructed by filling the region between two glass plates with a mixture of gases that usually includes neon. A series of vertical conducting ribbons is placed on one glass panel, and a set of horizontal ribbons is built into the other glass panel. Firing voltages applied to a pair of horizontal and vertical conductors cause the gas at the intersection of the two conductors to break down into glowing plasma of electrons and ions. Picture definition is stored in a refresh buffer, and the firing voltages are applied to refresh the

pixel positions 60 times per second. One disadvantage of plasma panels has been that they were strictly monochromatic devices, but systems have been developed that are now capable of displaying colour and grayscale.

LCD Technology

Borrowing technology from laptop manufacturers, some companies provide LCD (Liquid Crystal Display) displays. LCDs have low glare flat screens and low power requirements. The colour quality of an active matrix LCD panel actually exceeds that of most CRT displays. At this point, however, LCD screens usually are more limited in resolution than typical CRTs and are much more expensive. There are three basic LCD choices.

They are.....

- Passive matrix monochrome.
- Passive matrix colour.
- Active matrix colour.

In a LCD, a polarizing filter creates two separate light waves. In a colour LCD, there is an additional filter that has three cells per each pixels – one each for displaying red, green and blue.

The light wave passes through a liquid crystal cell, with each colour segment having its own cell. The liquid crystals are rod-shaped molecules that flow like a liquid. They enable light to pass straight through them. Although monochrome LCDs do not have colour filters, they can have multiple cells per pixel for controlling shades of grey.

In passive matrix LCD, each cell is controlled by electrical charges transmitted by transistors according to row and

column positions on the screen's edge. As the cell reacts to the pulsing charge, it twists the light wave, with stronger charges twisting the light wave more. In an active matrix LCD, each cell has its own transistor to charge it and twist the light wave. This provides brighter image than passive matrix displays because, the cell can maintain a constant, rather than momentary charge. However, active matrix technology uses more energy than passive matrix. With a dedicated transistor for every cell, active matrix displays are more difficult and expensive to produce. In both active and passive matrix LCDs, the second polarizing filter controls how much light passes through each cell. Cells twist the wavelength of light that passes through the filter at each cell, the brighter the pixel. The best colour displays are active matrix or thin film transistor panels, in which each pixel is controlled by three transistors for red, green and blue.

Raster-scan Systems

Interactive raster graphics systems typically employ several processing units. In addition to the central processing unit, a special-purpose processor, called the video controller or display controller is used to control the operation of the display device.

The video controller accesses the frame buffer to refresh the screen. In addition to the video controller, more sophisticated raster systems employ other processors as co-processors and accelerators to implement various graphics operations.

Video Controller

Frame buffer locations, and the corresponding screen positions, are referenced in Cartesian co-ordinates. For many graphics monitors, the co-ordinate origin is defined at the lower left screen corner. The screen surface is then represented as the first quadrant of a two-dimensional system, with positive x values increasing to the right and positive y values increasing from bottom to top. Scan lines are then labelled from y_{max} at the top of the screen to 0 at the bottom. Along each scan line, screen pixel positions are labelled from 0 to x_{max}. Two registers are used to store the co-ordinates of the screen pixels. Initially, the x register is set to 0 and the y register is set to ymax. The value stored in the frame buffer for this pixel position is then retrieved and used to set the intensity of the CRT beam. Then the x register is incremented by 1, and the process repeated for the next pixel on the top scan line. This procedure is repeated for each pixel along the scan line. After the last pixel on the top scan line has been processed, the x register is reset to 0 and the y register is decremented by 1. Pixels along this scan line are then processed in turn, and the procedure is repeated for each successive scan line. After cycling through all pixels along the bottom scan line (y = 0), the video controller resets the registers to the first pixel position on the top scan line and the refresh process starts over.

Raster-scan Display Processor

The organization of raster system containing a separate display processor, sometimes referred to as a graphics

controller or display co-processor. The purpose of the display processor is to free the CPU from the graphics chores. In addition to the system memory, a separate display processor memory area can also be provided. A major task of the display processor is digitizing a picture definition given in an application Programme into a set of pixel-intensity values for storage in the frame buffer. This digitization process is called scan conversion.

Characters can be defined with rectangular grids. The array size for character grids can vary from about 5 by 7 to 9 by 12 or more for higher quality displays. Display processors are typically designed to interface with interactive input devices such as mouse. In an effort to reduce memory requirements in raster systems, methods have been devised for organizing the frame buffer as a linked list and encoding the intensity information. One way to do this is to store each scan line as a set of integer pairs. One number of each pair indicates an intensity value, and the second number specifies the number of adjacent pixels on the scan line that are to have that intensity. This technique called run-length encoding. A similar approach can be taken when pixel intensities change linearly. Another approach is to encode the raster as a set of rectangular areas (cell encoding).

Random-scan Systems

The organization of a simple random-scan system. An application Programme is input and stored in the system memory along with a graphics package. Graphics commands in the application Programme are translated

by the graphics package into a display file stored in the system memory. This display file is then accessed by the display processor to refresh the screen. The display processor cycles through each command in the display file Programme once during every refresh cycle. Sometimes the display processor in a random-scan system is referred to as a display processing or a graphics controller.

Lines are defined by the values for their co-ordinate endpoints, and these input co-ordinate values are converted to x and y deflection voltages. A scene is then drawn one line at a time by positioning the beam to fill in the line between specified endpoints.

3

Computer Graphics in Java

Although computer graphics is a vast field that encompasses almost any graphical aspect, we are mainly interested in the generation of images of 3-dimensional scenes.

Computer imagery has applications for film special effects, simulation and training, games, medical imagery, flying logos, etc. Computer graphics relies on an internal model of the scene, that is, a mathematical representation suitable for graphical computations. The model describes the 3D shapes, layout and materials of the scene.

This 3D representation then has to be projected to compute a 2D image from a given viewpoint, this is the rendering step.

Rendering involves projecting the objects (perspective), handling visibility (which parts of objects are hidden) and computing their appearance and lighting interactions. Finally,

for animated sequence, the motion of objects has to be specified. We will not discuss animation in this document.

BACKGROUND OF COMPUTER GRAPHICS

Today there are very few aspects of our lives not affected by computers. Practically every cash or monetary transaction that takes place daily involves a computer. In many cases, the same is true of computer graphics. Whether you see them on television, in newspapers, in weather reports or while at the doctor's surgery, computer images are all around you.

"A picture is worth a thousand words" is a well-known saying, and highlights the advantages and benefits of the visual presentation of our data. We are able to obtain a comprehensive overall view of our data and also study features and areas of particular interest. A well-chosen graph is able to transform a complex table of numbers into meaningful results. Such graphs are used to illustrate papers, reports, and theses, as well as providing the basis for presentation material in the form of slides and overhead transparencies. A range of tools and facilities are available to enable users to visualise their data, and this document provides a brief summary and overview. Computer graphics are used in many disciplines and subjects but for the purpose of this document, we will split the topic of computer graphics into the following fields:

CHARTING

One of the prime uses for graphical software at the University is to produce graphs and charts. Everyone has

data of one kind or another, whether on paper, in the computer, or just in the mind. We often need to know the significance and properties of the data, or to be able to compare different parts of it against other data sets.

One of the simplest aspects of data display is the production of charts. This is where you would want to put your data into a graphical form to show relationships and comparisons between sets of values.

There may be a number of reasons why you would want to put your data into a chart:

- To illustrate differences between different sets of data,
- To show trends between two variables,
- To show patterns of behaviour in one variable.

There are basically two broad areas of graphs:

- Presentation charts and graphs of the kind used to illustrate a few principal points. We see these on news and current affairs programmes on television. A bar chart or a pie chart is used to indicate results of data obtained so far and the general trends. They are often liberally decorated with bright colours to increase their visual appeal and attractiveness to the viewers and to hold their attention. They are used for visual impact and getting a simple point over clearly and effectively.
- Scientific charts and graphs are more concerned with ensuring that the detail in the data is represented accurately and faithfully. We may have some results obtained from experimental

measurements and wish to display them. We may want to compare the results from the data measurements with the results we would expect according to a particular theoretical model. We may want to draw a curve through the data points (*i. e.* interpolate the data) and display this along with the original points.

The aims of the two are different, and so the facilities you will want from your charting package will also be different. Presentation charting has more to do with impressive presentation graphics where the aim is to put a salient point across to an audience. As a result the priority with this sort of charting is not always accuracy of representation. You want charts with strong colours, an impressive look and special effects. The effect of a presentation can be enhanced by using 3D graphs, adding pictures to the graph, or using pictograms. These sorts of charts are rarely produced in isolation but as part of a general presentation.

Therefore, some presentation packages also have their own charting module for this purpose. Word and PowerPoint use a module called Microsoft Graph and Excel's charting module has some very powerful presentation graphics features. Origin and Gsharp, both dedicated charting packages, also provide professional presentation charting facilities on the PC systems. Gsharp is also available on the UNIX systems.

In scientific charting you want to display data as accurately as possible in order to analyse it graphically or

demonstrate clearly your comparisons and results. As this sort of charting is done mainly for analysis, it is rarely an isolated activity but is often done alongside detailed numerical analysis of your data. Two of the most powerful charting packages available are Origin on the PC network and Gsharp on the PC and UNIX systems. Also, many numerical analysis packages have their own charting modules integrated with the rest of the package.

