Swansea University E-Theses

The development of problem solving environments for computational engineering.

Jones, Jason William

How to cite:

Jones, Jason William (2003) *The development of problem solving environments for computational engineering..* thesis, Swansea University.
http://cronfa.swan.ac.uk/Record/cronfa42503

Use policy:

This item is brought to you by Swansea University. Any person downloading material is agreeing to abide by the terms of the repository licence: copies of full text items may be used or reproduced in any format or medium, without prior permission for personal research or study, educational or non-commercial purposes only. The copyright for any work remains with the original author unless otherwise specified. The full-text must not be sold in any format or medium without the formal permission of the copyright holder. Permission for multiple reproductions should be obtained from the original author.

Authors are personally responsible for adhering to copyright and publisher restrictions when uploading content to the repository.

Please link to the metadata record in the Swansea University repository, Cronfa (link given in the citation reference above.)

http://www.swansea.ac.uk/library/researchsupport/ris-support/

School of Engineering
University of Wales Swansea

THE DEVELOPMENT OF PROBLEM SOLVING ENVIRONMENTS FOR COMPUTATIONAL ENGINEERING

JASON WILLIAM JONES
BSc.

Thesis submitted to the University of Wales in candidature for the degree of
Doctor of Philosophy

December 2002

ProQuest Number: 10801733

All rights reserved

INFORMATION TO ALL USERS
The quality of this reproduction is dependent upon the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript
and there are missing pages, these will be noted. Also, if material had to be removed,
a note will indicate the deletion.

ProQuest 10801733

Published by ProQuest LLC (2018). Copyright of the Dissertation is held by the Author.

All rights reserved.
This work is protected against unauthorized copying under Title 17, United States Code
Microform Edition © ProQuest LLC.

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 – 1346

UNIVERSITY OF
LIBRARY
WALES SWANSEA

Declaration

This work has not previously been accepted in substance for any degree and is not concurrently submitted in candidature for any degree.

Signed ---- .... (Candidate)

Date ....15-12-02..

Statement 1

This thesis is the result of my own work/investigation except where otherwise stated. Other sources have been acknowledged by footnotes giving explicit references. A bibliography is appended.

Signed .... (Candidate)

Date ....15-12-02..

Statement 2

I hereby give consent for my thesis, if accepted, to be available for photocopying and for inter-library loans, and for the title and summary to be made available to outside organisations.

Signed .... (Candidate)

Date ....15-12-02..

ACKNOWLEDGEMENTS

First and foremost I would like to thank Professor Nigel Weatherill, Professor Ken Morgan and Dr. Oubay Hassan for their guidance and friendship throughout my time in the group, as well as allowing me to join their research group in the first place. I have always felt it a great privilege to be part of such a high profile group in such a world class department.

The pleasant atmosphere in the department is, of course, also a product of all of the other members, some of which have moved onto pastures new. In particular, I would like to thank my good friends Ed Turner-Smith, Mike Sotirakos, Peter Brookes, Daniel van der Leer and Ben Larwood with whom I have had many fun times both in and out of work; and Lynette Jones and Anne Davies whose friendship I will always treasure.

I would also like to thank Yoon K Ho, Andy Wood, Peter Stow and Les Harper, from Rolls-Royce plc; David Rowse, from BAE Systems; and Luciano Fornasier and Herbert Rieger, from EADS. Their enthusiasm, input and continuous testing of the two environments has made them what they are today.

Of course, I would like to thank my parents whose continuous support and encouragement throughout my life has enabled me to be where I am today.

And last, but not in any way least, I would also like to thank my friends outside the University that have made my time in Swansea so enjoyable.

SUMMARY

This thesis presents two Problem Solving Environments that enable engineers in industry to utilise complex computational simulation algorithms during their design processes. The work addresses the issues of allowing the end user to interact with the algorithms in a user-friendly manner through the use of graphical user interface design and advanced computer graphics. Throughout this thesis major emphasis is placed on being able to tackle a wide range of problem sizes from routine to grand challenge simulations through the use of parallel computing hardware. The effectiveness of both the environments in their domain is demonstrated using a series of examples.

1. **CHAPTER 1. INTRODUCTION 10**
1.1. WHAT IS A PROBLEM SOLVING ENVIRONMENT (PSE)? 10
1.2. REVIEW OF PROBLEM SOLVING ENVIRONMENTS 11
1.2.1. Pre-processors to Conventional Programming Languages 12
1.2.2. New Programming Languages 12
1.2.3. Graphical PSE's for Specific Mathematical Domains 13
1.2.4. Visual Programming Environments 14
1.2.5. Graphical PSE's for Specific Applications 15
1.2.6. Futuristic PSE's 16
1.3. THE PROBLEM DOMAIN: COMPUTATIONAL SIMULATION 16
1.3.1. Mesh Generation 17
1.3.2. Computational Analysis 17
1.3.3. Mesh Refinement / Adaptation 17
1.4. REQUIREMENTS FOR A COMPUTATIONAL SIMULATION PSE 18
1.4.1. Geometry Preparation 19
1.4.2. Mesh Density Specification 21
1.4.3. Mesh Quality Evaluation and Repair 22
1.4.4. Boundary Condition Specification 22
1.4.5. Solver Monitoring 23
1.4.6. Solution Visualisation 23
1.4.7. Mesh Refinement Control 23
1.5. LAYOUT OF THESIS 23

2. **CHAPTER 2. TECHNICAL BACKGROUND 25**
2.1. BACKGROUND TO THREE-DIMENSIONAL GRAPHICS 25
2.1.1. Breakdown of the Rendering Process 25
2.1.2. Examples of Software Libraries for 3D Graphics 29
2.2. BACKGROUND TO PARALLEL COMPUTING 30
2.2.1. Shared Memory Parallel Architectures 30
2.2.2. Distributed Memory Parallel Architectures 31
2.2.3. Hybrid Distributed-Shared Memory Parallel Architectures 32
2.2.4. The Message Passing Programming Model 33
2.3. SUMMARY 34

3. **CHAPTER 3. PROMPT – AN IMPLEMENTATION OF A PROBLEM SOLVING ENVIRONMENT 35**
3.1. REQUIREMENTS 35
3.1.1. Current Status at Rolls-Royce (circa 1995) 36
3.1.2. Aims of the PROMPT Environment 38
3.2. SCOPE AND CONTEXT OF PROMPT 38
3.3. THE ARCHITECTURE OF PROMPT 40
3.3.1. The Structure of PROMPT 40
3.3.2. The Communication Mechanism used within PROMPT 41
3.3.3. Module Initiation and Termination 43
3.4. GLOBAL DATA STRUCTURES USED WITHIN PROMPT 45
3.4.1. Structured Curvilinear Grids (Single Block) 45
3.4.2. Structured Curvilinear Grids (Multi-Block) 46
3.4.3. Structured Curvilinear Grids with Local Refinement 49
3.4.4. Unstructured Hybrid Grids 51
3.5. THE VISUALISATION AND CONTROL MODULE 53
3.5.1. The Visualisation Window 53
3.5.2. Region and View Configurations 54
3.5.3. On-Screen Manipulation of Data 54
3.5.4. Feature Selection 56

3.5.5. *Feature Selection Algorithm* ... 57
3.5.6. *The Pull-down and Pop-up Menus* ... 60
3.5.7. *The Selection Gizmo Panel* ... 63
3.5.8. *The Colour Editor Gizmo Panel* ... 64
3.5.9. *The Appearance Gizmo Panel* ... 66
3.5.10. *The Lighting / Material Gizmo Panel* ... 68
3.5.11. *The Clipping Plane Gizmo Panel* ... 69
3.5.12. *The Clipping Plane Algorithm* ... 71
3.5.13. *The Print Gizmo Panel* ... 72
3.6. THE TASK DATABASE MODULE ... 73
3.6.1. *The Task Database Window* ... 73
3.6.2. *Loading Data Files into PROMPT* ... 74
3.6.3. *Saving Data Files to the Database* ... 75
3.6.4. *Deleting Data Files from the Database* ... 76
3.6.5. *Attaching User Comments to Data Files in the Database* ... 77
3.6.6. *Importing External Files into the Database* ... 78
3.6.7. *The Task Database Importing Mechanism* ... 80
3.7. MESH ANALYSIS ... 83
3.7.1. *The Mesh Analysis Window* ... 83
3.7.2. *Performing a Mesh Analysis* ... 83
3.7.3. *Fixing Areas of Poor Quality* ... 87
3.8. THE BOUNDARY CONDITION SPECIFICATION PANEL ... 88
3.8.1. *The Boundary Condition Window* ... 89
3.8.2. *Boundary Condition Definition for Structured Meshes* ... 90
3.8.3. *Boundary Condition Definition for Unstructured Meshes* ... 94
3.9. THE SOLVER EXECUTION PANEL ... 94
3.9.1. *The Solver Execution Panel Appearance* ... 95
3.9.2. *Solver Execution Mechanism* ... 97
3.10. SOLUTION VISUALISATION AND POST-PROCESSING ... 100
3.11. CONCLUSIONS AND EXAMPLE TEST-CASES USING PROMPT ... 101
3.11.1. *Agard B4 Test Case* ... 101
3.11.2. *Generic Engine for a Vertical Take-off Aircraft* ... 103
3.11.3. *Conclusions* ... 105

4. **CHAPTER 4. PSUE II – A PARALLEL PROBLEM-SOLVING ENVIRONMENT ... 107**

4.1. INTRODUCTION ... 107
4.2. CONTEXT OF PSUE II WITHIN THE JULIUS PROJECT ... 107
4.3. THE REQUIREMENT FOR A PARALLEL ENVIRONMENT ... 109
4.3.1. *An Example: The Equation Solver* ... 110
4.3.2. *An Initial Parallel Problem Solving Environment* ... 112
4.4. DISTRIBUTING THE DATA-SETS ... 114
4.4.1. *Sequential Geometry Format* ... 114
4.4.2. *Sequential Surface Mesh Format* ... 116
4.4.3. *Sequential Volume Mesh Format* ... 117
4.4.4. *Sequential Solution Format* ... 117
4.4.5. *Partitioned Geometry Format* ... 118
4.4.6. *Partitioned Surface Mesh Format* ... 118
4.4.7. *Partitioned Volume Mesh Format* ... 119
4.4.8. *Partitioned Solution Format* ... 122
4.5. SUMMARY ... 122

5. **CHAPTER 5. THE IMPLEMENTATION OF THE ENVIRONMENT (PSUE II V1.0) ... 123**

5.1. VISUALISATION AND INTERACTION ISSUES ... 123
5.1.1. *The Visualisation Pipe-Line* ... 123
5.1.2. *Distributing the Visualisation Pipe-Line* ... 125

5.2. INTERNAL DATA COMMUNICATION SYSTEM 130
5.2.1. UNIX Socket Transfer (TCP / IP) 131
5.2.2. UNIX Socket Transfer (UDP / IP) 131
5.2.3. MPI (Message Passing Interface) 131
5.2.4. PVM (Parallel Virtual Machine) 132
5.3. THE CONTROL STRUCTURE OF THE ENVIRONMENT 132
5.3.1. The Two Types of Communication 134
5.4. SUMMARY 134

6. CHAPTER 6. PSUE II V2.0 – AN IMPROVED ARCHITECTURE 136

6.1. IMPROVING FLEXIBILITY OF USE 136
6.1.1. The CORBA Architecture 137
6.1.2. The Use of CORBA within the PSUE II v2.0 141
6.2. REDUCING NETWORK COMMUNICATION 142
6.2.1. The Mesh Management Class Hierarchy 147
6.2.2. Mesh Manager Object 147
6.2.3. Mesh Server Object 147
6.2.4. Mesh Volume Object 148
6.2.5. 2D Mesh Object 149
6.2.6. Render Object 149
6.2.7. Co-operation between the Objects 149
6.3. IMPROVING LOAD BALANCING 150
6.4. INCREASING THE PERFORMANCE OF VOLUME DATA SET TRAVERSAL 154
6.5. INCREASING RENDERING SPEEDS 158
6.6. THE INTEGRATION OF THIRD-PARTY APPLICATIONS 160
6.6.1. Stage 1 – Application Execution 160
6.6.2. Stage 2 – Data File Transferral 162
6.6.3. Stage 3 – PSUE II and Application Interaction at run-time 163
6.7. SUMMARY 164

7. CHAPTER 7. THE FUNCTIONALITY OF THE PSUE II V2.0 165

7.1. THE MAIN DISPLAY 165
7.2. THE NESTED TOOLBARS 166
7.2.1. The PSUE II Nested Toolbar 167
7.3. THE 'CONFIGURATION' TOOLBAR 169
7.4. THE 'GEOMETRY' TOOLBAR 172
7.4.1. The 'Geometry Edit' Sub-Toolbar 175
7.5. THE 'SOURCES' TOOLBAR 177
7.6. THE 'MESH' TOOLBAR 178
7.7. THE 'BOUNDARY CONDITIONS' TOOLBAR 185
7.8. THE 'SOLUTION' TOOLBAR 186
7.9. THE 'POST-PROCESSING' TOOLBAR 190

8. CHAPTER 8. ADDRESSING THE ISSUES OF SOFTWARE PORTABILITY 194

8.1. LANGUAGE FEATURES 195
8.1.1. Fortran 77 Pointers 195
8.1.2. Fortran 77 Subroutine and Variable Names 196
8.2. GRAPHICS 197
8.3. THREADING INTERFACE 197
8.4. INPUT / OUTPUT 198
8.4.1. Unformatted I/O between Fortran 77 and C 198
8.4.2. Portability Issues due to Big and Little Endian Computers 199
8.5. INTER-PROCESS COMMUNICATION 200
8.6. SUMMARY 200

9. CHAPTER 9. EXAMPLE TEST-CASES 201

9.1. Explanation of Test-Cases 201
9.2. CFD Simulation over a Dassault Falcon 201
9.3. CFD Simulation over a complete F16 configuration 209
9.4. Pre-processing and Post-processing of a Grand-Challenge Simulation over a Dassault Falcon 217
9.5. Summary of Test Cases 221

10. CHAPTER 10. CONCLUSIONS AND FUTURE RESEARCH 223

10.1. Conclusions 223
10.2. Future Research 226

11. CHAPTER 11. BIBLIOGRAPHY 230

12. APPENDIX A. EQUATION EDITOR – EQUATE 242

A.1. Definition of a Generic Mathematical Expression in EQUATE 242
A.2. EQUATE Syntax and Semantics 243
A.3. Syntax and Semantics of Expressions 244
A.4. Syntax and Semantics of Operators 245
A.5. Syntax and Semantics of Functions 246
A.6. Syntax and Semantics of Variables 247
A.7. Syntax and Semantics of Constants 249
A.8. Some Implementation Details 249
A.9. Performance Figures 249

Figure 1 – A short ELLPACK program 12
Figure 2 – A typical session in MatLab 13
Figure 3 – Typical User Sessions in LSA and PDELab 14
Figure 4 – Typical Maps in AVS and IRIX Explorer 15
Figure 5 – Overlapping Surfaces 20
Figure 6 – Inter-Surface Gaps 20
Figure 7 – Sliver Surface at Wing Tip 20
Figure 8 – Sliver Surface on Fuselage 20
Figure 9 – Complex Geometry with Sources 21
Figure 10 – Mesh Smoothing 22
Figure 11 – Element Removal 22
Figure 12 – Face Swapping 22
Figure 13 – Non Planar Polygon transformed to a Self Intersecting Polygon 25
Figure 14 – The Vertex Transformation Pipeline 26
Figure 15 – Translation, Scaling and Rotation Matrices 27
Figure 16 – The Orthographic Projection Matrix 27
Figure 17 – The Perspective Projection Matrix 27
Figure 18 – Flat, Gouraud and Phong Shading Models 28
Figure 19 – The Shared Memory Architecture 31
Figure 20 – SMP Configurations from the Leading Vendors 31
Figure 21 – The Distributed Memory Architecture 32
Figure 22 – The ccNUMA Distributed-Shared Memory Architecture 33
Figure 23 – Memory Hierarchies of RISC and ccNUMA Architectures 33
Figure 24 – Influences on the Design of Military Nozzle / After-Bodies 36
Figure 25 – Thrust Vectoring Technologies 36
Figure 26 – An Example of the Solver Control Files (c.1995) 37
Figure 27 – Logical Processes performed within the PROMPT Environment 39
Figure 28 – The Options contained within PROMPT 39
Figure 29 – The Architecture of PROMPT 41
Figure 30 – Flattening of a typical hierarchical data structure 42
Figure 31 – Module Initiation within PROMPT 44
Figure 32 – The Top-Level Mesh Data Structure 45
Figure 33 – Data Structure for a Single-Block Structured Mesh 45
Figure 34 – An example of a Single-Block Structured Mesh with *Dead Nodes* 46
Figure 35 – Data Structure for a Single-Block Structured Mesh with *Dead Nodes* 46
Figure 36 – An example of a Multi-Block Mesh with Single Face Matching 47
Figure 37 – An example of a Multi-Block Mesh with Multiple Face Matching 47
Figure 38 – Data Structure for a Multi-Block Mesh with *Dead Nodes* 47
Figure 39 – Block and Face Coordinate Axes 48
Figure 40 – Numbering Convention for Block-Block Interfaces 49
Figure 41 – A Structured Mesh with Hanging Nodes (and *Dead Nodes*) 50
Figure 42 – Data Structure for a Multi-Block Mesh with Hanging and Dead Nodes 50
Figure 43 – The (i,j,k) structure of a mesh with hanging nodes 51
Figure 44 – The contents of the *nodeConn* array for the hanging nodes 51
Figure 45 – Data Structure for an Unstructured Hybrid Mesh 52
Figure 46 – Node Numbering Conventions used for the Unstructured Element Types 52
Figure 47 – A typical example of the Visualisation Window 53
Figure 48 – Possible Region Configurations 54
Figure 49 – The Selection Gizmo 57
Figure 50 – Point outside the polygon 58
Figure 51 – Point inside the polygon 58
Figure 52 – Node counted as one intersection 58
Figure 53 – Node counted as zero or two intersections 58
Figure 54 – The Pull-down menu hierarchy 60
Figure 55 – The Information Panels 60

FIGURE 56 – THE POP-UP MENU HIERARCHY 61
FIGURE 57 – AN EXAMPLE OF RESIZING REGIONS 62
FIGURE 58 – THE SELECTION GIZMO FOR UNSTRUCTURED MESHES 63
FIGURE 59 – THE SELECTION GIZMO FOR STRUCTURED, MULTI-BLOCK MESHES....63
FIGURE 60 – THE COLOUR EDITOR GIZMO PANEL....64
FIGURE 61 – THE RGB COLOUR SPACE....65
FIGURE 62 – THE HSV COLOUR SPACE....65
FIGURE 63 – THE APPEARANCE GIZMO PANEL....66
FIGURE 64 – VARIOUS APPEARANCE GIZMO PANEL SETTINGS FOR A MESH 68
FIGURE 65 – THE LIGHTING / MATERIAL GIZMO PANEL....69
FIGURE 66 – THE CLIPPING GIZMO PANEL 69
FIGURE 67 – MANIPULATING A CLIPPING PLANE THROUGH A MESH....70
FIGURE 68 – THE SAME MESH AFTER IT HAS BEEN CLIPPED 70
FIGURE 69 – HEXAHEDRON INTERSECTING A CLIPPING PLANE....72
FIGURE 70 – THE PRINT GIZMO PANEL....72
FIGURE 71 – A STANDARD FILE SELECTION BOX USED IN THE PRINT GIZMO PANEL....73
FIGURE 72 – THE TASK DATABASE WINDOW....74
FIGURE 73 – A SCHEMATIC OF DEPENDENCIES BETWEEN DATA FILES....74
FIGURE 74 – THE DEPENDENCIES AFTER A MESH HAS BEEN SELECTED....75
FIGURE 75 – THE FINAL SET OF SELECTABLE DATA FILES....75
FIGURE 76 – THE 'SAVE DATA' PANEL 76
FIGURE 77 – DELETING DATA FILES FROM THE TASK DATABASE 77
FIGURE 78 – EDITING THE ANNOTATION OF A DATA SET 78
FIGURE 79 – THE 'IMPORT DATA' PANEL....79
FIGURE 80 – SAVING IMPORTED FILES IN THE PROMPT FORMAT....80
FIGURE 81 – EDITING THE LIST OF AVAILABLE FILE CONVERTERS 80
FIGURE 82 – A SCHEMATIC OF THE IMPORTING PROCESS....81
FIGURE 83 – THE MESH ANALYSIS PANEL 83
FIGURE 84 – THE FOUR HISTOGRAM SELECTION METHODS....85
FIGURE 85 – ZOOMING INTO THE HISTOGRAM....86
FIGURE 86 – *BAD* ELEMENTS HIGHLIGHTED IN THE VISUALISATION WINDOW....87
FIGURE 87 – EDITING A MESH PLANE....88
FIGURE 88 – THE BOUNDARY CONDITION WINDOW FOR A STRUCTURED MESH....89
FIGURE 89 – THE BOUNDARY CONDITION WINDOW FOR AN UNSTRUCTURED MESH....90
FIGURE 90 – DEFINING A REGION ON A MESH PLANE FOR A BOUNDARY CONDITION 91
FIGURE 91 – THE CFDS INLET BOUNDARY CONDITION PANEL 92
FIGURE 92 – THE CFDS OUTLET BOUNDARY CONDITION PANEL....92
FIGURE 93 – THE CFDS WALL BOUNDARY CONDITION PANEL....92
FIGURE 94 – THE CFDS FREE STREAM BOUNDARY CONDITION PANEL....92
FIGURE 95 – THE CFDS REPEAT BOUNDARY CONDITION PANEL 93
FIGURE 96 – THE CFDS SYMMETRY BOUNDARY CONDITION PANEL 93
FIGURE 97 – THE CFDS CENTRE LINE BOUNDARY CONDITION PANEL 93
FIGURE 98 – AN EXAMPLE OF A 1D PARAMETER PROFILE FOR A BOUNDARY CONDITION 94
FIGURE 99 – AN EXAMPLE OF A 2D PARAMETER PROFILE FOR A BOUNDARY CONDITION 94
FIGURE 100 – THE 'FLOW PARAMETERS' PANEL FOR THE CFDS SOLVER....95
FIGURE 101 – THE 'RUNTIME CONTROL' PANEL FOR THE CFDS SOLVER....96
FIGURE 102 – THE MIXING-LENGTH TURBULENCE MODEL PANEL....96
FIGURE 103 – THE K-ε TURBULENCE MODEL PANEL....96
FIGURE 104 – THE K-*L* TURBULENCE MODEL PANEL....97
FIGURE 105 – AN 'INITIAL GUESS' PANEL FOR CONSTANT INITIAL VALUES....97
FIGURE 106 – AN 'INITIAL GUESS' PANEL FOR PROFILES OF INITIAL VALUES 97
FIGURE 107 – A TYPICAL CONVERGENCE HISTORY LOG OF A SOLVER 98
FIGURE 108 – A TYPICAL SET OF CONVERGENCE HISTORY PLOTS FOR THE CFDS SOLVER....98
FIGURE 109 – CONNECTING PROMPT TO A SOLVER USING A *NAMED PIPE*....99
FIGURE 110 – A TYPICAL SESSION USING VISUAL 3....100
FIGURE 111 – DEFINING NEW EQUATIONS USING EQUATE 100

FIGURE 112 – ILLUSTRATIONS OF THE AGARD B4 NOZZLE 101
FIGURE 113 – STRUCTURED MESH AROUND THE AGARD B4 NOZZLE 102
FIGURE 114 – UNSTRUCTURED MESH AROUND THE AGARD B4 NOZZLE 102
FIGURE 115 – COMPARISON BETWEEN SOLVER RESULTS AND EXPERIMENTAL DATA FOR THE AGARD B4 NOZZLE 103
FIGURE 116 – THE GEOMETRY OF THE VERTICAL TAKE-OFF ENGINE 104
FIGURE 117 – THE UNSTRUCTURED MESH AROUND THE ENGINE 104
FIGURE 118 – SOLUTION CONTOURS FROM THE CINDY SOLVER 104
FIGURE 119 – PRESSURE DISTRIBUTIONS FROM THE TWO SOLVERS 105
FIGURE 120 – AN ILLUSTRATION OF THE MEMORY USAGE FOR THE VARIOUS SOLVERS 106
FIGURE 121 – A SCHEMATIC OF THE 6S ENVIRONMENT SHOWING THE DATA FLOWS 108
FIGURE 122 – A SCHEMATIC SHOWING THE LOGICAL LAYOUT OF THE 6S ENVIRONMENT 109
FIGURE 123 – ESTIMATED EXECUTION TIME FOR TYPICAL SIMULATIONS 110
FIGURE 124 – TYPICAL MESH SIZE FOR SIMULATIONS ON A COMPLETE AIRCRAFT 111
FIGURE 125 – TYPICAL SEQUENCE OF OPERATIONS WITHIN A PARALLEL SOLVER 112
FIGURE 126 – THE BOTTLE-NECK PRODUCED BY A SEQUENTIAL PROCESS 113
FIGURE 127 – THE PSE WITH NO SEQUENTIAL BOTTLENECKS 114
FIGURE 128 – A SIMPLE GEOMETRY ILLUSTRATING ITS COMPONENTS 115
FIGURE 129 – A MORE COMPLEX GEOMETRY ILLUSTRATING THE CONCEPT OF TRIMMED SURFACES 116
FIGURE 130 – THE PATH FROM THE VOLUME MESH BACK TO THE GEOMETRY VIA THE SURFACE MESH 117
FIGURE 131 – EXAMPLES OF COMPLEX AEROSPACE CONFIGURATIONS 118
FIGURE 132 – A SOLVER RUNNING WITH 4 BALANCED PARTITIONS 120
FIGURE 133 – A SOLVER RUNNING ON 4 UNBALANCED PARTITIONS 120
FIGURE 134 – COMMUNICATION STRUCTURE BETWEEN MESH PARTITIONS 121
FIGURE 135 – THE LINK BETWEEN MESH PARTITIONS AND THE ORIGINAL SURFACE MESH 122
FIGURE 136 - THE VISUALISATION PIPE-LINE 124
FIGURE 137 - THE IMAGE DATA TRANSFER METHOD 126
FIGURE 138 - THE GRAPHICS DATA TRANSFER METHOD 128
FIGURE 139 - THE GEOMETRY DATA TRANSFER METHOD 129
FIGURE 140 – POSSIBLE HARDWARE SCENARIO ON WHICH TO USE THE PSUE II 130
FIGURE 141 – THE SEQUENCE OF OPERATIONS REQUIRED TO LOAD A SOLUTION FILE 133
FIGURE 142 – MAPPING OF SOLUTION VALUES TO COLOURS 134
FIGURE 143 – REQUEST BEING SENT VIA CORBA 138
FIGURE 144 – THE STRUCTURE OF THE REQUEST BROKER INTERFACES 139
FIGURE 145 – STEPS PERFORMED FOR A METHOD INVOCATION (USING C++) 141
FIGURE 146 – THE FINAL, CORBA-BASED ARCHITECTURE OF THE PSUE II 142
FIGURE 147 – A ROUGH CUTTING PLANE 143
FIGURE 148 – A SMOOTH CUTTING PLANE 143
FIGURE 149 – THE GEOMETRIC PRIMITIVES AFFECTED BY A CUTTING PLANE 144
FIGURE 150 – POSITION OF THE CUTTING PLANE DURING FOR COLLECTION OF STATISTICS 145
FIGURE 151 – THE HIERARCHY OF CLASSES USED TO MANAGE THE MESH DATA SETS WITHIN PSUE II v2.0 146
FIGURE 152 – FACE, EDGE AND NODE NUMBERING FOR A SIMPLE 2D MESH 148
FIGURE 153 – THE TWO TYPES OF EDGE-BASED DATA STRUCTURE FOR A MESH 152
FIGURE 154 – PARTITIONER PERFORMANCE GRAPHS 153
FIGURE 155 – THE 2D MESH 154
FIGURE 156 – THE QUAD-TREE DATA STRUCTURE 154
FIGURE 157 – MISSING AN ELEMENT IN A QUAD-TREE SEARCH 155
FIGURE 158 – THE IMPROVED QUAD-TREE DATA STRUCTURE 156
FIGURE 159 – A 200 MILLION ELEMENT MESH 158
FIGURE 160 – STATISTICS FOR THE 200 MILLION ELEMENT MESH 158
FIGURE 161 – STATISTICS FOR THE RENDERING DATA FOR THE MESH 159
FIGURE 162 – COMPARISON BETWEEN SINGLE PRIMITIVES AND STRIPS 159
FIGURE 163 – STATISTICS FOR THE RENDERING DATA USING TRIANGLE-STRIPS 160
FIGURE 164 – THE MAIN PSUE II GUI 165
FIGURE 166 – MAPPING OF THE PSUE II TOOLBARS TO THE SIMULATION PROCESS 167

FIGURE 165 – THE STRUCTURE OF THE NESTED TOOLBAR IN THE PSUE II 168
FIGURE 167 – REGION LAYOUT PANEL 169
FIGURE 168 – LIGHTING AND MATERIAL PANEL 170
FIGURE 169 – VARIOUS MATERIAL PROPERTIES 171
FIGURE 170 – GENERAL APPEARANCE PANEL 172
FIGURE 171 – A FILE SELECTION PANEL 173
FIGURE 172 – ENTERING THE NUMBER OF GEOMETRY SERVERS 173
FIGURE 173 – THE PARALLEL PLATFORM PANEL 173
FIGURE 174 – THE GEOMETRY APPEARANCE PANEL 174
FIGURE 175 – GEOMETRY COLOUR PANEL 175
FIGURE 176 – OUTER BOUNDARY EDITING PANEL 176
FIGURE 177 – THE TOPOLOGY EDIT PANEL 176
FIGURE 178 – THE EDIT SOURCES PANEL 177
FIGURE 179 – DRAGGING HANDLES FOR SOURCES 178
FIGURE 180 – LOADING A PARTITIONED VOLUME MESH 179
FIGURE 181 – THE SURFACE MESH GENERATION INFORMATION PANEL 181
FIGURE 182 – SPECIFYING THE PARAMETERS FOR THE PARALLEL DELAUNAY MESH GENERATOR 182
FIGURE 183 – THE MESH APPEARANCE PANEL 182
FIGURE 184 – THE MESH COLOUR PANEL 183
FIGURE 185 – THE MESH QUALITY ANALYSIS PANEL 183
FIGURE 186 – SELECTING A RANGE OF HISTOGRAM BARS 184
FIGURE 187 – THE SAME HISTOGRAM ZOOMED INTO THE SELECTED RANGE 184
FIGURE 188 – SOME VOLUME ELEMENTS HIGHLIGHTED IN THE MESH 185
FIGURE 189 – THE BOUNDARY CONDITION EDITOR PANEL 186
FIGURE 190 – THE SOLVER CONTROL PANEL 187
FIGURE 191 – THE VARIABLE SELECTION PANEL 188
FIGURE 192 – THE DEFAULT SOLUTION-COLOUR MAPPING 189
FIGURE 193 – THE USER-DEFINED SOLUTION-COLOUR MAPPING 189
FIGURE 194 – THE USER-DEFINED SOLUTION-COLOUR MAPPING WITH CONTOURING 189
FIGURE 195 – THE CUTTING PLANE PANEL 190
FIGURE 196 – A ROUGH CUTTING PLANE 191
FIGURE 197 – A SMOOTH CUTTING PLANE 191
FIGURE 198 – THE ISO-SURFACE PANEL 192
FIGURE 199 – AN ISO-SURFACE OF MACH 1.0 OVER THE GULF-STREAM 193
FIGURE 200 – BYTE LAYOUT FOR A 32-BIT QUANTITY 199
FIGURE 201 – CREATION OF THE OUTER BOUNDARY 202
FIGURE 202 – CREATION OF THE SOURCES 203
FIGURE 203 – THE SURFACE MESH OF THE DASSAULT FALCON 204
FIGURE 204 – THE VOLUME MESH (WITH INTERFACE SURFACES) 204
FIGURE 205 – THE MESH QUALITY GRAPH OF THE FALCON MESH 205
FIGURE 206 – HIGHLIGHTING THE *FLAT* ELEMENTS WITHIN THE MESH 206
FIGURE 207 – SOLUTION COLOURS OF MACH NUMBER 207
FIGURE 208 – SOLUTION CONTOURS OF MACH NUMBER 207
FIGURE 209 – A CUTTING PLANE THROUGH THE FALCON 208
FIGURE 210 – AN ISO-SURFACE OF MACH 1.0 209
FIGURE 211 – ILLUSTRATION OF THE COMPLEXITY OF THE F16 CONFIGURATION 209
FIGURE 212 – THE F16 CONFIGURATION WITH OUTER BOUNDARY 210
FIGURE 213 – THE SOURCES USED FOR THE F16 211
FIGURE 214 – THE SURFACE MESH OF THE F16 212
FIGURE 215 – A ZOOMED VIEW OF THE F16 SURFACE MESH 212
FIGURE 216 – A CUT THROUGH THE VOLUME MESH OF THE F16 213
FIGURE 217 – THE MESH QUALITY GRAPH FOR THE F16 213
FIGURE 218 – HIGHLIGHTING THE *POORER* QUALITY ELEMENTS 214
FIGURE 219 – THE FLOW SOLUTION OVER THE F16 215
FIGURE 220 – A CUTTING PLANE OVER THE F16 WING 216
FIGURE 221 – AN ISO-SURFACE OF MACH 1.0 OVER THE F16 216

FIGURE 222 – MODIFICATION OF THE OUTER BOUNDARY FOR THE CEM SIMULATION 217
FIGURE 223 – MODIFYING THE SOURCES FOR THE CEM SIMULATION 218
FIGURE 224 – THE SURFACE MESH FOR THE CEM SIMULATION 219
FIGURE 225 – THE SURFACE MESH ZOOMED IN ON THE FRONT OF THE ENGINE 219
FIGURE 226 – A CUT THROUGH THE VOLUME MESH 220
FIGURE 227 – A ZOOMED VIEW OF THE CUT AROUND THE ENGINE..221
FIGURE 228 – A SIMPLE PROTOTYPE OF A VPE WITHIN THE PSUE II...227
FIGURE 229 – EXAMPLE OF REPRESENTING EDGES BY FORMING NODE PAIRS 244
FIGURE 230 – EXAMPLE OF REPRESENTING TRIANGULAR FACES BY FORMING 3-TUPLES OF NODES............244

Chapter 1. INTRODUCTION

1.1. What is a Problem Solving Environment (PSE)?

In April 1991, a research conference was held from which a long report was issued [Gallopoulos94] exploring the field of PSE's. The definition of a Problem Solving Environment that emerged was:

> *"A Problem Solving Environment is a computer system that provides all the computational facilities needed to solve a target class of problems. These features include advanced solution methods, automatic and semiautomatic selection of solution methods, and ways to easily incorporate novel solution methods. Moreover, PSE's use the language of the target class of problems, so users can run them without specialised knowledge of the underlying computer hardware or software. By exploiting modern technologies such as interactive colour graphics, powerful processors, and networks of specialised services, PSE's can track extended problem solving tasks and allow users to review them easily. Overall, they create a framework that is all things to all people: they solve simple or complex problems, support rapid prototyping or detailed analysis, and can be used in introductory education or at the frontiers of science."*

This definition can be summarised by the formula:

"PSE = Natural Language + Solvers + Intelligence + Software Bus"

Here, the Software Bus represents the computing infrastructure, i.e. the computers, networks, etc.

The level of *intelligence* that a PSE requires depends on the number of options available and how much background information is required in order to make an informed choice. This does not mean that the PSE must be capable of answering complex questions such as:

- What is the best simulation software for this problem?
- What time step size is needed to achieve the accuracy required?
- Which computer should be used?
- Where is the data needed for this computation?

But at the very least, a PSE should be able to present the options in a manner that allows the user to make an informed choice without requiring a detailed knowledge of the algorithms.

The term 'Solver' refers to the class of algorithms that will actually compute the results for the particular class of problem. The term used in the equation is plural since a PSE will naturally have more than one solver available since it is unlikely that there will be one solver that is best for all cases in a given class of problems.

The 'Natural Language' component means that a PSE must communicate with the user in a language that is suitable for that application. This means the data that is input or displayed should be in a form that is readily understood by the user, rather than in a form that is used internally inside the algorithm. Examples of this can range from presenting simple data using SI units to using advanced two- and three-dimensional graphics in order to make sense of a large set of data.

The need for a PSE becomes obvious when the complexity of today's problems are considered. Problems involving a single discipline (such as fluid flow) are continuously being replaced by more complex, multi-disciplinary problems (such as fluid-structure interaction). Furthermore, these problems are increasingly being tackled by people in a commercial or industrial environment where there may be little or no expertise in the underlying algorithms. In these situations, there is a fundamental requirement for a software system that can guide the user as much as possible through the steps involved in solving the particular problem.

Due to the high level and wide ranging goals of a PSE it naturally leads to a large and complex software system. For this reason, unlike many of the actual tools within the PSE, the architecture of the system is of utmost importance in ensuring robustness, performance and flexibility.

1.2. Review of Problem Solving Environments

One of the difficulties of reviewing Problem Solving Environments is that for almost any problem domain that utilises computers one or more PSE's exist. Even if we restrict ourselves to the domain of mathematics, there are many PSE's for the many different branches.

If we restrict ourselves further to the domain of interest to this thesis, computational simulation, then there are still a large number of PSE's but they do mostly fall into one or more of six main categories depending on their generality:

- Pre-processors to conventional programming languages
- New programming languages
- Graphical PSE's for specific domains of mathematics
- Visual Programming Environments (VPE)
- Graphical PSE's for specific applications
- Futuristic PSE's.

1.2.1. Pre-processors to Conventional Programming Languages

The most general type of PSE takes the form of a pre-processor to a conventional programming language. A typical example of this type of PSE is ELLPACK [Purdue02a, Rice86] developed in the Department of Computer Sciences at Purdue University. This takes the form of a pre-processor to the FORTRAN language that, with the use of a large library of mathematical routines, converts a program in the form shown in Figure 1 into FORTRAN that can then be compiled and executed. Its purpose is to greatly reduce the programming effort required to solve 'routine' elliptic problems.

```
EQUATION.        UXX + Y*UYY + SIN(X+Y)*U = 1 - X + Y
BOUNDARY.        U = 0  ON    X = 0.
                 U = Y  ON    X = 1.
                 U = 0  ON    Y = 0.
                 U = X  ON    Y = 1.
GRID.            21 X POINTS $ 21 Y POINTS
DISCRETIZATION.  HERMITE COLLOCATION
SOLUTION.        LINPACK BAND
OUTPUT.          PLOT(U) $ TABLE(U)
END.
```

Figure 1 – A short ELLPACK program

Although this is not a conventional form of PSE, as it still requires considerable programming experience, it does relieve the need for a programmer to know all of the implementation details of the various methods of solving elliptic problems in order to obtain a solution.

1.2.2. New Programming Languages

A more sophisticated example of a general PSE is that of a domain specific programming language. Examples of these include MATLAB [MathWorks] (Figure 2), Maple [Maplesoft, Wright01] and Mathematica [Wolfram]. These systems consist of an environment in which mathematical problems can be expressed in mathematical notation rather than conventional programming language notation. This makes them more accessible to people without a programming background and can even be preferred by programmers over traditional languages for rapid prototyping purposes.

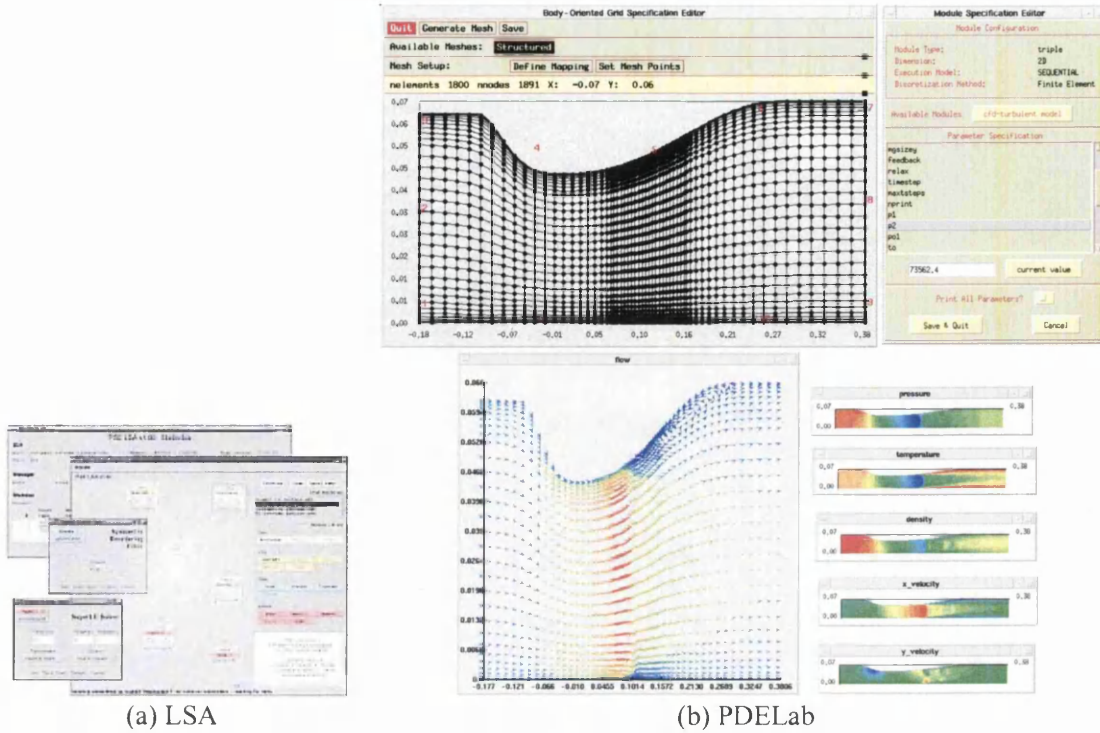
Figure 2 – A typical session in MatLab

1.2.3. Graphical PSE's for Specific Mathematical Domains

If the domain of the PSE is restricted still further then graphical environments appear in which the user can solve problems with little or no programming knowledge. Examples of this type of PSE are PDELab [Purdue02b] that solves systems of partial differential equations (PDE), and LSA [Bramley98] that solves sparse systems of linear equations.