It is clear that your choice of charting programme will depend very much on what purpose you want the chart to fulfil, and also what other programmes you are already using. On the whole, if you are already using a programme that has its own charting module, use that. The table below gives some rough guidelines on your choice of charting PC package, with the packages increasing in facilities and complexity going down the table.

Requirement	**Choice**
Simple bar, column, line or pie charts to integrate in a word processor	Microsoft Graph in Word, Charting Module in Excel
Charts for use in a presentation	Microsoft Graph in Word or PowerPoint, Charting Module in Excel, Origin
Raw data requiring good quality scientific charting	Origin, Gsharp
Data requiring simple mathematical or statistical analysis	Charting Module in Excel, Origin, Gsharp
Complicated statistical analysis and good quality scientific charts	Graphics module in SPSS

PRESENTATIONS

Presentation software is used to create material used in presentations, such as OHP transparencies and 35mm

slides. The term is also commonly used when a presentation is given using the output from a computer screen. The use of presentation software is becoming of increasing importance as higher standards become expected in courses and presentations. This will often include making use of colour, graphics and the University logo.

Course materials produced using presentation packages can be delivered in a number of ways. The simplest way is to print the material on a laser printer and then use a photocopier to produce overhead projector (OHP) acetates (first making sure that the photocopier can accept acetates). You can also use the output services produced by Information Systems Services and University Media Services to produce colour output or output on 35mm slides.

Alternatively you can give a desktop presentation using OHP projection tablets or projection systems to deliver a presentation using the output from a computer system directly. The simplest presentation software is a word processor. Word processing packages such as Word, which can produce text in a variety of sizes, can be used to create OHP transparencies.

Specialist presentation packages, such as PowerPoint, provide a wider range of facilities than word processors and, in general, are easier to use for the production of presentation materials. PowerPoint is a presentation software programme that helps you quickly and easily create professional quality presentations. Presentations can be transferred onto paper, overheads or 35mm slides,

or they can be shown on a video screen or computer monitor. PowerPoint's printing options include formats ranging from audience handouts to speaker's notes.

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.

BUSINESS ANALOGY-SERVICES PROVIDED BY AN ORGANISATION

You could think of classes as corresponding to departments in an organisation, and methods as being the services they provide. Let's take an example of calling the telephone information service to get someone's phone number. When you make the call, you pass information, the name of the person whose number you want, to the method (information service). This called "method" then does something, and returns a value to you (the desired phone number). Just as you don't have to know how the information service performs its job, you typically don't need to know exactly how a method does its work, unless of course, you are writing it.

Hierarchical Organisation: Computer programmes are structured in many ways like an organisation—higher levels rely on others to do much of the work. They "call" on lower levels to do the work, passing them all necessary "arguments". In a similar way, the top level of a computer programme, main, often consists largely of method calls, and those methods may in turn call on yet other methods.

TERMS: CALL AND RETURN

Call. When a method call is encountered in a programme, the programme remembers where it was and execution goes to the method (*calls* the method). After a small amount of initialisation, the statements in the method are executed starting at the beginning.

Return. When the end of the method is reached or a return statement is executed, the method *returns* to the where it was called from, and execution continues in the calling method from that point.

A method may return a value (eg, parseDouble) or not (showMessageDialog). The call-return terminology is almost universal.

STATIC (CLASS) METHODS

Static. This starts with *static* (also called *class*) methods because all applications start with the *static* method *main*, and many of the early library methods that you use are static methods. Static methods are different than *instance* methods because they don't have an extra object passed to them.

Instance methods are associated with an object (an "instance" of a class).

Identifying Methods

Parentheses follow name. You can identify a method name because it is always followed by left and right parentheses, which may enclose *arguments* (*parameters*). If you see a left parenthesis with a name preceding it, it will be a method call or definition, or a *constructor* (constructors are very similar to methods). In the following example each method name is highlighted.

When calling methods outside of the current class, eg, in the Java library, static methods are preceded by the class name (followed by a dot), and instance methods are

preceded by an object. When a static method is defined, the keyword "static" will preceded it. The example below has only static methods.

```
// File : methods/KmToMiles. java
1 // Purpose: Convert kilometers to miles. Use JOptionPane
for input/ output.
2 // Author: Fred Swartz
// Date : 22 Apr 2006
3 import javax. swing. *;
4 public class KmToMiles {
5                                                          //
============================================================
constants
6 private static final double MILES_PER_KILOMETER = 0.
621;
 7 //
============================================================
8 main
9 public static void main(String[] args) { //Note 1
10 //. . . Local variables
11 String kmStr; // String km before conversion to double.
12 double km; // Number of kilometers.
13 double mi; // Number of miles.
14 //. . . Input
15 kmStr = JOptionPane. showInputDialog(null, "Enter
kilometers. ");
16 km = Double. parseDouble(kmStr);
17 //. . . Computation
18 mi = km * MILES_PER_KILOMETER;
19 //. . . Output
20 JOptionPane. showMessageDialog(null, km + " kilometers
is "
21 + mi + " miles. ");
22 }
23 }
24
25
26
27
28
29
30
```

Notes:

- This defines a method called "main". Everything between the "{" on the end of this line to the matching "}" second from the end is the "body" of the method.

The above code *defines* the static main method, which someone (eg, the operating system) will call with KmToMiles. main(. . .). To do its work, main calls on other methods: showInputDialog, which is defined in the JOptionPane class, parseDouble, which is defined in the Double class, and showMessageDialog, which is also in the JOptionPane class. Whenever you call a static method in a different class, precede it with the name of the class containing its definition, followed by a dot. If you don't specify the class name, it assumes the method is defined in the current class.

Identifying Instance Methods (SB)

Object precedes. Instance method calls are identified in the following programme. Note that they are all preceded by an object reference. This object is used by the methods to do their work. In this case, the objects are strings.

In addition to supplying the object's data to the method, the class of the object, eg String, is where the method is defined.

```
// File : dialog/capitalise/Capitalise2. java
1 // Purpose: Capitalise first letter of each name. Declare
with first use.
2 // Author: Fred Swartz-placed in public domain.
3 // Date : 30 Mar 2006
4 import javax. swing. *;
```

```
5 public class Capitalise2 {
6 public static void main(String[] args) {
7 //. . Input a word
8 String inputWord = JOptionPane. showInputDialog(null,
"Enter a word");
9 //. . Process-Separate word into parts, change case,
put together.
10 String firstLetter = inputWord. substring(0, 1); //
Get first letter
11 String remainder = inputWord. substring(1); // Get
remainder of word.
12 String capitalised = firstLetter. toUpperCase() +
remainder. toLowerCase();
13 //. . Output the result.
14 JOptionPane. showMessageDialog(null, capitalised);
15 }
16 }
17
18
19
20
21
```

What's Before the Dot Tells Whether it's a Class or Instance Method

What's before the dot? If it's a class name, then it's a static (class) method; if it's an object, it's an instance method. Nothing at front when calling methods in same class.

When calling your own methods, you don't have to write anything before the method name. The compiler assumes the same class or object.

ACTUAL ARGUMENTS (PARAMETERS)

The values that are passed to a method are called *actual arguments* in the Java specification. However, it is very common for them to be called just *arguments*, *actual parameters*, or just plain *parameters*. These terms are so

used interchangeably so often that even the Java specification isn't entirely consistent. For a value which is passed in a call I'll try to stick to *actual argument* or just *argument*, which are generally regarded as the "best" terms.

Identifying method arguments

When you call a method, you can pass information for it to use.

These *actual arguments* are inside parentheses following the method name. Use commas to separate arguments if there is more than one.

The previous programme is shown below, but this time the arguments in the calls are highlighted.

```
// File : methods/KmToMilesArgs. java
1 // Purpose: Convert kilometers to miles. Use JOptionPane
for input/ output.
2 // Author: Fred Swartz
// Date : 22 Apr 2006
3 import javax. swing. *;
4 public class KmToMilesArgs {
5                                                          //
============================================================
constants
 6 private static final double MILES_PER_KILOMETER = 0.
621;
 7 //
============================================================
8 main public static void main(String[] args) {
9 //. . . Local variables
10 String kmStr; // String km before conversion to double.
11 double km; // Number of kilometers.
12 double mi; // Number of miles.
13 //. . . Input
14 kmStr = JOptionPane. showInputDialog(null, "Enter
kilometers. ");
15 km = Double. parseDouble(kmStr);
16 //. . . Computation
```

```
17 mi = km * MILES_PER_KILOMETER;
//. . . Output
18 JOptionPane. showMessageDialog(null, km + " kilometers
is "
+ mi + " miles. ");
19 }
20 }
21
22
23
24
25
26
27
28
29
30
```

Argument Evaluation

Before a method is called, the arguments are evaluated left-to-right. In the example above most arguments are simple values, except the second argument in the call to showMessageDialog. Before the call can be made, this argument expression must be evaluated by performing the conversions to string and the concatenations.

VOID AND VALUE-RETURNING METHODS

A method may return a value. In the example above, showInputDialog returns a String and parseDouble returns a double value.

These method calls can be used anywhere in an expression where a String or double value is required. Here they simply provide the value for the right side of an assignment void.

If a method has a "side effect", but doesn't produce a value, it is called a *void* method. The showMessageDialog

method shows something to the user, but doesn't return a value, and is a void method. When a method is defined, you need to specify the keyword void if it doesn't return a value. You can see this on line 13 where the main method definition starts.

DEFINING A STATIC METHOD

Method Header Syntax

A method header is the part of the method definition that occurs at the beginning. The following definition leaves out a few obscure features, but gives the syntax of ordinary method headers.