These environments allow the user to solve their particular mathematical problem using point-and-click techniques with the environment assisting the user in selecting the appropriate technique for the given problem. This alleviates the user from having to expend the effort to learn all of the subtleties of the individual mathematical techniques that are needed in order to solve the PDE or system of equations. Instead, the user can treat them purely as a tool that can be used in order to progress in their own field of expertise.

Figure 3 shows a typical session in LSA and PDELab.



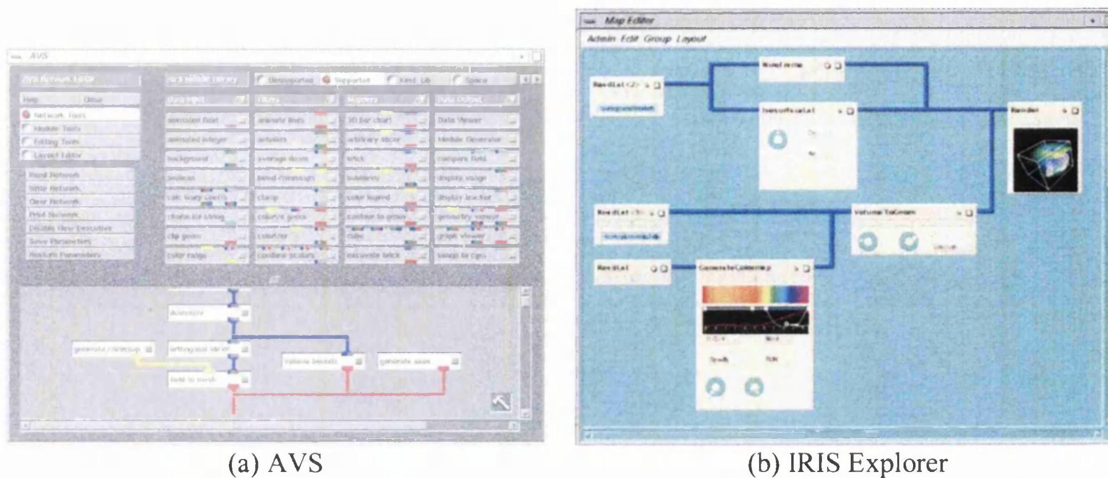
(a) LSA (b) PDELab

Figure 3 – Typical User Sessions in LSA and PDELab

1.2.4. Visual Programming Environments

A common style of graphical PSE is that of the general Visual Programming Environment (PSE). There are a number of implementations in the research domain, including SCIRun [Johnson98, Johnson00] and the Component Architecture Toolkit [Vilacis99]), and a number of commercial implementations including AVS [AVS] and IRIX Explorer [Nag].

Here, the user is presented with a number of modules that perform individual tasks. Each module is represented by a box, with the various inputs and outputs that are required for that module represented by connectors. The user can then take any combination of these modules and construct their environment (often called a *map*) by connecting the input and output ports of the various modules in a point-and-click manner. Figure 4 shows an example of a map in AVS and IRIX Explorer.



(a) AVS (b) IRIS Explorer

Figure 4 – Typical Maps in AVS and IRIX Explorer

The initial purpose of the above four environments was to enable the user to easily produce customised visualisations of their results. However they were designed with sufficient flexibility that, with sufficient programming experience, it is possible to create user-defined modules that can then be included in a map. This ability enables the VPE to potentially become a PSE for almost any problem domain. For AVS and IRIX Explorer in particular, there is a substantial collection of pre-written modules freely available in the public domain so even users who are not experienced programmers can extend the functionality of these environments to meet their own requirements.

However, even with modules available for a particular problem domain, the VPE is more suitable for the user who has an understanding of the process they are undertaking including the order in which the various components must be linked. For any user who is purely using a PSE as a means of obtaining a result for a particular class of problem and who needs guidance through the process, the VPE is not really a suitable method.

1.2.5. Graphical PSE's for Specific Applications

In order for a user with little or no knowledge of how to solve a particular problem, but whom requires the solution in order to further their own work, a graphical PSE that is tailored for a specific application is the most appropriate choice. For example, a team designing the shape of a car has to take into account how the moving car interacts with the stationary air. In order to obtain such results Computational Fluid Dynamics (CFD) would most likely be used. In an ideal world, these users are not interested in any aspect of the CFD process – they would prefer to input a geometry of a car with some details concerning its speed, wind direction, etc. and obtain results such as drag, wind noise, etc.

However, the current state of the algorithms that form the CFD simulation process means that this is not yet possible. In order to overcome this the PSE forms a compromise in which as much of the process as possible is fully automatic, and the tasks that need user involvement are made as simple and intuitive as possible.

In order to achieve this, a typical application-specific PSE includes all of the relevant tools for setting up the problem (pre-processing), solving the problem and visualising and interrogating the results (post-processing). These tools are generally integrated into one seamless Graphical User Interface (GUI). Numerous examples of this type of PSE exist in both the research domain [Gaither96, Brodersen98, Goel99, Gaither00, Ravishankar00, Shevare00, Ramakrishnan01] and the commercial sector (e.g. Fluent, MSC, CFX, Ansys, etc.).

1.2.6. Futuristic PSE's

The continuing increase in power of modern computers has allowed more and more complex simulations to be performed. Modern state-of-the-art simulations are invariably multi-disciplinary and involve heavy use of high-performance parallel computing technology. The next stage in this evolution is to connect computers that are geographically disparate in order that a much larger computer may be created and hence allow much larger simulations to be performed.

This continuing increase in size of the simulation and the complex issues involved in the management of such wide-ranging networks of computers will mean that the use of a PSE will be of fundamental importance. To this end, a number of projects have been initiated looking into the requirements of such a PSE. NASA and a number of the US National Laboratories started one of the largest projects, called the Information Power Grid (IPG). The aim of this project is to produce a general, all encompassing, prototype PSE that will allow the user to semi-transparently access the vast amounts of computation power available around the world in order to solve Grand Challenge problems. Several components have been developed, including GLOBUS [Foster01] for distributed resource management and a scientific software library, PetSc [Buschelman00], but at this time the overall PSE is more of a vision than a reality.

1.3. The Problem Domain: Computational Simulation

For this thesis, we concentrate on Problem Solving Environments for the computational simulation process, more specifically, for finite element or finite volume based algorithms for CFD and Computational Electromagnetics (CEM).

A typical computational simulation process for these types of applications often involves three main classes of algorithms:

- Mesh Generation,
- Computational Analysis and
- Mesh Refinement / Adaptation.

Traditionally, these tasks were performed using simple command-line driven tools with little or no graphical capability.

1.3.1. Mesh Generation

Assuming the geometry on which the mesh is to be generated is topologically correct, current unstructured mesh generation technology has made this process virtually automatic. The only user interaction required is the definition of the mesh density in the various regions of the domain. A common means of achieving this is by placing a number of point, line and planar sources in the domain at key positions. However, this process is very time-consuming for a complex geometry and requires the user to know in advance the dimensions of the domain and the coordinates of all of the key features.

Once the sources have been placed then the mesh generation process can begin. This is generally split into two main sub-tasks; surface and volume mesh generation. Surface mesh generation involves the generation of nodes and edges along all of the intersection curves. Further nodes are then placed on the interior of the geometry surfaces (including any symmetry planes and outer boundaries) and connected to form triangles and/or quadrilaterals. This mesh is then used as the starting point for the generation of the volume elements (i.e. tetrahedra, pyramids, prisms and hexahedra) that fill the entire computational domain.

1.3.2. Computational Analysis

Like the mesh generation phase, the majority of the Computational Analysis phase is fully automatic. Before the solver can be executed, regions of the computational domain need to be assigned solver specific properties called *boundary conditions*. These are used to inform the solver how to treat the various surfaces and/or volumes. For example, when performing a CFD analysis, the surfaces might represent entities such as engine inlets or exhausts, or viscous or inviscid solid walls. In a CEM analysis, portions of the volume mesh may be assigned different material properties. Other types of boundary condition may not represent physical entities at all; instead they might represent topological entities such as symmetry planes.

Once all of the geometry surfaces have had boundary conditions applied the solver may be initiated. The algorithms of the solver are extremely computationally intensive and are therefore often executed on a, possibly remote, super-computer.

1.3.3. Mesh Refinement / Adaptation

As with any algorithm that relies on numerical approximations to the governing equations, the accuracy of the result depends heavily on having an appropriate density of sampling points in regions where the solution changes rapidly. Knowing, in advance, the locations of all of these regions is either impossible, or at the very least requires a great deal of experience on the part of the user. The purpose of the Mesh Refinement phase is to analyse the solution with respect to the geometric spacing of the nodes of the mesh in order to produce an estimate of the error. These regions can then be refined using a number of algorithms, including *h*-refinement, *r*-refinement [Scott-McRae00] and local remeshing, with the new mesh being passed back to the solver again to continue from the existing solution and produce a more accurate solution.

1.4. Requirements for a Computational Simulation PSE

As stated above, the three main stages of the computational simulation are all fairly automatic processes once they have been initiated. The area where user interaction is beneficial is in the preparation of the input data for each stage and in the analysis of the output from each stage. If these are taken into account, then the number of stages involved in a simulation increases from three to ten:

- Geometry Preparation
- Mesh Density Specification
- *Mesh Generation*
- Mesh Quality Evaluation and Repair
- Boundary Condition Specification
- *Computational Analysis*
- Solver Monitoring
- Solution Visualisation
- Mesh Refinement Control
- *Mesh Refinement / Adaptation.*

Key

- Interactive Stages
- *Non-Interactive Stages*

It is during these interactive stages (blue) that the most time is spent during a simulation. In fact, it has been estimated that for a complex simulation, the time required for the preparation of the input data (stages 1 and 2) accounts for 90% of the total.

If the PSE is to be used to perform very large simulations of the order of 10's or 100's of millions of elements, then the use of parallel computing hardware throughout the simulation is essential. Executing and monitoring tasks on a remote parallel computer is much more difficult than performing the same task locally. This has been made even more difficult with the recent trend towards using clusters of PC's or workstations to form a parallel computer since the user has to decide on which computers to run the job.

Therefore, the key requirements of a PSE in the field of computational simulation can be summarised as follows:

- Problem set-up time must be reduced.
- The user must be guided through the simulation process.
- The details of the execution of tasks on remote parallel computers must be hidden from the user.

In order to achieve these requirements, a number of challenges must be overcome:

- *The User Interface must remain intuitive throughout the simulation*
  This means ensuring that all information is presented to the user in a manner to which he/she is accustomed, i.e. in engineering language rather than computer language.
- *All invalid routes through the environment should be disabled*

This means ensuring that only the options that are valid should be presented to the user. The validity of an option depends on whether the data sets required to perform that option are present or whether the option makes sense at the current stage of the simulation.

- *All three-dimensional rendering must be real-time*
  This is of paramount importance since any delays occurring whilst manipulating the model on the display will invariably cause user frustration. This should be achieved regardless of the size of the model or mesh. As the size of the data sets increases the PSE should automatically adapt its rendering in order to maintain interactive frame rates.
- *All interaction must remain as close to real-time as possible*
  This means ensuring that, as far as possible, any operation performed on a data set must happen in a timely fashion. Obviously, when performing complex operations on a large model some periods of unresponsiveness are inevitable but these should be minimised. For example, if the user performs the option to create an iso-surface then a small period of computation is expected. However, clicking on the model to select a geometry surface should be instantaneous.
- *The use of parallel computers should be (semi-) transparent*
  When working with large and complex models, with meshes of the order of 10's of millions of elements, the use of parallel computing technology is inevitable in order to satisfy the previous four conditions. However, this should not complicate the use of the PSE any more than necessary. The maximum amount of extra user interaction that should be tolerated is the selection of which computers should form the parallel computer. Even this extra interaction should be presented to the user in a user-friendly manner where the selection can be made purely by pointing and clicking. All of the details of executing the parallel processes and initiating the communication links should be completely hidden.

1.4.1. Geometry Preparation

In a typical industrial environment, the geometry is usually extracted directly from the CAD database. These geometries invariably arrive in a form that makes the later stages of mesh generation impossible due to the fact that they are tailored more for a manufacturing or prototyping purpose than for a computational simulation which have quite differing requirements. For example, geometries intended for the manufacturing process would include gaps around doors and hatches, or locations of rivets and bolts. For most simulations, for example, CFD, these features would need to be removed.

Regardless of their purpose, geometries that are specified by their boundary representation (B-Rep models) are defined as a combination of surface patches and intersection curves. For a geometry to be meshable, it needs to be topologically valid (or *closed*). A closed geometry is one in which every surface has one or more sets of intersection curves forming a closed loop and every intersection curve is attached to two, and only two, surfaces. This forms a domain in which the inside and outside of the domain can be determined unambiguously.

There are many features of a geometry that may make it topologically invalid, but they generally fall into one of two categories; overlapping surfaces or inter-surface gaps.

Overlapping surfaces are where any two adjacent surfaces that, through inaccuracies or design faults, do not meet along an intersection curve but overlap slightly. This is shown in Figure 5.

Inter-surface gaps are where any two adjacent surfaces that, for similar reasons, do not meet along an intersection curve but have a small gap between them. This is shown in Figure 6.

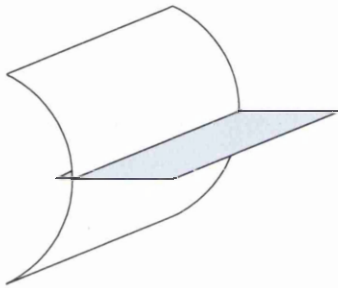


Figure 5 – Overlapping Surfaces

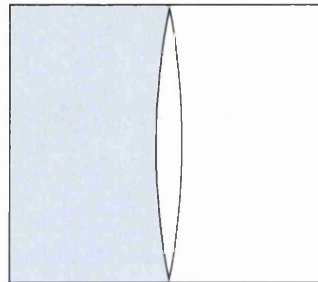


Figure 6 – Inter-Surface Gaps

Even if a geometry is valid, there may be features that inhibit the generation of a good quality mesh. These problems occur because most surface mesh generators operate a surface at a time. A typical example is a surface with a very small angle at a corner. Sometimes this is unavoidable due to the shape of the geometry, for example at the nose of an aircraft or then end of a wing (Figure 7), but often this is simply due to the choices made by the designer of the original CAD model (Figure 8).

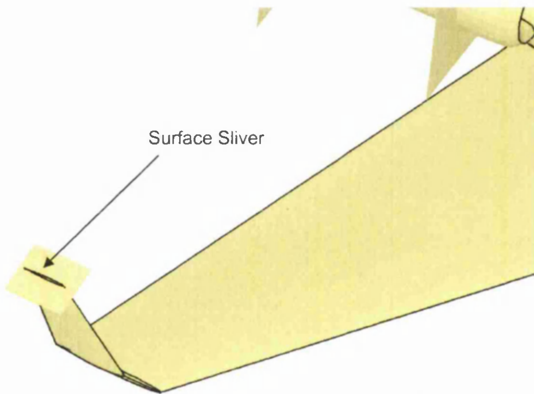


Figure 7 – Sliver Surface at Wing Tip

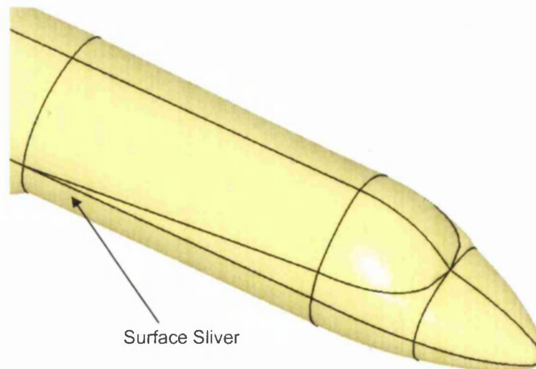


Figure 8 – Sliver Surface on Fuselage

The purpose of the Geometry Input phase is to convert a CAD geometry into a form on which a good quality mesh can be generated. This process is often referred to as *CAD Repair*. This process is, at best, semi-automatic in which the CAD repair algorithms can

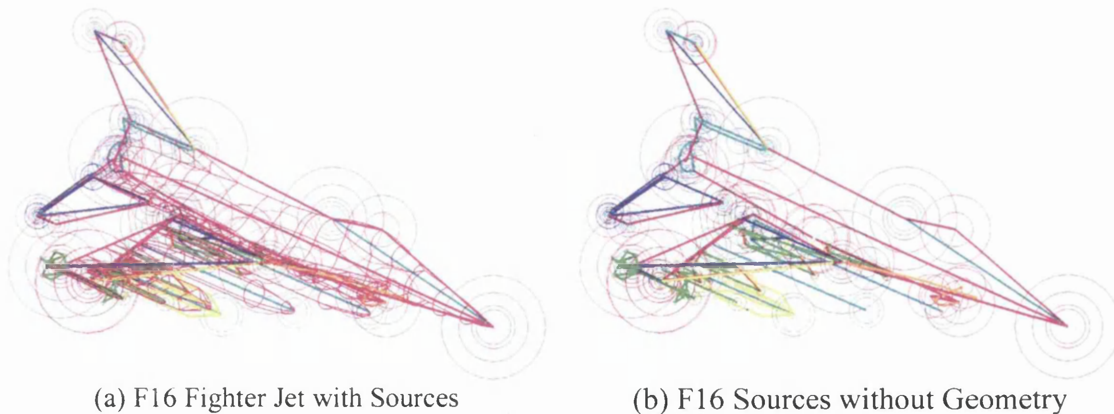
repair small anomalies leaving the user to repair larger areas through graphical interaction.

1.4.2. Mesh Density Specification

Once a suitable geometry has been obtained, the density of the mesh in the various regions of the domain needs to be defined. As stated previously, a common method is to place point, line and planar sources at the positions of key features [Weatherill94a].

A point source is defined as a 6-tuple, $\{x, y, z, r_1, r_2, s\}$. Geometrically, a source comprises two spheres centred on the point $[x, y, z]$, with radii r_1 and r_2. The mesh spacing (i.e. element edge length) within the inner sphere is defined as s. In the region between the inner and outer sphere the mesh spacing increases linearly from the spacing, s, and the background spacing of the mesh. Line sources are a linear combination of two point sources that, geometrically, form the shape of a sausage. Planar sources are a further extension of the point source created by linearly combining three point sources to form a triangular area.

A typical example is shown in Figure 9.



(a) F16 Fighter Jet with Sources (b) F16 Sources without Geometry

Figure 9 – Complex Geometry with Sources

This is a process heavily dependent on user interaction since the placement and strength of the sources depend on a number of factors including:

- The type of simulation to be performed
- The areas of interest
- The shape of the geometry and
- Any prior knowledge as to where interesting features occur (e.g. shock waves or vortices in CFD).

For this process to be performed efficiently, it is necessary for the PSE to be able to interact directly with three-dimensional models on the screen. The user must then be able to interactively place the sources at key locations and receive instant graphical feedback

as to how the source strength affects the density of the mesh by illustrating the mesh spacing or size of elements within the source in real-time.

Techniques are also available to automatically place these sources based on feature detection [Mezentsev00] such as curvature but results invariably need some user interaction to fine-tune their positions and strengths.

1.4.3. Mesh Quality Evaluation and Repair

Once a surface and volume mesh has been generated, it should be analysed to evaluate its suitability for the intended solver. This can be achieved by using a combination of statistical quality measures to identify any *poorly* formed elements, and graphical interaction with the model in order to identify the positions of these elements with respect to areas in the domain that are of particular interest.

If the poorly formed elements occur away from regions of interest then it may be decided that no action to improve the quality of the elements is required. However, if it is deemed necessary to improve the element quality then a number of techniques could be used to improve the quality of the mesh in that area [Hassan99c]. These include:

- *Mesh Smoothing* – The vertices of the elements that are local to the bad element are moved in order to minimise a pre-defined energy function (Figure 10).
- *Element Removal* – Thin elements (slivers) can often be collapsed completely and removed from the mesh resulting in an overall improvement in mesh quality (Figure 11).
- *Face Swapping* – Swapping the faces of two adjacent elements can often increase the quality of both elements (Figure 12).
- *Re-Meshing* – If the above techniques fail then the only alternative may be to re-generate the mesh in a region local to the poor element but with modified mesh density parameters.

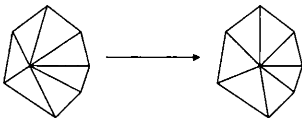


Figure 10 – Mesh Smoothing

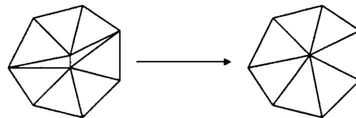


Figure 11 – Element Removal

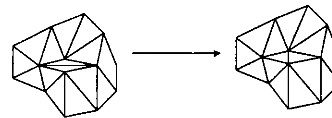


Figure 12 – Face Swapping

1.4.4. Boundary Condition Specification

The last interactive process before the computational analysis phase can proceed is the definition of the boundary conditions. This process, although reasonably quick, does benefit significantly from the use of graphical interaction in order to select the surfaces of the model on which to apply the various boundary conditions. Graphical feedback, through the use of colouring, helps identify which conditions have been applied to which surfaces helping to reduce any errors.

1.4.5. Solver Monitoring

During the execution of the solver, it is often useful for the user to be able to monitor the progress of the solver through the plotting of parameters, such as the residuals of the solver variables. Since a solver is often executed remotely and may take a number of hours or days to run, the monitoring tools should be able to connect and disconnect from the running solver at any time without affecting the execution of the solver in any way.

1.4.6. Solution Visualisation

Once the solver has finished, and a solution obtained, it is essential to represent the huge quantity of numbers in a form that can be analysed easily by the user. This can range from simple two-dimensional plots to the use of advanced three-dimensional graphics and feature extraction algorithms such as vortex detection, iso-surfaces, streamlines, etc. Regardless of the form of the output, it is essential to be able to interact with the model in an efficient manner in order to define the positions of these entities.

1.4.7. Mesh Refinement Control

If it has been deemed that mesh refinement is necessary, the user must be able to identify the areas of the mesh that must be refined. Although, in theory, this could be fully automated through the use of error estimation algorithms, in practise, the error estimation is only one factor in determining how much of the mesh is refined. Other factors include:

- *Mesh Size* – Refining based on an error estimator alone may produce a mesh that is too large to be able to continue the simulation.
- *Areas of Naturally High Error* – Mesh refinement at the outlet of an aircraft engine may always produce an unusually high error due to the very high gradients of the solution at that point. Refining the mesh in this region may be deemed a waste of resources.

In order to be able to define regions, in which mesh refinement should occur, a combination of the automatic error estimator and graphical interaction with the model is essential.

1.5. Layout of Thesis

The layout of the thesis encompasses eleven chapters that describe the design and implementation of two Problem Solving Environments. Throughout both of these projects, the industrial partners in each project, many of whom were leading European Aerospace companies, influenced the design of the PSE's.

Chapter 1 sets the scene by introducing the concept of a Problem Solving Environment along with the requirements that a PSE for computational simulation has to meet in order to be useful in an industrial environment.

Chapter 2 then gives a brief background of the two main technologies behind the two PSE's described in this thesis; three-dimensional graphics and parallel computing.

Chapter 3 describes the design and implementation of the first PSE, called PROMPT. This was an environment developed for the Nozzle After-body division of the Military Power-plant Technology group in Rolls Royce with the aim of enabling the engineers to use the simulation tools developed within, and for, the company.

After the PROMPT project, a European Project, called JULIUS, started in which a parallel environment was developed with the aim of being able to perform very large-scale simulations within an easy-to-use interface.

Chapter 4 introduces the aims of the JULIUS project, along with the role that University of Wales Swansea played. This is then followed by the requirements that a parallel environment had to fulfil. Finally the partitioning of the main data structures used throughout the environment is described.

Chapter 5 continues the theme by describing the design and implementation of the first version of the parallel environment, called PSUE II.

Chapter 6 then identifies the areas in which the PSUE II could be improved and describes the new design and implementation.

Chapter 7 gives an overview of the functionality and user interface of the second version of the PSUE II along with a description of the mechanism through which 3rd party applications can be integrated within the environment.

Chapter 8 concludes the description of the two environments by describing the various design and software issues that had to be overcome in order to ensure the environments were portable across all major UNIX platforms as well as platforms based on Microsoft Windows NT/2000.

Chapter 9 then presents some simulations that were performed using the two environments. These are accompanied by a number of statistics to illustrate the effectiveness of the two environments in their domain.

Chapter 10 and 11 then draw some conclusions and present ideas for future research into this growing field of research.

Chapter 2. TECHNICAL BACKGROUND

2.1. Background to Three-Dimensional Graphics

2.1.1. Breakdown of the Rendering Process

In order to represent a three-dimensional object on a two-dimensional display a number of operations need to be performed. In a general-purpose framework, these operations can be classified into four stages:

- Scene Construction.
- Scene Projection.
- Vertex / polygon based effects.
- Rasterisation with pixel based effects.

Scene Construction

Regardless of the original form of the data that needs to be rendered, it must be converted into a set of simple primitives that can then be passed on to the graphics sub-system of the computer. For most modern systems, these primitives are points, straight lines and triangles. Although most systems also allow convex quadrilaterals, they are generally regarded as being unsafe since a convex quadrilateral can change to a self-intersecting polygon from some angles if it is non-planar as shown in Figure 13

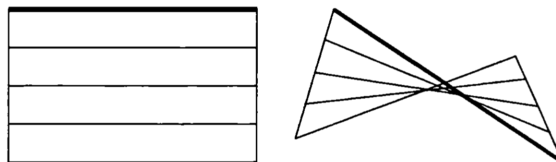


Figure 13 – Non Planar Polygon transformed to a Self Intersecting Polygon

The result of this stage is a simple sequence of primitives along with any associated colour and normal data.

Scene Projection

In order to render a three-dimensional scene, constructed of simple primitives, onto a two-dimensional display a series of transformations need to be applied. This is generally regarded as being analogous to taking a photo with a camera. These steps could be:

- Arranging the objects in the scene to be photographed into the desired composition and pointing the camera at the scene (modelling transformation).
- Choosing the camera lens or adjusting the zoom (projection transformation).
- Determining how large you want the final print to be (viewport transformation).

After these steps have been taken the picture can be taken, or the scene can be drawn.

In three-dimensional graphics, these transformations are represented as a series of 4x4 matrices with the vertices being represented as homogenous co-ordinates as shown in Figure 14.

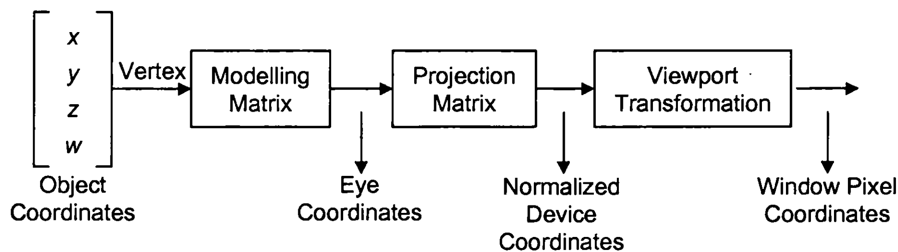


Figure 14 – The Vertex Transformation Pipeline

The modelling matrix is usually a combination of translation (Figure 15a), scaling (Figure 15b) and rotation (Figure 15c-e) operations in order to position the objects and the camera in the correct positions in relation to each other.

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

(a)
Translation by (x, y, z)

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

(b)
Scaling by (x, y, z)

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

(c)
Rotation around x-axis

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

(d)
Rotation around y-axis

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

(e)
Rotation around z-axis

Figure 15 – Translation, Scaling and Rotation Matrices

The projection transformation is invariably a choice between an orthographic projection matrix in which all lines that should be parallel are parallel, and a perspective projection matrix in which lines converge as they travel away from the viewer.

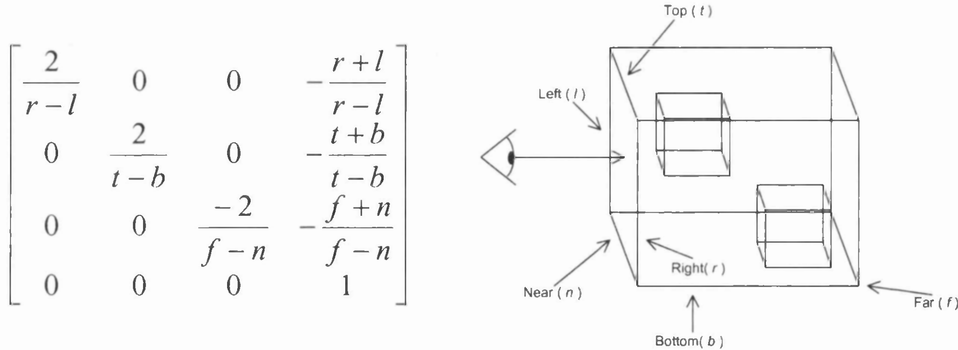


Figure 16 – The Orthographic Projection Matrix

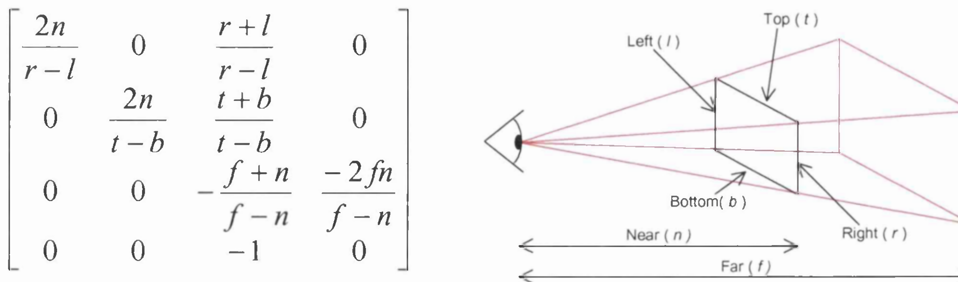


Figure 17 – The Perspective Projection Matrix

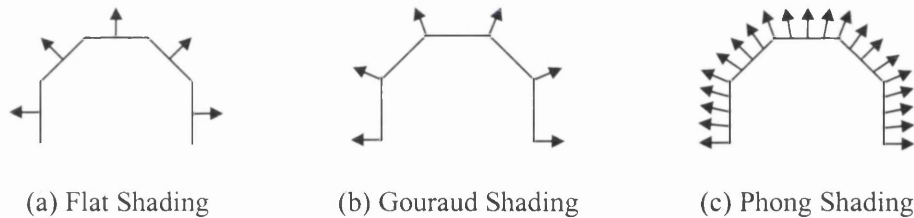
The viewport transformation is simply a two-dimensional scaling and translation in order to transform the objects coordinate system into the pixel coordinate system of the display.

These transformation matrices are multiplied together in order to form a single transformation matrix. Each vertex is then multiplied by this combined matrix in order to produce a position on the display.

Vertex / Polygon Based Effects

During the transformation of the scene from three to two dimensions, a number of other effects can be applied in order to produce a more realistic appearance. One of the most common effects is that of lighting. In its most basic form, this is a simple calculation based on the angle each light source makes with each vertex or polygon of the scene. The user positions the light sources within the scene using the same coordinate system as the model. The vertex or polygon normals are usually also supplied by the user and are used to determine the intensity of the light as it bounces off the model towards the display.

During the transformation of the scene into display coordinates, the contributions from the various light sources are transformed into colours for each vertex / polygon. Figure 18 shows the three most common lighting models used in modern graphics libraries.



(a) Flat Shading (b) Gouraud Shading (c) Phong Shading

Figure 18 – Flat, Gouraud and Phong Shading Models

The flat shading model uses polygon normals in order to compute a colour for the entire polygon. This is normally the quickest for the graphics system to calculate but for curved surfaces produces the least realistic appearance (Figure 18a). The second and third models use normals calculated at the vertices of the polygons. For the Gouraud shading model, these normals are then used to calculate the colours at the vertices of the polygons. These colours are then linearly interpolated across the polygon. This model requires more calculations than the flat shading model but give a smoother appearance (Figure 18b). The Phong shading model interpolates the normal across the polygon and at each pixel uses the interpolated normal to compute the required colour. This last model requires the most calculations but does produce the most realistic effects (Figure 18c). Until recently, the only models that were implemented in hardware in most graphics systems were the flat shading and Gouraud shading models. However, recent advances in graphics hardware have meant that Phong shading has become a viable alternative.

Rasterisation with Pixel Based Effects

The last stage in the rendering process is the rasterisation procedure. Essentially, this takes the transformed coordinates, and colours, and produces the image in the frame buffer, which is then rendered on the display. During this process, a process known as depth buffering is often applied.

Whereas the frame buffer stores the colour information for every pixel on the display, the depth buffer stores a depth value for every pixel on the screen. When drawing each pixel, its depth buffer value is compared with that already in the depth buffer. If it is less (i.e. closer to the viewer) then the pixel is drawn into the frame buffer and its depth is stored in the depth buffer. This is a very simple method of rendering scenes with hidden surfaces.

Other effects, which are beyond the scope of this introduction, can also be applied at this stage, including texture-mapping, transparency, anti-aliasing, bump mapping, etc.

2.1.2. Examples of Software Libraries for 3D Graphics

Over the years, a number of software libraries have aimed at standardising the interface to the graphics sub-system of each computer. Each library has approached this problem at various levels ranging from libraries that allow the user to define the scene as collections of objects using many representations from simple primitives to complex bi-cubic patches, down to libraries that restrict the user to simple primitives and operations. There have been many such libraries over the years. GKS (Graphics Kernel System) [ANSI85] was the first ever library to be officially standardised by ANSI in 1985. However, it was limited to two dimensions. An extension, GKS-3D [ISO88], added three-dimensional graphics and also became an ANSI standard in 1988. Later, more complex libraries that allowed nested groupings of primitives appeared including PHIGS (Programmer's Hierarchical Interactive Graphics System) [ANSI88] and PHIGS+ [PHIG88], an extension to PHIGS.

Nowadays, there are two main standards for low-level graphics. These are DirectX and Open-GL [OGL-ARB92, Neider93].

DirectX [Microsoft95] was introduced in 1995. It is a suite of multimedia API's (Application Programming Interface) developed by Microsoft and built into the Windows operating systems. These API's give applications easy access to two and three dimensional graphics, but also go much further by incorporating interfaces to sound cards, joysticks, keyboards, mice, etc.

Open-GL was introduced in 1992 and was designed as a streamlined, hardware-independent interface to be implemented on many different hardware platforms. It includes two and three-dimensional graphics incorporating a large number of visualisation effects. It is a very stable interface with additions to the standard being well controlled by a review board including representatives from leading companies such as SGI, Hewlett-Packard, IBM, Intel, NVidia and Microsoft. It is a very scalable and

portable graphics standard with implementations on every conceivable platform ranging from low-end PC's and Macs to multi-million pound graphics super-computers from companies such as SGI, and every conceivable language including C, C++, Fortran, Python, Perl and Java. It is undisputedly the most widely adopted graphics standard in existence.

The higher level libraries that allow the user to describe entire scenes, rather than low level primitives, are then built on top of these libraries. Examples include Open Inventor, OpenGL Volumiser, OpenGL Optimiser and OpenGL Performer.

2.2. Background to Parallel Computing

Parallel computing is the division of work into smaller tasks, assigning these smaller tasks to multiple processors to work on simultaneously. Its main goals are to solve much larger problems in less time. This concept is summed up well by a quote from Grace Hopper (1906-1992) during one of her many public presentations:

> *"In pioneer days they used oxen for heavy pulling, and when one ox couldn't budge a log, they didn't try to grow a larger ox. We shouldn't be trying for bigger computers, but for more systems of computers."*

The power of parallel computing, especially modern Massively Parallel Computers (MPPs), is illustrated clearly in the current Top500 supercomputers in the world (www.top500.org). Here, an overwhelming 456 out of the top 500 super-computers in the world are parallel computers based on standard, scalar processors[1].

Although there are many different types of parallel computing hardware available, they all fall into one of three categories:

- Shared Memory
- Distributed Memory and
- A hybrid of the two called Distributed-Shared Memory.

2.2.1. Shared Memory Parallel Architectures

A standard shared memory architecture is shown in Figure 19. Here, there is one global pool of memory that each processor can access with equal priority and is often referred to as an SMP (Symmetry Multi-Processing) architecture. This architecture is by far the simplest one to construct a parallel program for a number of reasons:

- The data on which the program operates can stay in one piece rather than having to be partitioned into many smaller pieces.
- A sequential program can be parallelised one routine at a time in an incremental fashion. This makes it easy to ensure that at each stage the program is still producing the same results as the sequential version.

[1] Statistics as of 10th December 2002.

- For many shared memory architectures, the compiler can actually perform most of the parallelisation automatically.

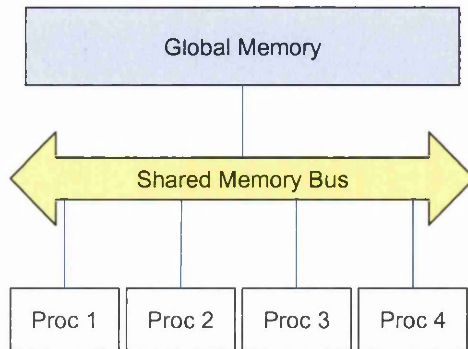


Figure 19 – The Shared Memory Architecture

However, the shared memory architecture does have one major drawback; it is not very scalable. True shared memory architectures do not scale beyond 64 processors and are most commonly found in configurations of two or four processors. This is illustrated by Figure 20 that shows the largest SMP configuration for each of the major vendors.

Manufacturer	Model Name	Max CPUs	Source of Information
HP (Compaq)	AlphaServer GS320	32	http://www.compaq.com/alphaserver/
Sun	Enterprise 10000	64	http://www.sun.com/servers/comparison/enterprise/index.html
SGI	Power Challenge	36	http://www.sgi.com

Figure 20 – SMP Configurations from the Leading Vendors

This is due to the contention between the processors to access the memory. As the number of processors increases this contention increases to the extent that no further performance increase occurs.

2.2.2. Distributed Memory Parallel Architectures

A distributed memory architecture (Figure 21) is a stark contrast to the shared memory architecture in that the advantages and disadvantages are the exact reverse. Here, each processor has its own pool of memory with the only means of accessing data from memory in the other processors is through explicit message passing. This means that parallelising a program for this type of architecture is more difficult for the following reasons:

- The data on which the program operates must be distributed in order for each processor to be able to work simultaneously. The way in which this data is sub-divided can often be a major research effort in itself.
- A sequential program often has to be parallelised in one go. In fact, the parallel version of the program is often so different in structure from the sequential version

that two versions are often maintained. This can make incremental testing and debugging virtually impossible.

- Although, there are compilers and pre-processors available that will attempt to parallelise a sequential program automatically for a distributed architecture, they are often limited in their application and / or produce poor scalability and performance [Berthou97, Sturler97, Mehrotra98].

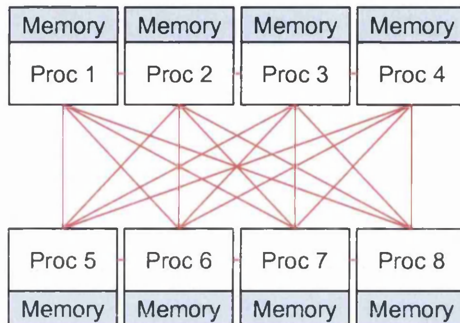


Figure 21 – The Distributed Memory Architecture

However, unlike a shared memory architecture, the scalability of distributed memory architectures is phenomenal with most vendors being able to scale well above 1000 processors.

2.2.3. Hybrid Distributed-Shared Memory Parallel Architectures

The aim of the distributed, shared memory architecture is to try to lever the advantages of both the shared and distributed memory architectures and produce an architecture that is both as easy to program as a shared memory architecture and as scalable as a distributed memory architecture.

One of the most successful implementations of this type of architecture is the Origin family of computers from SGI. As Figure 22 shows, these have a shared and distributed memory architecture but the operating system hides this and presents a single, shared-memory image to the user.

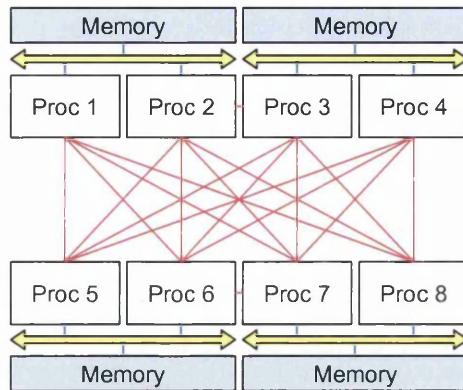
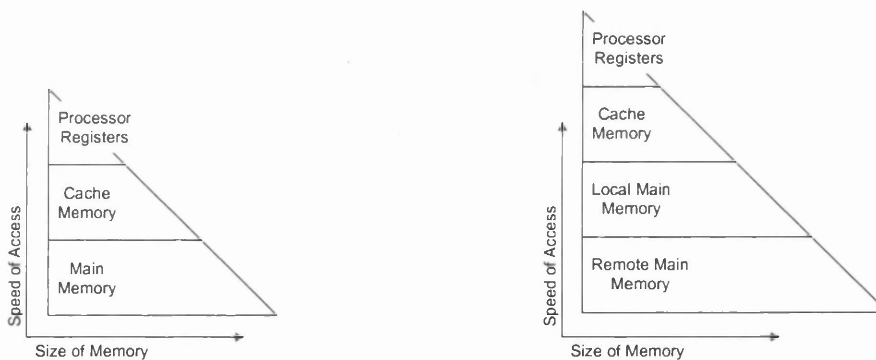


Figure 22 – The ccNUMA Distributed-Shared Memory Architecture

The architecture of the Origin is called ccNUMA (cache-coherent Non-Uniform Memory Architecture). To the user, this means that the memory hierarchy of modern RISC computers is extended one more level to include memory on remote processors as illustrated in Figure 23.