Syntax Notation

There are several ways to describe syntax that have been very popular. It's essential for compiler writers to have a very accurate description of the syntax of a programming language, and there are a number of tools for reading a syntax description and turning it into executable parsers.

Following are three ways to show the syntax of a *method header*.

EBNF

Here are the syntax rules for EBNF.

```
Non-terminals Written as italicised text.
Terminals Written in quotes, as in "private".
| The left and right sides are alternatives.
( ) Parentheses are used for grouping.
[ ] Brackets enclose optional material.
{ } Braces enclose material to be repeated 0 or more
times.
= Defines symbol on left by notation on the right.
```

```
. Ends each definition.
spaces Used only for readability.
```

Definition of a Method Header

```
methodHeader
=
[visibility] ["static"] returnType methodName "(" 
[parameterList] ")".
visibility
=
"public" | "private" | "protected".
ParameterList
=
parameterDeclaration {", " parameterList}.
parameterDeclaration
=
type ParameterName.
returnType
=
"void" | type
```

RAILROAD DIAGRAMS

An attractive, readable, syntax notation can be found in *railroad/railway diagrams*. Just follow the lines. The flow is left to right or top-to-bottom. Traditionally they go primarily left-to-right because that's how programming statements are written, but this results in very wide diagrams, requiring lines to snake to the beginning of the next line or some connection notation.

Advantage: You'll note in the crude ASCII railroad diagrams below that there are not as many non-terminal symbols as in EBNF.

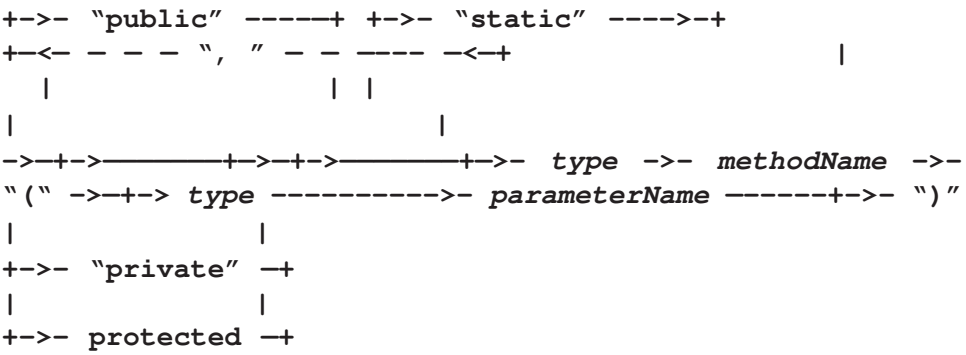
Problems: There are two big problems with railroad digrams:

- They are hard to produce. The only software I've seen to produce general images (Bungisoft Diagram

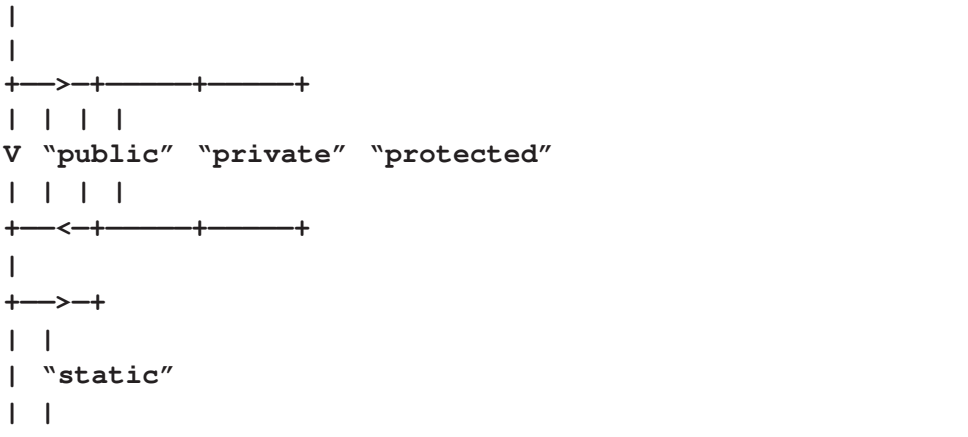
Visualiser) seems to no longer be available. Perhaps this isn't surprising given the poor quality and high price. I mentioned this market opportunity that there might be 17 people who would be interested in this sort of tool. Well, it would make a good programming project for someone.

- An audience that you might think this would appeal to are compiler writers, but that isn't so. The compiler writers want a text file that can be read by parser generators, which means something closer to EBNF.

Horizontal ASCII Railroad Diagram



Vertical ASCII Railroad Diagram



```
+--<--+
|
type
|
methodName
|
"("
|
+-<-+
| |
type |
| ", "
parameterName |
| |
+->-+
|
")
```

How to Define your Own Method

The previous programme is rewritten below to define a method to convert from kilometers to miles. The method call, and the first line (header) of the method definition are highlighted.

```
// File : methods/KmToMilesMethod. java
1 // Purpose: Convert kilometers to miles using a method.
JOptionPane IO.
2 // Highlight call and method definition header.
// Author: Fred Swartz
3 // Date : 22 Apr 2006
4 import javax. swing. *;
5 public class KmToMilesMethod {
/                                                        /
============================================================
6 constants
7 private static final double MILES_PER_KILOMETER = 0.
621;
8                                                       //
============================================================
main
9 public static void main(String[] args) {
10 //. . . Local variables
```

```
11 String kmStr; // String km before conversion to double.
12 double km; // Number of kilometers.
13 double mi; // Number of miles.
14 //. . . Input
15 kmStr = JOptionPane. showInputDialog(null, "Enter
kilometers. ");
16 km = Double. parseDouble(kmStr);
17 //. . . Computation
18 mi = convertKmToMi(km);
19 //Note 1
20 //. . . Output
21 JOptionPane. showMessageDialog(null, km + " kilometers
is "
22 + mi + " miles. ");
23                                                    //
============================================================
24 convertKmToMi
25 private static double convertKmToMi(double kilometers)
{
26 //Note 2
27 double miles = kilometers * MILES_PER_KILOMETER;
28 return miles;
29 }
30 }
31
32
33
34
35
36
```

Notes:

- Call our own method below to do the conversion. We could have qualified the name with our class name, Km To Miles Method. convert Km To Mi (km), but this is unnecessary when calling a static method in the same class.
- Altho this method is trivial, just a multiplication, it is good practice to separate the "model", or "logic", from the user interface. As programmes become larger, this separation becomes essential.

ANATOMY OF THE CONVERTKMTOMI METHOD HEADER

We'll take a look at each of the parts of the method header in order.

Visibility-public, private, or package

```
private static double convertKmToMi(double kilometers) {
double miles = kilometers * MILES_PER_KILOMETER;
return miles;
}
```

For greatest reliability and flexibility in your programmes, you should always give methods the lowest visibility to others that you can. When you define a method, you should think about who can use it.

Generally you want to choose the lowest level of visibility that makes your programme usable, either private or the default (*package*). Here are the four options, from least visible to most visible.

- *Private:* If you don't want any other class to use it, declare it private. This is a good choice.
- *None (package):* If you don't specify anything, the default visibility allows only classes in the same package (directory) to see it. This is a common choice. It's common to use public visibility when *package* visibility is more appropriate—I do it myself. The lack of a keyword for package visibility makes it a little harder to read.
- *Protected:* Don't use this protected, except in certain cases to let a child class see it. Even then, its use is controversial.
- *Public:* Let's anyone see it. Choose this if you've defined a method that will be used by others outside

of your project. Note that main must be declared public so the run-time system can call it.

Class (static) or instance method

```
private static double convertKmToMi(double kilometers) {
double miles = kilometers * MILES_PER_KILOMETER;
return miles;
}
```

A method should be declared static if it doesn't user instance variables or methods. A static method must use only parameters, local variables, and static constants, and other static methods in the same class. If the static keyword is omitted, the method will be an *instance* method. This example uses static, but soon you will learn about instance methods too.

Return Type:

```
private static double convertKmToMi(double kilometers) {
double miles = kilometers * MILES_PER_KILOMETER;
return miles;
}
```

Method Name:

```
private static double convertKmToMi(double kilometers) {
double miles = kilometers * MILES_PER_KILOMETER;
return miles;
}
```

Method names should begin with a lowercase letter. Method names are typically verbs, whereas variable names are usually nouns.

Parameter(s):

```
private static double convertKmToMi(double kilometers) {
double miles = kilometers * MILES_PER_KILOMETER;
return miles;
}
```

Parameters are enclosed in parentheses following the method name. They are also called *formal parameters*). There is only one parameter in this example-kilometers,

but if there are more, they must be separated by commas. The *type* of each parameter is specified before the name (eg, double). Parameters are local variables that only exist inside the method. They are assigned initial values from the arguments when the method is called.

Method body:

```
private static double convertKmToMi(double kilometers) {
double miles = kilometers * MILES_PER_KILOMETER;
return miles;
}
```

The *body* of a method is the statements which are executed when the method is called are enclosed in braces following the method header. Additional local variables may be defined (eg, miles).

Return Statement:

```
private static double convertKmToMi(double kilometers) {
double miles = kilometers * MILES_PER_KILOMETER;
return miles;
}
```

A method returns to the caller after it has done what it wants. If the method returns a value (not a void method), it must contain a return statement that specifies a value to return. When execution reaches the return statement, control transfers back to the calling method, passing a return value to it.

Returning an Expression:

The above example returns the value in the local variable *miles*. The return statement can be followed by any expression of the appropriate type, not just a single value. For example, this method body could have been written as a single return statement.

```
private static double convertKmToMi(double kilometers) {
return kilometers * MILES_PER_KILOMETER;
}
```

Order of Method Definitions Doesn't Matter

If you define multiple methods in a class, you don't have to worry about the order of the definitions, unlike some other languages.

LOCAL VARIABLES

Now that we've written two methods, main and convertKmToMi, you should know a little more about the variables in them. Variables that are declared in a method are called *local variables*. They are called *local* because they can only be referenced and used locally in the method in which they are declared. In the method below miles is a local variable.

```
private static double convertKmToMi(double kilometers) {
double miles = kilometers * MILES_PER_KILOMETER;
return miles;
}
```

Visibility: Only in Defining Method

No code outside a method can see the local variables inside another method. There is no need, or even possibility, of declaring a local variable with a visibility modifier—local variables are automatically known only in the method itself.

Lifetime: From Method Call to Method Return

Local variables are created on the call stack when the method is entered, and destroyed when the method is exited. You can't save values in local variables between calls. For that you have to use instance variables, which you'll learn about a little later.

Initial Value: None

Local variables don't have initial values by default—you can't try to use their value until you assign a value. It's therefore common to assignment a value to them when they're declared.

Compiler Error

If you try to use a local variable before it's been assigned a value, the compiler will notice it and give an error message. But the compiler doesn't really know the exact order of execution in a programme, so it makes some conservative assumptions.

These assumptions can sometimes be too conservative, and there are cases where you must initialise a local variable even though you know it will have a value before it's referenced.

```
// BAD BAD BAD BAD BAD BAD BAD BAD BAD BAD BAD BAD BAD BAD
BAD
private static double convertKmToMi(double kilometers) {
double miles;
return miles; // Won't compile because nothing was assigned
to miles.
}
```

PARAMETERS ARE PREINITIALISED LOCAL VARIABLES

Method parameters are basically implemented as local variables.

They have the same visibility (none outside the method) and lifetime (created on method call, destroyed on method return).

Preinitialised. The difference is that parameters are initialised from the corresponding argument values.

```
// Both kilometers and miles are implemented as local
variables.
private static double convertKmToMi(double kilometers) {
double miles = kilometers * MILES_PER_KILOMETER;
return miles;
}
```

Style: Don't Assign to a Parameter

You can assign to a parameter variable, just as you would to a local variable, but this is often considered bad style because it can deceive the casual reader in two ways:

1. *Unexpected Meaning Change:* Programmers assume parameter variables represent actual argument values. Assigning to parameters breaks that assumption.
2. *Doesn't Change Actual Argument:* Because formal parameter variables are really local variables, assigning new values to them doesn't have any effect on the actual parameters.

However, in some programming languages assignment to a parameter can assign to the corresponding actual parameter (eg,

C++ reference parameters). Therefore if you write an assignment to a formal parameter variable, it may mislead the careless programmer with a C++ background. Or the reader may pause and try to decide if you thought you were assigning to the actual argument. In either case it reduces the readability.

Example. The example below shows how a parameter could be reused. The overhead of declaring an extra variable is just about zero, so this really isn't more efficient, and even this small example is astoundingly misleading.

```
// BAD BAD BAD BAD BAD BAD BAD BAD BAD BAD BAD BAD BAD BAD
BAD
private static double convertKmToMi(double kilometers) {
kilometers = MILES_PER_KILOMETER * kilometers; // BAD-
Don't do this, altho it works.
return kilometers;
}
```

Style: Final Keyword Prevents Assignment

Some programmers recommend using the final keyword for each parameter. This prevents assignment to the parameter. Few programmers do this because it adds extra clutter, which in a different way reduces the readability.

The use of self-restraint in assigning to parameters is usually sufficient, but specifying final isn't a bad idea.

```
private static double convertKmToMi(final double kilometers)
{
double miles = kilometers * MILES_PER_KILOMETER;
return miles;
}
```

DYNAMIC CHANGES IN THE CALL STACK MEMORY ALLOCATION

The table below shows how the call stack changes as calls and returns in the KmToMilesMethods programme are made. This shows the first 8 changes to the call stack after *main* is entered.

There is actually something before *main* on the call stack, and the library methods that are called call many methods of their own, which isn't shown here because we don't need to know what they call. Stack frame. Each box represents the information that's stored on the call stack for each method. This block of information is often

called a *stack frame*. There is internal information associated with the method, for example, it saves the place to resume execution in the calling method.

1	2	3	4	5	6	7	8
Main	**Main**	**Main**	**Main**	**Main**	**Main**	**Main**	**Main**
args	args	args	args	args	args	args	args
kms	kms	kms	kms	kms	kms	kms	kms
miles	miles	miles	miles	miles	miles	miles	miles
	getDouble	*getDouble*	*getDouble*	*getDouble*	*getDouble*		*convert*
	prompt	prompt	prompt	prompt	prompt		*KmToMi*
	str	str	str	str	str		kilometers
		show		*parseDouble*			miles
		InputDia-		???			
		log					
		???					

Each stack frame is labelled with the method name and a list of parameters and local variables that are allocated on the stack. "???" is written when we don't know (or care) what the local variables are that are used by a library method.

TYPICAL CALL SEQUENCE

- *Evaluate Arguments left-to-right:* If an argument is a simple variable or a literal value, there is no need to evaluate it. When an expression is used, the expression must be evaluated before the call can be made.
- *Push a new Stack frame on the Call Stack:* When a method is called, memory is required to store the following information.
  - Parameter and local variable storage. The storage that is needed for each of the parameters and local variables is reserved in the stack frame.

  - Where to continue execution when the called method returns. You don't have to worry about this; it's automatically saved for you.
  - Other working storage needed by the method may be required. You don't have to do anything about this because it's handled automatically.
- *Initialise the Parameters:* When the arguments are evaluated, they are assigned to the local parameters in the called method.
- *Execute the Method:* After the stack frame for this method has been initialised, execution starts with the first statement and continues as normal. Execution may call on other methods, which will push and pop their own stack frames on the call stack.
- *Return from the Method:* When a *return* statement is encountered, or the end of a void method is reached, the method returns. For non-void methods, the return value is passed back to the calling method. The stack frame storage for the called method is popped off the call stack. Popping something off the stack is really efficient-a pointer is simply moved to previous stack frame. This means that the current stack frame can be reused by other methods. Execution is continued in the called method immediately after where the call took place.

Example with Three Methods

Here is another variation of the programme, this time using three methods. Altho there is no real need for these methods in such a small programme, large programmes are in fact composed of many small methods. It is the essential way that all code is structured.

Each of the user-defined method names, both in the call and the definition, is hilited:

- One method is *void*, which means it doesn't return a value.
- Three methods call other methods.
- The main programme consists mostly of calls to other methods.

Source Code

```
// File : methods/KmToMilesMethods. java
1 // Purpose: Converts kilometers to miles using two
methods.
// Author: Fred Swartz-placed in public domain
2 // Date : 22 Apr 2006
3 import javax. swing. *;
4 public class KmToMilesMethods {
//=========================================================
5 constants
private static final double MILES_PER_KILOMETER = 0. 621;
6 //
=========================================================
7 main
8 public static void main(String[] args) {
9 double kms = getDouble("Enter number of kilometers. ");
}
10 //=========================================================
double miles = convertKmToMi(kms);
11 convertKmToMi
// Conversion method-kilometers to miles.
displayString(kms + " kilometers is " + miles + " miles.
");
```

```
12 private static double convertKmToMi(double kilometers)
{
double miles = kilometers * MILES_PER_KILOMETER;
13 return miles;
}
1                  4                  /                  /
=========================================================
15 getDouble
16 // I/O convenience method to read a double value.
17 private static double getDouble(String prompt) {
18 String tempStr;
19 tempStr = JOptionPane. showInputDialog(null, prompt);
20 return Double. parseDouble(tempStr);
21 }
22//=======================================================
23 displayString
24 // I/O convenience method to display a string in dialog
box.
25 private static void displayString(String output) {
26 JOptionPane. showMessageDialog(null, output);
27 }
28 }
29
30
31
32
33
34
35
36
37
38
```

OVERLOADING

Here is a small programme which simply computes the average of three numbers. It uses three overloaded methods to read the numbers. For such a small programme you would not use three different methods, of course, but this shows how overloaded methods are defined and used. It's very common for one overloaded method to call another. Another variation of the programme, this time

using three methods. Altho there is no real need for these methods in such a small programme, large programmes are in fact composed of many small methods. It is the essential way that all code is structured.

Each of the user-defined method names, both in the call and the definition, is hilited:

- One method is *void*, which means it doesn't return a value.
- Three methods call other methods.
- The main programme consists mostly of calls to other methods.

Good Practices

- *Coherence:* It's important that all the methods do the "same" thing, so that the programme is human comprehensible. All methods sharing the same name should return the same value, have the same side effects, and all be either static or instance methods. The language doesn't require this, but doing otherwise is asking for trouble.
- *Call Each Other:* Because all overridden methods should be doing the same thing, it is very common for there to be calls from one to another, supplying extra default parameter values as required.
- *Default Parameter Values:* Some programming languages allow you to specify default values for parameters, and if a parameter is not supplied, the default value is used. Java doesn't have default parameters, but you can easily implement those using overloaded methods.