(a) Memory Hierarchy of RISC processor (b) Memory Hierarchy of ccNUMA architecture

Figure 23 – Memory Hierarchies of RISC and ccNUMA Architectures

This architecture has many of the advantages of the shared memory architecture but has been shown to scale beyond 512 processors.

2.2.4. The Message Passing Programming Model

Ironically, regardless of which parallel architecture is being used, by far the most common parallel-programming model is message passing. This is probably due to a number of key issues:

- The message-passing model can be implemented efficiently on any of the above architectures.
- There are programming libraries that hide many of the details of the underlying communications hardware, and present a standard, hardware-independent interface.

- Due to the fact that the data is sub-divided amongst the processes, cache utilisation is likely to be better than with a global data structure. This results in a program based on the message-passing model often outperforming an equivalent program based on a shared memory model even on a shared memory platform.

2.3. Summary

The previous two sections present a basic introduction to the fields of three-dimensional graphics and parallel processing. Both of these technologies play an important role in the design and implementation of the two PSE's described in this thesis and will be expanded upon in later chapters.

Chapter 3. PROMPT – An Implementation of a Problem Solving Environment

The purpose of this chapter is to describe an implementation of a Problem Solving Environment called PROMPT (PRe-processing Option for Military Power-plant Technology). This environment was developed for Rolls-Royce plc and DERA (Defense Evaluation Research Agency) with the aim of enabling the actual design engineers to make use of the existing numerical analysis software already developed in the two companies.

The first section of this chapter will provide an overview of the PROMPT environment by placing it in the context of the two companies. This will then be followed by a description of the overall architecture of the PROMPT environment along with the global data structures used throughout the environment. Finally, the operation and implementation of each of the modules is described.

3.1. Requirements

PROMPT was primarily developed for the Nozzle After-body division of the Military Power-plant Technology group. The nozzle / after-body is a key element to a successful engine – airframe integration. There are many factors influencing its design (Figure 24) but a key factor is its drag since it can make up between 30% and 50% of the total drag of the aircraft.

- The main design requirements are:
- High internal performance
- Low drag
- Low cooling flow
- Low weight
- Low maintenance
- Maintained engine matching
- Observables management and
- Thrust vectoring (Figure 25).

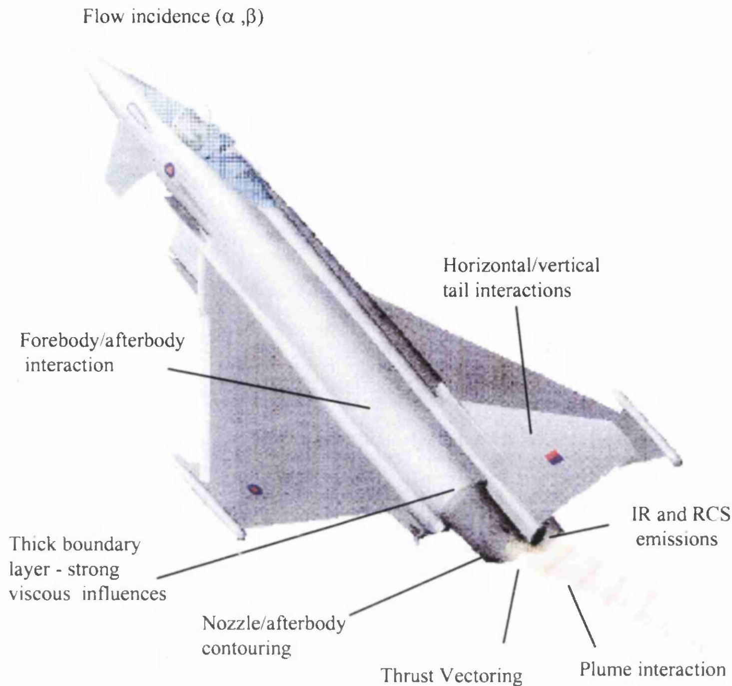


Figure 24 – Influences on the Design of Military Nozzle / After-Bodies

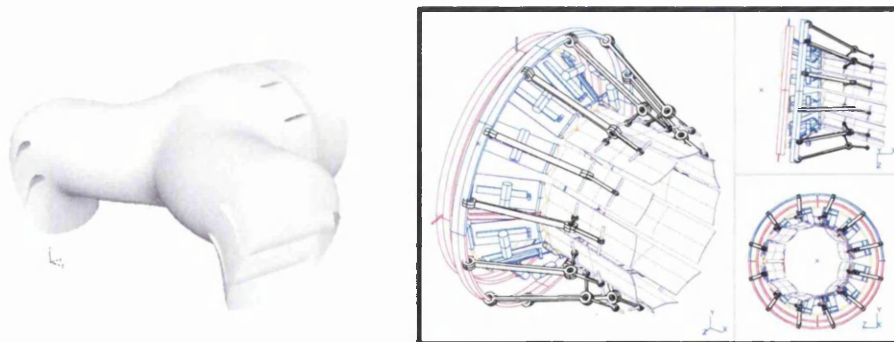


Figure 25 – Thrust Vectoring Technologies

3.1.1. Current Status at Rolls-Royce (circa 1995)

Considerable effort has been expended over a number of years in developing proprietary computer simulation technology in order to be able to perform many more iterations of the design process much more quickly and cheaply than was possible with traditional experimental processes such as wind tunnels. However, these tools were primarily suited to the applications of civil aerospace design and turbine blade design where the changes in shape and topology between different products were minimal. When these tools were subsequently applied to the wide variety of military nozzle designs, the meshes had to be

created by individually authored Fortran codes involving upwards of 1000 lines. This process could take a skilled person at least one month and upwards of six months in unskilled hands.

Once the meshes were generated they were visualised using command-line driven applications, which restricted the view, in many cases, to two dimensions, and the quality of the mesh was judged by eye. In order to execute the flow solver, the user was required to edit up to eight command files, each containing obscure file notation only made intelligible by referencing the solvers user guide (Figure 26). This was highly susceptible to user error.

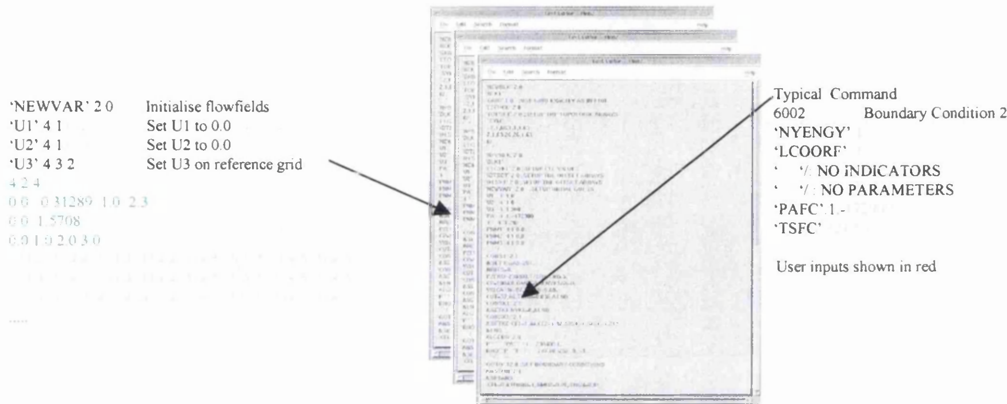


Figure 26 – An Example of the Solver Control Files (c.1995)

The post-processing of the results was then performed using a combination of four simple, graphical tools and command-line driven tools in which the user had to recall features of the mesh in terms of (I, J, K) plane or (x, y, z) notation.

The simulation process as described above was far from ideal. It required a skilled programmer (for the mesh generation), a very experienced user (to judge the mesh quality) and a combination of a very experienced user and the design engineer to perform the simulation, which was highly susceptible to user error through the use of unintuitive control files. The combination of all of these different stages, including the time to bring the relevant people together, meant that simulations could take of the order of 25-30 weeks which was unacceptable and meant that the take-up of the software by the design engineers was slow. The need for a Problem Solving Environment, as defined in Chapter 1, was obvious if these simulations were to be performed:

- In an error-free manner
- In a reasonable time frame
- By the design engineer and
- Without the need for a skilled programmer/user.

3.1.2. Aims of the PROMPT Environment

To address these issues, the PROMPT project was started with the following aims[2]:

- Enable CFD computations for Nozzle / After-body configurations to be prepared, initiated and examined within an intuitive workstation environment by non-specialist personnel.
- To allow meshes from a range of sources to be submitted to the Rolls Royce production solvers thereby avoiding the memory and CPU time overheads characteristic of commercial codes.
- To enable exploitation of the best of current and future in-house, commercial and University mesh generation and solver developments.
- To dramatically reduce the time needed to apply CFD to nozzle / after-body configurations.
- To minimise cost and lost time arising from pre-processing errors.
- To provide portability across SGI and Hewlett Packard workstations[3].

3.2. Scope and Context of PROMPT

These aims were subsequently expanded into the processes that needed to be performed within the PROMPT environment (Figure 27). This defined the PROMPT environment as "An intuitive user interface and graphics environment encompassing":

- Mesh and Solution file Input / Output Translators
- Grid Visualisation and Diagnostics
- Grid Refinement
- Boundary Condition and Solver Control Definition
- Multiple, embedded solvers accessed from the same interface and graphics tools
- Convergence Monitoring
- Solution Visualisation
- Data Plotting
- Data Integration
- Output Translators to non-embedded solvers and special-purpose post-processors and
- On-line Help.

The logical processes encapsulated in the PROMPT environment are shown in Figure 27.

[2] Extracted from the brochure titled 'Applied Research Package 07b Milestone M83501 Review'
[3] The range of Sun workstations was later added to this list.

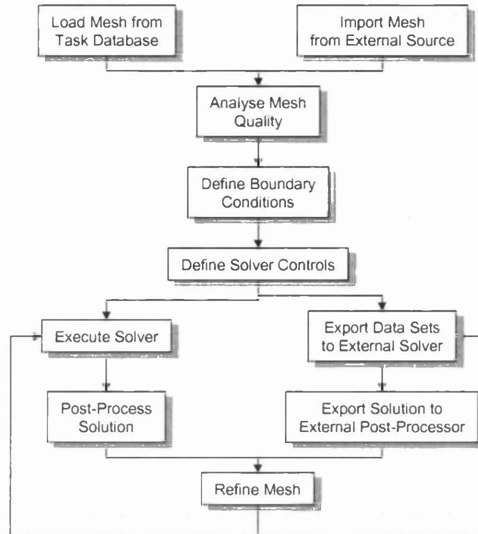


Figure 27 – Logical Processes performed within the PROMPT Environment

Figure 28 shows the number of other systems both proprietary to Rolls Royce, and commercial, that PROMPT needed to interact with.

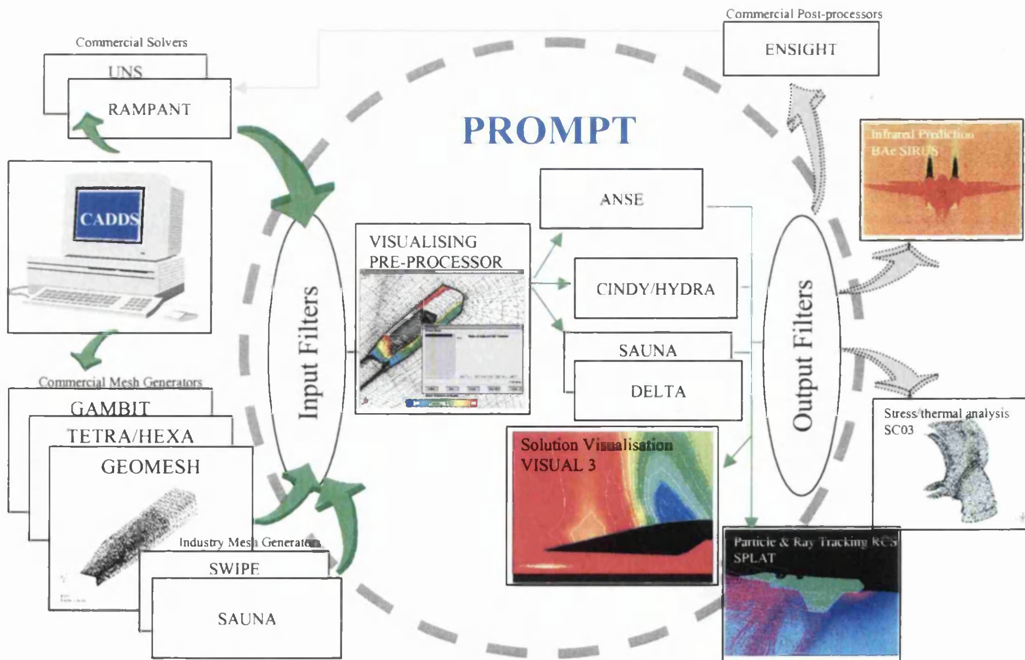


Figure 28 – The Options contained within PROMPT

3.3. The Architecture of PROMPT

Once the aims of PROMPT were defined, the next decision to make was how to perform the implementation in order to meet the objectives and provide for future expandability and ease of maintenance in an industrial environment.

3.3.1. The Structure of PROMPT

The first decision to be made was whether the environment was to be implemented as a single module or whether it should be a number of closely coupled modules. Although a single module would be conceptually simpler to implement, splitting it up into a number of modules was seen to have a number of benefits:

- Each module would be smaller and easier to manage during initial implementation and future maintenance.
- The overall robustness of the system would be increased. This was based on the assumption that the central module, which would store the computational data sets, would be both as simple as possible and would not change much throughout the life of PROMPT. These two design considerations would combine to minimise the occurrence of any software errors (bugs). This would mean that regardless of the robustness of the other, more complex, modules the main data sets would remain intact.
- It would allow PROMPT to fit into the software structure currently in place in Rolls Royce. The current system (called SWIPE [Bradley91a, Bradley91b, Northall02]) employs a configurable Menu Module that allows the user to select the required task. The module that then performs that task would then be executed. If PROMPT was a single module then it would only be shown as a single option in this Menu Module, which would make PROMPT look like a completely separate environment from the SWIPE system. However, if it was a number of separate modules, one for each logical process in the CFD analysis, then this could be represented as a number of options in the Menu Module. This would make PROMPT look more integrated into the SWIPE system thus allowing the system to be configured easily for each type of user. This was important because designers of different parts of the aircraft propulsion system had different requirements in terms of mesh generators and solvers. In order to accommodate these different requirements, PROMPT either had to present each user with all of the possible options (which could cause confusion) or be tuneable to each users needs.
- In order to ensure portability across the different UNIX platforms mentioned previously, all source code will be in ANSI C, the graphical user interface would be written using OSF / Motif [Nye88, Nye90, Nye93, OSF93, OSF95] and all three-dimensional graphics will be rendered using OpenGL.

The resulting architecture is illustrated in Figure 29 and is composed of nine main modules:

- Main Menu Module
- Data Storage Module
- The Visualisation Module

- The Task Database
- The Mesh Quality Analysis Module
- The Boundary Condition Definition Module
- The Solver Controls and Solver Execution Module
- The Solution Post-Processing Module and
- The Mesh Refinement Module.

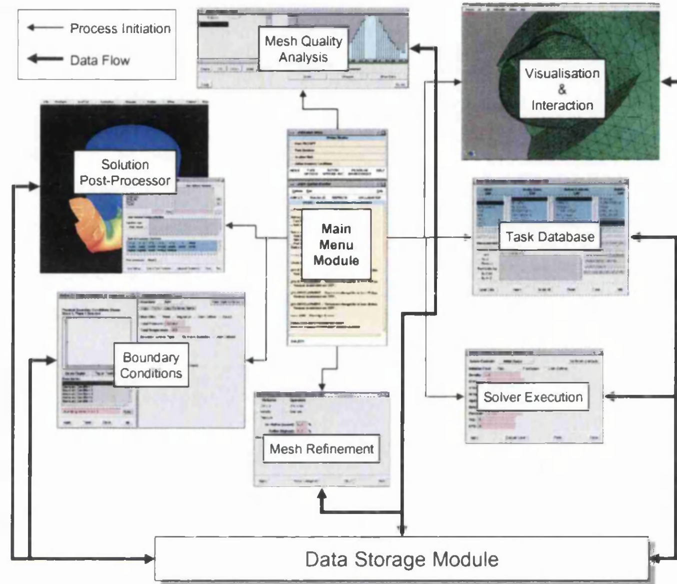


Figure 29 – The Architecture of PROMPT

Here, the two key modules are the Rolls Royce Main Menu Module, which initiates each of the modules, and the Data Storage Module, which is responsible for storing all of the mesh and solution data sets. The user then initiates the other modules in order to perform one of the logical processes in the simulation.

3.3.2. The Communication Mechanism used within PROMPT

The second decision was how the various modules within the PROMPT environment would co-operate. A number of alternative solutions were investigated:

- MPI (Message Passing Interface) [Dongarra95, Gropp99a, Gropp99b],
- PVM (Parallel Virtual Machine) developed at Oak Ridge National Laboratory and University of Tennessee [Sunderam90, Beguelin94],
- Native Shared Memory [Stevens90] and
- Native UNIX Sockets [Stevens90].

During the execution of PROMPT, modules will be initiated and then terminated. Upon initialisation, they will need to connect to each other in order to co-operate. This dynamic nature meant that MPI was not suitable since it imposed a static set of processes throughout the execution of the environment.

PVM does allow dynamic process configurations but imposes the restriction that a process can only be added to the group of communicating processes by one already in that group. As mentioned previously, the Main Menu Module, which is not in a PVM group, will initiate the PROMPT modules and then connect them to the Data Storage Module, which would be in a PVM group. This means that PVM is also not suitable.

The third alternative was to use the native shared memory mechanism provided by the Operating System. This would allow both the dynamic connection and disconnection of processes without imposing the restrictions of PVM. However, the shared memory implementation on most platforms does have its own drawbacks:

- At the time, there was no standardised interface to the shared memory mechanism thus using shared memory in a manner that was portable to a number of different platforms would be difficult.
- The same was true of the locking mechanisms supplied by the various Operating Systems in order to ensure that only one process could access the shared regions at any one time.
- The number of individual shared memory segments that could be allocated per process was limited (sometimes as little as 4-5). This meant that a naturally hierarchical set of data structures would have to be flattened in order to be stored in a small set of contiguous memory regions. The flattening of a typical tree-like data structure is illustrated in Figure 30. The flattening of the data structures would have a major impact on the dynamic nature of the data structures. For example, increasing the size of one mesh block would mean shifting all of the data stored in that shared memory segment in order to make room. Adding any extra data to a shared memory segment, requires allocating an entirely new segment with space for the extra data, the existing data must be copied into the new segment and then the old segment released back to the system. This has significant memory and speed penalties.
- If the environment does fail for any reason then it is very difficult to ensure that any shared memory segments are freed cleanly since they are not freed automatically by the OS upon the program exiting. These can then build up over time until codes that use shared memory will no longer run. This can only be cured by either using special commands to force memory segments to be freed (assuming it is known which ones are not currently in use by another program) or rebooting the workstation.

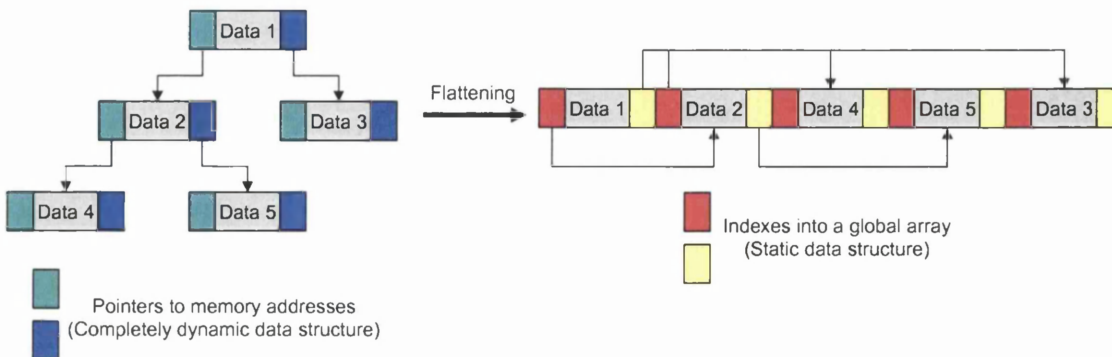


Figure 30 – Flattening of a typical hierarchical data structure

The fourth alternative was to use the native socket communication system supplied by the OS. This method has the advantages of having a standard interface throughout all UNIX workstations, imposing none of the restrictions mentioned above, and the OS performing all necessary cleanup operations if a program exits prematurely. The only requirement is that each program can gain access to a string and a number that uniquely identifies the communication port of the Data Storage Module. This requirement is easily met by storing this data in a file in a known location with a known name. An example could be `"/tmp/promptcomm.<uid>"` where `<uid>` is the user's unique identifier.

3.3.3. Module Initiation and Termination

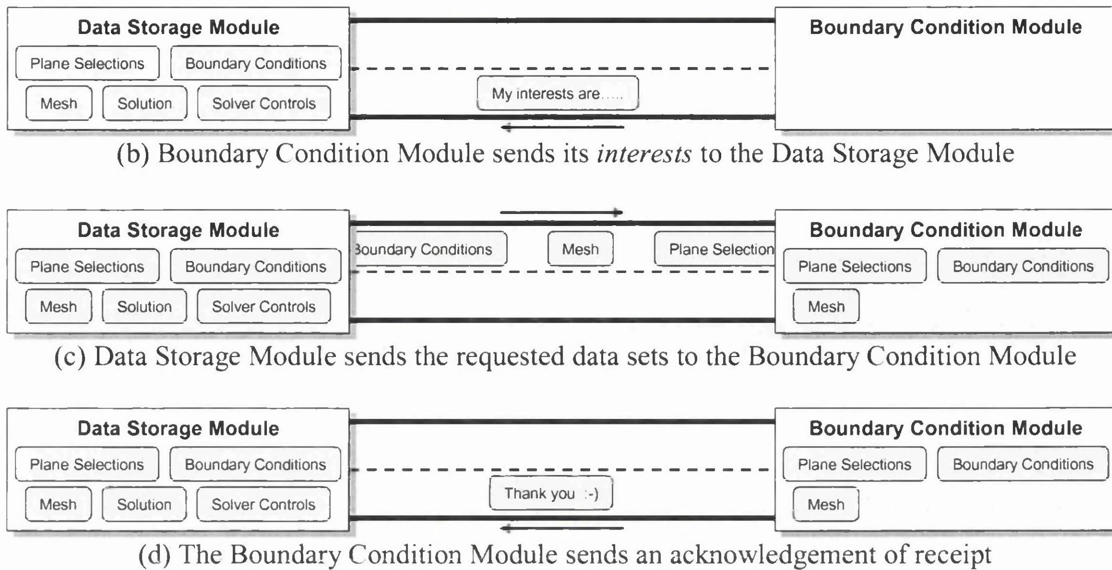
The method chosen for implementation within the PROMPT environment was the UNIX socket method. This means that when a module is initiated through the Main menu Module, it must establish a communication path to the Data Storage Module in order to receive any necessary data. This is achieved by reading a small text file in the "`/tmp`" directory initially written out by the Data Storage Module. This text file contains two items, the IP address of the computer on which the Data Storage Module is running and a port number. The new module uses this data to request a connection. On receiving this request, the Data Storage Module accepts the connection and a new communications path is formed.

The new module then informs the Data Storage Module of its *interests*. These interests include data sets, such as meshes, boundary conditions and solutions, and events, such as the user selecting a mesh plane via the Visualisation Module. Expressing a modules' interests in this way has two advantages:

- The Data Storage Module knows which data sets the module needs and, thus, does not need to communicate all of the data sets. This improves efficiency.
- The Data Storage Module doesn't need to know in advance the interests of every module that will be connected to it. This enables the Data Storage Module to adapt to future developments in each of the modules in PROMPT without having to be changed. This improves the robustness of the Data Storage Module, and hence, the robustness of the entire environment.

An example of this process is shown in Figure 31. Here the module that is responsible for allowing the user to define boundary conditions is initiated. This module needs the mesh and current boundary condition data sets, and needs to be told when the user, in the Visualisation Module, has selected a mesh plane.

(a) Boundary Condition Module is initiated



(b) Boundary Condition Module sends its *interests* to the Data Storage Module

(c) Data Storage Module sends the requested data sets to the Boundary Condition Module

(d) The Boundary Condition Module sends an acknowledgement of receipt

Figure 31 – Module Initiation within PROMPT

At any time during a module's execution, it may:

- Change its interests by sending a new list to the Data Storage Module. This may cause the Data Storage Module to send new data sets, if necessary.
- Update the Data Storage Module with new data. This would cause any other connected modules that are interested in that data set to be updated.

When a module exits cleanly, through the user pressing the 'Close' button on the panel, it informs the Data Storage Module of its intentions, closes its end of the communication path and then exits. The Data Storage Module receives the modules exit signal and closes its end of the communication path.

If a module exits prematurely for any reason, it will not have the opportunity to inform the Data Storage Module. The OS will automatically close the module's end of the communication path during its automatic cleanup operations. However, the Data Storage Module will not have been informed. To overcome this scenario, the Data Storage Module has two defences:

- It periodically pings each of its communication paths to test whether the module at the other end responds. If no response is received within a given time period (approximately 5 seconds), it closes the connection to that module.
- If data is inadvertently sent along a path to a module that has exited, then the OS automatically sends a signal to the sending process. If this is not caught and handled correctly, it results in an automatic termination of that process. The Data Storage Module registers a handler for this signal upon start-up and, thus, recovers from these communication errors by assuming the module at the other end has exited and closing the communication path.

3.4. Global Data Structures used within PROMPT

In order to ensure PROMPT continues to be in full use for as long as possible there was a need to implement data structures that could represent the majority of meshing topologies and solver requirements in use today. This list includes:

- Structured Curvilinear Grids (Single-Block),
- Structured Curvilinear Grids (Multi-Block),
- Structured Curvilinear Locally Refined Grids and
- Unstructured Grids with mixed cell types.

To allow the storing of these in a consistent manner throughout the execution of PROMPT, the top-level of the data structure is shown in Figure 32. This allowed both structured and unstructured meshes to be referred to as a *mesh*. The structures `StructuredMesh` and `UnstructuredMesh` are in a `union`, which means that they use the same segment of memory. The actual type of mesh that is stored is decided by the variable, `mesh_type`. The lower-level data structures that describe the specifics of each type of mesh are described in the following sections.

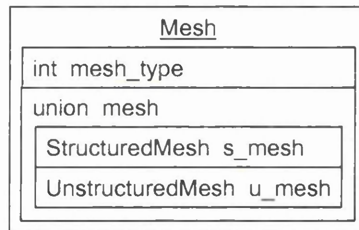


Figure 32 – The Top-Level Mesh Data Structure

3.4.1. Structured Curvilinear Grids (Single Block)

Although the single-block, structured grid imposes strict limitations on the complexity of the geometry that can be represented, its simplicity means it is still frequently used where possible. The data structure used to store the mesh is also very simple as shown in Figure 33. It is essentially a mapping between Cartesian space and parametric space.

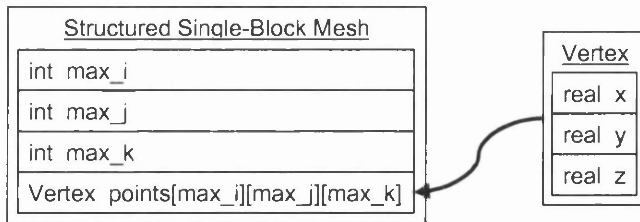


Figure 33 – Data Structure for a Single-Block Structured Mesh

For all but the simplest geometries, the use of a single block mesh is too restrictive. However, instead of choosing the much more complex multi-block strategy it is common

to generalise the single block approach by allowing nodes of the mesh to be placed inside the solid regions of the geometry. These are then flagged to the solver as *dead* nodes and can then be subsequently ignored for the computation. A simple 2D example of this is shown in Figure 34. The accompanying data structure, which is a simple extension of the previous data structure, is shown in Figure 35.

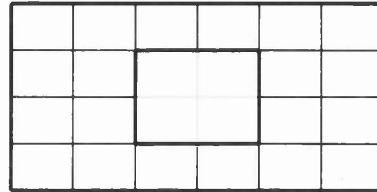


Figure 34 – An example of a Single-Block Structured Mesh with *Dead Nodes*

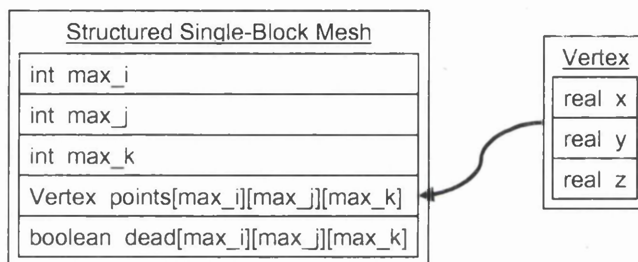


Figure 35 – Data Structure for a Single-Block Structured Mesh with *Dead Nodes*

3.4.2. Structured Curvilinear Grids (Multi-Block)

Structured multi-block grids allow the application of structured mesh generation and solver technology to much more complex configurations. A multi-block grid is composed of many single-block grids adjacent to each other, each with their own local parametric co-ordinate systems. These are stored along with information detailing how these blocks interface with each other. Figure 36 shows a simple multi-block mesh that imposes the limitation that any face on any block must be adjacent to at most one other block. This restriction is often relaxed so that any block face can be adjacent to any number of other blocks as shown in Figure 37. As can be seen, by relaxing this restriction the number of blocks required to represent the geometry is halved. Since a complex geometry may require many hundreds of blocks, these savings can be very significant, albeit with an increase in the complexity of the algorithms and data structures used within the actual solver.

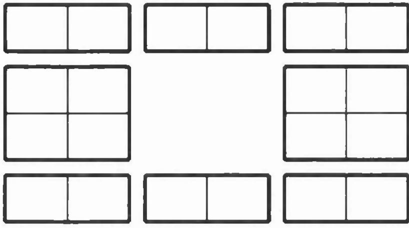


Figure 36 – An example of a Multi-Block Mesh with Single Face Matching

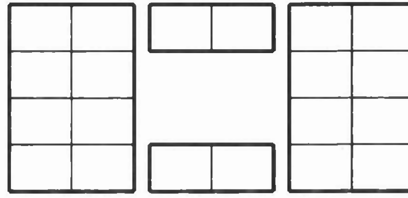


Figure 37 – An example of a Multi-Block Mesh with Multiple Face Matching

The data structures implemented within PROMPT allows any of the above multi-block configurations to be utilised. The data structure for a single or multi-block mesh is shown in Figure 38.

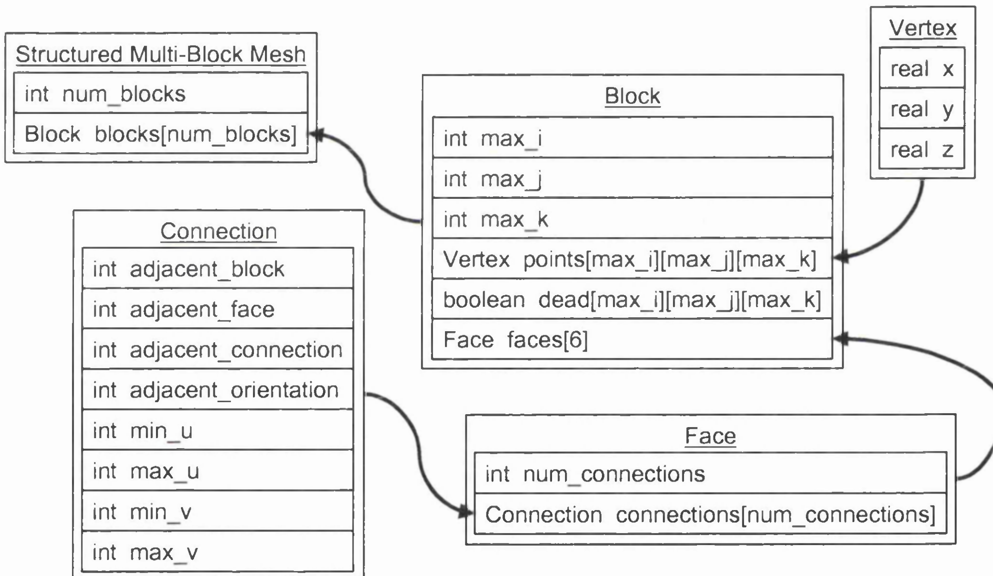


Figure 38 – Data Structure for a Multi-Block Mesh with *Dead Nodes*

The upper two data structures allow the mesh to comprise any number of blocks. The `block` structure also contains a set of six `face` structures, one for each face of the block. Each `face` structure then contains a set of *n* `connection` structures, one for each adjacent block face. The `connection` structure defines how the grid lines propagate through adjoining blocks. The first field, `num_connections`, specifies the number of adjacent block faces. The other fields are then dimensioned appropriately and contain all of the necessary details of the connection of each adjacent block. The fields are:

- `adjacent_block` – The number of the adjoining block.

- `adjacent_face` – The face number of the block specified in `adjacent_block`. Figure 39 shows the conventions used within PROMPT for the numbering of the faces of a block.
- `adjacent_connection` – The connection number of the face specified in `adjacent_face`.
- `adjacent_orientation` – Each face of a block is given a unique local (u, v) coordinate system as shown in Figure 39. The `adjacent_orientation` field contains a code that uniquely identifies the mapping between the two interfacing blocks. Figure 40 shows the eight possibilities.
- `min_u`, `max_u`, `min_v`, `max_v` – These fields define the region of nodes that are coincident with the adjacent block.

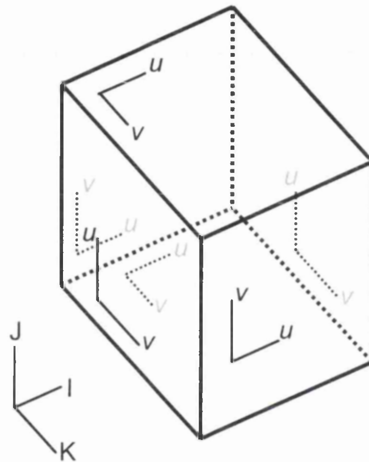


Figure 39 – Block and Face Coordinate Axes

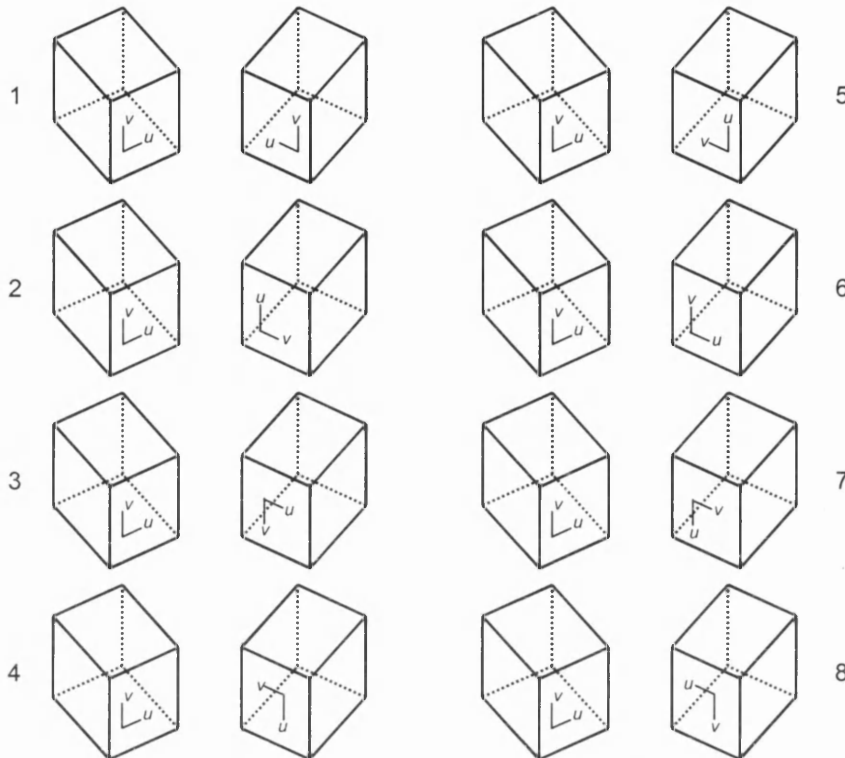


Figure 40 – Numbering Convention for Block-Block Interfaces

3.4.3. Structured Curvilinear Grids with Local Refinement

One of the problems with structured grids is that of mesh refinement. As mentioned previously, a common path through the latter stages of the simulation process is to cycle between the steps of:

1. Calculate the solution using the current grid
2. Interrogate the solution to find regions of the grid that need more resolution
3. Increase the density of the nodes in the regions indicated
4. Interpolate the solution from the old grid to the new grid and then
5. Obtain another solution on the new mesh using the interpolated solution as the starting point.

This cycle is repeated until the solution is of an acceptable accuracy. When using a structured mesh, if any cells are sub-divided then the i,j,k planes which are created by the new node must be propagated throughout the entire mesh in order to maintain conformance. This invariably causes regions, in which there was little solution change, to be refined unnecessarily thus placing an extra burden on the solver.

To overcome this deficiency, a technique known as *local refinement* is often used. This enables any new nodes to be added to the mesh with the new mesh planes only being

propagated as far as required. This approach introduces into the mesh a set of *hanging nodes* as shown in Figure 41.

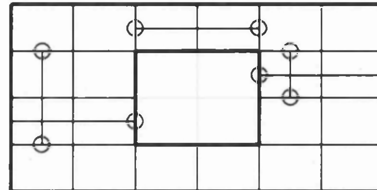


Figure 41 – A Structured Mesh with Hanging Nodes (and *Dead Nodes*)

In order to maintain the structured nature of the grid, and enable rapid traversal of the nodes, the data structure in Figure 42 was implemented.

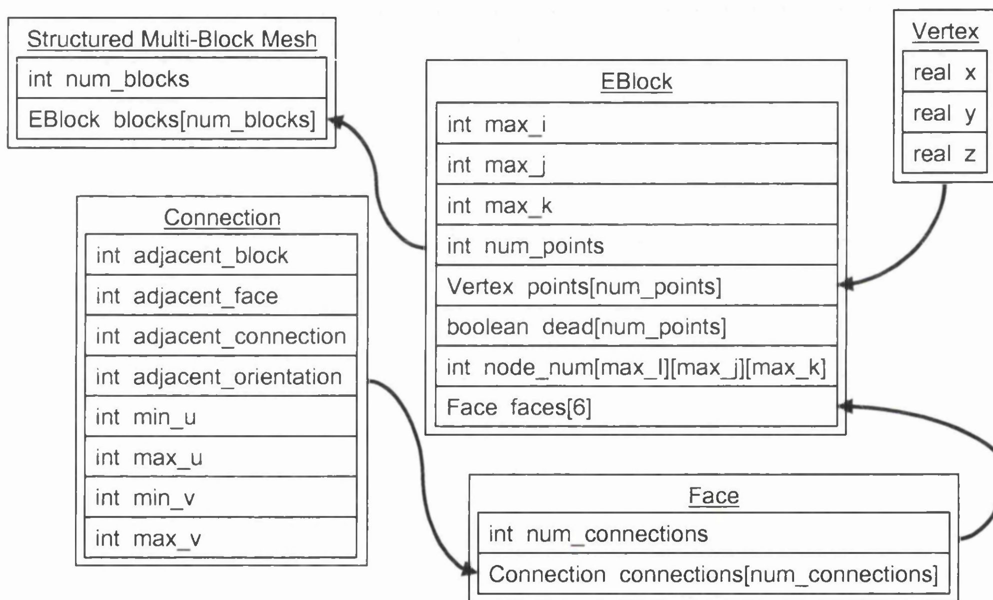


Figure 42 – Data Structure for a Multi-Block Mesh with Hanging and Dead Nodes

This contains an index array, `nodeNum`, which maintains the (i,j,k) structure of the grid as if the new planes were propagated. The coordinates of the real nodes are then stored in a linear array, `points`, which is indexed by the array `nodeNum`. Any entry in the `nodeNum` array that does not represent a real node contains –1. In order to resolve any ambiguities, an additional array, `nodeConn`, was introduced. This array is structured in the same way as the `nodeNum` array and either, contained a 0 if no hanging node is present, or an encoding of the mesh planes that meet at the hanging node. This encoding is defined as:

$$nodeConn = connection(I) + 4 * connection(J) + 16 * connection(K)$$

Where

$$connection(A) = 0$$

If node is connected in +ve A direction **then** $connection(A) = connection(A) + 1$
If node is connected in -ve A direction **then** $connection(A) = connection(A) + 2$

Figure 43 and Figure 44 show an example of a simple grid. The grid on the left shows the (i,j,k) structure of the grid with any added planes being propagated throughout the mesh. The grid on the right is the same but with examples of the values stored in the `nodeConn` array at the hanging nodes.

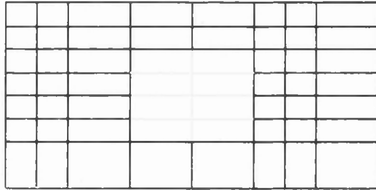


Figure 43 – The (i,j,k) structure of a mesh with hanging nodes

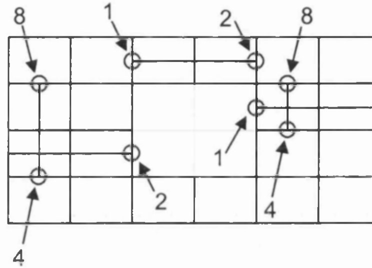


Figure 44 – The contents of the *nodeConn* array for the hanging nodes

3.4.4. Unstructured Hybrid Grids

Unlike their counterparts, unstructured grids have no mapping between parametric space and Cartesian space. Instead, an unstructured grid comprises a set of nodes irregularly placed within the domain. A connectivity table is then used to define the joining of these points in order to form surface and volume elements. A *hybrid* mesh usually contains a combination of element types chosen from the set of volume elements (tetrahedra, pyramid, prism and hexahedra) and a set of surface elements (triangles and quadrilaterals).

In order to accommodate the use of hybrid meshes within PROMPT, the data structure shown in Figure 45 was implemented. The conventions used within PROMPT for the node numbering of each element type is shown in Figure 46.

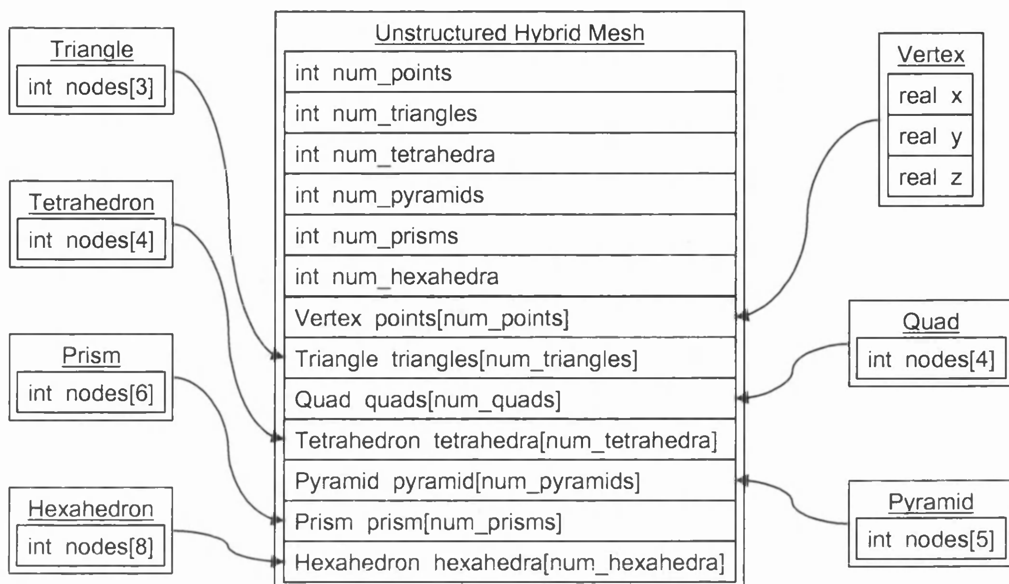


Figure 45 – Data Structure for an Unstructured Hybrid Mesh

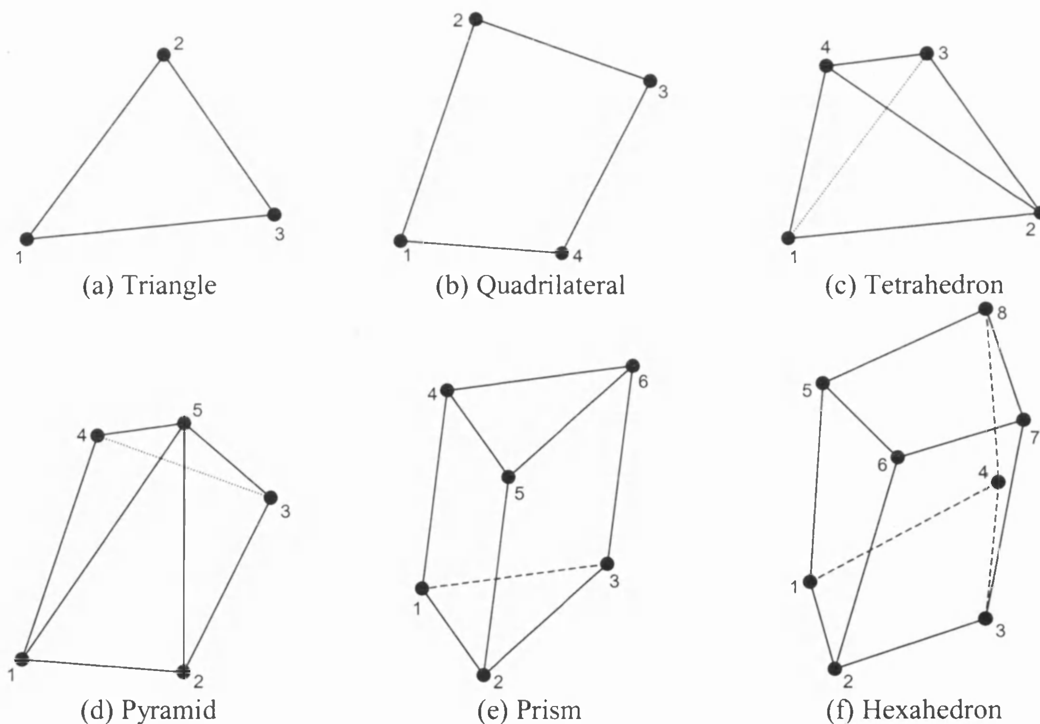


Figure 46 – Node Numbering Conventions used for the Unstructured Element Types

3.5. The Visualisation and Control Module

The most important module in PROMPT is the Visualisation and Control (V&C) Module. It provides the main front-end for all of the other modules; it displays and allows the user to interact with the 2D and 3D data sets; it contains all of the data sets within PROMPT and it co-ordinates the communication between all of the other modules within PROMPT. Throughout the development of the V&C Module a large emphasis has been placed on speed; not only in the manipulation of data on the screen, but also in the selection of features of that data. At each stage of the development, the design of the data structures and algorithms have been oriented towards providing an increase in speed for operations that are performed frequently, even if it is at the possible expense of some operations that are performed infrequently. An example of this is the increase in computation necessary to construct the data structures for a mesh when loaded from disk in order to increase the performance of any screen updates or feature selections.

3.5.1. The Visualisation Window

A typical example of the Visualisation Window is shown in Figure 47. As can be seen, the Visualisation Window covers the full display of the workstation. This allows the user to examine and manipulate the 3D data sets in more detail.

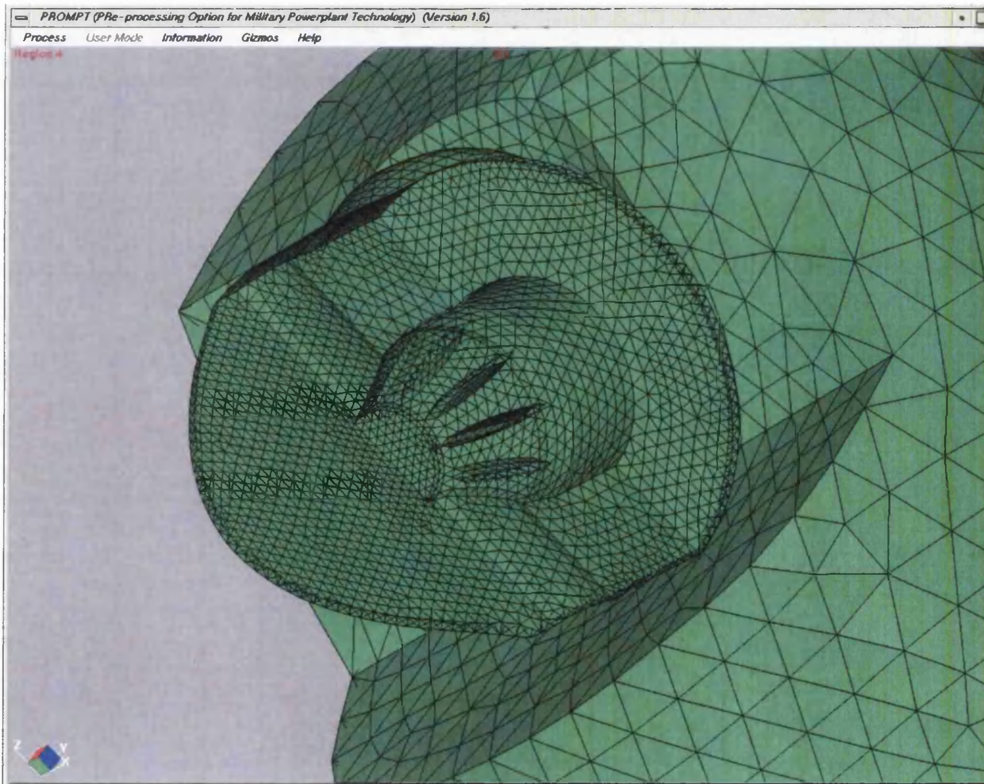


Figure 47 – A typical example of the Visualisation Window

3.5.2. Region and View Configurations

During the early development of PROMPT, the need to have different ways to view the object became apparent. For example, when the operations consist entirely of looking at or selecting features in a mesh then a single, full-screen view of the mesh is desirable since this allows small features to be distinguished easily. Whereas when a position in space is being identified then the only way of specifying the x, y and z coordinates is to have three views of the mesh, the convention being front, top and left views.

In PROMPT, this flexibility is taken two stages further. Firstly, the user can configure the display to show any number of regions between 1 and 4; and second, the user can configure any of the visible regions to show any view of the object. The possible combinations of regions available in PROMPT are shown in Figure 48. Within each of these regions the user can choose between one of the six orthographic projections (front, rear, left, right, top and bottom) and a fully rotateable view.

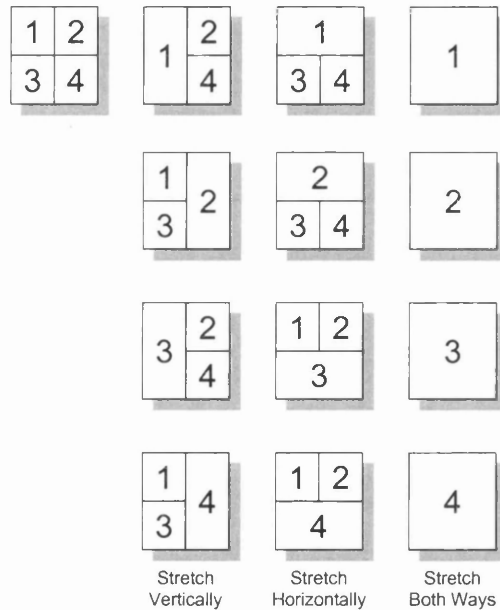


Figure 48 – Possible Region Configurations

3.5.3. On-Screen Manipulation of Data

Objects on the screen in the Visualisation Window may be translated, enlarged or rotated. All of these operations are performed using the movement of the mouse with the middle mouse button pressed and a combination of the *SHIFT*, *CTRL* and *ALT* keys. All three of these manipulations are performed in a manner that is deemed most intuitive to the user.

Scaling

The scaling of the model is the simplest manipulation to perform. The current scaling factor is determined in an incremental fashion using the formula, $S = MAX(0.0001, S + D/100)$, where S is the current scaling factor and D is the

distance the mouse has travelled since the last time its position was recorded. This formula achieves two intuitive features:

- As the scaling factor increases so does the amount it increases for a given mouse movement. This allows the user to zoom into an area of the model without needing to use an excessive amount of mouse movement.
- Performing a mouse movement and then returning the mouse to the original position returns the model to the original scale.
- The '*MAX*' part of the expression ensures that the scale never reaches zero, thus causing the model to disappear, and it never becomes negative, causing the model to invert.

During the rendering process, the scaling operation is always the first to be applied. This means that the user always zooms into the centre of the display. This was felt to be more intuitive than applying the operation last which would cause the user to zoom into the centre of the model regardless of where it was positioned on the display.

Translation

The most intuitive form of translation of the model is for it to behave as it was attached to the mouse pointer, i.e. if the user clicks on a feature of the model and then moves the mouse, that feature will stay positioned under the mouse pointer. This is achieved by simply maintaining a translation vector that has its x and y components incremented by the distance the mouse pointer moves in each direction. Maintaining the model under the mouse pointer regardless of the size of the display or the model is achieved by a simple scaling of the mouse movements taking into account the difference between the co-ordinate system in which the model is stored and the pixel co-ordinate system of the display.

In order for the model to travel in the direction of the mouse pointer regardless of its current rotation, the translation is always performed second. Performing it before the scaling would cause the translation to be multiplied by the current scale factor and, thus, would result in uncontrollable behaviour at high scale factors. Performing it after the rotation would cause the model to be translated along its own local axes rather than the global axes of the display.

Rotation

The rotation of the model is the most complex operation to make intuitive. The aim is to rotate the model around the global vertical axis when the mouse is moved from left to right and around the global horizontal axis when the mouse is moved up and down. This is achieved by maintaining a *current* rotation matrix and continuously pre-multiplying it by the rotation matrix constructed by the current mouse movements as shown in the following equation:

$$R' = \begin{bmatrix} \cos\delta y/2 & 0 & \sin\delta y/2 & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\delta y/2 & 0 & \cos\delta y/2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\delta x/2 & -\sin\delta x/2 & 0 \\ 0 & \sin\delta x/2 & \cos\delta x/2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} R$$

Here the first matrix performs the rotation around the y-axis, the second performs the rotation around the x-axis and R is the current rotation matrix. The variables δx and δy are the current movement of the mouse pointer in their respective directions.

Global Transformation

Combining each of these transformations the matrix pipe-line is:

$$\begin{bmatrix} 1 & 0 & 0 & -x_c \\ 0 & 1 & 0 & -y_c \\ 0 & 0 & 1 & -z_c \\ 0 & 0 & 0 & 1 \end{bmatrix} R \begin{bmatrix} 1 & 0 & 0 & x_t \\ 0 & 1 & 0 & y_t \\ 0 & 0 & 1 & z_t \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} s & 0 & 0 & 0 \\ 0 & s & 0 & 0 \\ 0 & 0 & s & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_m \\ y_m \\ z_m \\ 1 \end{bmatrix}$$

Where

$[x_c \quad y_c \quad z_c]$	is the centre of the model,
$[x_t \quad y_t \quad z_t]$	is the current translation vector,
R	is the current rotation matrix,
s	is the current scaling factor and
$[x_m \quad y_m \quad z_m]$	is each coordinate of the model.

3.5.4. Feature Selection

Another operation performed in the Visualisation Window is the selection (or *picking*) of features in the mesh. At first, this operation may seem to be trivial; the user places the mouse pointer over the required feature and then clicks the left mouse button. However, that action can be inherently ambiguous. For example, when a user clicks on a section of a multi-block, structured mesh the software has to decide whether the user wishes to select a face of a cell, a cell, a plane of a mesh block, an outer surface of a mesh block or the entire mesh block. In PROMPT, the user decides this by selecting the feature of interest in the Selection Gizmo. As can be seen in Figure 49, the Selection Gizmo consists of a number of buttons each representing a feature of the mesh that the user might need to select for a given operation. The interests of the module currently connected to the V&C Module decide the options available to the user; the remaining being ghosted.

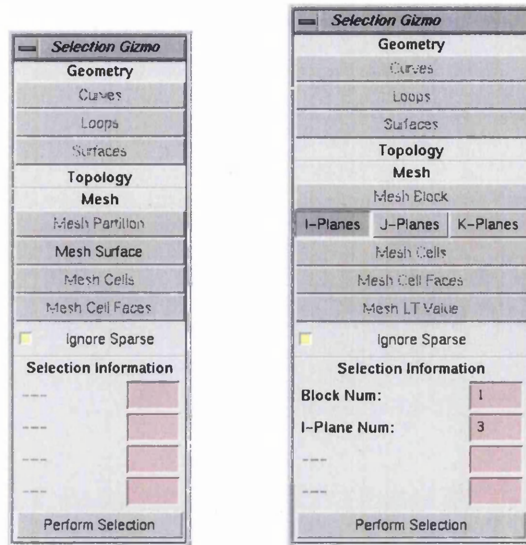


Figure 49 – The Selection Gizmo

3.5.5. Feature Selection Algorithm

The algorithm used to decide which item is being selected by the mouse pointer depends on whether the item consists of points / lines or faces. Regardless of the algorithm, each of the primitives to be tested is projected from the three-dimensional space of the model to the two-dimensional space of the display using the matrices described above. If the items to be selected are points or lines, then the algorithm simply selects the item with the smallest perpendicular distance to the cursor.

If solid faces are to be selected then the Crossings Test [Shimrat62] is used on the two-dimensional projected polygons. This algorithm simply projects an infinitely long line from the cursor along the x-axis. The number of times this line intersects with a polygon then determines whether the cursor is inside or outside of the polygon. If the line crosses the polygon an even number of times then it is outside, otherwise it is inside. Both cases are illustrated in Figure 50 and Figure 51.

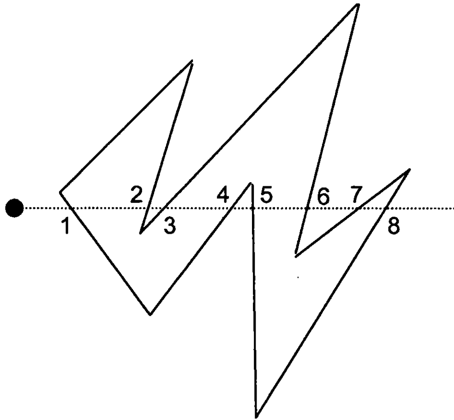


Figure 50 – Point outside the polygon

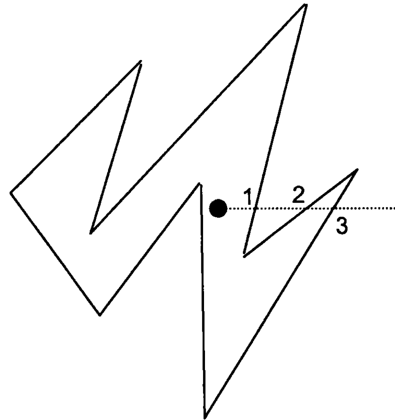


Figure 51 – Point inside the polygon

If a cursor is deemed to be inside the polygon then the projected *z*-coordinate of the intersection of the cursor and the polygon determines the front-most polygon.

This algorithm is very simple to implement; however, it does have an ambiguity if the infinite line crosses a polygon exactly on a node. If this occurs then there are two possibilities. If the nodes adjoining this node are on opposite sides of the infinite line then the node is counted as one (Figure 52), otherwise it is counted as either zero or two, since both produce the same result (Figure 53).

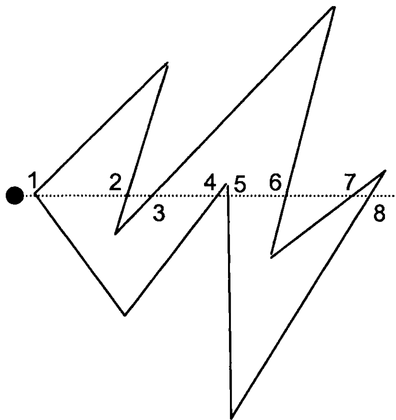


Figure 52 – Node counted as one intersection

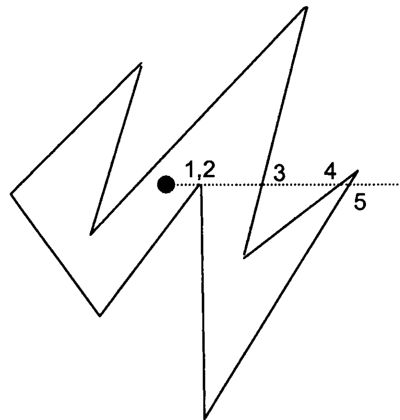


Figure 53 – Node counted as zero or two intersections

Although this algorithm is simple and very efficient, the need to traverse every polygon within the mesh in order to determine which has been selected can still lead to performance degradation. In order to overcome this, the polygons are grouped by their block number, plane direction and plane number for structured meshes; and by surface number for unstructured meshes. The selection algorithm is then modified to test the cursor against the projected bounding box of each of these groups with the individual

polygons being tested only if the cursor falls within its bounding box. This grouping enables a large number of polygons to be disregarded without the need to test each one individually.

A number of other algorithms were considered for feature selection.

Angle Summation Test

This algorithm forms the sum of the signed angles formed at the test point with the endpoints of each edge. If the sum is near zero then the point is outside otherwise it is inside. Although this algorithm is simple it is computationally expensive since for each edge, a square root, arc cosine, division, dot and cross product must be computed.

Open-GL Selection Mechanism

The Open-GL system has a facility to perform image-based selection [Neider93, OGL-ARB92]. When in this mode the image on the display is untouched. Instead when the drawing commands are issued, each primitive can be assigned a unique index and the system logs the list of primitives that intersect with the viewing volume. Using the projection matrix, the viewing volume can be shrunk to form a small square around the cursor. Any primitive that intersects with this volume could be considered a candidate for selection. The speed of this method depends on the speed of the graphics hardware that, with modern workstations, is sufficient even for large models. However, if more than one primitive intersects with the viewing volume then there is no easy way of determining which primitive was in front (for solid faces) or the closest to the cursor (for points and lines).

Framebuffer Selection Technique

If the image is drawn into an off-screen buffer using a colour-index[4] mode rather than an RGB[5] mode, then each polygon could be drawn with a unique index. Picking a polygon is then just a simple case of reading the colour index from the frame buffer at the required position [Hanrahan90].

This technique is very simple to implement and with accelerated graphics hardware one polygon can be picked in a fraction of a second even when millions of polygons exist. However, it does have two major drawbacks:
On workstations with low to medium range graphics capabilities, the range of indices can be limited, sometimes as low as 256 (for 8-bit displays, or 65536 for 16-bit displays). This means that polygons must be grouped with each group being drawn with the same colour index. This limits the ability of the user to be able to pick single polygons.

[4] Colour Index mode allows the user to specify colours as an integer index that is then used in a lookup table to form the actual colour.
[5] RGB mode allows the user to specify colours as triples of red, green and blue. This value is then either stored in the frame buffer or is approximated by stippling if the frame buffer cannot represent the colour precisely.

This method does not work for selecting points or lines because if the user clicks on an area close to a point or line then the colour index read from the frame buffer will be that of the background and no point or line will be selected.

3.5.6. The Pull-down and Pop-up Menus

In PROMPT, there are two types of menu used; the pull-down menu and the pop-up menu. The pull-down menus are used for *global* operations such as initiating the other PROMPT modules, retrieving size statistics about the data sets and initiating the various Gizmo panels (these will be described later). The pop-up menus are used to control operations pertaining to the region in the display area that currently contains the mouse pointer.

The hierarchy of the pull-down menus is shown in Figure 54. The first menu provides the list of modules in PROMPT that may be initiated. These modules will be described in detail in later sections.

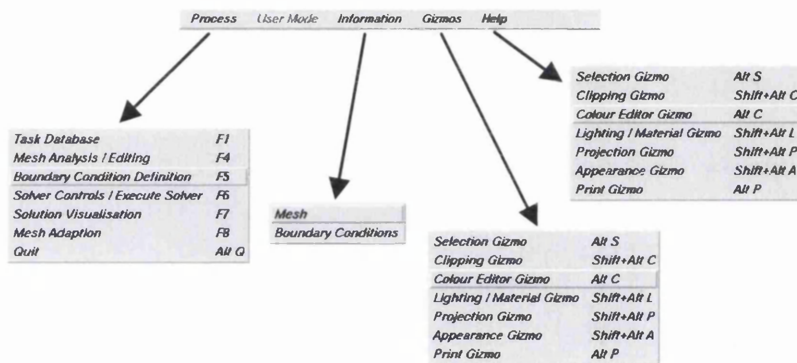


Figure 54 – The Pull-down menu hierarchy

The second menu allows the user to retrieve various size statistics about the mesh, boundary conditions, solution, etc. The panels for these are shown in Figure 55.

Figure 55 – The Information Panels

The third menu allows the user to open the various Gizmo panels available within the V&C Module. The Gizmo panels are used to split the functionality of the V&C Module into logical groups. Each of these Gizmo panels will be described later. The last menu pane provides the user with context sensitive on-line help for the various operations available within PROMPT. A Help button is also available at the bottom of every panel. This takes you directly to the help associated with that panel.

The Pop-up menu contains options pertaining to the region that the mouse pointer currently occupies. The hierarchy of options is shown in Figure 56.

Views

This allows the user to choose the view that will be displayed within the current region.

Regions

This allows the user to choose the size for the current region. This is performed by doubling the region in the horizontal and / or vertical directions or leaving it as a single area. The other regions shrink or expand as necessary in order to fill any gaps. Figure 57 shows the result of the user performing these operations on the top-left region.

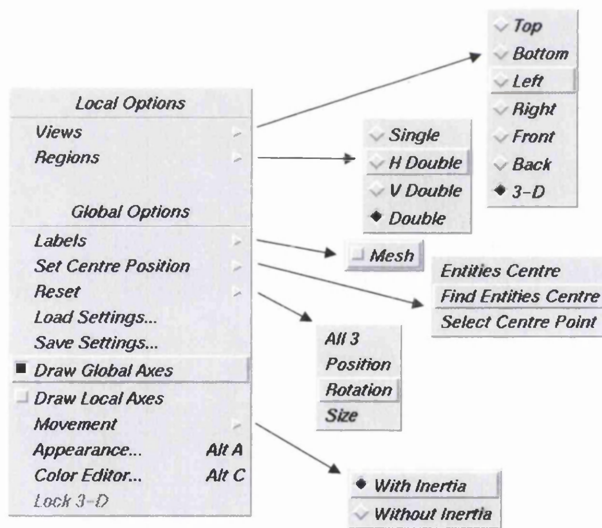


Figure 56 – The Pop-up menu hierarchy

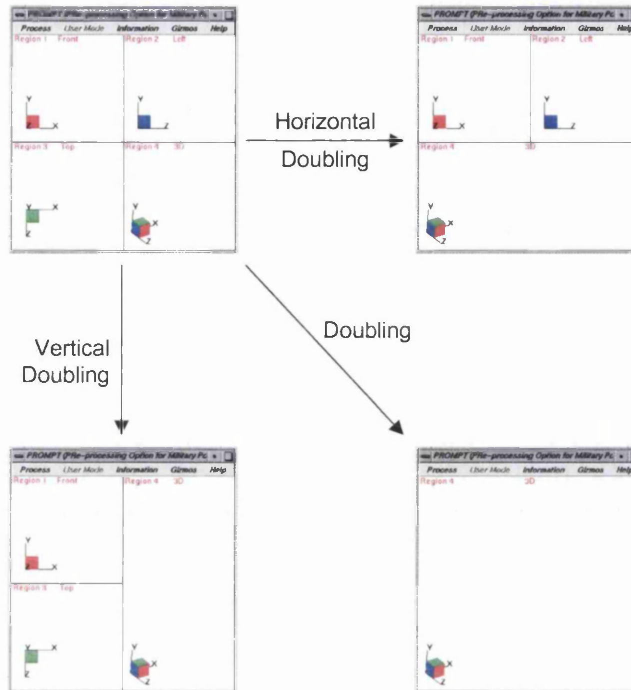


Figure 57 – An example of resizing regions

Set Position

This menu allows the user to set the centre of gravity of the objects currently being displayed. This is then used as the centre of any subsequent rotations. The first item, 'Find Entities Centre', recalculates the centre of gravity based on the portions of the objects currently visible. This is most often used when a clipping plane has been used to cut away a section of the mesh. By default, the centre of gravity is not changed and the object will continue to centre on the same point as before. Selecting this object re-centres the rotation. The second item, 'Centre on Selection' causes the centre of gravity to be placed at the centre of the feature that is currently selected (for example, a mesh plane or a mesh block). This allows the user to examine a portion of the mesh more closely without it disappearing from view when performing a rotation.

Reset

This menu allows the user to reset the position, size or the rotation of the objects on the screen back to their default positions. It should be noted that this does not reset the centre of gravity defined by the previous menu item. It merely sets $[x_t \quad y_t \quad z_t]$ to 0, S to 1 and R to the identity matrix.

Draw Global Axes

This toggles whether a set of axes showing the current x, y and z directions is displayed.

Draw Local Axes

This toggles whether a set of axes will be overlaid on top of each structured mesh block showing the current i, j and k directions.

Movement

This allows the user to switch *inertia* on or off. Inertia causes the object to keep performing any manipulation that was currently being performed when the middle mouse button was released. For example, if the user was rotating the object then the object will keep spinning in the same direction and at the same speed until the user stops it by briefly clicking the middle mouse button without moving the mouse. If the mouse was stationary when the middle mouse button was released then the object remains stationary. Turning this option off causes the object to become stationary when the user releases the middle mouse button regardless of the current motion of the mouse.

3.5.7. The Selection Gizmo Panel

The Selection Gizmo Panel allows the user to select which one of the possible features of the mesh can be selected by clicking the left mouse button in the visualisation window. The set of features, from which to choose, is decided by the *interests* of the module currently linked to the V&C Module. For example, for a structured mesh the Boundary Condition Definition (BCD) Module requires an *i*, *j*, or *k* mesh plane to be selected on which a boundary condition may be applied. Upon start-up, the BCD Module registers these interests with the V&C Module that causes the Selection Gizmo to disable all but the mesh plane selection buttons. Figure 58 and Figure 59 show the two typical appearances of the Selection Gizmo Panel.

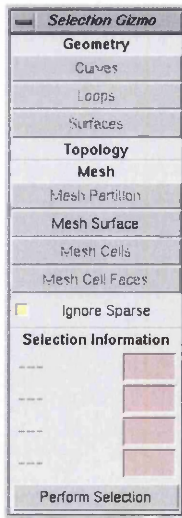


Figure 58 – The Selection Gizmo for Unstructured Meshes

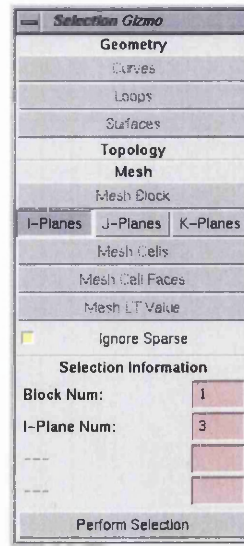


Figure 59 – The Selection Gizmo for Structured, Multi-Block Meshes

The bottom region of the Selection Gizmo Panel provides both a numerical verification of the feature that has been selected (e.g. mesh plane number), and a numerical means by which the user may select the chosen feature.

3.5.8. The Colour Editor Gizmo Panel

The Colour Edit Gizmo Panel provides a means by which the user can edit the colour of any of the objects displayed in the visualisation window. The panel is divided into five sections as shown in Figure 60.

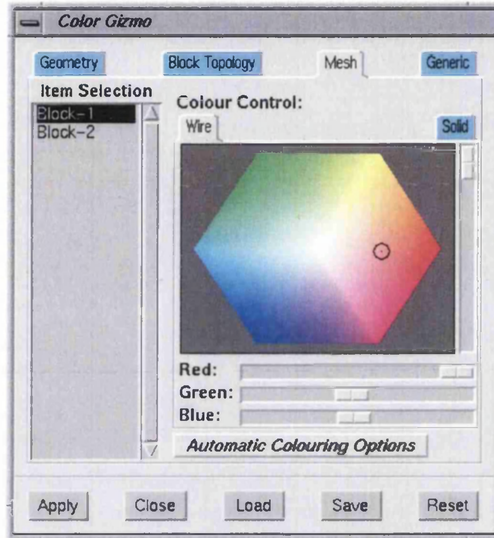


Figure 60 – The Colour Editor Gizmo Panel

Entity Type Selection

The first selection to make in this panel is the type of object that is to have its colour edited. Currently in PROMPT, there are two options; Mesh and Generic. The Mesh option allows the user to edit the colours of various parts of the mesh. The Generic option allows the user to edit the colour of the ancillary items such as any text labels, the background, the axes, etc.

Sub-Entity Type Selection

Under the Mesh option, the user needs to choose whether the colours should be altered for the edges drawn around each of the cells in the mesh (Wire) or the solid faces of the mesh (Solid).

Entity Selection

This list contains all of the items contained within the selected entity type. For a structured mesh, it is a numbered list of the mesh blocks; for an unstructured mesh, it is the surfaces contained within the mesh and for the Generic entity, it

contains the list of generic features (e.g. background, text colour, etc.). Selecting one or more of these items causes the Colour Editor section to be enabled.

Colour Editor

The Colour Editor section allows the user to alter the colour of the selected items using either the HSV (Hue, Saturation and Value) colour model or the RGB (Red, Green and Blue) colour model. These two colour spaces are illustrated in Figure 61 and Figure 62.

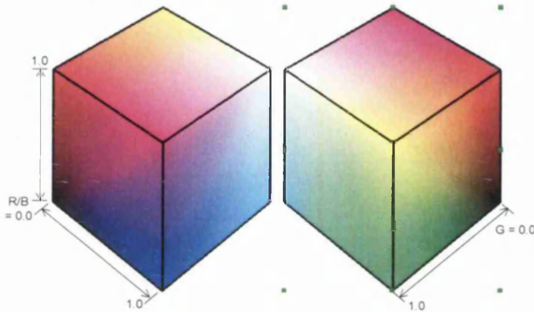


Figure 61 – The RGB Colour Space

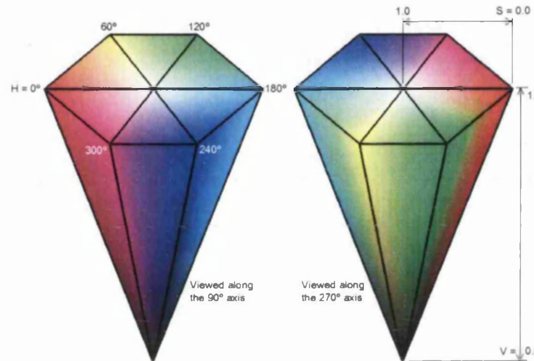


Figure 62 – The HSV Colour Space

The RGB colour space is the native colour space of any monitor but the HSV colour model is much more intuitive to the user since it mimics the way different coloured paints may be mixed to produce new colours.
To convert from the RGB colour space to the HSV colour space, the following equations are used [Yang92, Foley90]:

$$H = 60 * \begin{cases} (G-B)/\alpha & \text{if } R = \max(R,G,B) \\ 2+(B-R)/\alpha & \text{if } G = \max(R,G,B) \\ 4+(R-G)/\alpha & \text{if } B = \max(R,G,B) \end{cases}$$

where

$$\alpha = \max(R,G,B) - \min(R,G,B)$$

$$S = (\max(R,G,B) - \min(R,G,B))/\max(R,G,B)$$

$$V = \max(R,G,B)$$

To convert back again:

$$
\begin{aligned}
i &= \text{floor}(H/60) && \text{where floor}(i)\text{ returns the largest integer} \leq i \\
f &= H/i && f \text{ is the fractional part of } H \\
p &= V*(1-S) \\
q &= V*(1-(S*f)) \\
t &= V*(1-(S*(1-f)))
\end{aligned}
$$

$$
\text{if } i = \begin{cases}
0 & \text{then } (R,G,B) = (v,t,p) \\
1 & \text{then } (R,G,B) = (q,v,p) \\
2 & \text{then } (R,G,B) = (p,v,t) \\
3 & \text{then } (R,G,B) = (p,q,v) \\
4 & \text{then } (R,G,B) = (t,p,v) \\
5 & \text{then } (R,G,B) = (v,p,q)
\end{cases}
$$

Automatic Colouring

The last section of the panel provides some quick short-cuts to commonly used colour schemes such as giving each mesh block / surface a different colour. All of the options in this menu can be performed using the manual features of the panel but will just take much longer.

3.5.9. The Appearance Gizmo Panel

The Appearance Gizmo Panel is similar in layout to the Colour Editor Gizmo Panel and is used to allow the user to edit the non-colour-related attributes of the mesh. As shown in Figure 63, the panel is also divided into five sections.

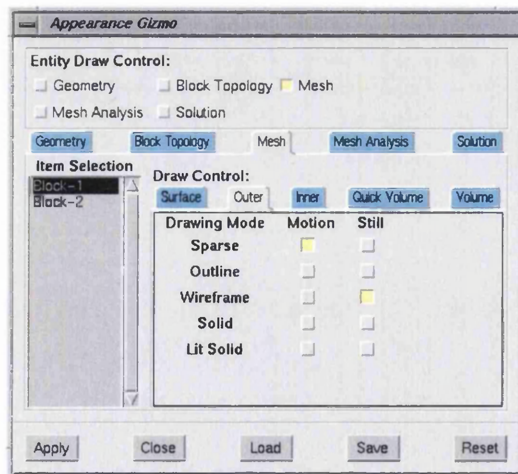


Figure 63 – The Appearance Gizmo Panel

Draw Control

If PROMPT currently contains a number of entities, such as mesh, mesh analysis data, etc., then drawing them all at once would make the display very cluttered.

To overcome this, the Draw Control section at the top of the panel allows the user to selectively turn on or off these individual entities.

Entity Type Selection

As with the Colour Editor Gizmo Panel, the user must choose the feature whose appearance is to be altered by selecting one of the tabs. This changes the options that are available in the next three sections.

Entity Render Mode

Under the 'Entity Type Selection' section, the features of the current entity that are to be drawn are selected. For a structured, multi-block mesh, the options are:

- 'Surface' – This only draws the faces of the mesh cells that actually are identified as being on geometrical surfaces. This data is contained as part of the mesh.
- 'Outer' – This draws the cell faces that appear on the outer faces of the mesh blocks, i.e. the face of a block with no adjacent block.
- 'Inner' – This causes the cell faces that appear on all six faces of each mesh block to be drawn regardless of whether there is an adjacent block or not.
- 'Quick Volume' – This effectively draws only the faces that would be seen if the every cell in the volume mesh had been drawn using hidden-line removal. This produces similar results to the previous options until clipping planes are introduced.
- 'Volume' – This actually draws every cell in the volume mesh. This option isn't often chosen since it reduces the rendering performance significantly and can cause a cluttered display for fine meshes.

Item Selection

This list has an identical purpose to the list in the Colour Editor Gizmo Panel. It used to select the items within the selected entity for which any changes in appearance will affect.

Item Drawing Mode

The last section of this panel allows the user to alter how the selected items will be drawn. The choice does depend on the entity type that is selected but usually includes:

- 'Sparse' – Vertices are drawn as single dots.
- 'Grid' – Any edges are drawn as lines.
- 'Solid' – Any faces are drawn as solid polygons.
- 'Solid Lit' – Drawn as 'solid' but lit from a single, user-configurable light source.

Most of these options can be combined, for example 'Solid Lit' can be selected along with 'Grid' to produce a solid lit entity with a grid of lines overlaid on top. Figure 64 shows some examples of a mesh drawn using a number of the appearance combinations.

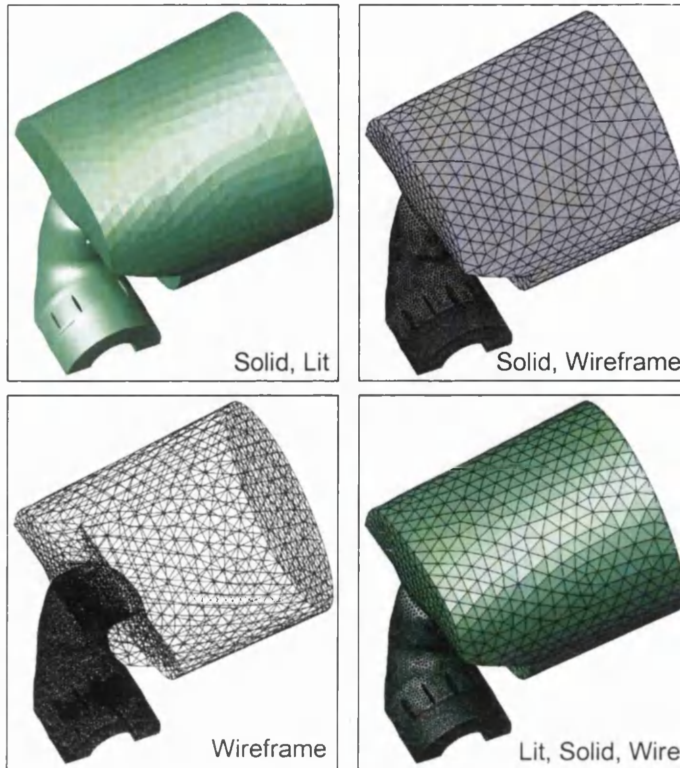


Figure 64 – Various Appearance Gizmo Panel settings for a mesh

For the convenience of the user PROMPT associates two drawing modes with each entity; one for when it is stationary and one for when it is in motion. This allows a lower detailed view of the mesh that provides a higher frame rate whilst it is being manipulated by the user and a higher detailed representation when it is stationary without the need to repeatedly open the Appearance Gizmo Panel to change the settings.

3.5.10. The Lighting / Material Gizmo Panel

As mentioned previously, PROMPT has the ability to render objects as thought they are lit from a single, white light source at an infinite distance[6]. The Lighting / Material Gizmo allows the user to fine tune the direction from which the light is coming and the reflective properties of the objects on the screen. Figure 65 shows the panel with the light source pointing into the screen from the top, left corner and with the objects material being quite metallic.

[6] This is representative of the light from the sun, which is far enough away that all of the light rays can be assumed parallel. This is in stark contrast to a local spotlight whose rays emanate from a central point.