Example of Overloading-averaging Three Values

```
// File : methods/avg3/AvgThreeOverloaded. java
1// Description: Averages three numbers—meaningless, but
// Purpose: Show an overloaded method, getDouble, with
three
2 definitions,
// differing in the number of parameters.
3 // Issues: Input isn't checked for legality (non-null
number) because
4 // the point is to show overloading.
// Author: Fred Swartz-2007-01-11-placed in public domain
5
import javax. swing. *;
6
public class AvgThreeOverloaded {
7
/                                                          /
============================================================
8 main
public static void main(String[] args) {
9//. . . Read three numbers using the three different
methods.
// Using three different methods is only to show
10 overloading.
double n1 = getDouble();
11 double n2 = getDouble("Enter the second number. ");
double n3 = getDouble("Enter last number. ", 0. 0, 100.
0);
double average = (n1 + n2 + n3)/ 3. 0;
12 displayString("Average is " + average)
13 }
14                                                        //
============================================================
getDouble
15 // I/O convenience method to read a double value.
// This version of the getDouble method simply calls on
16 another
// version passing it a generic input message.
17 private static double getDouble() {
return getDouble("Enter a number");
18 }
19                                                        //
============================================================
getDouble
```

```
20 // I/O convenience method to read a double value given
a prompt.
// This version of getDouble displays the user supplied
21 prompt.
private static double getDouble(String prompt) {
22 String tempStr;
tempStr = JOptionPane. showInputDialog(null, prompt);
23 return Double. parseDouble(tempStr);
}
24                                                    //
============================================================
25 getDouble
// I/O convenience method to read a double value in a
range.
26 // It builds a new prompt and calls another version to
get
// the value, looping until a value in the range is found.
27 private static double getDouble(String prompt, double
low, double high) {
28 double result;
String rangePrompt = prompt + " Value must be in range "
29 + low + " to " + high;
30 //. . . Read and loop back if the number is not in the
right range.
31 do {
result = getDouble(rangePrompt);
32 } while (result < low || result > high);
33 return result;
}
34
// I/O convenience method to display a string in dialog
box.
35 String(String output)
36 private static void display
JOptionPane. showMessageDialog(null, output);
37 }
38 }
39
40
41
42
43
44
45
46
```

```
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
```

Don't Confuse Overloading and Overriding

This two terms are easily confused because they both have to do with multiple definitions of methods. Better terms would have been nice, but these are what we have. *Overloading* is making multiple method definitions which differ in the number or types of parameters, as described here. *Overriding* is redefining a method in a super class, using exactly the same number and types of parameters.

THE COMPUTER GRAPHICS PIPELINE

The process that goes into the production of a fully realised 3D movie character or environment is known by industry professionals as the "computer graphics pipeline. " Even though the process is quite complex from a technical standpoint, it's actually very easy to understand when illustrated sequentially. Think of your favourite 3D movie character.

It could be Wall-E or Buzz Lightyear, or maybe you were a fan of Po in *Kung Fu Panda*. Even though these three characters look very different, their basic production sequence is the same.

In order to take an animated movie character from an idea or storyboard drawing to a fully polished 3D rendering, the character passes through six major phases:

PRE-PRODUCTION

In pre-production, the overall look of a character or environment is conceived. At the end of pre-production, finalized design sheets will be sent to the modeling team to be developed.

- Every Idea Counts: Dozens, or even hundreds of drawings & paintings are created and reviewed on a daily basis by the director, producers, and art leads.
- Colour Palette: A character's colour scheme, or palette, is developed in this phase, but usually not finalized until later in the process.
- Concept Artists may work with digital sculptors to produce preliminary digital mock-ups for promising designs.
- Character Details are finalized, and special challenges (like fur and cloth) are sent off to research and development.

3D MODELLING

With the look of the character finalized, the project is now passed into the hands of 3D modellers. The job of a

modeller is to take a two dimensional piece of concept art and translate it into a 3D model that can be given to animators later on down the road.

In today's production pipelines, there are two major techniques in the modeller's toolset: polygonal modelling & digital sculpting.

- Each has its own unique strengths and weaknesses, and despite being vastly different, the two approaches are quite complementary.
- Sculpting lends itself more to organic (character) models, while polygonal modelling is more suited for mechanical/architectural models.

The subject of 3D modelling is far too extensive to cover in three or four bullet points, but its something we'll continue covering in depth in both the blog, and in the Maya Training series.

SHADING AND TEXTURING

The next step in the visual effects pipeline is known as shading and texturing. In this phase, materials, textures, and colours are added to the 3D model.

- Every component of the model receives a different shader-material to give it an appropriate look.
- Realistic materials: If the object is made of plastic, it will be given a reflective, glossy shader. If it is made of glass, the material will be partially transparent and refract light like real-world glass.
- Textures and colours are added by either projecting a two dimensional image onto the model, or by

painting directly on the surface of the model as if it were a canvas. This is accomplished with special software (like ZBrush) and a graphics tablet.

LIGHTING

In order for 3D scenes to come to life, digital lights must be placed in the scene to illuminate models, exactly as lighting rigs on a movie set would illuminate actors and actresses. This is probably the second most technical phase of the production pipeline (after rendering), but there's still a good deal of artistry involved.

- Proper lighting must be realistic enough to be believable, but dramatic enough to convey the director's intended mood.
- Mood Matters: Believe it or not, lighting specialists have as much, or even more control than the texture painters when it comes to a shot's colour scheme, mood, and overall atmosphere.
- Back-and-Forth: There is a great amount of communication between lighting and texture artists. The two departments work closely together to ensure that materials and lights fit together properly, and that shadows and reflections look as convincing as possible.

ANIMATION

Animation, as most of you already know, is the production phase where artists breathe life and motion into their characters.

Animation technique for 3D films is quite different than traditional hand drawn animation, sharing much more common ground with stop-motion techniques:

- Rigged for Motion: 3D characters are controlled by means of a virtual skeleton or "rig" that allows an animator to control the model's arms, legs, facial expressions, and posture.
- Pose-to-Pose: Animation is typically completed pose-to-pose. In other words, an animator will set a "key-frame" for both the starting and finishing pose of an action, and then tweak everything in between so that the motion is fluid and properly timed.

Jump over to our computer animation companion site for extensive coverage of the topic.

RENDERING AND POST-PRODUCTION

The final production phase for a 3D scene is known as rendering, which essentially refers to the translation of a 3D scene to a finalized two dimensional image. Rendering is quite technical, so we won't spend too much time on it here. In the rendering phase, all the computations that cannot be done by your computer in real-time must be performed.

This includes, but is hardly limited to the following:

- Finalizing Lighting: Shadows and reflections must be computed.
- Special Effects: This is typically when effects like depth-of-field blurring, fog, smoke, and explosions would be integrated into the scene.

- Post-processing: If brightness, colour, or contrast needs to be tweaked, these changes would be completed in an image manipulation software following render time.

4

Graphics System

Let us consider the organization of a typical graphics system we might use. As our initial emphasis will be on how the applications programmer sees the system, we shall omit details of the hardware. The model is general enough to include workstations, personal computers, terminals attached to a central time-shared computer, and sophisticated image-generation systems. In most ways, this block diagram is that of a standard computer. How each element is specialized for computer graphics will characterize this diagram as one of a graphics system, rather than one of a general-purpose computer.

THE PROCESSOR

Within the processor box, two types of processing take place. The first is picture formation processing. In this stage, the user programme or commands are processed.

The picture is formed from the elements (lines, text) available in the system using the desired attributes. Such as line colour and text font. The user interface is a part of this processing. The picture can be specified in a number of ways, such as through an interactive menu-controlled painting programme or via a C programme using a graphics library. The physical processor used in this stage is often the processor in the workstation or host computer.

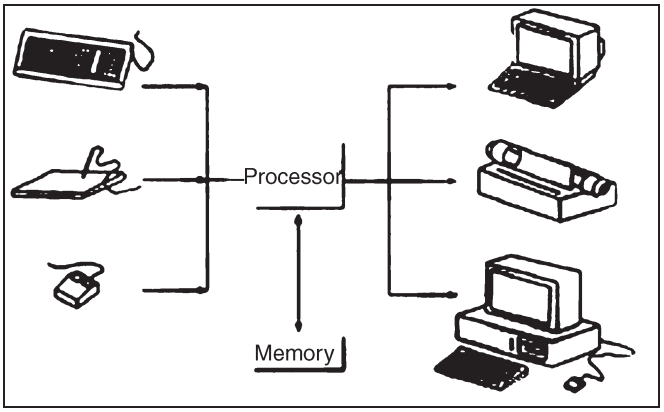


Fig. The Graphic System

The second kind of processing is concerned with the display of the picture. In a raster system, the specified primitives must be scan converted. The screen must be refreshed to avoid flicker. Input from the user might require objects to be repositioned on the display. The kind of processor best suited for these jobs is not the standard type of processor found in most computers. Instead, special boards and chips are often used. As we have already noted, one of the elements that distinguishes real-time graphics systems is their use of display processors. Since we have agreed to stay at the block-diagram level for now, however, we shall not explore these architectures in any detail until later.

MEMORY

There are often two distinct types of memory employed in graphics systems. For the processing of the user programme, the memory is similar to that of a standard computer, as the picture is formed by a standard type of arithmetic processing. Display processing, however, requires high-speed display memory that can be accessed by the display processor, and, in raster systems, memory for the frame buffer.

This display memory usually is different in both its physical characteristics and its organization from what is used by the picture processor. At this point, we need not consider details of how memory can be organized. You should be aware that the way the internals of our processor and memory boxes are organized distinguishes a slow system from a real-time picture-generating system, such as a flight simulator. However, from our present perspective, we shall emphasize that all implementations have to do the same kinds of tasks to produce output.

OUTPUT DEVICES

Our basic system has one or more output devices. As raster displays are the dominant type, we shall assume there is a raster-scan CRT on our system. We shall consider the frame buffer to be part of the display memory. In a self-contained system such as a workstation, the display is an integral part of the system, so the transfer of information from the processor to the display will happen rapidly. When the display is separate, such as with a

graphics terminal, the speed of the connection is much slower. Terminals with raster displays usually must have their own frame buffers, so the displays can be refreshed locally. In our simple system, we might also have other displays, such as a plotter, to allow us to produce hardcopy.

INPUT DEVICES

A simple system may have only a keyboard to provide whatever input is necessary. Keyboards provide digital codes corresponding to sequences of keystrokes by a user. These sequences are usually interpreted as codes for characters. If individual keystrokes or groups of keystrokes are interpreted as graphical input, the keyborad can be used as a complex input device.

For example, the "arrow" keys available on most keyboards can be used to direct the movement of a cursor on the screen. Most graphics systems will provide at least one other input device.

The most common are the mouse, the lightpen, the joystick, and the data tablet. Each can provide positional information to the system and each usually is equipped with one or more buttons to provide signals to the processor. From the programmer's perspective, there are numerous important issues with regard to the input and output devices. We must consider how the programme can communicate with these devices.

We must decide what kinds of input and output can be produced. We will be interested in how to control

multiple devices, so that we can choose a particular device for our input, and can direct our output to some group of the available output devices.

DIRECT GRAPHICS COORDINATE SYSTEMS

You can specify coordinates to IDL in one of the following coordinate systems:

DATA COORDINATES

This coordinate system is established by the most recent PLOT, CONTOUR, or SURFACE procedure. This system usually spans the plot window, the area bounded by the plot axes, with a range identical to the range of the plotted data. The system can have two or three dimensions and can be linear, logarithmic, or semi-logarithmic.

DEVICE COORDINATES

This coordinate system is the physical coordinate system of the selected plotting device. Device coordinates are integers, ranging from (0, 0) at the bottom-left corner to (V_x -1, V_y -1) at the upper-right corner. V_x and V_y are the number of columns and rows addressed by the device. These numbers are stored in the system variable !D as !D.X_SIZE and !D.Y_SIZE.

NORMAL COORDINATES

The normalized coordinate system ranges from zero (0) to one (1) over each of the three axes.

Almost all of the IDL graphics procedures accept parameters in any of these coordinate systems. Most procedures use the data coordinate system by default. Routines beginning with the letters TV are notable exceptions. They use device coordinates by default. You can explicitly specify the coordinate system to be used by including one of the keyword parameters/DATA,/DEVICE, or/NORMAL in the call.

TWO-DIMENSIONAL COORDINATE CONVERSION

The system variables !D, !P, !X, !Y, and !Z contain the information necessary to convert from one coordinate system to another. The relevant fields of these system variables, and formulae are given for conversions to and from each coordinate system.

In the following discussion, *D* is a data coordinate, *N* is a normalized coordinate, and *R* is a raw device coordinate.

The fields !D.X_VSIZE and !D.Y_VSIZE always contain the size of the visible area of the currently selected display or drawing surface. Let V_x and V_y represent these two sizes.

The field !X.S is a two-element array that contains the parameters of the linear equation, converting data coordinates to normalized coordinates. !X.S is the intercept, and !X.S is the slope. !X.TYPE is 0 for a linear *x*-axis and 1 for a logarithmic *x*-axis. The *y*- and *z*-axes are handled in the same manner, using the system variables !Y and !Z.

Also, let D_x be the data coordinate, N_x the normalized coordinate, R_x the device coordinate, V_x the device X size

(in device coordinates), and X_i = !X.S<sub< nobr=""> style ="font-style: italic" class ="cSubScript" > i (the scaling parameter). </sub<>

With the above variables defined, the linear two-dimensional coordinate conversions for the x coordinate can be written as follows:

Coordinate Conversion	***Linear***	***Logarithmic***
Data to normal	$N_x = Xo + X_1D_x$	$N_x = Xo + X_1 \log D_x$
Data to device	$Rx = V_x(X_o + X_1D_x)$	$Rx = V_x(X_o + X_1 \log D_x)$
Normal to device	$Rx = N_x V_x$	$R_x = N_x V_x$
Normal to data	$D_x = (N_x - X_o)/X_1$	$D_x = 10(N_x - X_0)/X_1$
Device to data	$Dx = (R_x/V_x - X_o)/X_1$	$Dx = 10(R_x/V_x - X_o)/X_1$
Device to normal	$N_x = R_x/V_x$	$N_x = R_x/V_x$

The *y*- and *z*-axis coordinates are converted in exactly the same manner, with the exception that there is no *z* device coordinate and that logarithmic *z*-axes are not permitted.

CONVERT_COORD Function

The CONVERT_COORD function provides a convenient means of computing the above transformations. It can convert coordinates to and from any of the above systems. The keywords DATA, DEVICE, or NORMAL specify the input system. The output coordinate system is specified by one of the keywords TO_DATA, TO_DEVICE, or TO_NORMAL.

For example, to convert the endpoints of a line from data coordinates (0, 1) to (5, 7) to device coordinates, use the following statement:

```
D = CONVERT_COORD([0, 5], [1, 7],/DATA,/TO_DEVICE)
```

On completion, the variable *D* is a (3, 2) vector, containing the *x*, *y*, and *z* coordinates of the two endpoints.

X Versus Y Plots-PLOT and OPLOT

This section illustrates the use of the basic *x* versus *y* plotting routines, PLOT and OPLOT. PLOT produces linear-linear plots by default, and can produce linear-log, log-linear, or log-log plots with the addition of the XLOG and YLOG keywords.

Data used in these examples are from a fictitious study of Pacific Northwest Salmon fisheries. In the example, we suppose that data were collected in the years 1967, 1970, and from 1975 to 1983.

The following IDL statements create and initialize the variables SOCKEYE, COHO, CHINOOK, and HUMPBACK, which contain fictitious fish population counts, in thousands, for the 11 observations:

```
SOCKEYE=[463, 459, 437, 433, 431, 433, 431, 428, 430,
431, 430]
COHO=[468, 461, 431, 430, 427, 425, 423, 420, 418, 421,
420]
CHINOOK=[514, 509, 495, 497, 497, 494, 493, 491, 492,
493, 493]
HUMPBACK=[467, 465, 449, 446, 445, 444, 443, 443, 443, 443, 445]
; Construct a vector in which each element contains
; the year of the sample:
YEAR = [1967, 1970, INDGEN(9) + 1975]
```

If you prefer not to enter the data by hand, run the batch file plot01 with the following command at the IDL prompt:

```
@plot01
```

The following IDL commands create a plot of the population of Sockeye salmon, by year:

```
PLOT, YEAR, SOCKEYE, $
```

```
TITLE='Sockeye Population', XTITLE='Year', $
YTITLE='Fish (thousands)'
```

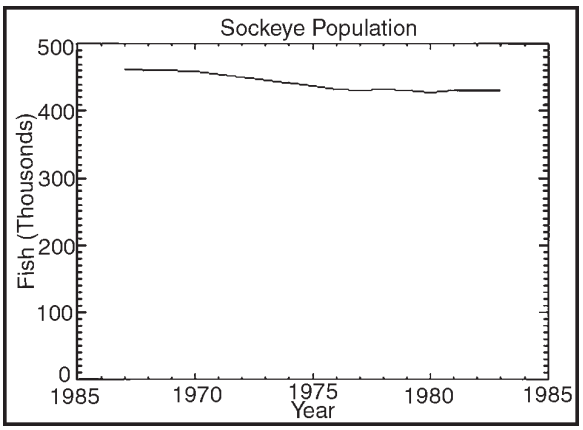
The PLOT procedure, which produces an *x* versus *y* plot on a new set of axes, requires one or two parameters: a vector of *y* values or a vector of *x* values followed by a vector of *y* values.

The first attempt at making a plot produces. Note that the three titles, defined by the keywords TITLE, XTITLE, and YTITLE, are optional.

Axis Scaling

The fluctuations in the data are hard to see because the scores range from 428 to 463, and the plot's *y*-axis is scaled from 0 to 500.

Two factors cause this effect. By default, IDL sets the minimum *y*-axis value of linear plots to zero if the *y* data are all positive.

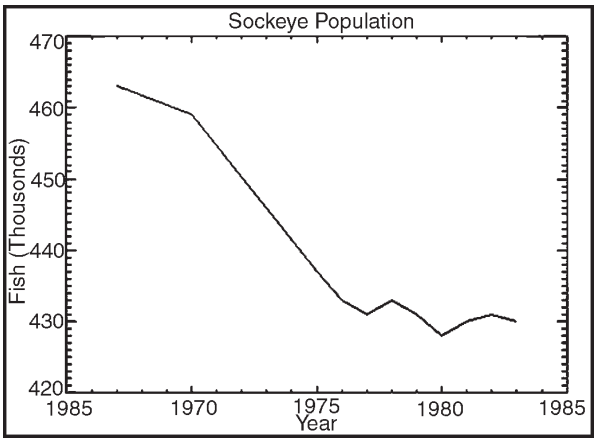


The maximum axis value is automatically set by IDL from the maximum *y* data value. In addition, IDL attempts to produce from three to six tick-mark intervals that are

in increments of an integer power of 10 times 2, 2.5, 5, or 10. In this example, this rounding effect causes the maximum axis value to be 500, rather than 463. The YNOZERO keyword parameter inhibits setting the *y*-axis minimum to zero when given positive, nonzero data. The data plotted using this keyword.