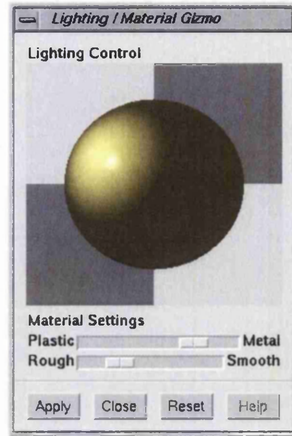


Figure 65 – The Lighting / Material Gizmo Panel

The direction the light is coming from is shown on the sphere at the top of the panel. Simply clicking within the region of the sphere and dragging can change this direction. PROMPT will stop any attempts to make the light source go behind the object since this would make all the objects on the screen appear very dark.

The two scroll bars at the bottom of the panel changes the degree to which the objects appear as plastic or polished metal, and whether the surface is rough or smooth in appearance.

3.5.11. The Clipping Plane Gizmo Panel

PROMPT has the facility to define planes that clip away portions of the mesh. This valuable tool allows the user to investigate the interior of a volume mesh. These clipping planes can be manipulated on the screen in real-time using the mouse in a similar fashion to manipulating the mesh itself, i.e. translation and rotation. The Clipping Gizmo Panel is sub-divided into two sections as shown in Figure 66.

Figure 66 – The Clipping Gizmo Panel

Clip Operations

This section deals with the creation and deletion of clipping planes. Clicking on the 'New' button causes a new clipping plane to be defined. This is drawn on the display as a white, translucent rectangle that performs a rough clipping of the

mesh. This is shown in Figure 67. Selecting the 'Remove' button will cause the currently selected clipping plane to be deleted and any portions of the mesh clipped by that plane will be restored. To fix the position of the current clipping plane the 'Fix' button should be selected. This will remove the white rectangle and perform the final clipping of the mesh. An example of this is shown in Figure 68.

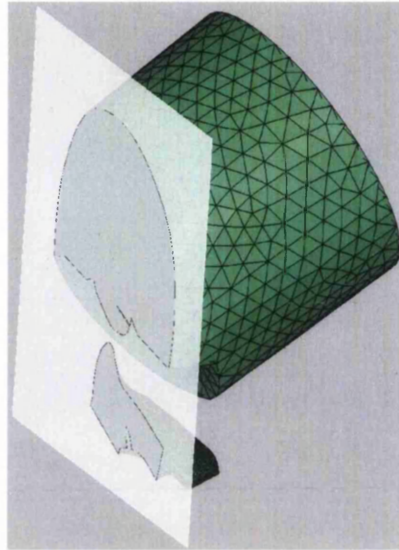


Figure 67 – Manipulating a Clipping Plane through a mesh

Figure 68 – The same mesh after it has been clipped

Manipulation

In order to position the current clipping plane the user must translate or rotate it using the mouse and the middle mouse button along with the *SHIFT, CTRL* and *ALT* keys in an identical manner to manipulating the mesh itself. As the plane is manipulated, the rough clipping of the mesh will update in real-time to give the user instant feedback. The 'Manipulation' buttons allow the user to, either manipulate the clipping plane with the mouse whilst keeping the mesh stationary, manipulate the mesh whilst keeping the clipping plane stationary or move both at the same time. The latter is most used to provide a different view of the scene without disturbing the relative positions of the mesh and the plane.

3.5.12. The Clipping Plane Algorithm

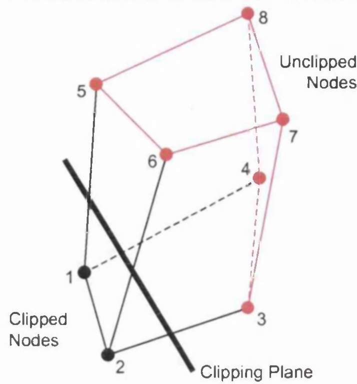
Regardless of whether the mesh originated as a structured, multi-block mesh or an unstructured mesh the clipping plane algorithm treats each element as an unstructured element. The algorithm is shown below:

```
for each element in the mesh do
    Count the number of nodes on the unclipped side of the clipping plane
    if the number of unclipped nodes = 0 then
        Do nothing {The cell does not intersect the plane}
    else if the number of unclipped nodes = the number of nodes in the element then
        Do nothing {The cell does not intersect the plane}
    else
        Set mask = ( 1 and ( node 1 clipped ) )
                 + ( 2 and ( node 2 clipped ) )
                 + ( 4 and ( node 3 clipped ) )
                 + ( 8 and ( node 4 clipped ) )
                 +  and so on for elements with more nodes
        Use the value of mask in a lookup table to determine the face numbers to draw
        Use a second lookup table to determine the node numbers for each face
        Add face primitive to the render list
    end if
end for
```

The first lookup table is a two dimensional array with its dimensions defined as two raised to the power of the number of nodes in the element and the maximum number of faces that can be drawn after nodes have been clipped. In the case of the hexahedron, the former is 256 and the latter is 3. The use of the first lookup table using the `mask` variable is illustrated in Figure 69.

This result of this algorithm is a complete surface of triangles and/or quadrilaterals that are formed from the closest set of element faces to the clipping plane. The same procedure is applied to the surface faces of the mesh in order to clip unwanted portions of the surface. The need to test the clipping plane against every element is alleviated in the same manner as for the feature selection algorithm (described previously) by grouping

the elements into mesh blocks, testing the blocks against the clipping plane and only testing the elements if the block intersects the plane.



<u>Algorithm Results</u>

- Nodes 1 and 2 are clipped
- mask = 3.
- Lookup table entry 3 contains:

{ 2, 6, -1 }

where 2 and 6 are the faces that should be drawn (shown in red) and –1 signifies a blank entry since only two faces are drawn for this combination of clipped nodes.

Figure 69 – Hexahedron intersecting a Clipping Plane

3.5.13. The Print Gizmo Panel

The Print Gizmo Panel (Figure 70) allows the user to easily create a hard copy snap-shot of the graphical display without the need for any third-party screen grabbing utilities.

Figure 70 – The Print Gizmo Panel

Using the panel the user can select between a bitmap image as a TIF (Tagged Image Format) file or an EPS (Encapsulated PostScript) file [Adobe90]. The user also has a number of options that allow a trade-off between quality and image size:

- The user can choose between a full colour image and a grey-scale image,
- The user can choose the number of colours used to produce the image. PROMPT provides four levels of colour quality, and
- The user can control the dimensions of the final image. The actual image produced is always the full visualisation window. The change in image dimensions represents a change in resolution in which the image is rendered.

Whilst the user is altering these settings, the dimensions of the image and the size (in Kb) of the final image file are constantly updated.

The four colour quality levels represent the image being rendered using a 24-bit (16,777,216 colours), a 16 bit (65,536 colours), a 12 bit (4,096 colours) or an 8-bit (256 colours) representation. Combining the red, green and blue components of the full-colour image into a single luminescent value for the grey-scale image generates the grey-scale

representation. This is performed using the NTSC standard [Yang92, Yang97] where the luminescence is defined by:

$$Y = 0.299R + 0.587G + 0.114B$$

The different weightings given to the three primary colours directly correspond to the human eyes differing sensitivity to same three colours.

Once the settings are satisfactory the 'Save' button is selected. This opens a standard File Selection box (Figure 71) that is used to determine the name and directory of the image file.

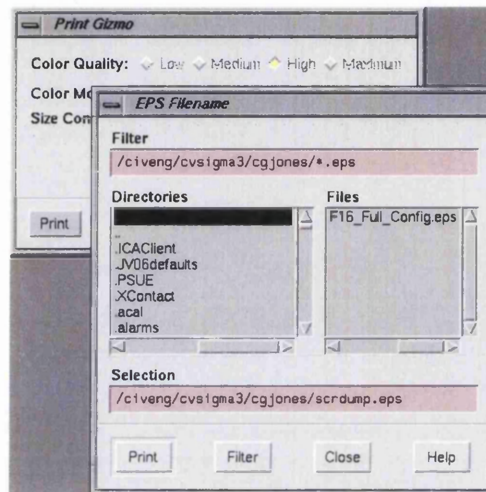


Figure 71 – A standard File Selection box used in the Print Gizmo Panel

3.6. The Task Database Module

The purpose of the Task Database is to store all of the data files related to sessions within PROMPT. The approach the Task Database takes is to take control over how these various data files are placed. This allows the Task Database to keep track of the dependencies between the different files. For example, if a solution is obtained on a particular mesh then that solution will be associated with that mesh and the user will not be allowed to mistakenly overlay it on a different mesh. This relieves the user from having to remember which data files go together and, therefore, significantly reduces the number of consistency checks that need to be made by PROMPT when loading data files.

3.6.1. The Task Database Window

When the Task Database is opened, the panel will appear like Figure 72. The operations performed using this panel fall into five main categories:

- Loading data files into PROMPT,
- Saving data files to the database,
- Deleting data files from the database,
- Attaching User Comments to data files currently residing in the database and

- Importing data files from outside sources into the database.

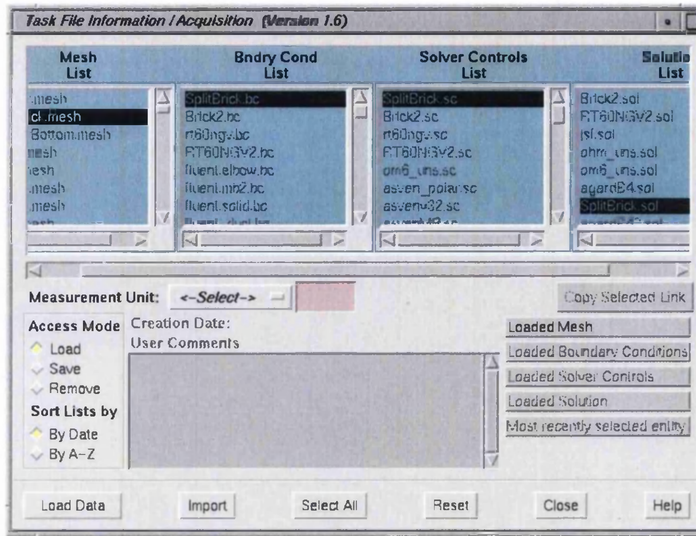


Figure 72 – The Task Database Window

3.6.2. Loading Data Files into PROMPT

As mentioned previously, the dependencies between data files are tracked automatically by the Task Database. This means that selecting the correct set of files to load together is very easy and error-free. To load a set of data files the user simply selects the respective entries in the 'Entity Lists'. As data files are selected, all other entries that are not compatible with the selected entries are ghosted, and thus cannot be selected. This has the effect that as the user selects data files from each of the columns the choice remaining decreases. This effect is shown in Figure 73 where the dependencies between data files are shown schematically. At first, all of the data files are selectable. If the user selects a mesh then all of the data files except for those related to the selected mesh are disabled (Figure 74). This process then continues when the user selects one of the remaining solution files (Figure 75).

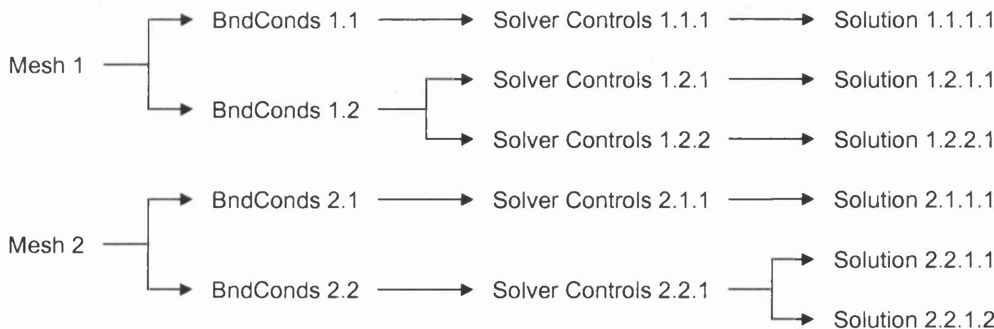


Figure 73 – A Schematic of Dependencies between Data Files

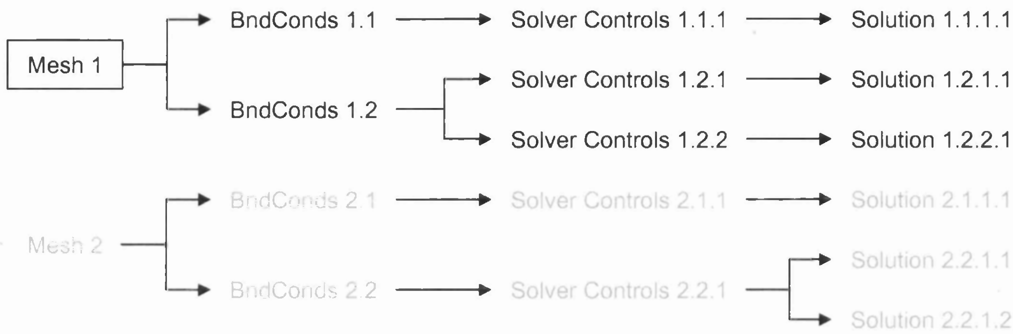


Figure 74 – The Dependencies after a mesh has been selected

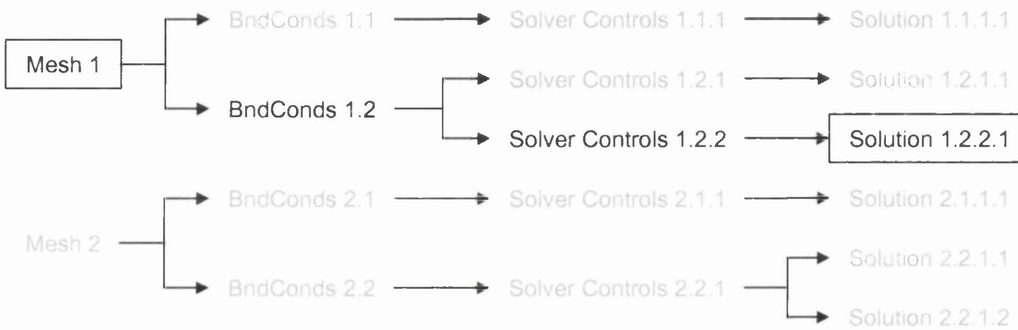


Figure 75 – The final set of Selectable Data Files

3.6.3. Saving Data Files to the Database

To save entities currently residing in PROMPT to the Task Database, the user must first select the correct *access mode* in the 'Access Box' then select the 'Save Data' button. This will cause a panel to open (Figure 76) into which the names of the data files are typed.

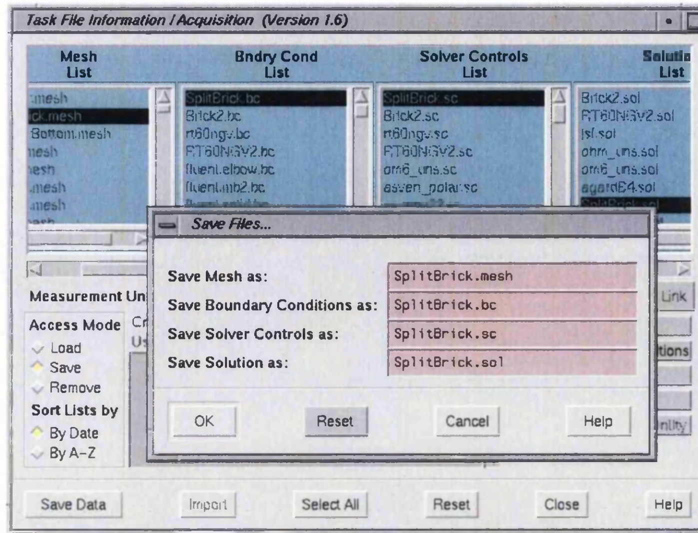


Figure 76 – The 'Save Data' Panel

At this stage, a number of checks are performed by the Task Database to ensure the consistency of the dependencies:

1. If none of the names currently reside in the Task Database then the data files are simply saved and their relationships are stored.
2. If the name of a data file does exist in the Task Database then that file is compared to the currently saved version. If they are identical and their dependencies are compatible then the remaining data files are saved and their relationships are merged.
3. If the name of a data file exists and that file is either different or their dependencies are not compatible then the user is requested to choose another name.

3.6.4. Deleting Data Files from the Database

Since the links between data files are controlled by the Task Database, it is not enough to just manually remove the files from the disk using a UNIX shell. In PROMPT, the only way to remove data files cleanly is to use the Task Database.

To perform this action, the user must first select the correct access mode using the 'Access Mode' box, and then select the data files in the 'Entity Lists' that are to be removed. Finally clicking on the 'Delete' button will remove the selected data files and modify the links appropriately. In order to maintain consistency in the Task Database, each data file that depends on the data files selected for removal are also removed. Since this could lead to more files than the user intended, a dialog box (Figure 77) is shown which confirms the users actions before actually performing them.

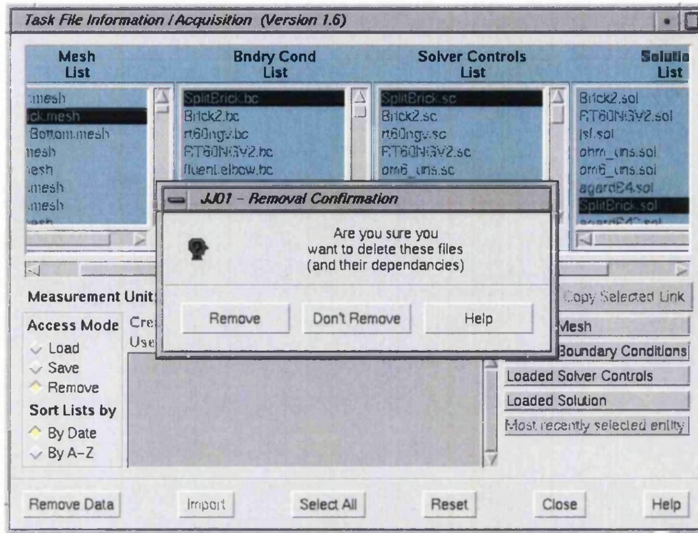


Figure 77 – Deleting Data Files from the Task Database

3.6.5. Attaching User Comments to Data Files in the Database

Often when a data file, such as a solution, is obtained it is preferable to be able to annotate it with a small message describing the purpose for which it was created. Traditionally, these might be encoded, rather cryptically, in the filename somehow or a separate text file might be created to contain the annotation. Both of these methods have disadvantages. The filename approach limits the amount of information that can be stored due to the limit on the number of characters imposed by the file system. Whereas creating a separate file that sits alongside the solution data file has the disadvantage that it could easily become out of sync or even lost over time.

In the Task database, an annotation may be stored along with any data file. When a data file is selected in one of the entity lists its annotation automatically appears in the 'User Comments' box from where it can be easily edited by the user.

The actual information is stored in a fixed length block of 2000 characters at the front of the data file. Although choosing a fixed size for this block does limit the length of the annotation, it does have a major advantage over a variable sized block. When the user selects a data file only the annotation section of the file is read, and if this is subsequently edited then the new fixed length block of characters is overlaid on top of the original. This removes the need to read in and write out the entire data file every time the user highlights it in a list to view or edit its annotation. This procedure is shown in Figure 78.

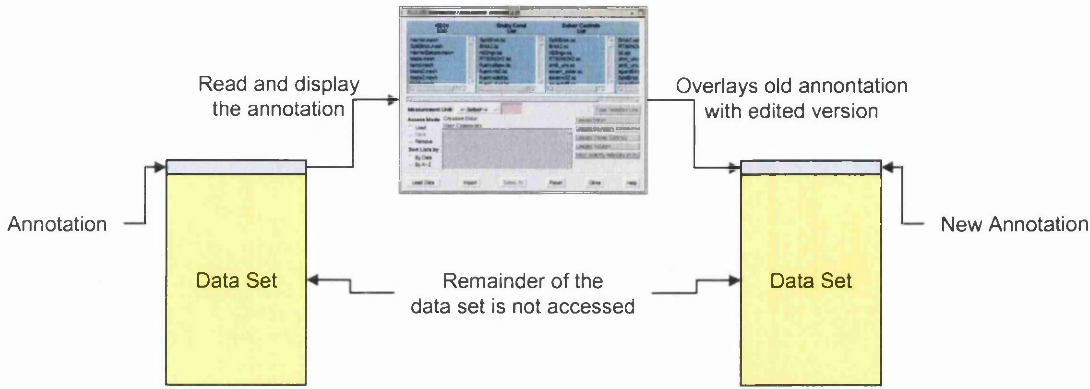


Figure 78 – Editing the Annotation of a Data Set

3.6.6. Importing External Files into the Database

As described previously, the Task Database controls the storage of any data files used within the PROMPT environment. As well as maintaining the dependencies between the data files, PROMPT also enforces the format in which these data files are stored. This is to ensure that the data files are syntactically correct and efficient to read and write. The PROMPT file formats also accommodate data sets that are commonly used by the modules within PROMPT but may be time consuming to compute. In order to be able to use data (in particular meshes and/or solutions) from outside the PROMPT environment, the Task Database provides a mechanism for translating the external files in a format foreign to PROMPT into the internal PROMPT format that can then be stored within the Task Database.

In order to activate this mechanism the 'Import' button needs to be selected. This opens a panel like the one shown in Figure 79.

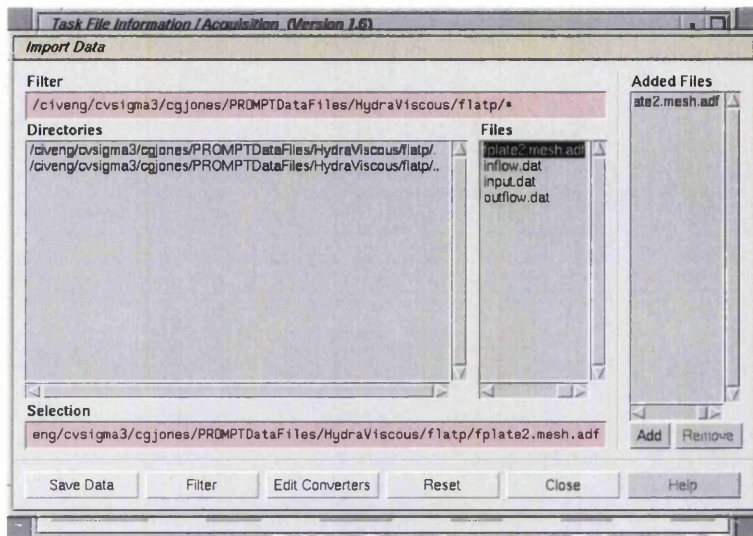


Figure 79 – The 'Import Data' Panel

The left-hand side of the panel looks and behaves like a standard UNIX file selection panel with the directory listing on the left and the file listing on the right. On the right-hand side there is a list that contains the files currently selected for importing. Double clicking on a file in the file selection section or clicking on a file and selecting the 'Add' button adds it to the currently selected list. Selecting an item in this list and clicking on the 'Remove' button removes it from the list. Selecting multiple files to import behaves in exactly the same manner as opening the Import Data panel multiple times and selecting one file each time.

To import the selected files the user simply clicks on the 'Save data' button. This cycles through the selected files, imports them into the Task Database and presents the user with a panel (Figure 80) into which the names may be entered.

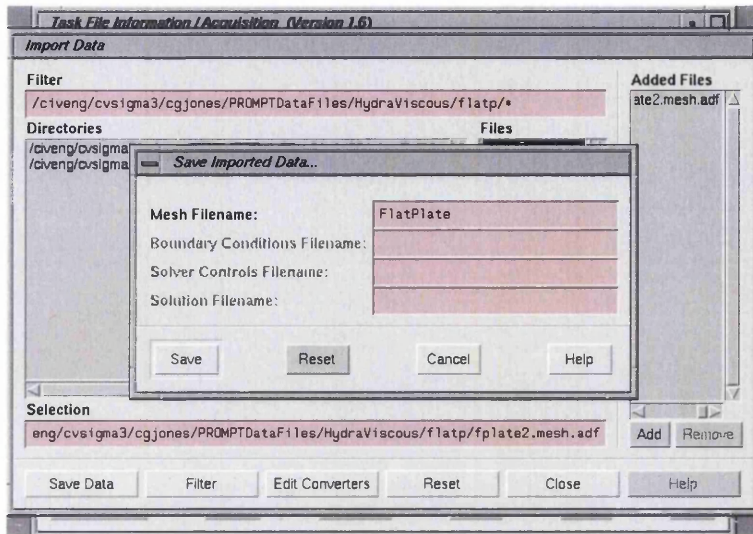


Figure 80 – Saving Imported Files in the PROMPT format

In order to choose the correct conversion algorithm PROMPT uses the extension of the imported file to identify it. The user can change this mapping between file extension and conversion algorithm by selecting the 'Edit Converters' button in the Import Data panel. This opens a panel as shown in Figure 81. The majority of the panel is used to display the current file extension mappings. New mappings may be added by entering the file extension (with or without the '.') and then selecting one of the available converters. Mappings can be edited and removed in a similar fashion. In order to remove any possible ambiguities, multiple extensions may be mapped to one converter but only one converter is allowed to be mapped to a given file extension.

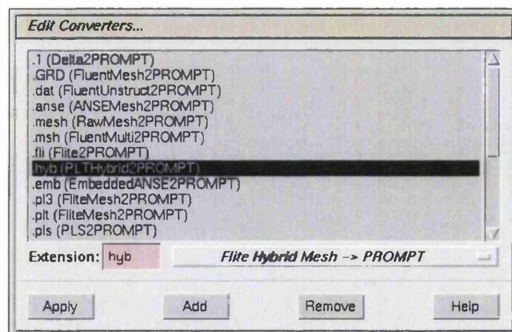


Figure 81 – Editing the List of Available File Converters

3.6.7. The Task Database Importing Mechanism

During the development of PROMPT, it became apparent that data files originating from a wide, and rapidly increasing, range of sources might need to be imported into the PROMPT environment. To satisfy this requirement there were two possibilities that were considered:

- A conversion algorithm could be developed for each foreign file format and included within the actual Task Database, or
- A conversion algorithm could be developed for each foreign file format and compiled into a separate executable, which would take, as input, the filename for the file to be imported and piped out the data sets in the PROMPT format.

The first method would be simple but it would require that the Task Database was modified and redistributed every time a new file format was added. With a large number of file formats, this could also make the Task Database very large. The second method would require a more sophisticated architecture but would reduce the size of the Task Database to a minimum. When a new file format was added then a new stand-alone conversion module could simply be developed and distributed. A schematic of the method used in PROMPT is shown in Figure 82.

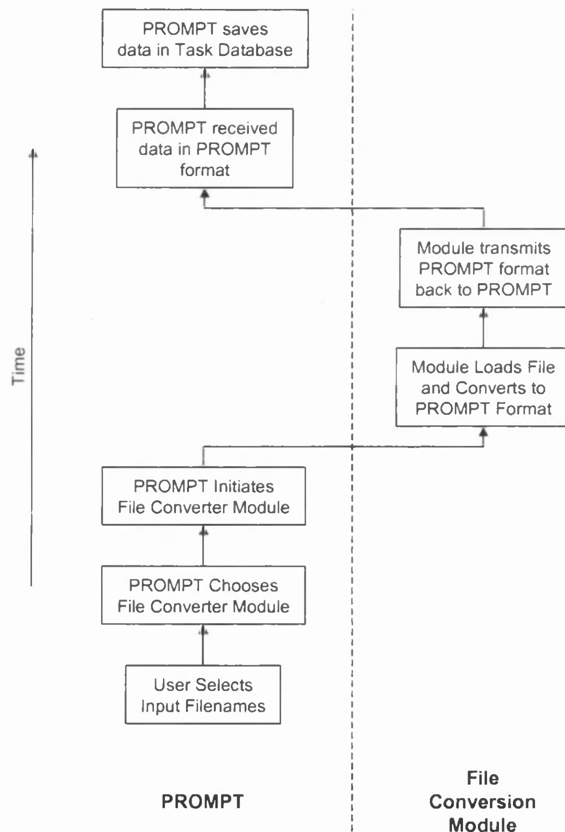


Figure 82 – A Schematic of the Importing Process

The process starts at the bottom-left with the filenames selected by the user. These are then mapped onto the appropriate file conversion module by the Task Database. The module is then initiated and passed the filename as a command-line argument. The *stdout* port of the conversion module is connected to the Task Database via the standard UNIX pipe mechanism. The Task Database then waits for the data in the appropriate PROMPT

file format to flow along the pipe before saving them in the database. Meanwhile, the conversion module reads the supplied file, converts it to the appropriate PROMPT file format and then uses the standard PROMPT I/O routines to write the file to the pipe in an identical manner to writing it to a file.
The current list of file conversion modules, and their associated file extensions, is shown in the table below. The algorithms used to perform the file conversion are detailed in Appendix A.

Mesh Type	Extension	Description
CFDS ANSE Mesh	.1	Single-block CFDS-ANSE mesh[7]
Fluent Mesh	.GRD	Single-block structured mesh from Fluent[8]
Fluent Mesh	.msh	Multi-block structured mesh from Fluent
FLITE Mesh	.pl3	Unstructured tetrahedral mesh from the FLITE system[9]
CFDS ANSE Embedded	.emb	Single-block CFDS-ANSE mesh using hanging nodes
SAUNA Multi-Block	.xyz / .3d	Structured multi-block SAUNA mesh[10]
CINDY Unstructured	.oxd	Unstructured tetrahedral grids in the Oxford CINDY format[11]
Fluent Unstructured	.cas	Unstructured grids from the Fluent system
Fluent Unstructured Solution	.dat	Solution files associated with Fluent unstructured meshes
CINDY Unstructured Solution	.oxs	Solution files obtained using the Oxford CINDY solver
HYDRA Unstructured	.adf	Unstructured meshes in the Oxford HYDRA format[12]
HYDRA Unstructured Solution	.adf	Solution files obtained using the Oxford HYDRA solver
DELTA Structured	.geom / .1	Structured multi-block mesh files in the Loughborough DELTA format[13]
DELTA Structured Solution	.geom / .1	Solution files obtained from the Loughborough DELTA solver.

[7] CFDS-ANSE is a proprietary solver from Rolls-Royce
[8] Fluent is a company that develops software for the pre-processing and solution of many types of finite element problems.
[9] FLITE is a proprietary suite of tools including an unstructured mesh generation capability and various CFD solution algorithms.
[10] SAUNA is a proprietary suite of tools developed by ARA and DERA for the pre-processing and solution of structured multi-block and unstructured CFD problems.
[11] CINDY is an unstructured CFD solver developed by Oxford University Computing Labs.
[12] HYDRA is the sequel to the CINDY solver developed by Oxford University Computing Labs.
[13] DELTA is a structured multi-block CFD solver developed by Loughborough University.

3.7. Mesh Analysis

Once a mesh has been imported into PROMPT, it may be necessary to verify that the mesh is of a suitable *quality* for the intended solver; and if not, then highlight to the user the areas that fall short of the required standard.

3.7.1. The Mesh Analysis Window

When the Mesh Analysis (MA) Module is first started, it shows the mesh analysis features of the module (Figure 83). The left of the panel contains a list of the blocks comprising the mesh from which the user may select one or more on which to perform the mesh analysis. The right-hand portion of the panel contains a histogram of the currently selected mesh quality measure. The current mesh quality measure may be selected from the pull-down menu above the histogram.

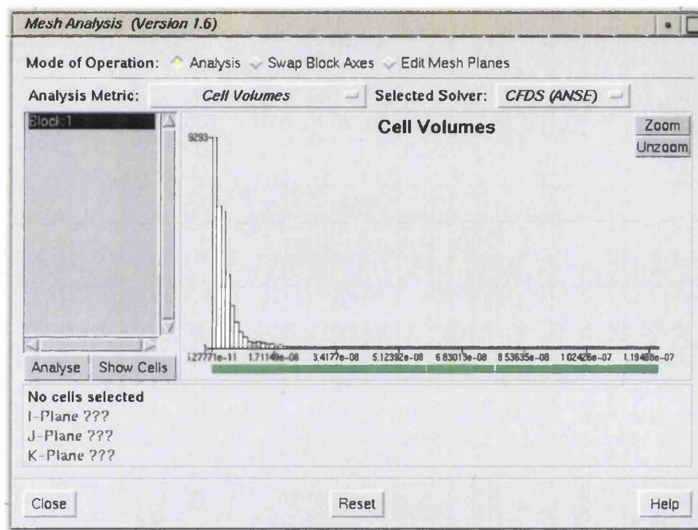


Figure 83 – The Mesh Analysis Panel

3.7.2. Performing a Mesh Analysis

In order to perform a quality analysis of a mesh, a number of steps need to be performed.

Choosing the Mesh Quality Metric

The first step is to choose the quality measure by using the pull-down menu above the histogram. The current choice of measures includes:

- Ratio of Adjacent Element Volumes
- Element Skewness
- Element Aspect Ratio

The quality metrics [Fol91, Belytschko84, Haimes93] were chosen to highlight mesh elements that:

Deviated from the ideally shaped element
Were neighbouring elements of a significantly different size and
Could be calculated for each of the element types.

Choosing the Mesh Blocks

For a structured, multi-block mesh, the next choice to make is on which block the analysis should be performed. This is done by selecting the required blocks in the list on the left of the panel. For a structured, single-block mesh or an unstructured mesh this option is disabled since there is no choice to be made.

Performing the Analysis

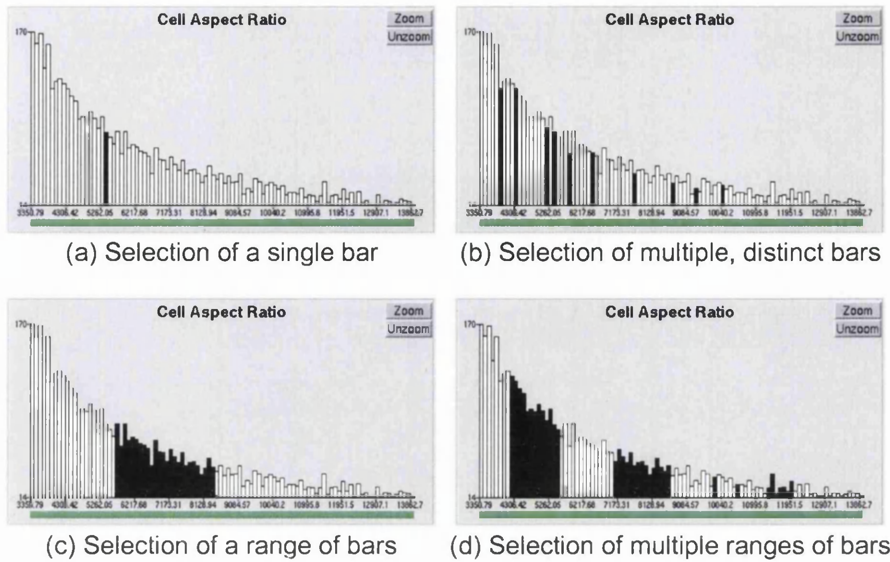
The next step is to perform the analysis of the mesh. This is achieved by selecting the 'Analyse' button. After a small delay, while the calculations are being performed, the results appear in the histogram on the right of the panel. The results are displayed as a set of 50 bars, each representing a $1/50^{th}$ of the total range of values. The height of each bar indicates the number of elements that fall into each range.

Although the histogram gives a graphical representation of the values of the chosen quality metric, this is often not enough for the inexperienced user since he/she may have no experience of which range of values is *good* for a particular solver and which are *bad*. In the MA Module, a horizontal green and red bar always accompanies the histogram. This indicates to the user which ranges of values are valid for the particular solver. These ranges are solver specific and can be changed by an experienced user by editing a simple ASCII file.

Selecting Ranges on the Histogram

As well as just displaying the mesh analysis results, the histogram also allows ranges of values to be selected on which further operations may be performed. There are four methods of selecting histogram bars:

- Single Selection – To select a single bar in the histogram the user just simply clicks on it. This will cause any previously selected bars to be de-selected. This is shown in Figure 84(a).
- Multiple Selection – Holding the *SHIFT* key whilst clicking on a bar will cause the bar to be added to the list of currently selected bars, i.e. previously selected bars will remain selected. This is shown in Figure 84(b).
- Drag Selection – Pressing and holding the left mouse button while dragging the pointer over a range of bars will select the entire range and any bars outside this range will be de-selected. This is shown in Figure 84(c).
- Multiple Drag Selection – Performing Drag Selection whilst holding the *SHIFT* key will add the newly selected range to the list of currently selected bars. This is shown in Figure 84(d).



(a) Selection of a single bar

(b) Selection of multiple, distinct bars

(c) Selection of a range of bars

(d) Selection of multiple ranges of bars

Figure 84 – The Four Histogram Selection Methods

Zooming In and Out of the Histogram

When a mesh analysis is performed, the initial histogram covers the entire range of metric values. For a large mesh, each histogram bar will represent a large number of elements, perhaps 1000's or 10,000's. When this occurs it may be preferred to narrow the range of values being displayed to view them in more detail. To *zoom in* to the histogram the user first needs to select the range of bars of interest as described in the previous section. The leftmost and rightmost selected bar will determine the new value range. Pressing the 'Zoom' button will, after a short delay, redraw the histogram with the 50 bars now representing the selected range of values. This procedure may be repeated up to ten times thus allowing the user to focus in on a small number of elements regardless of the size of the overall mesh.

To undo the previous zoom operation, the 'Unzoom' button may be selected. This may be repeated until the histogram returns to displaying the full range of analysis values. A typical sequence of zooming into the histogram is shown in Figure 85.

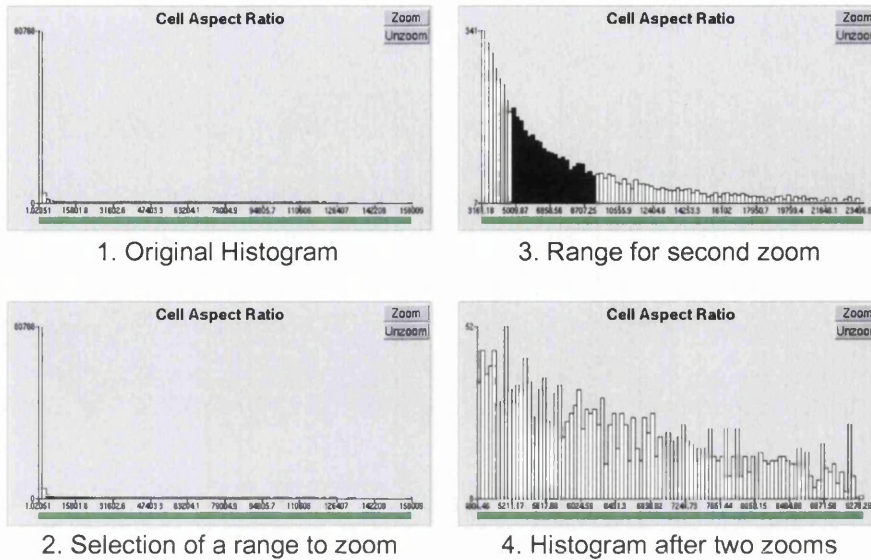


Figure 85 – Zooming into the Histogram

Highlighting *Bad* Cells in the Mesh

After the histogram has been manipulated so that the bars in the area of interest represent a reasonably small number of elements, it might be preferable to view the positions of the elements with respect to the rest of the mesh. For example, a bad element might be acceptable if it occurs in a region where the solution is constant but would be unacceptable if it occurred in a region of particular interest.

In order to highlight the appropriate cells, they need to be selected in the histogram as described previously. Selecting the 'Show Cells' button will then cause the elements whose quality metrics fall within the selected range to be highlighted in the mesh display in the Visualisation Window. As can be seen in Figure 86, the selected elements are colour coded according to their quality metric and a colour scale is overlaid at the bottom of the window for quantitative purposes [Haimes93].

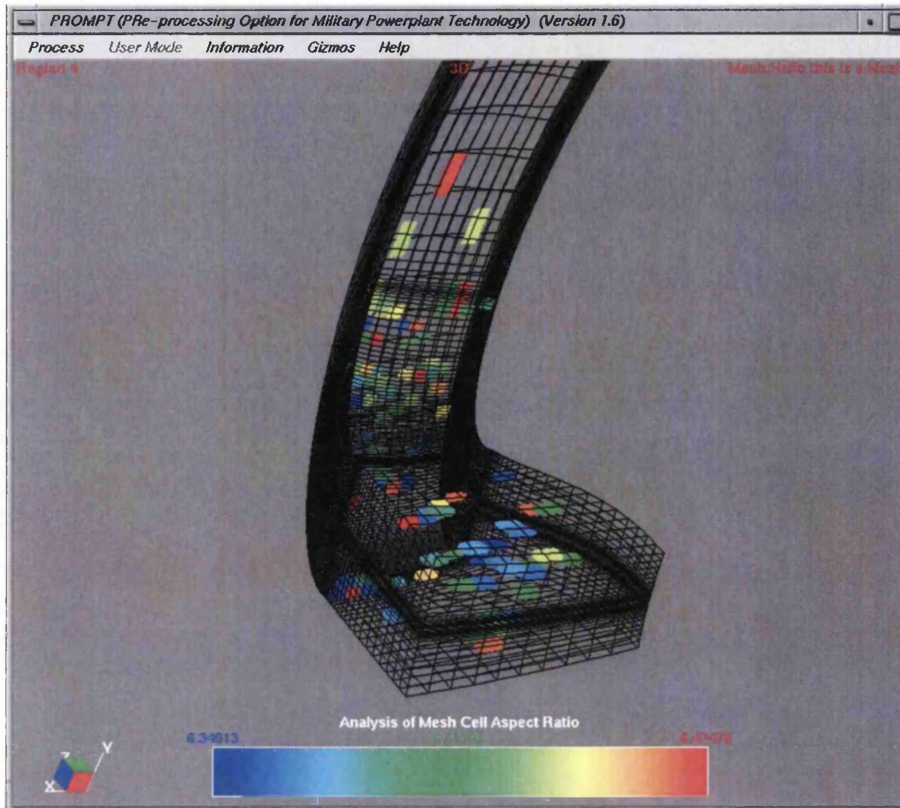


Figure 86 – *Bad* Elements highlighted in the Visualisation Window

3.7.3. Fixing Areas of Poor Quality

For structured, multi-block meshes a number of facilities are available within the PROMPT system to post-process the mesh in order to attempt to remove as many of the poorly shaped elements as possible.