The *y*-axis now ranges from 420 to 470, and IDL creates tick-mark intervals of 10.

```
;Define variables:
@plot01
PLOT, YEAR, SOCKEYE,/YNOZERO, $
 TITLE='Sockeye Population', XTITLE='Year', $
 YTITLE='Fish (thousands)'
```



Multiline Titles

The graph-text positioning command !C, starts a new line of text output. Titles containing more than one line of text are easily produced by separating each line with this positioning command.

The main title could have been displayed on two centred lines by changing the keyword parameter TITLE to the following statement:

```
TITLE = 'Sockeye!CPopulation'
```

Note: When using multiple line titles you may find that the default margins are inadequate, causing the titles to run off the page. In this case, set the [XY]MARGIN keywords or increase the values of !X.MARGIN or !Y.MARGIN.

Range Keyword

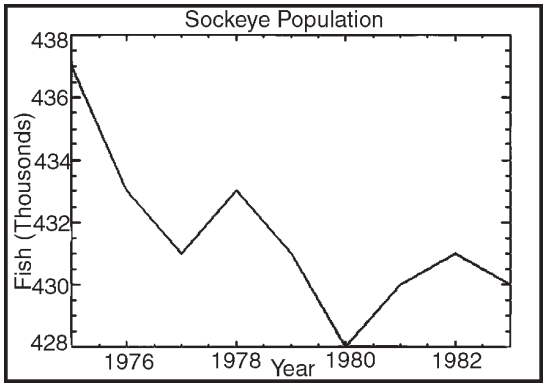
The range of the *x*, *y*, or *z* axes can be explicitly specified with the [XYZ] RANGE keyword parameter. The argument of the keyword parameter is a two-element vector containing the minimum and maximum axis values.

IDL attempts to produce even tick intervals, and the axis range selected by IDL may be slightly larger than that given with the RANGE keyword. To obtain the exact specified interval, set the axis style parameter to one (YSTYLE = 1).

The effect of the YNOZERO keyword is identical to that obtained by including the keyword parameter YRANGE = [MIN(Y), MAX(Y)] in the call to PLOT. You can make/ YNOZERO the default in subsequent plots by setting bit 4 of !Y.STYLE to one (!Y.STYLE = 16).

The STYLE field of the axis system variables !X, !Y, and !Z. Briefly: Other bits in the STYLE field extend the axes by providing a margin around the data, suppress the axis and its notation, and suppress the box-style axes by drawing only left and bottom axes.

For example, to constrain the x-axis to the years 1975 to 1983, the keyword parameter XRANGE = [1975, 1983] is included in the call to PLOT.



Note that the *x*-axis actually extends from 1974 to 1984, as IDL elected to make five tick-mark intervals, each spanning two years. The *x*-axis style is set to one, the plot will exactly span the given range.

The call combining all these options is as follows:

```
; Define variables:
@plot01
PLOT, YEAR, SOCKEYE,/YNOZERO, $
 TITLE='Sockeye Population', XTITLE = 'Year', $
 YTITLE = 'Fish (thousands)', XRANGE = [1975, 1983],/
XSTYLE
```

Note: The keyword parameter syntax/XSTYLE is synonymous with the expression XSTYLE = 1. Setting a keyword parameter to 1 is often referred to as simply setting the keyword.

Overplotting

Additional data can be added to existing plots with the OPLOT procedure. Each call to PLOT establishes the plot window (the rectangular area enclosed by the axes), the plot region (the box enclosing the plot window and its annotation), the axis types (linear or log), and the scaling. This information is saved in the system variables !P, !X, and !Y and used by subsequent calls to OPLOT.

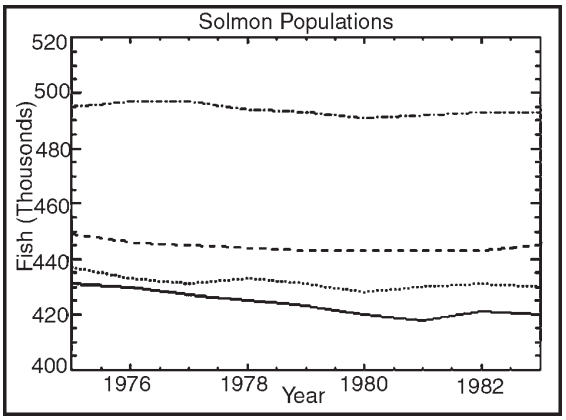
Frequently, the colour index, line style, or line thickness parameters are changed in each call to OPLOT to distinguish the data sets.

The *IDL Reference Guide* contains a table describing the line style associated with each index.

A plot showing all four data sets. Each data set except the first was plotted with a different line style and was produced by a call to OPLOT.

In this example, an (11, 4) array called ALLPTS is defined and contains all the scores for the four categories using the array concatenation operator.

Once this array is defined, the IDL array operators and functions can be applied to the entire data set, rather than explicitly referencing the particular sample.



First, we define an *n*-by-4 array containing all four sample vectors. (This array is also defined by the plot01 batch file.)

```
ALLPTS = [[COHO], [SOCKEYE], [HUMPBACK], [CHINOOK]]
```

The plot in the preceding figure was produced with the following statements:

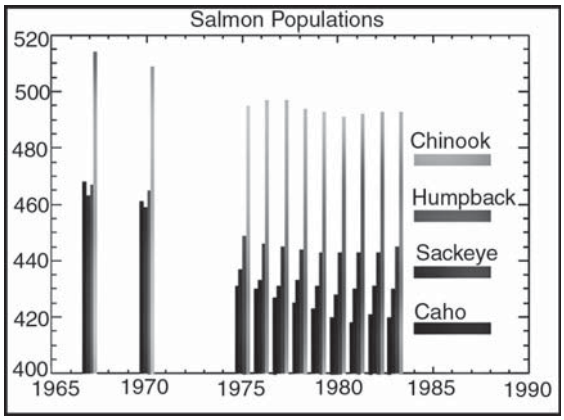
```
; Define variables:
@plot01
```

```
; Plot first graph. Set the y-axis min and max
; from the min and max of all data sets. Default linestyle
is 0.
PLOT, YEAR, COHO, YRANGE = [MIN(ALLPTS), MAX(ALLPTS)], $
 TITLE='Salmon Populations', XTITLE = 'Year', $
 YTITLE = 'Fish (thousands)', XRANGE = [1975, 1983], $
 /XSTYLE
; Loop for the three remaining scores, varying the
linestyle:
FOR I = 1, 3 DO OPLOT, YEAR, ALLPTS[*, I], LINE = I
```

Bar Charts

Bar (or box) charts are used in business-style graphics and are useful in comparing a small number of measurements within a few discrete data sets. Although not designed as a tool for business graphics, IDL can produce many business-style plots with little effort.

The following example produces a box-style chart showing the four salmon populations as boxes of differing colours or shading. You do not need to type these commands in yourself; they are collected in the files plot05.pro, which contains the two procedures, and plot06, which contains the found in the examples/doc subdirectory of the IDL distribution.



First, we define a procedure called BOX, which draws a box given the coordinates of two diagonal corners:

```
; Define a procedure that draws a box, using POLYFILL,
; whose corners are (X0, Y0) and (X1, Y1):
PRO BOX, X0, Y0, X1, Y1, colour
; Call POLYFILL:
 POLYFILL, [X0, X0, X1, X1], [Y0, Y1, Y1, Y0], COL =
colour
END
Next, create a procedure to draw the bar graph:
PRO BARGRAPH, minval
; Define variables:
 @plot01
; Width of bars in data units:
 del = 1./5.
; The number of colours used in the bar graph is
; defined by the number of colours available on your
system:
 ncol=!D.N_COLORS/5
; Create a vector of colour indices to be used in this
procedure:
 colours = ncol*INDGEN(4)+ncol
; Loop for each sample:
 FOR iscore = 0, 3 DO BEGIN
; The y value of annotation. Vertical separation is 20
data
; units:
 yannot = minval + 20 *(iscore+1)
; Label for each bar:
 XYOUTS, 1984, yannot, names[iscore]
; Bar for annotation:
 BOX, 1984, yannot - 6, 1988, yannot - 2, colours[iscore]
; The x offset of vertical bar for each sample:
 xoff = iscore * del - 2 * del
; Draw vertical box for each year's sample:
 FOR iyr=0, N_ELEMENTS(year)-1 DO $
   BOX, year[iyr] + xoff, minval, $
   year[iyr] + xoff + del, $
   allpts[iyr, iscore], $
   colours[iscore]
 ENDFOR
END
```

Enter the following at the IDL prompt to compile these two procedures from the IDL distribution:

```
.run plot5.pro
```

To create the bar graph on your screen, enter the following commands.

```
; Load a colour table:
LOADCT, 39
```

As in the previous example, the PLOT procedure is used to draw the axes and to establish the scaling using the NODATA keyword.

```
PLOT, year, CHINOOK, YRANGE = [MIN(allpts),MAX(allpts)],
$
 TITLE = 'Salmon Populations',/NODATA, $
 XRANGE = [year[0], 1990]
; Get the y value of the bottom x-axis:
minval = !Y.CRANGE[0]
; Create the bar chart:
BARGRAPH, minval
```

MECHANISMS AND METHODS

Scan conversion involves changing the picture information data rate and wrapping the new picture in appropriate synchronization signals.