Mesh Plane Movement

Figure 87 shows an example of moving a mesh plane. The panel shows a selected block in the mesh and allows the user to interactively select and drag the mesh plane along the appropriate axis. The nodes of the plane being moved are defined as a simple weighted linear interpolation of the planes either side. Any changes made to the position of a plane are propagated through all adjoining mesh blocks automatically.

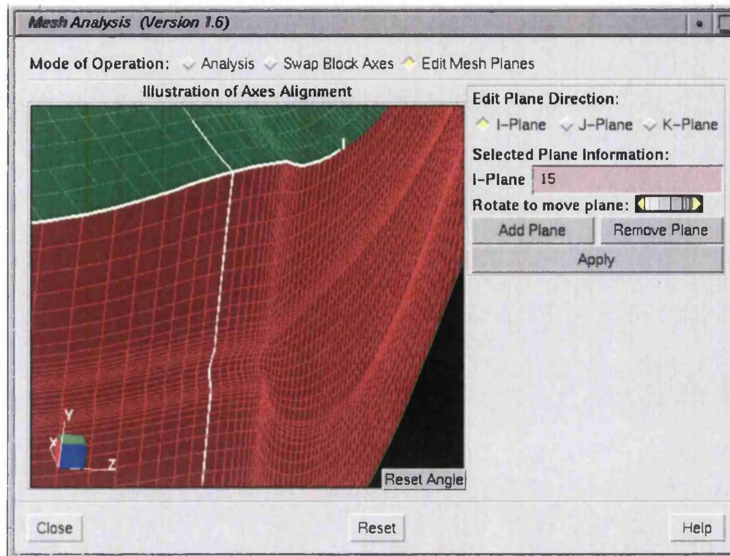


Figure 87 – Editing a Mesh Plane

Adding / Removing Mesh Planes

If moving a mesh plane is not sufficient to repair poorly shaped elements then the same panel can be used to add or remove mesh planes. To add a mesh plane, the user simply selects the mesh plane to the left of the plane to be added and then selects the 'Add' button. A new plane is then created as a linear interpolation of the two planes either side of it. Removal of a plane is performed in a similar manner by selecting the plane and clicking on the 'Remove' button. After a plane has been added or removed then the remaining planes may be moved interactively as described above.

If all of the above methods fail then the mesh must be regenerated in order to try to produce a mesh of a higher quality.

3.8. The Boundary Condition Specification Panel

Once the mesh has satisfied any quality criteria, it is necessary to define the boundary conditions for the relevant solver. These instruct the solver which criterion should be applied to a given section of the mesh boundary. For example, if a mesh represents the exterior of an aircraft then the majority of the boundary would represent a solid wall, with small sections representing the inlets and outlets of the engines. The solid wall criterion tells the solver that no air is allowed to pass through that section, whilst the engine inlet section prescribes the amount of air that will pass into the engine.

In PROMPT, the means by which the boundary conditions are applied depends on the intended solver. For a structured, multi-block mesh this ranges from being able to apply one boundary condition to each external block face (i.e. a block face not adjoining another block) through to being able to apply a boundary condition to any section of any

plane within the mesh. For unstructured meshes, the process is restricted to applying a single boundary condition to each group of boundary faces[14].

3.8.1. The Boundary Condition Window

Due to the different methods of placing the boundary conditions, the Boundary Condition Window has a different appearance depending on whether a structured or unstructured mesh currently resides in PROMPT. Figure 88 shows a typical example of a structured mesh boundary condition and Figure 89 shows an equivalent for an unstructured mesh.

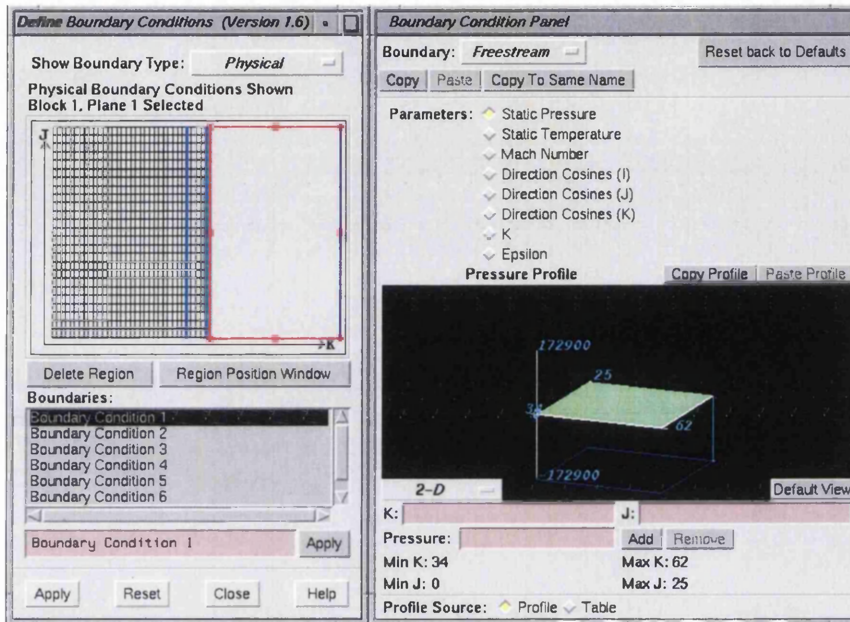


Figure 88 – The Boundary Condition window for a Structured Mesh

[14] The grouping of faces is performed outside of the PROMPT environment and is usually defined by the mesh generator as the geometric surface patch on which the boundary face lies.

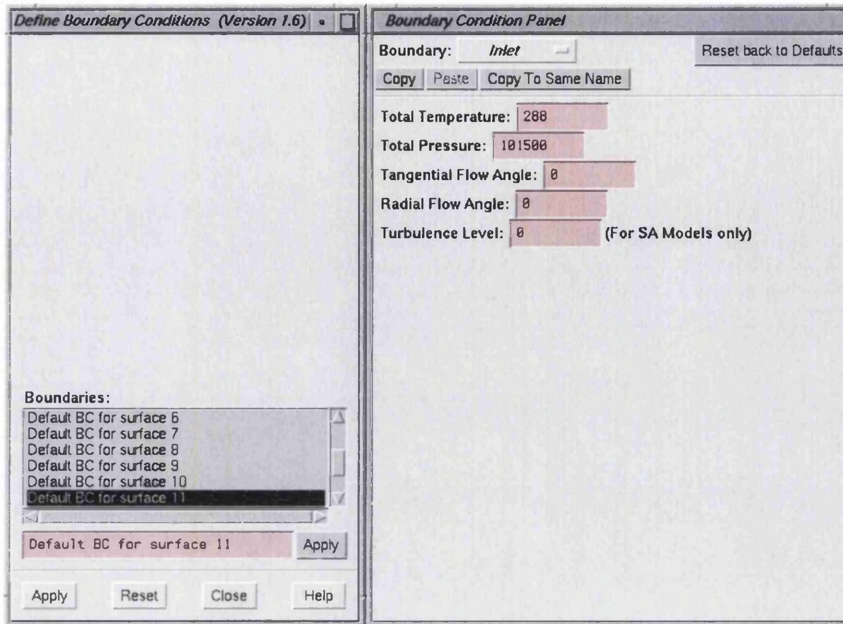


Figure 89 – The Boundary Condition window for an Unstructured Mesh

The next two sections describe how these panels are used to define boundary conditions for structured and unstructured meshes.

3.8.2. Boundary Condition Definition for Structured Meshes

In order to define a boundary condition on a structured, multi-block mesh a number of steps have to be performed:

Selecting a Mesh Plane

The selection of mesh planes is performed in the Visualisation Window and is described more clearly in Section 3.5.4. To summarise, the mesh plane direction (i.e. i, j or k) must be selected via the Selection Gizmo. The appropriate mesh plane may then be selected by simply clicking on it in the Visualisation Window. The chosen plane is highlighted in the mesh and a planar, Cartesian representation of it is shown in the top-left corner of the panel (Figure 90). If the solver is restricted to defining boundary conditions on exterior block faces then any selection in the Visualisation Window will also be restricted.

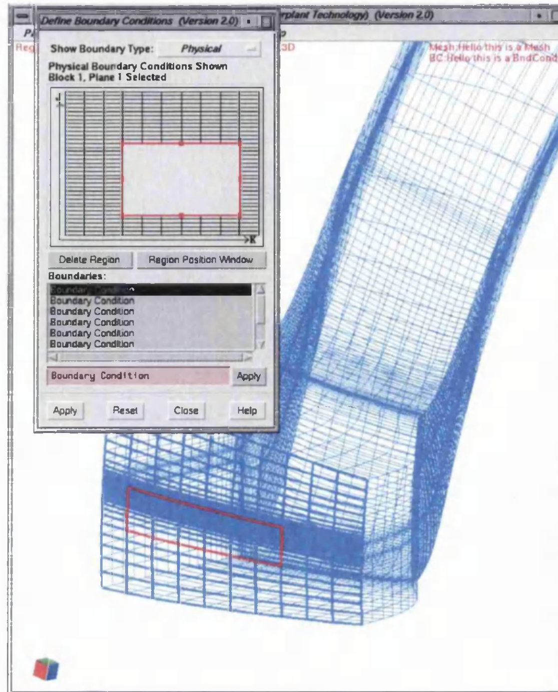


Figure 90 – Defining a Region on a Mesh Plane for a Boundary Condition

Selecting a Region on the Plane

Once a plane has been selected, the next step is to define a rectangular region on the Cartesian representation of the plane in which the boundary condition will be applied. The operations that can be performed on this Cartesian representation can be likened to a simple desktop window manager. Regions may be created (like windows) and they may be moved, resized and deleted.

Simply clicking where one corner of the region should be and then dragging to define the position of the opposite corner of the region creates a new region. A region may be resized by clicking on and dragging any corner or side, and moved by clicking and dragging anywhere in the interior of the region. A region may be selected by simply clicking on it; this allows the associated boundary condition to be edited.

During this process the defined region is also updated on the three-dimensional model in real-time. This gives the user the best possible feedback as to where any boundary conditions are defined. This is shown in Figure 90.

Selecting the Boundary Condition

Once a region has been selected, it is possible to edit the boundary condition associated with it. The boundary condition type may be selected using the pull-down menu at the top-right of the panel. This alters the appearance of the rest of

the right-hand side of the panel in order to reflect the parameters that need to be defined.

In PROMPT, a boundary condition is either physical or topological. A physical boundary condition *represents real-world* features of the mesh such as solid wall, inlet, outlet, etc. In contrast, a topological boundary condition exists purely to reduce the size of problem that is to be solved. For example, a symmetry boundary condition is a topological boundary condition that is used when the solution to a problem would have reflective symmetry. Solving half the problem and introducing the symmetry boundary condition along the plane of symmetry halves the amount of computation needed but still gives the same result. For the CFDS solver, there are four physical boundary conditions (Inlet, Outlet, Wall and Free Stream) and three topological boundary conditions (Repeat, Symmetry and Centre Line). The panels for each of these boundary conditions are shown Figure 91 – Figure 97.

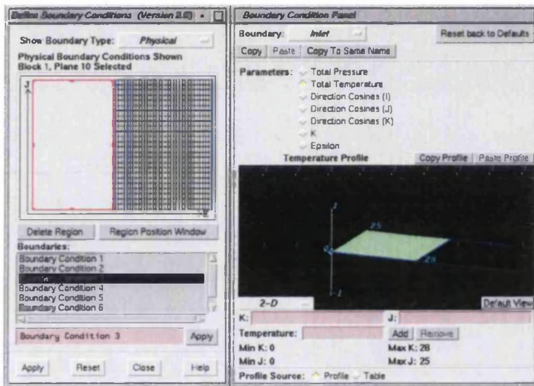


Figure 91 – The CFDS Inlet Boundary Condition Panel

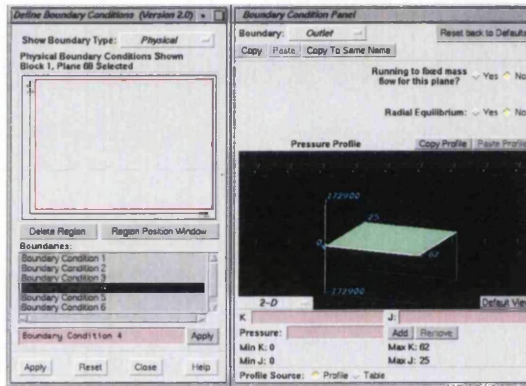


Figure 92 – The CFDS Outlet Boundary Condition Panel

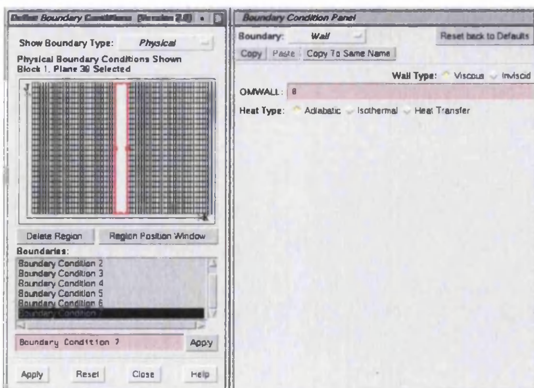


Figure 93 – The CFDS Wall Boundary Condition Panel

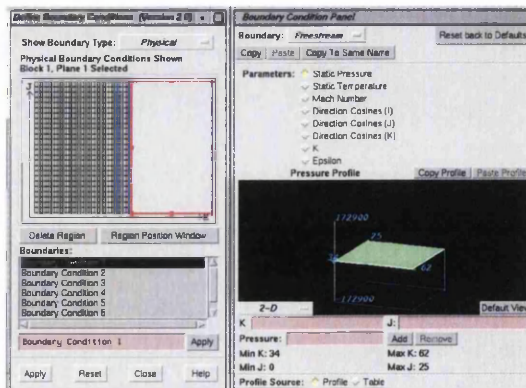


Figure 94 – The CFDS Free Stream Boundary Condition Panel

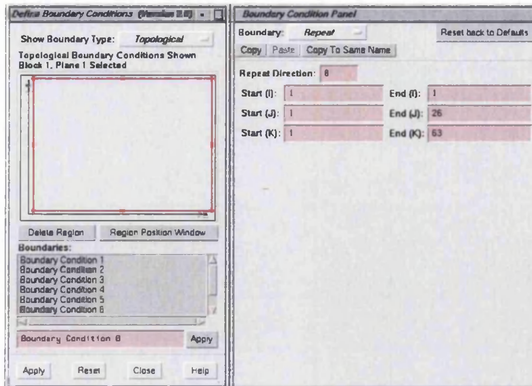


Figure 95 – The CFDS Repeat Boundary Condition Panel

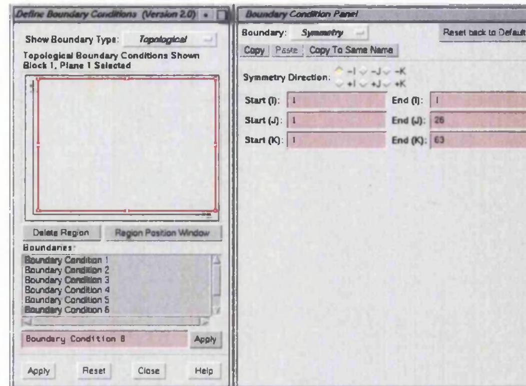


Figure 96 – The CFDS Symmetry Boundary Condition Panel

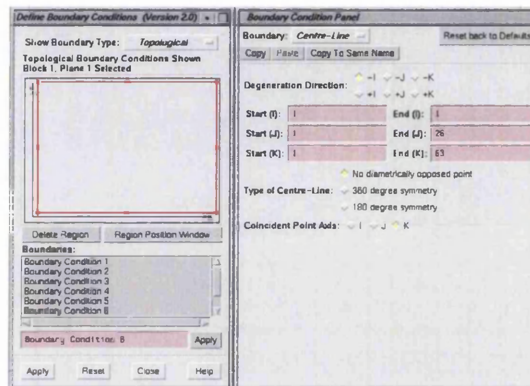


Figure 97 – The CFDS Centre Line Boundary Condition Panel

Editing Profiles of Values

Instead of just entering single values for boundary condition parameters that are then applied to the whole region, many of the parameters of the physical boundary conditions have the added capability to define profiles of values. For this purpose, PROMPT includes a Profile Graph Editor. The Profile Graph Editor allows the user to define a 1D or 2D grid of points each with an attached value. These points do not have to coincide with mesh points but can be anywhere within the region. Once a profile has been defined, the values in the profile are interpolated onto each of the mesh nodes.

Points are created and moved by interactively clicking and dragging them and simply selecting a point and entering a new value can alter its height.
Figure 98 shows an example of a 1D profile of values for a typical region and Figure 99 shows a typical example of a 2D profile.

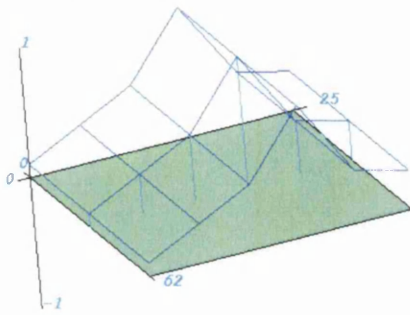


Figure 98 – An example of a 1D parameter profile for a Boundary Condition

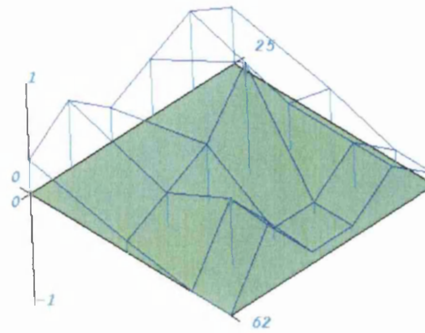


Figure 99 – An example of a 2D parameter profile for a Boundary Condition

3.8.3. Boundary Condition Definition for Unstructured Meshes

Due to the restrictions of where boundary conditions can be applied on an unstructured mesh, the number of steps is reduced to:

Selecting a Mesh Surface

The selection of unstructured mesh surfaces is very similar to the selection of structured mesh planes. The Selection Gizmo Panel in the Visualisation and Control Module is used first to enable mesh surface selection and then surfaces are selected simply by clicking on them in the Visualisation Window. The selected mesh surface is highlighted in the Visualisation Window and the boundary condition attached to it is displayed in the right-hand portion of the Boundary Condition Panel.

Selecting the Boundary Condition

Unlike structured meshes, the restriction that boundary conditions are applied to whole surfaces means that there is no equivalent to the creation of sub-regions. Instead, the process jumps straight to the selection of the Boundary Condition type. This is performed in the same manner as for structured meshes except that no parameters require profiles of values to be entered.

3.9. The Solver Execution Panel

The last step before the solver can be executed is the definition of the overall flow and run-time parameters of the solver. Like the Boundary Condition definition stage, these parameters vary considerably between different solvers. For the purposes of organisation, the Solver Execution (SE) Panel splits the various parameters into four types:

- General Flow Parameters,
- Runtime Control Parameters,

- Turbulence Model Control Parameters (only applicable if a turbulence model is selected) and
- The Set-up of the Initial Guess.

3.9.1. The Solver Execution Panel Appearance

The 'Flow Parameters' Panel

When first opened the SE Panel appears showing the various options categorised as Flow Parameters. Figure 100 shows a typical example for the ANSE-CFDS structured solver. The options in this panel are generally regarded as defining the overall physical constants that are independent of the numerical scheme used in the solver. Examples of these are the specific heat capacity of the fluid, the universal gas constant, etc.

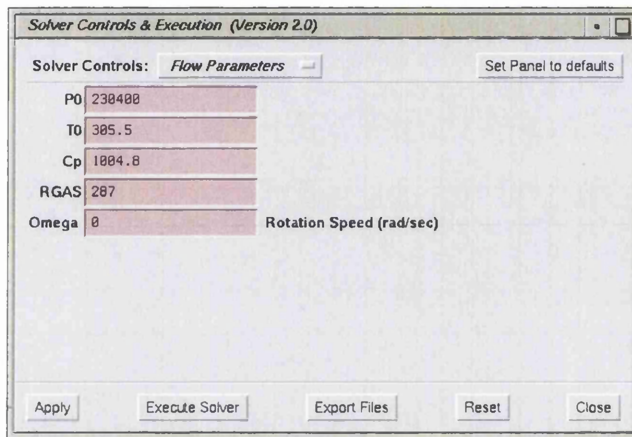


Figure 100 – The 'Flow Parameters' Panel for the CFDS solver

The 'Runtime Control' Panel

The Runtime Control Panel contains the options that control the numerical operation of the particular solver. Examples of these options are:

- Number of iterations,
- Relaxation Coefficients (such as CFL) and
- Multi-grid control options.

Figure 101 shows the Runtime Control Panel for the CFDS solver.

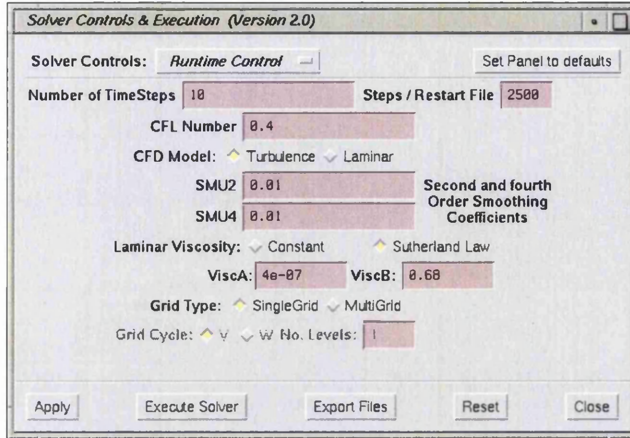


Figure 101 – The 'Runtime Control' Panel for the CFDS solver

The 'Turbulence Model' Panels

The Turbulence Model panels allow the user to control which turbulence model is used by the solver. Having chosen a turbulence model, the various parameters that fine-tune this model can then be altered. For the CFDS solver, there are three turbulence models available:

- Mixing-Length turbulence model,
- k-ε turbulence model and
- k-*l* turbulence model.

The CFDS panels for these options are shown in Figure 102, Figure 103 and Figure 104.

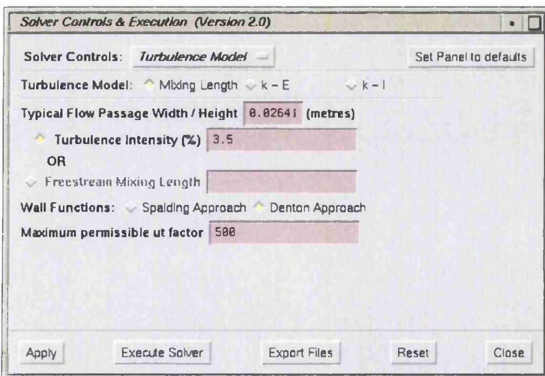


Figure 102 – The Mixing-Length Turbulence Model Panel

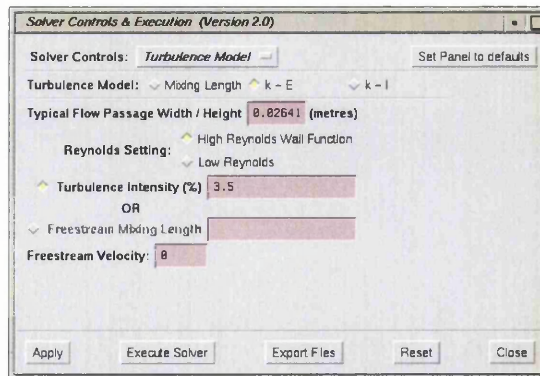


Figure 103 – The k-ε Turbulence Model Panel

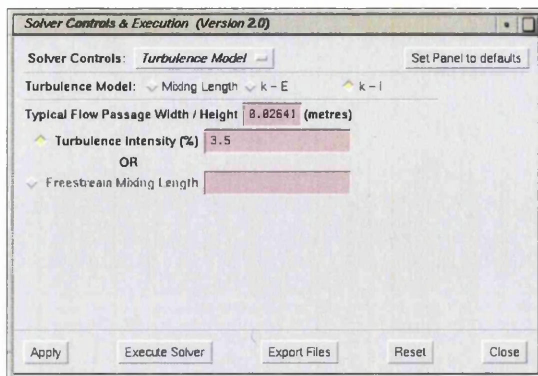


Figure 104 – The k-*l* Turbulence Model Panel

The 'Initial Guess' Panel

In order to converge, a number of solvers need to start from an initial solution that is not too far from the final solution. Often this can simply mean that the whole domain is filled with the free-stream values. For some, more complicated flows, constant values throughout the domain are not sufficient and profiles of values are necessary. For a solver that only needs constant values, the panel may look something like Figure 105, whereas for the CFDS solver which often needs profiles of values the panel is shown in Figure 106.

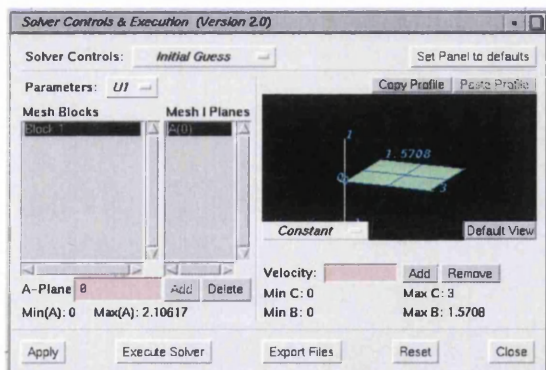


Figure 105 – An 'Initial Guess' Panel for constant initial values

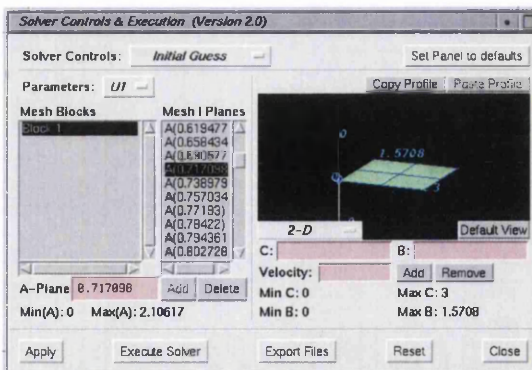


Figure 106 – An 'Initial Guess' Panel for profiles of initial values

For panels that allows the definition of constant initial values, the only input that is necessary is a number for each of the solver variables. For panels that require the user to define profiles of values, the operation is the same as that described for defining profiles of boundary condition values in the previous section.

3.9.2. Solver Execution Mechanism

Once all of the parameters have been entered, the next stage is to actually execute the required solver. In general, as a solver completes each iteration, it produces a trace log of its convergence. This is normally in the form of a table of numbers where the rows are

the iterations and the columns contain various residual values. A typical example of this is shown in Figure 107.

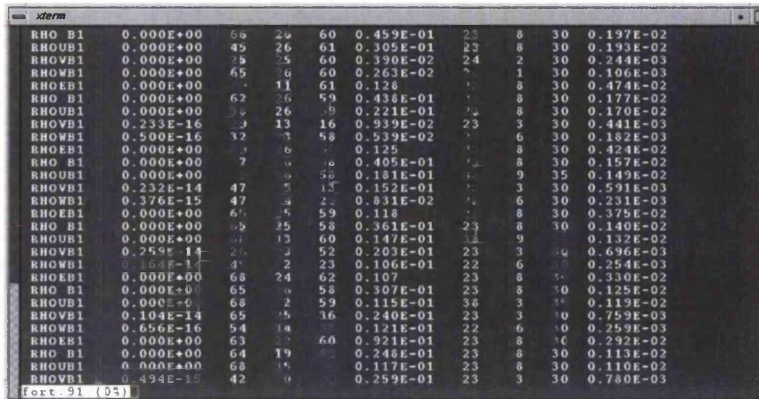


Figure 107 – A typical convergence history log of a solver

Traditionally this log was examined periodically to see if the solver was converging to a solution or diverging. In PROMPT, this periodic examination is performed automatically and a set of convergence plots is continuously updated on the screen. Figure 108 shows an example of this for a typical execution of the CFDS solver.

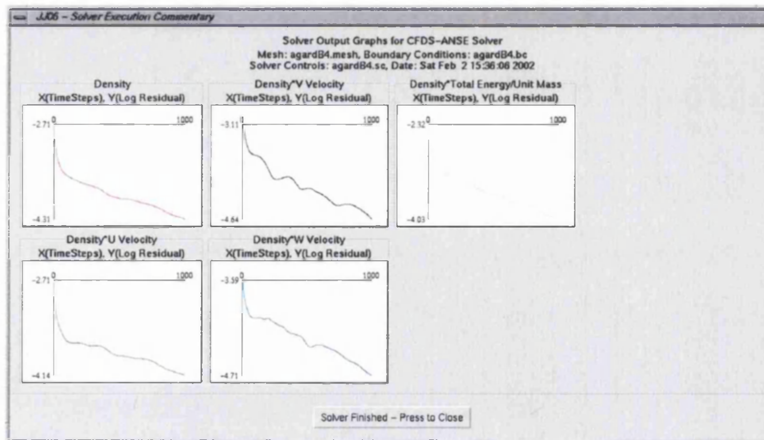


Figure 108 – A typical set of Convergence History Plots for the CFDS Solver

In order for PROMPT to be able to do this, without requiring the solvers to be modified in any way, it is necessary for PROMPT to be able to gain access to the log being output by the solver. The most obvious way of performing this is for PROMPT to periodically open the log file, read the entire file and then update the display. Unfortunately, this approach has a number of flaws:

- Since the solver may have this log file continuously open for writing some systems may not allow the file to be opened by another process for reading.

- When a large number of iterations have been performed, opening the file and reading its entire contents every time the graph is updated will place a load on the processor, thus inducing a performance penalty on the solver.
- Determining the frequency at which the solver log file should be read is difficult since the time it takes to perform one step can vary considerably with the type of solver and the size of the problem.

In order to solve the above problems a mechanism called a *named pipe* is used. This method effectively allows PROMPT to read the history log produced by the solver directly without creating an intermediate file. The first step is to create a special kind of file (called a FIFO) with the same name as that used by the solver. PROMPT then opens this file for reading and, when executed, the solver will open the file for writing. The file will never actually contain any data; instead, any data that is written to the file by the solver is fed immediately to PROMPT for reading. Effectively the file is used purely as a point for the two codes to convene. This method is shown in Figure 109.

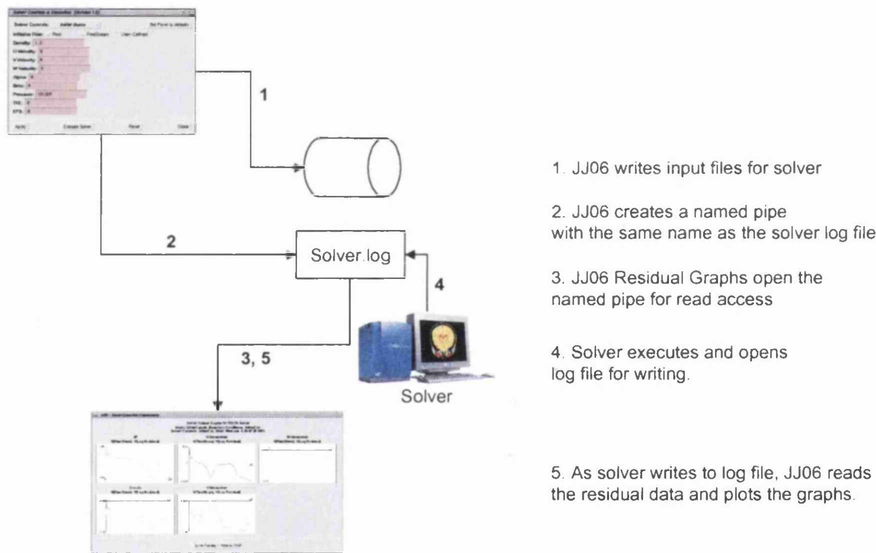


Figure 109 – Connecting PROMPT to a solver using a *Named Pipe*

This approach eliminates any of the previous three flaws:

- Clashes between PROMPT and the solver for access to the file are no longer a problem since this is the sole purpose of a named pipe [Stevens90].
- The named pipe means that PROMPT *sleeps* until data has been written to the pipe by the solver. This is then read by PROMPT, line-by-line, in order to update the display. In effect, PROMPT behaves as if it were reading a convergence history log once from an extremely slow disk.
- The fact that PROMPT sleeps until data is ready to be read means that no decision has to be made about how often the display gets updated.

When the solver has completed it will close the log file and this is signalled to PROMPT by the end-of-file condition. This has the added advantage that PROMPT knows immediately when the solver has finished regardless of whether it terminates after the required number of iterations or terminates abnormally during its execution.

3.10. Solution Visualisation and Post-Processing

Once the solver has finished and a solution obtained it needs to be analysed and presented in a meaningful way to the user. The purpose of the Solution Post-Processing (SPP) Module is to allow the user to define the method used to present the solution and then traverse the large set of solution values in order to extract the necessary information. Most post-processing software packages contain a large number of options to extract features from the solution and present it in a meaningful way to the user. During the development of PROMPT it was decided to utilise the capabilities of Visual 3 [Haimes91a-c, Haimes98a, Haimes98b] for the following reasons:

- It contained all of the necessary post-processing features, e.g. iso-contours, iso-surfaces, cutting planes, vector tufts, stream lines [Darmofal92], particle tracing [Plansky95, Haimes95], etc..
- It was very efficient in terms of both memory and performance.
- It was actively supported so new features were *free*.
- There were a number of other projects within Rolls-Royce that fed into the Visual 3 system in order to enhance it.

An example of the use of Visual 3 is shown in Figure 110. The SPP Module also used EQUATE (EQUATion Editor) [Jones98a] (Figure 111) developed during this project to allow the user to enter any mathematical expression based on the generic solver variables in order to create new variables of interest, e.g. Mach number, pressure, entropy, etc.. EQUATE is described fully in Appendix A.

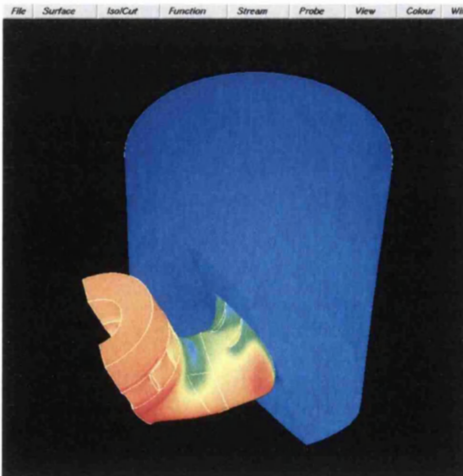


Figure 110 – A typical session using Visual 3

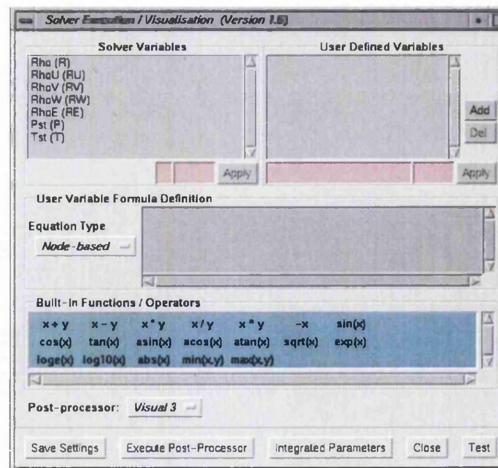


Figure 111 – Defining new equations using EQUATE

3.11. Conclusions and Example Test-Cases using PROMPT

The aim of this section is to show how the use of PROMPT enabled the simulation of two test cases from Rolls Royce. In both cases the actual design engineers, for whom PROMPT was intended, performed the entire simulation. This was in order to give a real estimate of the time taken for a typical simulation when performed by a user that was not a computer specialist.

3.11.1. Agard B4 Test Case

The Agard B4 single nozzle (Figure 112) is one of a set of nozzle geometries in the public domain for which experimental data exists in order to test computational simulation results for realistic test cases.

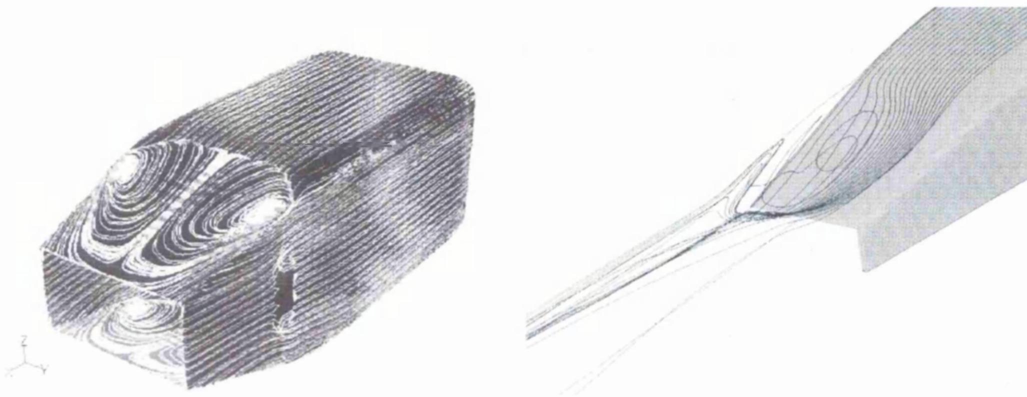


Figure 112 – Illustrations of the Agard B4 nozzle

This geometry was simulated using a structured single-block (using hidden cells), structured multi-block and unstructured meshes. Figure 113 shows a structured mesh around the Agard B4 nozzle and Figure 114 shows the equivalent unstructured mesh. In both cases the geometry is a quarter of the whole with the structure of the nozzle being represented as the region with no mesh elements.

UNIVERSITY OF
LIBRARY
WALES SWANSEA

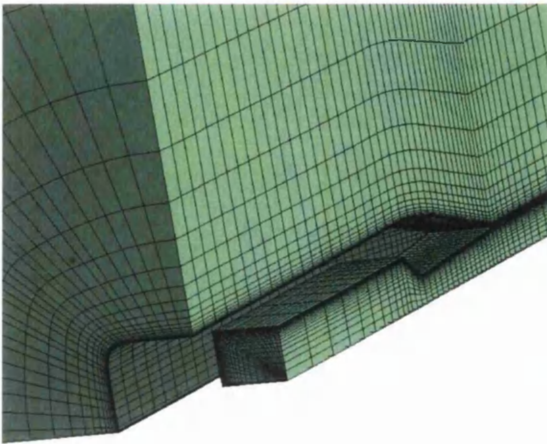


Figure 113 – Structured Mesh around the Agard B4 nozzle

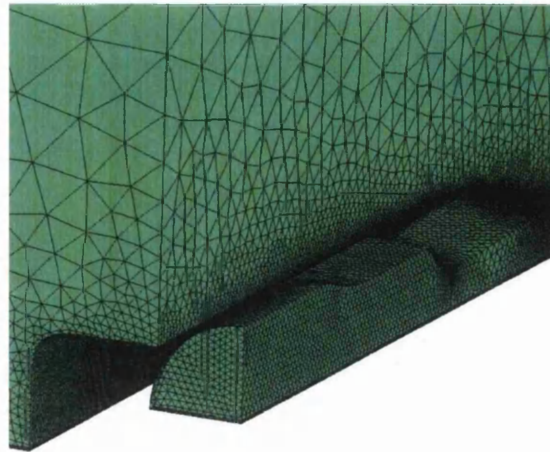


Figure 114 – Unstructured Mesh around the Agard B4 nozzle

The structured mesh was run both through PROMPT, using two versions of an in-house structured multi-block flow and heat transfer solver called CFDS-ANSE, and through a commercial solver, Rampant, developed by Fluent Inc. The unstructured mesh was run through Rampant only. The table below shows the typical run-times and memory usage of the four runs. It was not run through any of the in-house unstructured solvers because they did not have the required heat transfer capability at the time.

	Rampant (Structured)	*Rampant (Unstructured)*	*CFDS-ANSE 1 (Single-block)*	*CFDS-ANSE 2 (Multi-Block)*
Mesh Size (Cells)	212236	556714	93525	215858
Memory Used (Mb)	190	328	78	104
Grid Setup Time (Hours)	4	8	5	10
Convergence (Cycles)	400	400	5000	5000
Solution Clock Time (Hours)	30	100	11	22
Problem Turn-around	3 days	6 days	2 days	3 days

There are a number of points to note about these figures:

- The meshes were generated individually for each run using a size that was considered to give a solution with the required accuracy for the given solver.
- The memory requirements of CFDS-ANSE were considerably less than for Rampant.
- The solution clock time was also considerably less. This coupled with the reduction in memory usage, means that larger simulations are possible on a given platform when using PROMPT with CFDS-ANSE.

- The problem turn-around times were similar. The longer time for the unstructured test case was caused by the excessive run-time of the solver.

Figure 115 shows a comparison between the various simulations and the experimental results. This problem was chosen because it involved complex shock / boundary layer interactions, and as such all of the results stray from the experimental data with similar degrees of error.

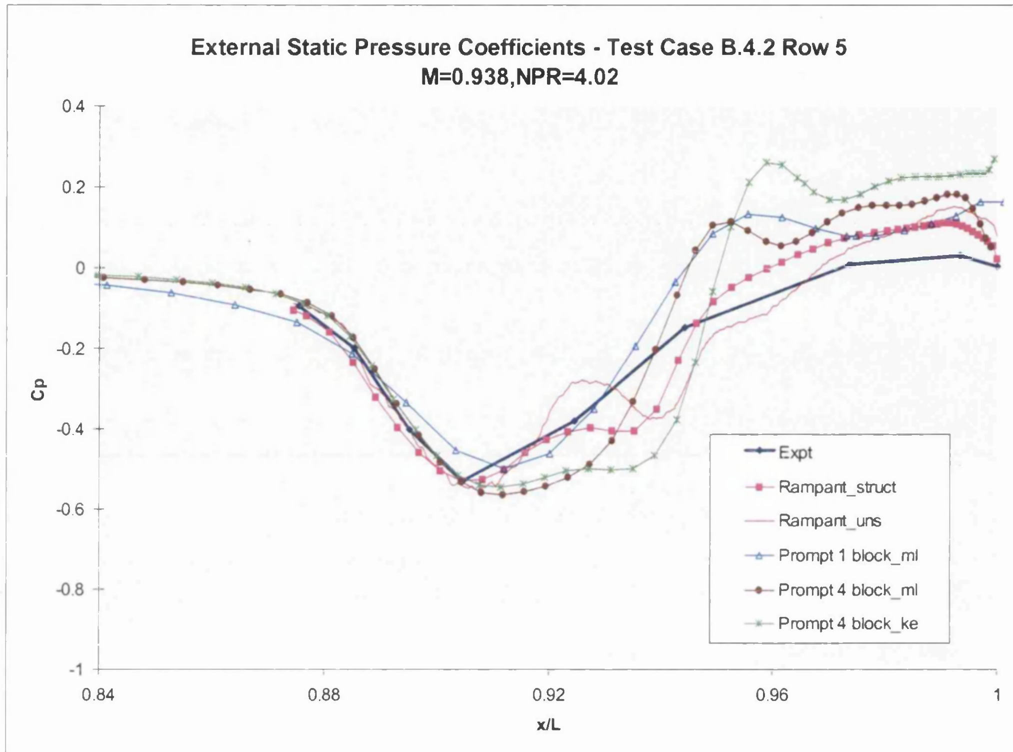


Figure 115 – Comparison between Solver Results and Experimental Data for the Agard B4 Nozzle

3.11.2. Generic Engine for a Vertical Take-off Aircraft

This test case was designed to demonstrate the capability of PROMPT, with the in-house flow solvers, to perform simulations on complex geometries. Figure 116 shows the geometry of the test case.

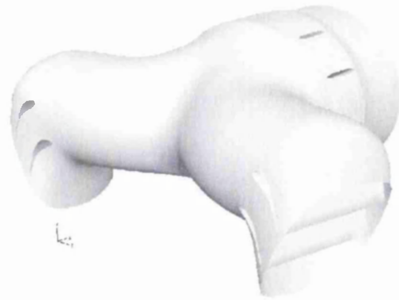


Figure 116 – The Geometry of the Vertical Take-Off Engine

The simulations were performed using only an unstructured, inviscid mesh (Figure 117) due to the complexity of creating a multi-block structured mesh for such a configuration. Only half of the geometry was meshed in order to decrease solution clock time (Figure 118).

Figure 117 – The Unstructured Mesh around the Engine

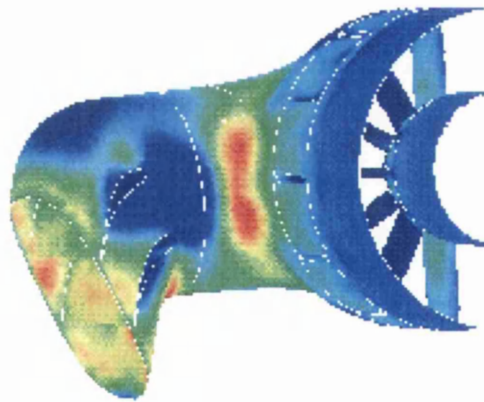


Figure 118 – Solution Contours from the Cindy solver

The simulation was performed using both the Rampant solver and an in-house unstructured solver, Cindy. The table below shows the run times and memory usage as before.

	Rampant	*Cindy*
Mesh Size (cells)	78494	78494
Memory Used (Mb)	54	17
Grid Setup Time (days)	2.5	2.5
Convergence (Cycles)	200	100
Solution Clock Time	2 hours	3.5 hours

With this test case, the two solvers were run using the same mesh. Here the commercial solver, Rampant, converged to a solution in less time but required over three times the memory of the in-house solver, Cindy. This means that, like the previous test case, larger simulations can be performed using the in-house solvers through PROMPT for a given platform. As can be seen in Figure 119, Cindy produces results virtually identical to those of Rampant despite the reduced memory usage.

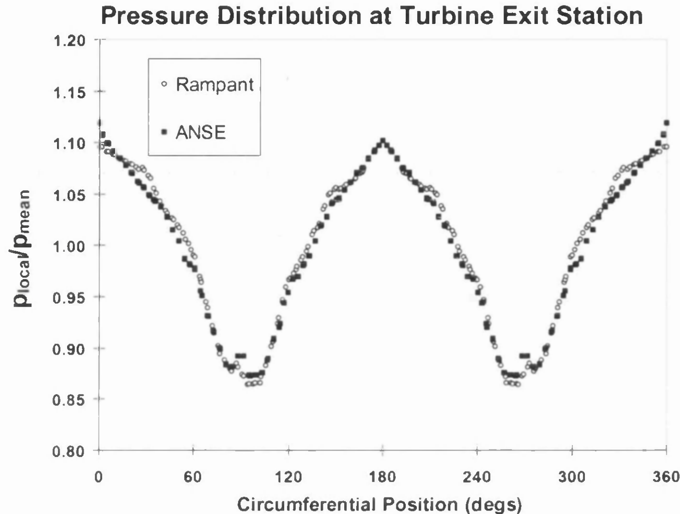


Figure 119 – Pressure Distributions from the Two Solvers

3.11.3. Conclusions

As stated at the beginning of this chapter, the design requirements of PROMPT were to:

- *Enable CFD computations for Nozzle / After-body configurations to be prepared, initiated and examined within an intuitive workstation environment by non-specialist personnel.*

  Feedback from the design engineers that actually use PROMPT on a day-to-day basis is positive. This has been reinforced by the considerable interest shown in PROMPT by a number of other groups within Rolls Royce, namely:

  - LP Compression Systems
  - Radar Cross Section (RCS) and Infra-Red (IR) Assessment
  - Civil Powerplant Group to perform analyses of installed nacelles and
  - Turbine Blading Group.

- *To allow meshes from a range of sources to be submitted to the Rolls Royce production solvers thereby avoiding the memory and CPU time overheads characteristic of commercial codes.*

  Meshes have been imported from a number of sources including GeoMesh (Fluent Inc.), ICEM CFD, Sauna (DERA) and in-house mesh generators from Rolls Royce. These meshes have been successfully processed and passed to a

number of in-house structured and unstructured solvers. The results have then been post-processed using the Visual 3 library within PROMPT. The memory overheads when using the PROMPT approach as opposed to commercial solvers are illustrated in Figure 120[15].

- *To enable exploitation of the best of current and future in-house, commercial and University mesh generation and solver developments.*
  The modular structure of PROMPT has enabled the integration of further solvers and mesh generators from University of Wales Swansea, Oxford University Computing Labs and Loughborough University after the end of the PROMPT project in further support projects.
- *To dramatically reduce the time needed to apply CFD to nozzle / after-body configurations.*
  The turn-around times of the two example test cases shown above are of the order of a few days. That is a significant reduction compared with the turn-around times that were measured in weeks when the PROMPT project started.
- *To minimise cost and lost time arising from pre-processing errors.*
  This has been shown to have been achieved through the reduction in turn-around time and the fact that this was achieved by the actual design engineers rather than specialist computer personnel.
- *To provide portability across SGI and Hewlett Packard workstations.*
  The use of industry standards such as ANSI C, OSF Motif and Open GL have ensured portability between all of the major UNIX vendors.

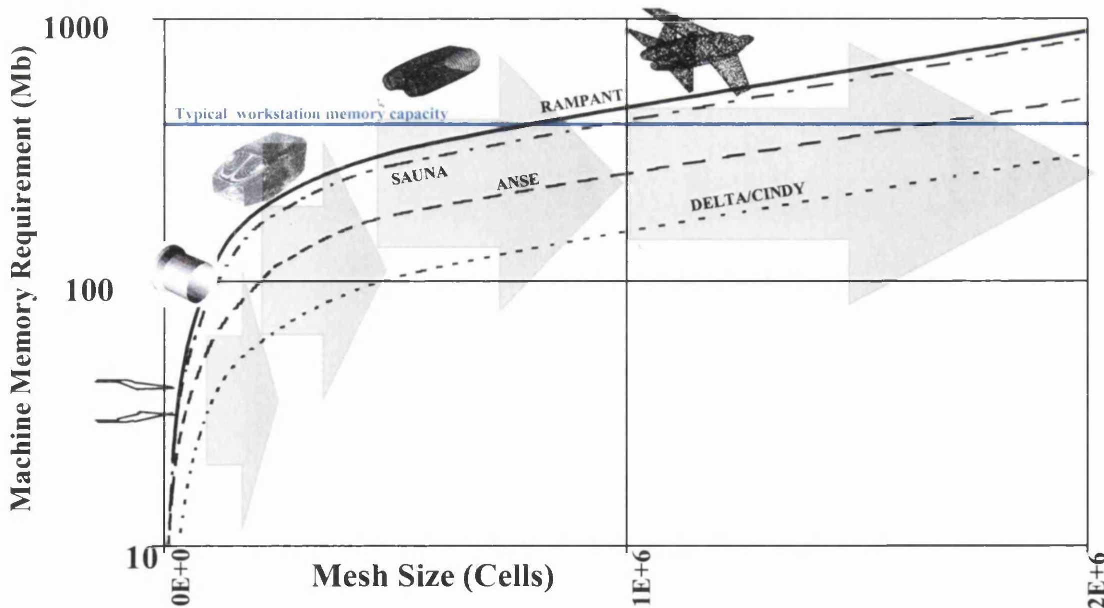


Figure 120 – An Illustration of the Memory Usage for the various Solvers

[15] Extracted from the brochure titles 'Applied Research Package 07b Milestone M83501 Review'

Chapter 4. PSUE II – A PARALLEL PROBLEM-SOLVING ENVIRONMENT

4.1. Introduction

The previous chapter described a Problem Solving Environment called PROMPT that was developed with funding from Rolls Royce and DERA. The purpose of PROMPT was to enable simulations to be performed quickly and easily by the end-user engineer on aerospace components such as nozzle after-bodies, turbine blades, etc. This was achieved by embedding the numerical algorithms already available in these two companies within a user-friendly environment that hid all of the complexities of problem set-up and post-processing from the end-user.

During this time, another environment, called PSUE (Parallel Simulation User Environment) [Turner-Smith96a, Turner-Smith96b, Marchant96, Turner-Smith98, Weatherill99, Zheng00], was being developed within a large European ESPRIT project called CAESAR [Risk96]. This environment had very similar aims but was targeted at a more general-purpose market and used the unstructured grid technology available within University of Wales Swansea.

Both of these environments were very successful at achieving their objectives, but near the end of their development it became clear that there was a growing need for a general-purpose PSE that could handle much larger simulations than could be performed on a single workstation. To address the problem a follow-on project to CAESAR was conceived called JULIUS [Rowse00] in which an integrated PSE would be developed which contained all of the numerical tools necessary to perform very large-scale simulations fully utilising parallel computer hardware.

This and the next two chapters describe the developments that took place in order to produce such an environment. This chapter introduces the aims of the work along with some of the initial design decisions that had to be made before any development could commence. The next chapter then continues with a description of the initial implementation of the PSUE II [Jones00, Weatherill00a, Weatherill01a, Weatherill01b, Weatherill02]. The third chapter then continues with a description of the improvements made to Version 2 of the PSUE II.

4.2. Context of PSUE II within the JULIUS Project

The JULIUS Project (ESPRIT 25050) was a large European project involving many partners, both in the form of large industries and smaller research institutions[16]. The aim of the project was to produce a seamless environment in which the end-user engineer can

[16] The partners involved in the JULIUS project were BAE Systems (UK), EADS (Germany), Dassault Aviation (France), ESI (France), Genias (Germany), IPK (Germany), University of Wales Swansea (UK), University of Oxford (UK), SMR (Switzerland) and NAG (UK).

focus more creatively on their design goals rather than on the software and hardware complexities of managing computational simulations on parallel platforms.

Consequently, its objectives were:

- To develop an integrated functional HPCN environment for simulation in multiple disciplines,
- To provide and demonstrate HPCN tools and engineering simulation tools that efficiently work together in this environment,
- To put developments in place to remove the major limitations and bottlenecks in engineering simulations and
- To demonstrate the entire system working with embedded applications software for realistic, industrial problems.

The resulting environment was named 6S (Sixth Sense) and was designed to have a modular structure into which 3rd-party applications could be integrated.

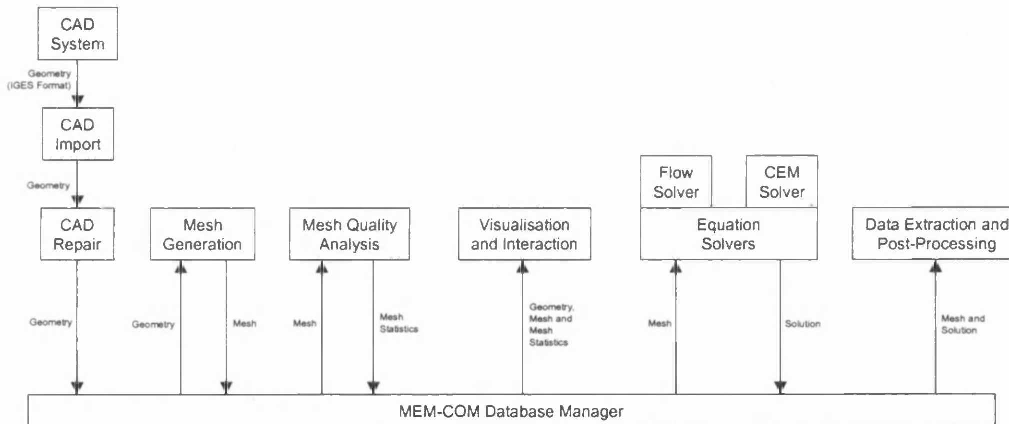


Figure 121 – A Schematic of the 6S Environment showing the data flows

Figure 121 shows a schematic illustrating both the various data flows through the 6S Environment and the individual modules that were combined to form the environment. Although this gives a good indication of the size and complexity of the structure of the environment, it does not show the structure in terms of how the end-user would see it. Obviously, if an environment is to appear seamless to the end-user it needs a central focus in which most, if not all, of the user interaction takes place. The PSUE II (Parallel Simulation User Environment 2) performs this central role. Its generic functionality encompassed the modules developed by the University of Wales Swansea, with the remainder being linked in as 3rd-party modules in a seamless manner. As can be seen in Figure 122, this provided a central focus to 6S encompassing the following functionality:

- Visualisation and Interaction with the various data sets involved in a computational simulation,
- Mesh Generation [Weatherill94a, Hassan96, Weatherill00b, Larwood01],

- Equation Solving (both CFD [Hassan01, Sorenson01] and CEM [Hassan02, Morgan00]),
- Mesh Refinement [Weatherill94b] and
- Integration of 3rd party applications.

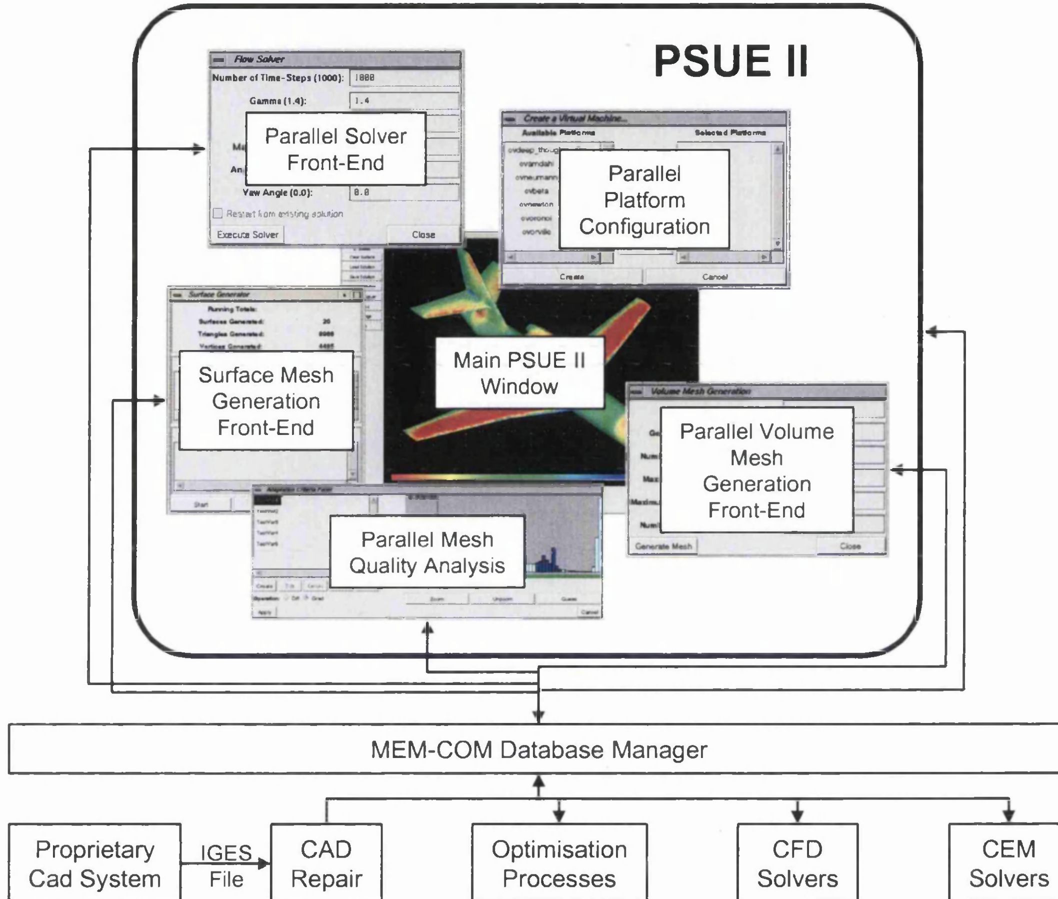


Figure 122 – A schematic showing the logical layout of the 6S Environment

4.3. The Requirement for a Parallel Environment

Although Problem Solving Environments, such as PROMPT, do enable the spread of computational simulation software into the domain of the *traditional* engineer, there is an inherent limitation on the complexity of simulation that may be performed. This is due to the architecture of the environment being designed to run exclusively on a standard graphical workstation. This isn't a problem when simulating the behaviour of individual components where the size of the meshes involved is small, such as turbine blades in the aerospace industry.

However, as the use of such environments becomes more commonplace, there is an increasing requirement to be able to perform simulations using meshes of a much larger magnitude. This increase in size has come about in two ways:

Simulating Larger Portions of the Problem

When engineers became accustomed to the benefits of using computational simulation in the design of turbine blades, it was natural to want to apply the same techniques to the design of the whole engine. Moving to the airframe, engineers began using computer simulation to design aircraft wings, which then naturally, led to the requirement to simulate the entire aircraft.

The Strive for Increased Accuracy

As computational simulation was used for geometries that were more complex, there was a need to capture smaller features of the solution more accurately. In CFD, for example, this has led to inviscid simulations being replaced by viscous, turbulent calculations, which, in turn, require much larger meshes.

4.3.1. An Example: The Equation Solver

As numerical simulations are getting larger and more complex, they place a huge burden on the available computational resources. Despite this, the requirement for rapid turn-around of solutions is still paramount. Ideally, a turn-around time from a given geometry to the solution should be of the order of a few days. In terms of computer time, by far the most expensive section of the whole process is the equation solver. It is, therefore, essential that as simulations get more complex that special attention is paid to the run-time of the solver.

Figure 123 and Figure 124 illustrate some typical scenarios of large simulations along with their required mesh sizes and computational demands for the solver [Hassan02]. The figures are based on performing a simulation over an entire aircraft configuration.

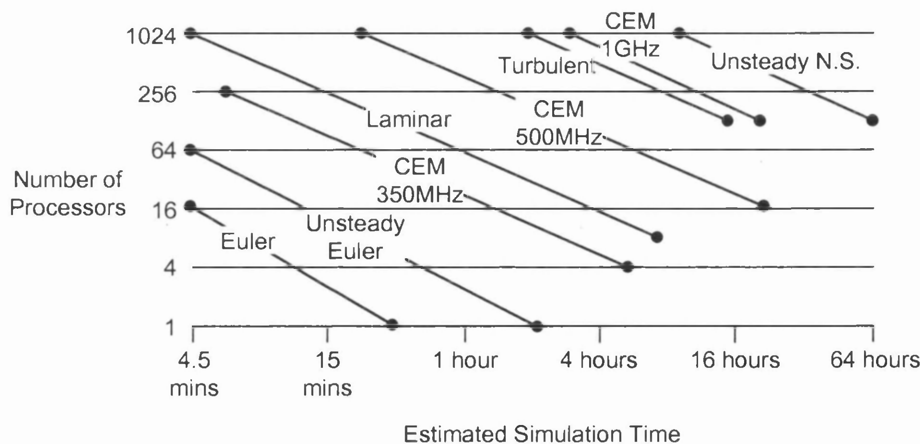
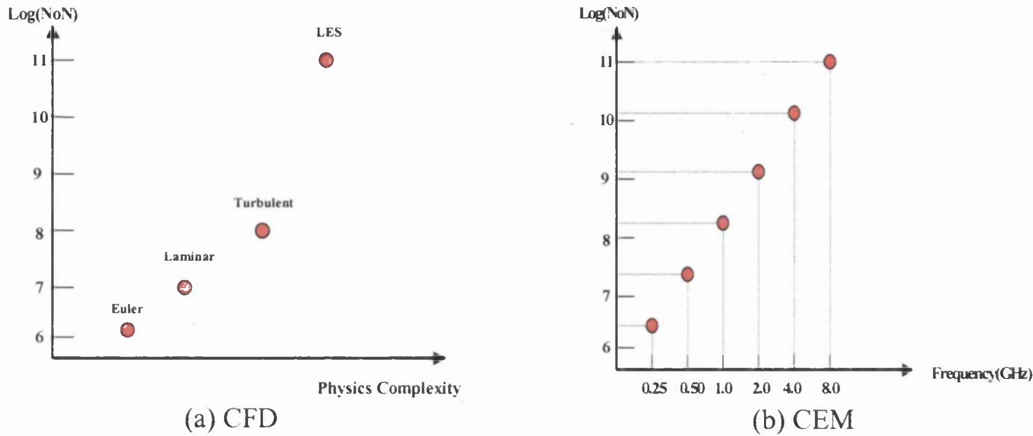


Figure 123 – Estimated Execution Time for Typical Simulations



(a) CFD (b) CEM

Figure 124 – Typical Mesh Size for Simulations on a Complete Aircraft

This clearly shows that, even without considering linking together multiple simulations, as the simulations get more complex, the size of the meshes involved and the demands the solvers place on the available computing power increase dramatically. Therefore, it is obvious that if state-of-the-art calculations are to be performed on a regular basis then considerable computing power is required; far more than can be obtained using a standard workstation / PC. For these types of calculations it is clear that parallel computing platforms cannot be beaten in terms of performance (for dedicated parallel computers) or performance / cost ratio (for clusters of networked workstations / PCs).

As mentioned in a previous chapter, in order to design a module, such as a solver, for a parallel computer, the most scalable, and certainly the most flexible, strategy to use is the message-passing paradigm, which assumes a distributed memory architecture. This means that the data sets on which the solver operates must be split amongst the various processors. Each processor can then spend most of the time working on its own sub-set of data producing a local solution, only occasionally passing data along a network between the processors to complete the global solution. A typical sequence of operations is shown in Figure 125.

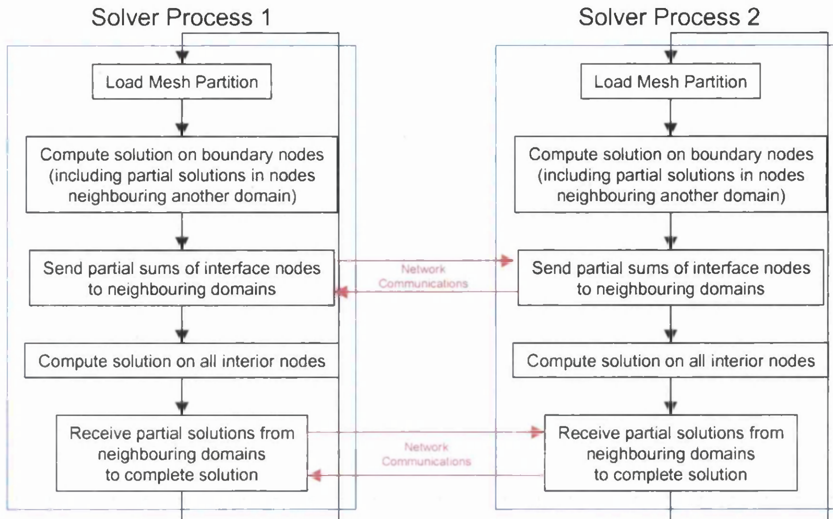


Figure 125 – Typical sequence of operations within a Parallel Solver

The methods used inside the PSUE II to split the various data sets are described in Section 4.4, titled 'Distributing the Data-Sets'.

4.3.2. An Initial Parallel Problem Solving Environment

An initial extension to the architecture of a PROMPT-like environment could be implemented by leaving the graphical front-end of the environment running on the graphical workstation but move the computationally expensive modules, such as the solver or mesh generator, to the parallel computer.

This would be a simple extension just requiring the addition of a facility to execute modules either locally or remotely on another computer. In this environment, each module is treated as a *black box* by the PSE, just requiring an indicator as to whether the module should be executed locally on the workstation or remotely on the parallel computer. The extra work necessary for the parallel module in order to spawn *n* copies of itself and partition the data set could conceivably be done by a script outside the environment. This scenario does make the assumption that there is a common file-store between the parallel computer and the workstation, but this can often be achieved in a closed environment through the use of NFS (Network File System) or similar.

Although this implementation of a Parallel PSE would deliver a performance increase over an environment that was purely sequential, the sizes of simulation are still limited to what can be physically stored and manipulated on the graphical workstation. This is illustrated in Figure 126 where the partitioned data sets from a parallel mesh generator are passed to the parallel solver after the boundary conditions have been applied inside the PSE.

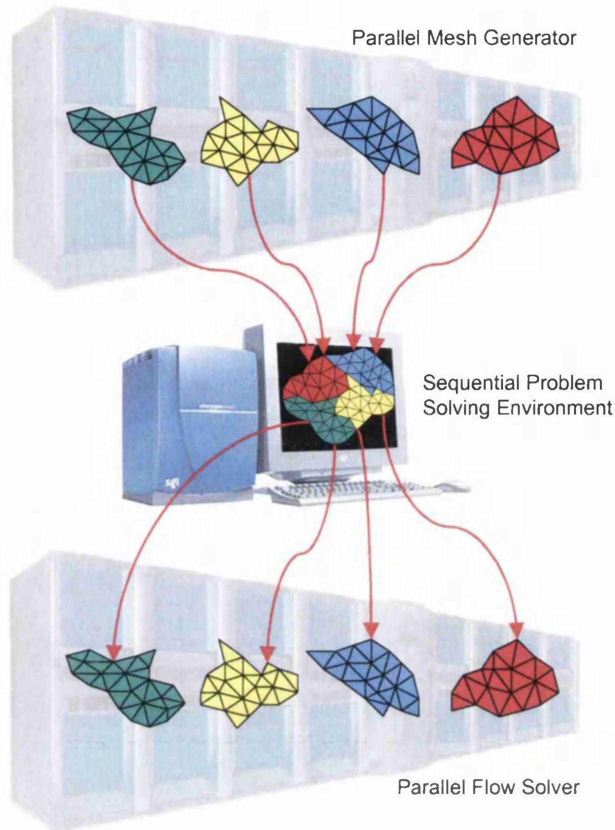


Figure 126 – The Bottle-neck produced by a Sequential Process

In order to be able to perform large-scale, state-of-the-art calculations in a user-friendly Problem Solving Environment, a new architecture that utilises the power of the parallel computer throughout the simulation process is necessary. An overview of such a structure is shown in Figure 127.

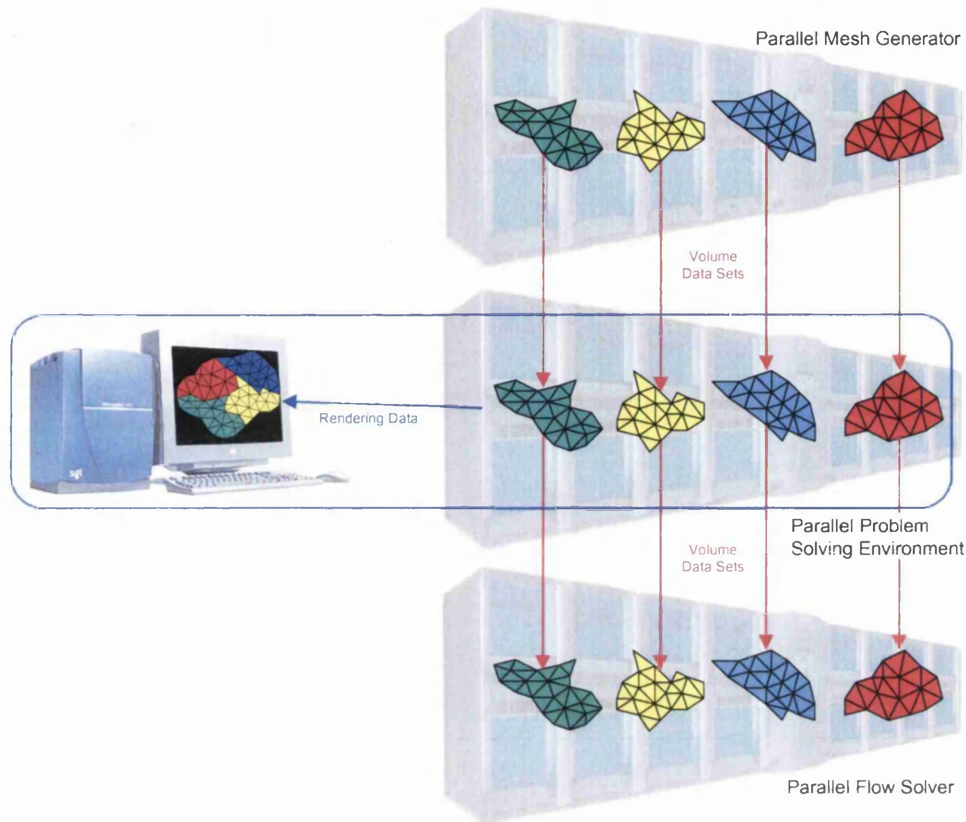


Figure 127 – The PSE with no Sequential Bottlenecks

Here, the same steps are performed except that the actual PSE runs on the parallel computer, thus removing any sequential bottlenecks. This means that the partitioned data sets are never recombined which, in theory, limits the size of simulation that can be performed to the size of the parallel computer.

4.4. Distributing the Data-Sets

The four major data sets used within a PSE are the geometry, the surface mesh, the volume mesh and the solution. This section will begin by describing, in the usual order of creation, how each of these four data sets are stored as global entities. Afterwards, the manner in which these four data sets are partitioned will be described.

4.4.1. Sequential Geometry Format

The first data set to be created during the simulation process is the geometry. In fact, the actual creation of this data set is often deemed to have occurred before the simulation process begins. Regardless of which geometrical representation was used to construct the geometry it must be converted to use a Boundary Representation (BREP) model based on a Ferguson patch representation [Hassan99a, Hassan99b] before it can be imported into the PSUE II.

This form of representation is constructed from three main entities; surfaces, curves and topological information. The combination of all three allows the construction of a *watertight* model on which mesh generation can be performed. Figure 128 shows a simple example consisting of half a sphere adjoining a flat plane.

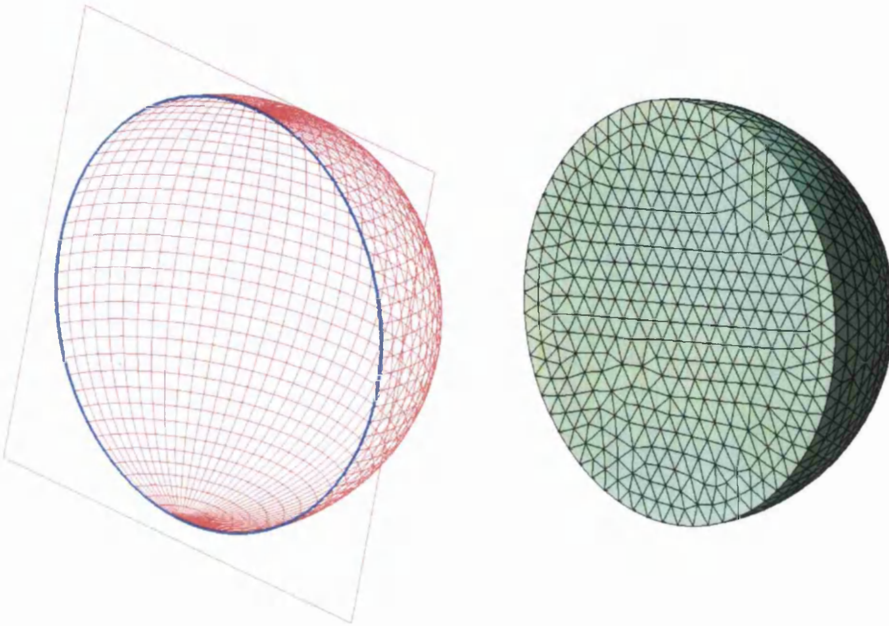


Figure 128 – A simple geometry illustrating its components

As illustrated, the main structure of the geometry is made up of the two parametric surfaces (*red*). In order for the model to be topologically valid, a number of curves (*blue*) must also be present to denote the boundary of the surface on which they appear. In order to be topologically valid and form a closed domain the following rules must apply:

- Each curve must be associated with two, and only two, surfaces.
- Each trimmed portion of a surface must be completely bounded by a whole number of curves.

The resultant surface mesh is shown alongside. This illustrates how the curves that interface the half sphere and the plane cut a disc out of the plane in order to form a watertight solid. The generation of the volume mesh would produce a solid half-sphere.

Figure 129 illustrates the concept of trimming portions of a surface. Here the two half-spheres trim a halo out of the plane with the resultant volume mesh producing a hollow half-sphere.

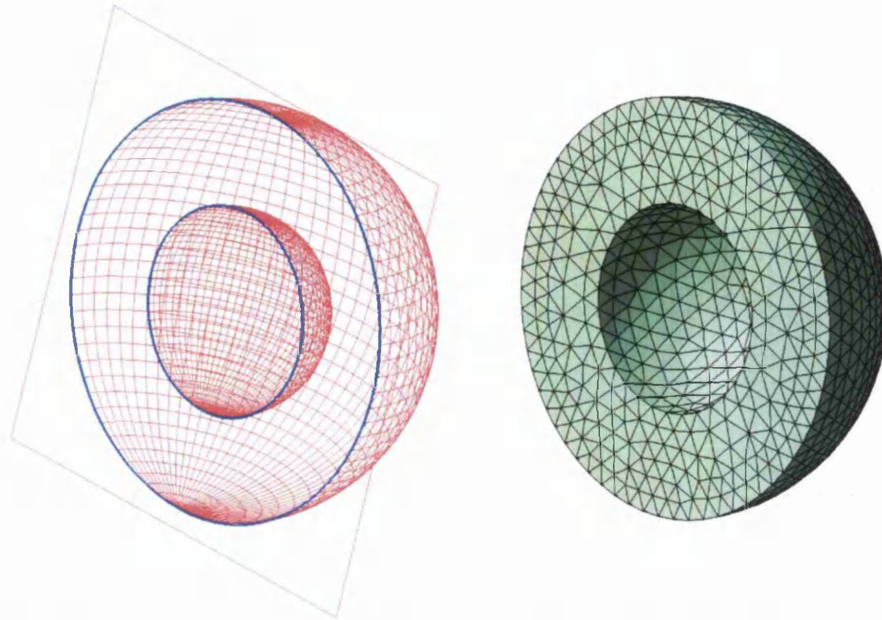


Figure 129 – A More Complex Geometry illustrating the concept of Trimmed Surfaces

In order to complete the description of the geometry, topological information is needed to associate the curves and surfaces in order to produce valid volumes. This is simply stored as a sequence of pairs, one for each curve, with each pair representing the two surfaces associated with it.

4.4.2. Sequential Surface Mesh Format

In its simplest form, the surface mesh approximates the surface of the original geometry with a collection of 2D linear elements (triangles and/or quadrilaterals). Together, they completely enclose the one or more domains in which the volume mesh will be generated. These elements are stored in two parts using a standard connectivity table. The first table stores the coordinates at the vertices of the elements. A second table is then used to define the elements by specifying their vertices using the indices from the coordinate table.

However, during the surface mesh generation process the mapping between the geometry surfaces and curves, and the surface mesh is created as a by-product. This information comes in the form of a set of tables, one for each surface and curve. Each table lists the nodes in the surface mesh that lie on the particular surface, or curve, and their respective parametric coordinates, (u,v) pairs for surfaces and u coordinates for curves. This information is stored alongside the surface mesh since it is both very useful during later stages of the simulation (e.g. calculating normals at mesh points or point projection during mesh refinement), and difficult to compute at a later stage.

4.4.3. Sequential Volume Mesh Format

The sequential volume mesh format is very similar to the surface mesh in that it represents the mesh as a collection of vertex coordinates and elements referencing the appropriate coordinates. The volume mesh is described using three tables. The first lists the coordinates of all of the vertices in the mesh. The second references the vertex table in order to define the volume elements (tetrahedral, pyramids, prisms and hexahedra). The final table defines the surface elements (triangles and quadrilaterals). These are defined in exactly the same order as the surface mesh data format and contain three sets of data:

- Face – Node connectivities - These refer to the nodes of the volume mesh instead of the surface mesh.
- Surface Number – As with the surface mesh, this contains the geometry surface on which the face lies.
- Parent Element – This references the volume element on which this boundary face lies.

This small amount of duplication provides a convenient path from the volume mesh back via the surface mesh, through the duplication of surface elements, to the geometry, through the inclusion of the parametric coordinates of the surface nodes in the surface mesh format. This path is illustrated in Figure 130.

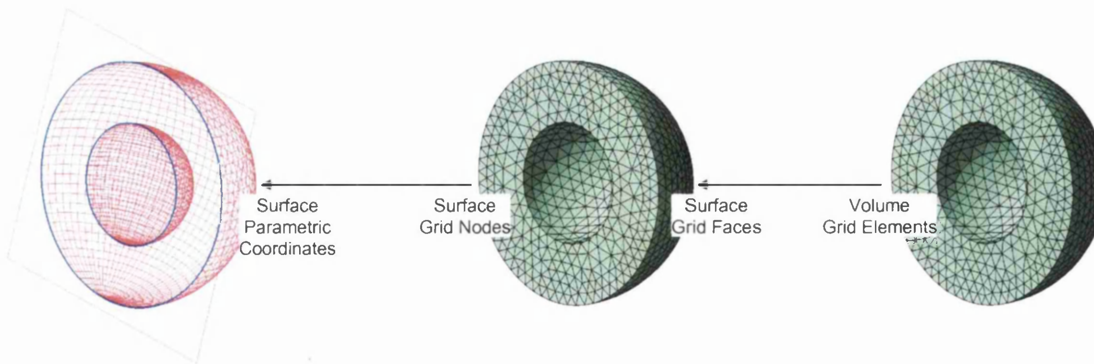


Figure 130 – The Path from the Volume Mesh back to the Geometry via the Surface Mesh

4.4.4. Sequential Solution Format

The solution data set is the last of the major data sets, and has the simplest structure. The solution data set falls into one of two categories based on whether it is overlaid on a surface or volume mesh. Regardless of the type, it simply consists of a set of tables, each containing the values of one variable at each of the nodes in the surface or volume mesh.

4.4.5. Partitioned Geometry Format

Although the geometry is often the smallest of the four main data sets in the simulation process, for complex configurations the number of surfaces and bounding curves can be very significant. As an example, Figure 131 shows a model of the F16 fighter jet and the Airbus A3XX, both of which consist of over 500 parametric surfaces and over 1000 curves.

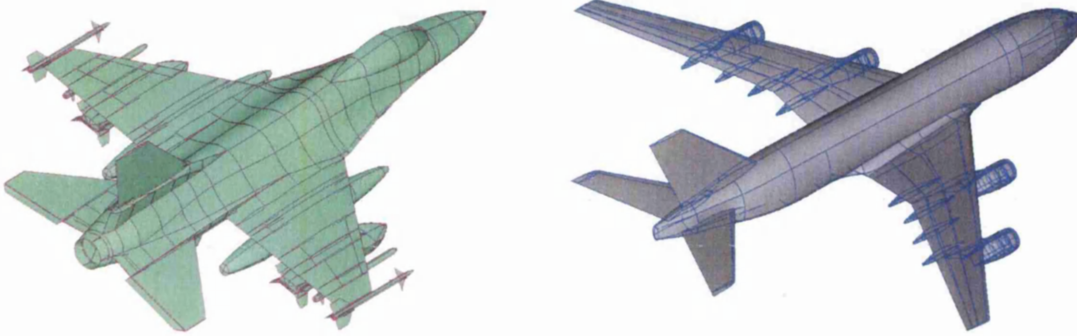


Figure 131 – Examples of Complex Aerospace Configurations

Performing calculations, or searches, on geometries of this size can be quite time-consuming. It is therefore, for speed, rather than memory capacity, that these geometries are partitioned and stored on the parallel computer.