There are two distinct methods for changing a picture's data rate:

- Analog Methods (Non- retentive, memory-less or real time method)

This conversion is done using large numbers of delay cells and is appropriate for analog video.

- Digital methods (Retentive or buffered method).

In this method, a picture is stored in a line or frame buffer with n1 speed (data rate) and is read with n2 speed, several picture processing techniques are applicable when the picture is stored in buffer memory including kinds of

interpolation from simple to smart high order comparisons, motion detection and … to improve the picture quality and prevent the conversion artifacts.

How to Realize

The process in practice is applicable only using integrated circuits in LSI and VLSI scales.

Timing, interference between digital and analog signals, clocks, noise and exact synchronization have important roles in the circuit. Digital conversion method needs the analog video signal to be converted to digital data at the first step.

A scan converter can be made in its basic structure using some high speed integrated circuits as a circuit board however there are some integrated circuits which perform this function plus other picture processing functions like scissoring, change of aspect ratio and … an easy to use example was SDA9401

Some examples:

A VGA to TV scan converter box like this turns enhanced-definition or high-definition signals into standard-definition signals.

Up conversion (interpolation):

- In many LCD monitors there is a native picture mode, however the monitor can display different graphical modes using a scan converter.
- In a 100 Hz/120Hz analog TV, there is a scan converter circuit which converts the vertical frequency (refresh rate) from standard 50/60Hz to 100/120Hz to achieve a low level of flicker which is important in large screen (high inch) TVs.
- An external TV card receives the TV signals and converts them to VGA or SVGA format to display on monitor.

Down conversion (decimation):

- Many graphic cards have output for standard-definition television. Here there is a conversion from computer graphical modes to TV standard formats.
- Other graphic cards lack an SDTV output, but their VGA outputs can still be connected to an SDTV through an external scan converter (pictured).

Scan conversion serves as a bridge between TV and computer graphics technology.

GRAPHIC CARD

A graphics card is the component in your computer that handles generating the signals that are sent to the monitor or "graphics". It is responsible for generating all the text and pictures that are displayed on your screen. It is called a "card" because most PCs will have a physical card that

is inserted in a PCI slot on the motherboard. Some motherboards have built-in graphics cards with is something of a misnomer since, it is built in as part of the motherboard and no longer a separate "card". 2D, or two dimensional graphics are the kind of graphics displayed when you use a web browser, check e-mail or work on a spreadsheet. For 2D graphics the major factors are resolution and refresh rate. Resolution determines how many little dots are used to draw the image on the screen. For example, 640 × 480 means that the whole screen is drawn using 307,200 little dots in 640 columns and 480 rows. The more dots that are used, the finer the detail. Thus, higher resolutions provide for great detail and image quality.

Another factor is colour depth. It expressed as a third parameter such as 640 × 480 × 256. This means 640 columns x 480 rows x 256 colours. Colour depth is usually a number that is 2 raised to the power of a multiple of 8 up to 32. *i.e.*, 2^8, 2^16, 2^24 or 2^32... or 256, 65,536, 16M or 4G colours. Obviously, the more colours the great the detail again. Finally refresh refers to how many times a second the image on the screen is redrawn. 60Hz means that the image on the screen is drawn 60 times every second. This becomes important in fast moving video games where the action needs to look really. Also, for CRT type monitors refresh rates of 60Hz and less tend to have a noticeable flicker even on stationary images. This flicker can lead to headaches for many people. Higher refresh rates are better. Of course, having a graphics card with the higher possible resolution and refresh rate doesn't do

much good if your monitor doesn't also support this capabilities. 3D or three dimensional graphics are what all first-person-shooter type games use.

Of course, current monitor technology still only really displays a 2D image, but the player is immersed in a landscape where they can moved their characters head and objects in all directions and move around within this world. This type of display capabilities requires some pretty intense mathematical calculation to be done very fast. The value of a good 3D graphics card is that it offloads most of this work from the computer's main processor and a specialized processor on the graphics card handles these calculations. This allows for faster, slicker looking graphics. Also, newer 3D cards handle all kinds of additional functions that gives surfaces texture, make water transparent, *etc.*

Scan Converting a Line

You know that a line in computer graphics typically refers to a line segment, which is a portion of a straight line that extends indefinitely in opposite directions. You can define a line by its two end points and by the line equation y = mx + c, where m is called the slope and c the y intercept of the line. Let the two end points of a line be P1(x1, y1) and P2(x2, y2). The line equation describes the coordinates of all the points that lie between the two endpoints.

A simple approach to scan convert a line is to first scan convert P1 and P2 to pixel coordinates (x1', y1') and (x2', y2') respectively. Then let us set m = (y2'- y1')/(x2'- x1')

and b = y1'-mx1'. Find |m| and if |m| ? 1, then for every integer value of x between and excluding x1' and x2', calculate the corresponding value of y using the equation and scan convert (x, y). If |m| >1, then for every integer value of y between and excluding y1' and y2', calculate the corresponding value of x using the equation and scan convert (x, y).

Design Criteria of Straight Lines

From geometry we know that a line, or line segment, can be uniquely specified by two points. From algebra we also know that a line can be specified by a slope, usually given the name m and a y-axis intercept called b. Generally in computer graphics, a line will be specified by two endpoints. But the slope and y-intercept are often calculated as intermediate results for use by most line-drawing algorithms.

The goal of any line drawing algorithm is to construct the best possible approximation of an ideal line given the inherent limitations of a raster display. Before discussing specific line drawing algorithms, it is useful to consider general requirements for such algorithms. The desirable characteristics needed for these lines.

The primary design criteria are as follows:

- Straight lines appear as straight lines
- Straight lines start and end accurately
- Displayed lines should have constant brightness along their length, independent of the line length and orientation.
- Lines should be drawn rapidly.

Scan Converting a Point

A mathematical point (x, y) where x and y are real numbers within an image area, needs to be scan converted to a pixel at location (x', y'). This may be done by making x' to be the integer part of x, and y' to be the integer part of y. In other words, x' = floor(x) and y' = floor(y), where function floor returns the largest integer that is less than or equal to the arguments. Doing so in essence places the origin of a continuous coordinate system for (x, y) at the lower left corner of the pixel grid in the image space.

All the points that satisfy $x' \geq x \geq x' + 1$ and $y' \geq y \geq y' + 1$ are mapped to pixel (x', y'). Let us take for example a point P1(1.7, 0.8). It will be represented by pixel (1, 0). Points P2 (2.2, 1.3) and P3(2.8, 1.9) are both represented by pixel (2, 1). Let us take another approach to align the integer values in the coordinate system for (x, y) with the pixel coordinates. Here we can convert (x, y) by making x' = floor(x + 0.5) and y' = floor(y + 0.5). This approach places the origin of the coordinate system for (x, y) at the centre of pixel(0, 0). All points that satisfy $x' - 0.5 \geq x \geq x' + 0.5$ and $y' - 0.5 \geq y \geq y' + 0.5$ are mapped to pixel (x', y'). This means that points P1 and P2 are now both represented by pixel (2, 1), whereas point P3 is represented by pixel (3, 2).

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.

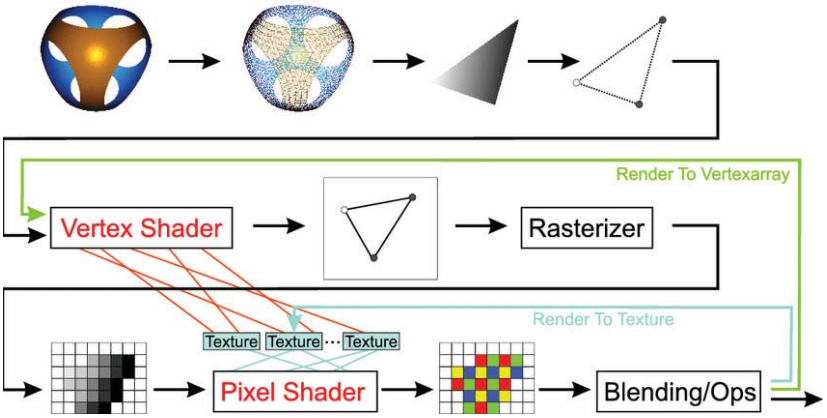
The majority of these products call themselves, "Mind Mapping" software.

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.

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.

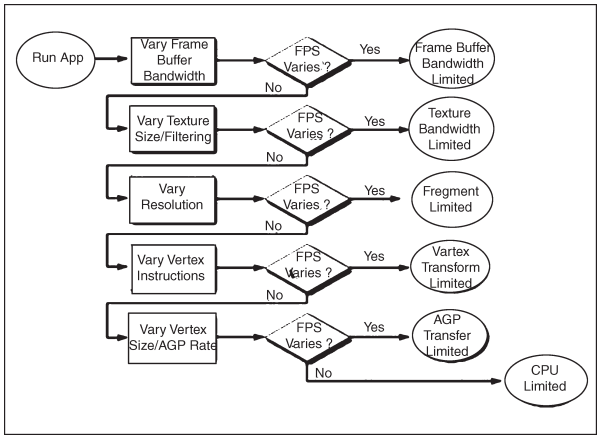


Fig. Bottleneck Flowchart

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.