As mentioned previously, a geometry is composed of three main components: surfaces, curves and topology information. When partitioned, the surfaces are spread amongst the processors, as are the curves. The topology information is then partitioned in a manner such that every surface knows on which processors the curves that surround it reside, and each curve knows which processors contain its two adjacent surfaces.

It is acknowledged that partitioning a geometry surface by surface is not an ideal method for partitioning in terms of load balancing, as compared to splitting individual surfaces and/or curves amongst processors, but it does have some major advantages:

- The partitioning algorithm is extremely quick and simple.
- There are usually many more surfaces and curves than processors, so by distributing the largest entities first, a pseudo-load balancing is achieved.
- Many operations performed on geometries require information about a whole curve or surface at a time so performing a distribution where these entities are kept as a whole reduces inter-process communication to a minimum.

4.4.6. Partitioned Surface Mesh Format

Invariably surface mesh generators generate surface meshes by placing points along the bounding curves and then discretising the individual surfaces in a sequential manner.

This allows the surface mesh to be easily partitioned in a similar manner to the geometry data set, in that collections of surface elements are grouped together depending on which geometry surface they were generated. These are then distributed as whole entities in the same manner as the actual geometry surfaces. This has the same disadvantage in that load balancing may be far from ideal but also has similar advantages:

- The partitioning algorithm is quick and simple.
- There are usually many more surfaces than processors so a pseudo-load balancing can often be achieved.

It should be noted that although the surface mesh is sub-divided based on the original geometry surfaces, the partitioning algorithm would invariably not place corresponding geometry and mesh surfaces on the same processor. This is because a more complex surface description does not necessarily result in the generation of a large number of surface mesh elements. For example, a symmetry plane will often have a very large number of surface elements generated on it but, geometrically, it is a very simple entity defined using only four points.

4.4.7. Partitioned Volume Mesh Format

The partitioned volume mesh is by far the largest of the four major data sets. In fact, as could be seen from Figure 124 the size of this data set can easily reach many 100's of Megabytes. Due to its size, and the fact that it is the primary data set for many of the parallel modules, it is the most important data set to be partitioned for use in a parallel environment.

Obviously, the volume mesh could not be partitioned in the same manner as the surface mesh since the majority of the elements do not touch any surface so a new strategy had to be adopted. It became obvious, looking through the literature, that there have been many algorithms developed over the years for partitioning volume meshes in preparation for the execution of parallel solvers [Hsieh95, Karypis98, Karypis02, Walshaw97, Walshaw01, Walshaw02a, Walshaw02b]. Despite the wide variety of methods, they all have a number of common goals:

Balanced Partitions

> If a number of processors are all performing the same operations on a sub-set of the mesh then it is obvious that these sub-sets all need to be approximately the same size in order to make full use of processing power. If partitions are unbalanced then processors operating on the smaller partitions will finish sooner than the larger partitions and will wait for the larger partitions to finish. For example; given a small mesh consisting of just 16 elements partitioned into 4 sub-domains, Figure 132 shows a plot of processor usage per solver iteration when the partitions are balanced, each with 4 elements. Figure 133 shows the same plot when the partitions contain two, eight, two and four elements respectively.

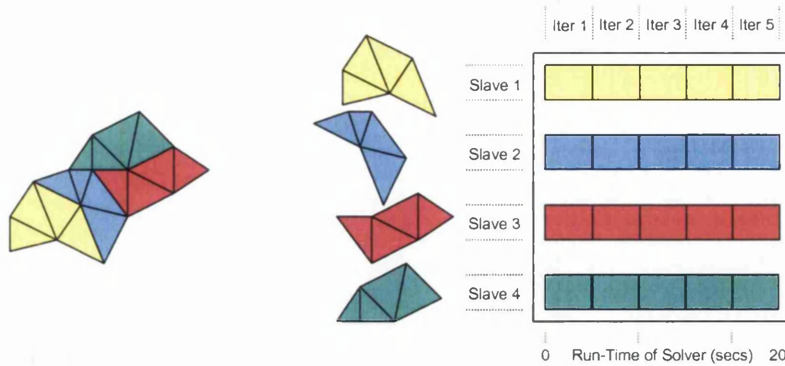


Figure 132 – A solver running with 4 balanced partitions

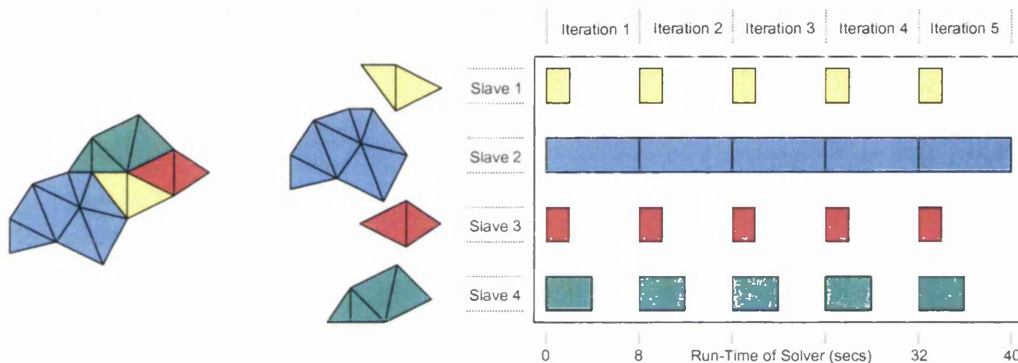


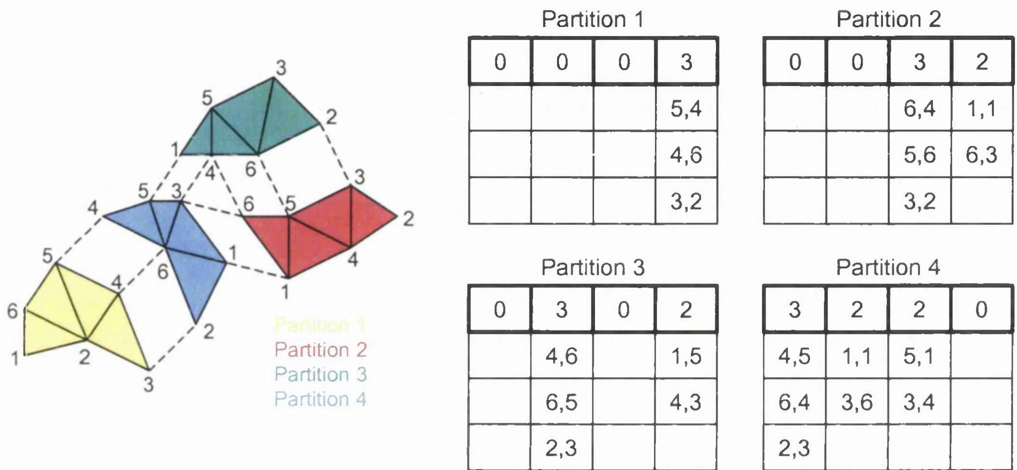
Figure 133 – A Solver running on 4 unbalanced partitions

Minimum Communication

When a process needs information from a neighbouring partition, that data needs to be transferred between the two processors along a network. Depending on the type of parallel computer, the speed at which this can be achieved can vary from approximately 10Mb/s, for a network of clustered PCs, to over 10Gb/s for dedicated connections inside an MPP (Massively Parallel Platform) such as the Cray T3E [CrayInc01]. However, regardless of the speed of this connection, it is not comparable to the speed of the actual processor so minimising communication is still of fundamental importance. Therefore, an ideal partitioning of a mesh minimises the number of interface nodes.

The structure in which the partitioned mesh is stored on each processor was largely determined by the requirements of existing parallel solvers. This is illustrated in 2D in Figure 134. Here, the mesh is sub-divided so that each partition always contains complete elements. This is done in a similar manner to a jigsaw where, if the partitions were solid objects, they could fit back together again to form the original mesh. This has the side effect that nodes along the boundaries formed by the partitioning process (interface nodes) are duplicated in each partition.

For each partition, a set of communication tables exist, one for each neighbouring partition; where each table contains pairs of node indices, the first being the interface node in the local partition and the second being the node index of the coincident point in the neighbouring domain. The ordering of the nodes in these tables is also significant. Each process lists its interface nodes in the same order as its neighbouring partitions. This allows the sending process to pack any data that needs to be sent to its neighbours by simply traversing the communication tables. The receiving process then unpacks the data by simply traversing its own communication table. This allows bulk data to be communicated without the need to send neighbouring node numbers.



Partition 1

0	0	0	3
			5,4
			4,6
			3,2

Partition 2

0	0	3	2
		6,4	1,1
		5,6	6,3
		3,2	

Partition 3

0	3	0	2
	4,6		1,5
	6,5		4,3
	2,3		

Partition 4

3	2	2	0
4,5	1,1	5,1	
6,4	3,6	3,4	
2,3			

Figure 134 – Communication Structure between Mesh Partitions

This information, along with the partitioned meshes, are all most parallel modules need in order to perform their calculations. For modules that need to link back to any geometrical information a link between the surface meshes of the partitions and the original surface mesh is needed. Due to the introduction of extra surface faces on inter-partition boundaries the surface mesh for a partition is no longer an exact duplicate of the original surface mesh. This is overcome by the existence of a set of tables, one for each partition, that map between the surface node numbers of the partition and their corresponding nodes in the original surface mesh. This link is shown in Figure 135.

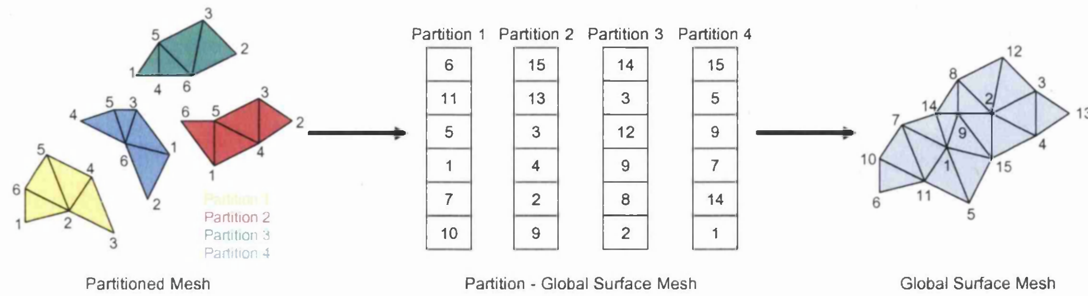


Figure 135 – The link between Mesh Partitions and the Original Surface Mesh

The aim of the Parallel PSE is to always store and operate on these volume meshes as individual partitions since they are considered too large to ever be combined into one global mesh.

4.4.8. Partitioned Solution Format

The partitioning of a solution data set depends heavily on whether it is associated with a surface or volume mesh. Regardless of its type, the solution data set is always partitioned in the same manner as the mesh data set on which it is based.

4.5. Summary

This chapter has described the requirements and basic design considerations of the parallel PSE called the PSUE II. Chapters 5 and 6 continue the theme by describing the challenges facing the design and implementation of such an environment in greater detail.

Chapter 5. THE IMPLEMENTATION OF THE ENVIRONMENT (PSUE II V1.0)

Chapter 4 introduced the basic requirements of the PSUE II along with the definition of the major data sets. Before it was possible to implement a PSE that could handle simulations with many 10's of millions of elements, a number of key issues need to be addressed:

- The method used to enable the real-time visualisation and interaction with these very large data sets,
- The facilities utilised to provide the internal data communications within the Environment.
- The use of these communication facilities to control the overall flow of operations between the various modules of the environment.

5.1. Visualisation and Interaction Issues

As with PROMPT, the ability to visualise and intuitively interact with the various three-dimensional data sets throughout the entire Computational Simulation process is of fundamental importance. Obviously, the techniques used within PROMPT to visualise moderate data sets on one workstation do not lend themselves well to visualising very large data sets distributed across multiple processors. In order to provide real-time visualisation and interaction capabilities with this type of data requires the algorithms to be modified in order to both make efficient use of the extra performance available on the parallel platform, and the rendering ability of the graphics workstation.

Unfortunately, unlike many parallel, number crunching algorithms, the Visualisation Process is a prime example of Amdahl's Law [Amdahl67]. This is due to the necessary existence of the single graphics workstation with which the user interacts with the model. This means that no matter how many processors are available on the parallel platform, the graphics workstation always introduces a sequential bottleneck.

5.1.1. The Visualisation Pipe-Line

The visualisation process is often described in terms of a pipeline, i.e. a process consisting of a number of stages in which the output from one stage is used as the input to the next. Before we can attempt to parallelise this pipeline, a clearer understanding of the various stages is required.

One possible breakdown is shown in Figure 136. This representation defines the pipeline as a three-stage transformation that uses the *Data Reduction Process*, the *Rendering Process* and the *Imaging Process* to convert the volume data sets to a rendered image on the display.

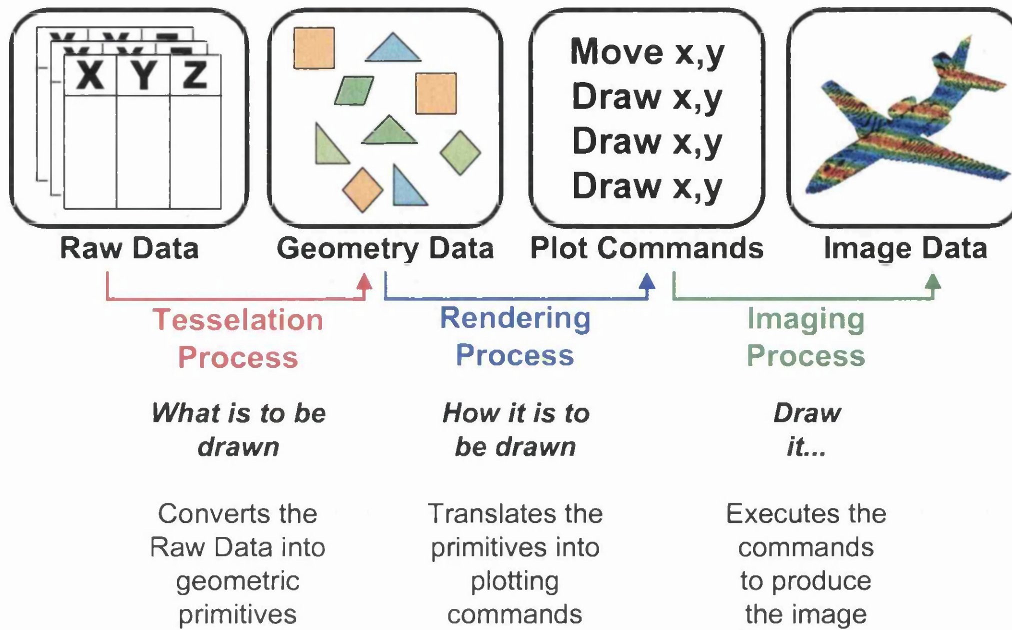


Figure 136 - The Visualisation Pipe-Line

The Data Reduction Process

This involves the conversion of the raw data sets into a number of sets of simple geometric primitives, e.g. points, straight lines, triangles or planar quadrilaterals. The operations that are executed in order to perform this conversion are wholly dependent on the chosen representation of the data sets.

For example, if the data set represents a geometry then it invariably takes the form of a series of co-ordinates and/or coefficients which define the parametric form of a patchwork of bi-cubic surfaces (e.g. Ferguson Patches, NURB surfaces, etc.) which, together, form the boundary surface of the model. With this representation, the Data Reduction Process involves calculating the approximation of the curved surfaces with a set of quadrilaterals and/or triangles. This process always involves a trade-off between the need to maintain a faithful reproduction of the original surface and the need to minimise the number of generated polygons in order to maintain an interactive frame-rate.

For an unstructured mesh, the data-sets take the form of a list of co-ordinates of the nodes in the mesh along with the connectivity information which defines how these nodes are connected together to form the various surface and volume elements which comprise the mesh. For this representation, the elements already form triangles and quadrilaterals so no tessellation is necessary. However, requiring the graphics workstation to render every face inside a mesh is not only inefficient, but serves no purpose except to produce a cluttered display. In order to reduce the number of primitives that are drawn and, at the same time, produce a clearer picture of the mesh, a decision is usually made to only render the

outermost parts of the visible mesh. The Data Reduction Process does this by extracting only the faces that are on the boundary of the mesh along with any that lie on user-defined features within the mesh, e.g. cutting planes, iso-surfaces, etc.

The Rendering Process

This process uses the output from the Data Reduction Process and converts the series of geometric primitives into actual plotting / drawing commands which can later be turned into an image. Since the Rendering Process only has to perform operations on sets of geometric primitives, and not the multitude of original representations, its operation is simplified considerably. The only extra information that needs to be provided at this stage are the various appearance attributes of the primitives. These attributes include the colour used to draw the primitives, the detail used for rendering (e.g. sparse points representing the nodes, a wire-frame outline or a solid, lit representation).

The Imaging Process

This process uses the plotting commands from the Rendering Process to produce the final image on the display of the workstation. It is here that the final transformations take place to convert the three-dimensional coordinates to a two-dimensional screen taking into account the various zoom, translation and rotation operations performed by the user.

So to summarise, the Data Reduction Process decides *what is to be drawn*, the Rendering Process decides *how it is to be drawn* and the Imaging Process *draws it.* The distribution of this pipeline can be thought of as partitioning these three processes between the parallel platform and the graphics workstation. This needs to be done in a manner that ensures the most efficient use is made of both types of computer, and equally important, to ensure that the amount of traffic communicated along the inter-connecting network is minimised.

5.1.2. Distributing the Visualisation Pipe-Line

In this section three approaches to the distribution of the visualisation pipe-line are discussed; each being classified by the type of data that is transferred between the parallel computer and the workstation.

Scenario 1 – The Image Data Transfer method

The first scenario involves the parallel computer performing almost the entire visualisation pipeline. As Figure 137 illustrates, each processor in the parallel computer performs the entire Data Reduction Process, the entire Rendering Process and a significant part of the Imaging Process. As with the other methods each processor performs these operations on the portions of the data set that resides locally on that processor. The result is a set of images, each showing part of the global domain. These partial images then need to be re-combined to form the full image. This task is performed by the workstation using the z co-ordinate (depth value) associated with each pixel in each image. This process is shown in Figure 137.

At first, this method seems to make full use of the both the large memory capacity of the parallel computer to store the original data sets and the computing power in order to create the partial images. The amount of communication between the parallel computer and the workstation is also purely dependent on the spatial and colour resolution of the final images. Since the individual images are all produced in parallel leaving the workstation very little work to do, it seems that this method is very scalable. Another advantage is that the workstation does not need to have an advanced 3D graphics capability. Any workstation, or PC, which could combine a set of images, would be suitable.

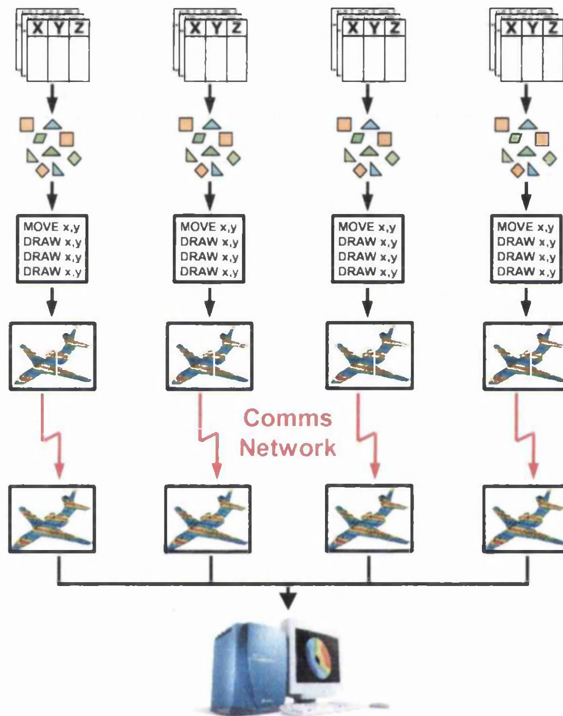


Figure 137 - The Image Data Transfer Method

However, the Image Data Transfer scenario does have a number of serious drawbacks that are all related to the network that connects the parallel computer to the workstation. As mentioned previously, the amount of data that needs to be transferred over the network is proportional to the size of the eventual image. Assuming the details of an average workstations display is:

- A resolution of 1280 * 1024 pixels,
- A minimum of 65536 colours (i.e. 16-bit colour) and
- A minimum *z*-buffer resolution for a complex model is 16 bits.

Then the amount of information required to represent one of the images is exactly 5Mb. Assuming a minimum refresh rate of 10 frames per second is needed for

real-time interaction then a continuous network bandwidth of 50Mb per second is required. With this method, each processor is generating its own image of its mesh partition that then needs to be transmitted to the workstation. Although each image will, on average, be smaller than the complete image there will invariably be a considerable overlap thus the total image data that is sent to the workstation is considerably more than 5Mb. This situation gets worse as the number of processors, and therefore partitions, increases since the amount of overlap between sub-images also increases.

It is obvious from the above figures that this bandwidth is well beyond the capabilities of any general purpose, departmental network and is not even sustainable on all but the fastest dedicated networks. However, despite these drawbacks, this method is used for the visualisation of large data sets in a number of applications whose volume data sets consist of *voxels* [Hancock97, Robb99, Sommer99, Xiao00]. However, many of these applications either suffer from a combination of limited resolution and / or limited frame rate, or require large graphics super-computers such as the Onyx^2 from SGI.

Scenario 2 – The Graphics Data Transfer method

The second approach attempts to move some of the workload from the parallel computer onto the less powerful workstation with hope of reducing the required bandwidth on the inter-connecting network. As with the previous scenario, the parallel computer performs the entire Data Reduction Process and the Rendering Process. However, this time, the plotting commands resulting from the Rendering Process are transmitted to the workstation. The workstation receives a set of plotting commands from each processor and then creates the final image by using the 3D rendering capability found in most modern workstations and PCs. This is shown in Figure 138.

Unlike the previous method, the workload is more evenly shared with each type of computer utilising its strengths. The storage and performance of the parallel computer is used to traverse the large data sets and generate the more manageable sets of plotting commands, and the rendering capabilities of the workstation are used to produce the image.

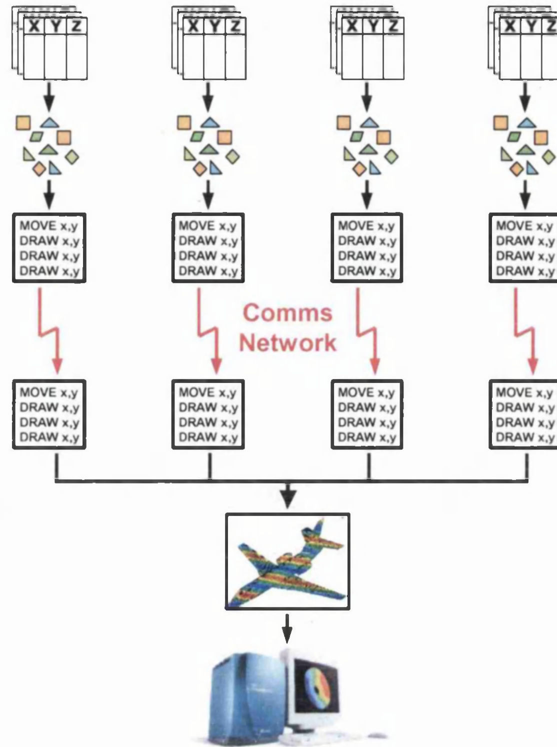


Figure 138 - The Graphics Data Transfer Method

The main disadvantage of this approach is that the network bandwidth is now dependent on the number of geometric primitives used to create the final rendered image that, for a realistic geometry and mesh, can be quite considerable. However, this problem can be solved using a combination of two methods:

1. Instead of transmitting the plotting commands to the workstation every time the image needs to be redrawn, they can be cached locally on the workstation. This means that a new set need only be transmitted when the appearance of the model is altered, e.g. drawing mode (for example, sparse to wire-frame), colour change, cutting-plane definition, etc. When the user just alters the viewpoint, through rotation, translation or zooming, then the cached data can be used to redraw the model. This reduces the necessary bandwidth in two ways; the network traffic now takes the form of occasional short bursts rather than a continuous stream, and even if the transmission takes of the order of 4-5 seconds, it is acceptable to the user since it only happens occasionally.
2. The number of geometric primitives that are needed to render the image can be reduced [Cignoni98, Hoppe98, Reinhard98]. This can occur most often with very fine mesh data sets since the number of mesh faces required to produce an accurate solution far exceeds the number needed for rendering. This approach also has the added benefit that it reduces the number of primitives that actually need to be rendered by the workstation for each frame thus improving frame-rate.

Scenario 3 – The Geometry Data Transfer method

The Geometry Data Transfer method [Haimes94, Hirsch94, Haimes97, Jones98b, Jones98c, Jones99, Jones02] is very similar to the previous method. As can be seen in Figure 139, this method continues the trend of moving more of the workload on to the workstation. Instead of the plotting commands being transmitted across the network, the actual geometric primitives are transmitted.

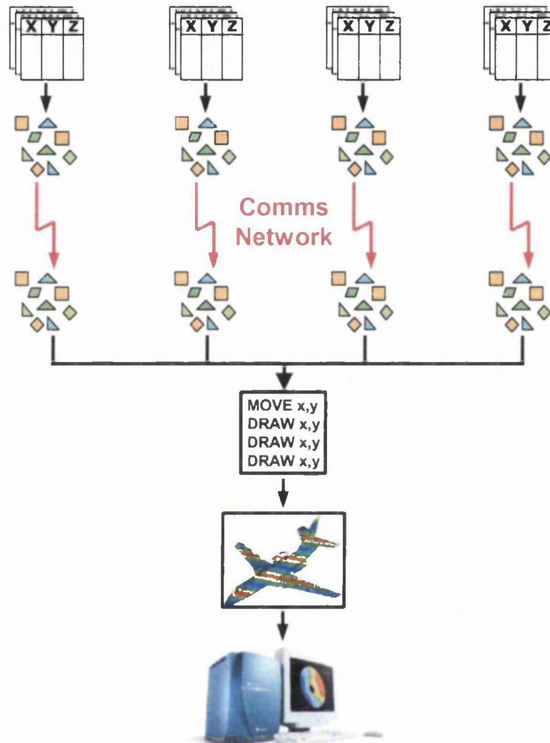


Figure 139 - The Geometry Data Transfer Method

Although, at first, this method might seem identical to the previous method, there are a number of important advantages:

1. The amount of data that needs to be transmitted for a given set of geometric primitives is reduced. For example, given the need to communicate 5 million triangle primitives, the Graphics Transfer Method requires 15 million sets of coordinates, colours and normals to be sent, each requiring three real numbers for representation. This requires over 510Mb of information.

   The Geometry Data Transfer method can transmit the triangles as tables of coordinates and connectivities leaving the workstation to traverse the data to produce the drawing commands. The colour information can also be reduced by transmitting a single index to a colour lookup table rather than the actual three components (red, green, blue) of each colour. This allows the overall data size to be reduced to approximately 123Mb. This difference is

proportional to the number of faces that are sent so can become quite considerable for a realistic model.

2. No material or drawing mode information needs to be sent from the parallel computer since this can be stored locally on the workstation and inserted when performing the Rendering Process.
3. Since only the positional and colour information of the data sets is transmitted from the parallel computer, this need only be done when significant changes are made to the visible portions of the data sets, i.e. mesh cutting plane, change of solution variable, etc.. Any lesser changes, such as changing the drawing mode from a wire-frame appearance to a lit, solid appearance can be performed locally.

This reduction in both the amount of data that needs to be transferred, and the reduction of the frequency with which the communication takes place, means that this approach lends itself very well to the process of visualising large, distributed, unstructured data sets when used in a standard working environment.

5.2. Internal Data Communication System

In order to produce an efficient implementation of the chosen visualisation strategy there is a need to make the most efficient use of the slowest component of the entire system, the inter-connecting hardware. In order to meet the original aims of the system, that is to be able to run on any type of parallel MIMD architecture, the communication mechanism must be able to cope with a configuration such as the one shown in Figure 140.

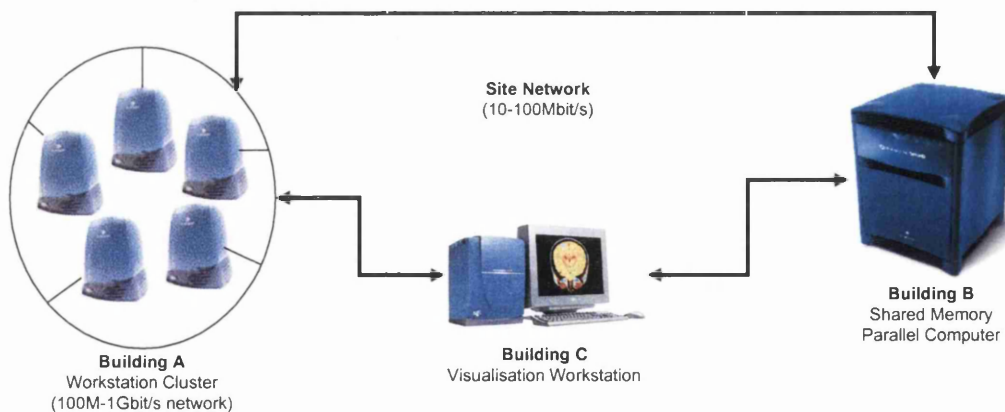


Figure 140 – Possible Hardware Scenario on which to use the PSUE II

As the figure shows, each of the processors comprising the parallel platform must be able to communicate with each other; and each processor must be able to communicate with the graphics workstation. The four most appropriate mechanisms for achieving this are:

- UNIX Socket Transfer (using the TCP / IP protocol) [Stevens90],

- UNIX Socket Transfer (using the UDP / IP protocol) [Stevens90],
- MPI (Message Passing Interface) [Dongarra95, Gropp99a, Gropp99b] and
- PVM (Parallel Virtual Machine) [Sunderam90, Beguelin94].

5.2.1. UNIX Socket Transfer (TCP / IP)

Implementing the communication architecture for the PSUE II using the TCP/IP protocol involves the initialisation of one communication channel between every pair of processes comprising the server and one channel from each of the server processes to the client process running on the graphics workstation. Each channel consumes one file handle at each end, which are used to send and receive data. The TCP/IP protocol ensures that data written to one end of a channel is received at the other end in the same order with the assumption that the communication is performed with no transmission errors. This provides an intuitive means by which inter-process communication can be achieved. However, the OS restricts the number of file handles available to a process to sometimes as little as 30. If, for example, we need eight for file I/O operations (including the standard three channels for console input, output and error reporting), this limits the number of processors that can communicate to 22, including the graphics workstation. Since the PSUE II is designed for very large, parallel problems, it was decided that limiting the size of the parallel platform to 20-30 processors was too restrictive.

5.2.2. UNIX Socket Transfer (UDP / IP)

Another standard UNIX communications protocol is UDP/IP. This is referred to as a *connectionless* protocol since there are no longer any defined channels between processors. Instead, the data stream to be transmitted needs to be broken down into small fragments (or *packets*) of about 500 bytes. These are then transmitted to the required destination where they are reassembled into the original stream. The UDP/IP protocol eliminates the file handle restriction since it requires only one per process regardless of the number of processes. However, it is an unreliable protocol. This means that there is no guarantee that any transmitted packets are received and there is no guarantee as to the order of the received packets. This unreliability would impose a heavy burden on the communication algorithms within the PSUE II since they would need to detect and automatically re-send lost packets and then ensure they are in the correct order. This was deemed too difficult a task to implement efficiently and robustly and, since it would form the core of the entire environment, any software bugs would be unacceptable.

5.2.3. MPI (Message Passing Interface)

To hide these, and many other complexities inherent in the use of native UNIX socket communication, a number of libraries have been developed, most of which are freely available in the public domain. The two most commonly used libraries for parallel architectures are MPI (Message Passing Interface) and PVM (Parallel Virtual Machine).

As well as providing an easy-to-use means of inter-process communication, they can also utilise any platform specific features that enable increased performance. This includes using protocols specially designed for high-speed networks (Myrinet, Gigabit, ATM, FDDI, etc.), using blocks of shared memory as a communication means on shared

memory parallel platforms or using native communication hardware on distributed MPP's (Massively Parallel Platforms).

Unfortunately, the MPI-1 standard did not contain any capabilities to change the parallel configuration dynamically at run-time. This is necessary in order for the PSUE II to execute and connect to both the visualisation server processes, and any external parallel modules, such as mesh generators or equation solvers.

5.2.4. PVM (Parallel Virtual Machine)

The second parallel communication library that was investigated was PVM. This library includes all of the benefits of MPI, but also includes the abilities to dynamically execute and connect to processes and, in the case of workstation clusters, add extra computers into the parallel configuration.

5.3. The Control Structure of the Environment

It is clear from its interactive nature that, unlike many parallel computationally intensive codes, the communication infrastructure of a Parallel Problem Solving Environment is used for more than pure data transfer. It is also used as a means of controlling the timing and order of operations during its execution.

As with any Problem Solving Environment, the control of the environment originates from the user through interaction with the keyboard and mouse. These inputs are passed directly to the master process running on the graphical workstation. For simple operations, in which the set of rendering primitives stays constant, the master process performs all of the necessary rendering operations with no interaction necessary with the parallel slave processes. Operations such as moving or rotating the object on the screen, or changing whether the primitives are drawn as wire-frame outlines or solid facets fall into this category.

However for more complex operations, which involve the creation of new rendering primitives from the distributed volume data sets, the master needs to be able to *instruct* the slaves to perform the required operations and then receive the new set of primitives produced as a result of those operations. A simple example of this is a user requesting a solution file to be loaded and overlaid on the mesh.

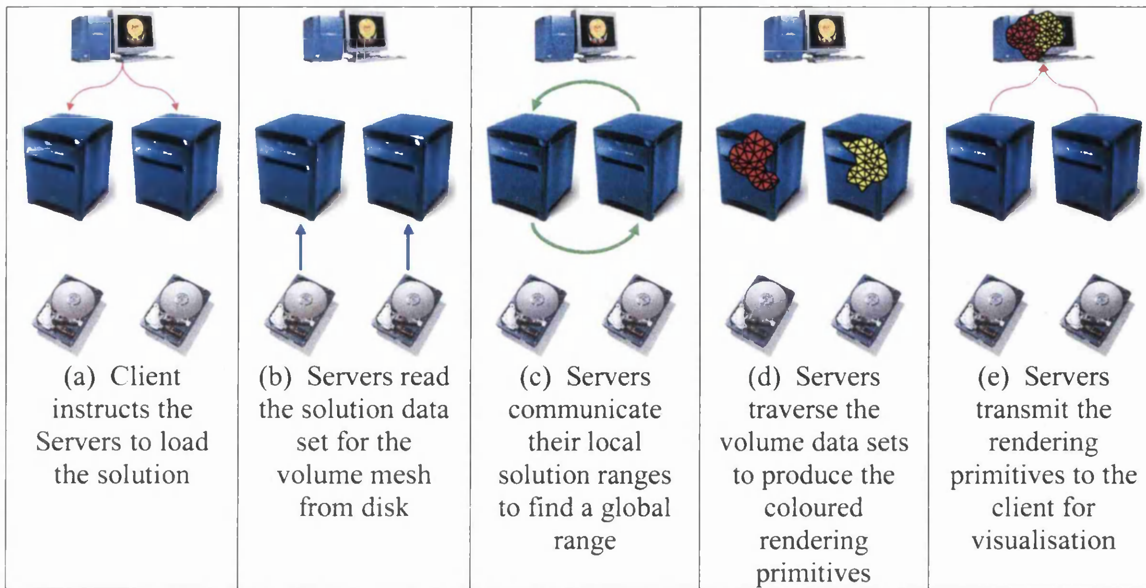


Figure 141 – The Sequence of Operations required to load a Solution File

As can be seen in Figure 141, the sequence of operations needed to perform this operation comprises five main steps.

The Client instructs the Servers to Load the Solution File

Whilst the server processes are idle they wait to receive an instruction from the client via PVM. This instruction is purely an integer constant followed by any relevant data (in this case a filename).

When the user selects the required solution file, the client sends each server process the appropriate instruction code followed by the filename. The client then becomes idle whilst waiting for a response from the servers.

The Servers load the Solution File

When the server processes receive the instruction code and filename they independently load their solution data sets and compute their local minimum and maximum solution values.

The Servers communicate with each other

In order to construct the colourful rendering primitives representing the solution values on the mesh, it is necessary for a global minimum and maximum to be computed for each variable. The server processes achieve this by communicating with each other to perform a global reduction operation.

The Servers construct the Rendering Primitives

Once the global minimum and maximum values have been computed, each server processes then traverses the volume data sets to construct the rendering primitives

required for visualisation. The solution values are mapped to a colour scale as shown in Figure 142. This operation is performed independently with no communication.

Figure 142 – Mapping of Solution Values to Colours

The Servers return the Rendering Primitives to the Client

When the rendering primitives have been constructed, they are then sent back to the client process. The client process, which has been idle waiting for this data, now receives the data from each server in turn and returns control back to the user.

5.3.1. The Two Types of Communication

Without clouding the scenario with any algorithmic details, this sequence of operations clearly shows the two types of communication used within the PSUE II; *control-flow* and *data-flow*. Step 1 forms the initiation of a control-flow in which an instruction token is passed from the client to each of the servers along with the specified filename. This type of communication is analogous to a subroutine call in a sequential process with the filename being passed as an *in* argument.

Step 3 forms a data-flow in which each slave needs to know the global minimum and maximum of the solution variable before any colour coding of the rendering primitives can be performed. However, only local minima and maxima can be calculated from the list of variable values in each solution file. The global range can only be found by communicating the local ranges between the servers. Here, no control information is passed, only required data. This is analogous to copying data items within a sequential process and is the type of communication used in most batch parallel processes such as equation solvers.

Step 5 represents the culmination of the control-flow initiated in step 1. Although data is transmitted from the servers to the client, it is analogous to the return value, or *out* argument, of a subroutine call in a sequential process and therefore is deemed a control-flow operation.

5.4. Summary

This chapter has described the means by which the visualisation of the very large data sets is achieved, and the technologies used to perform the communication between the various processes comprising the initial version of the PSUE II. Chapter 6 details the improvements made to the design and implementation in order to produce a more

flexible, parallel environment capable of handling data size an order of magnitude greater than was possible with the initial implementation.

Chapter 6. PSUE II v2.0 – An Improved Architecture

The previous chapter briefly described an implementation strategy of the PSUE II that used the PVM communications library throughout the system. Whilst this was being implemented, a number of improvements to the strategy became apparent:

- *Improving Flexibility of use*
  This involved adding the ability to connect and disconnect from parallel processes at any time during their execution even if they were implemented with a communication library other than PVM, e.g. periodically monitoring the status of solvers during their execution.
- *Reducing Network Bandwidth*
  A more intelligent strategy for sending the geometric primitives from the slaves to the master process was needed. In the initial implementation, if an operation, such as a defining a cutting plane, were performed then all of the geometric primitives would be sent from the slaves to the master process. This is obviously not the ideal situation since the master would already have many of these primitives. A more complex data management scheme would be needed to ensure that only the primitives that had changed would be transmitted.
- *Re-partitioning the mesh.*
  The previous strategy used the partitions as generated by the parallel mesh generator. This had two disadvantages; the partitions were not necessarily very well balanced, and it restricted the number of slave processes to the number of partitions that were originally generated. These two disadvantages would be overcome by re-partitioning the mesh to the required number of slaves at run-time.
- *Improving the performance of traversing the volume data sets*
  In order to be able to interactively manipulate data sets consisting of hundreds of millions of elements the ability to geometrically search for elements had to be improved from a linear search as in the previous strategy.

The sections in this chapter describe how each of the above improvements was achieved.

6.1. Improving Flexibility of Use

Since the PSUE II is designed for very large problems requiring large parallel computers, it is inevitable that the compute-intensive portions of the simulation process (i.e. the mesh generators, equation solvers, etc.) will take a significant amount of time to execute. The solvers, in particular, can often run for many days.

The traditional way of running these codes would be to run them on a remote parallel computer and then log-out and leave them, occasionally logging in to check on their progress. If the PSUE II is going to be used in this kind of environment then it needs to assist the user with this process. Ideally the PSUE II would allow the user to execute a

solver on a remote computer and then perform the periodic checking of its progress automatically, leaving the user to perform other tasks either within the PSUE II or not.

In order for the PSUE II to be able to do this, it must have the ability to disconnect from a set of processes and then re-connect at will. In fact, this ability should be extendable to the execution of many solvers on many different computers.

With the exclusive use of PVM as the means of connecting to computers, spawning and terminating processes, and inter-process communication, this kind of functionality is not feasible.

The use of PVM has another unwanted side effect when trying to spawn 3rd party applications that use another communication library since most communication libraries used for parallel computing require their own method of starting the processes. For example, PVM requires that slaves be started using the subroutine '`PVM_Spawn`', whereas MPI requires that all processes be started using the command '`mpirun`'. These two requirements are incompatible.

In order to alleviate all of these problems a communication system must be used that is both independent of the method used to start the processes, and allows disconnection and reconnection at will regardless of how the processes were initiated.

Since none of the communication libraries intended for parallel computing possess the required flexibility, there was a need to look outside the parallel computing community, where performance is paramount, and look in the distributed computing community where this kind of flexibility is often required.

Although there are a number of groups researching into the problem of distributed computing, if you require a system that is flexible and robust there are only three major contenders: DCOM [Microsoft95, Microsoft97, Brockschmidt95], JAVA/RMI [Daconta96] and CORBA [Schmidt95a, Schmidt95b, Yang96, Vinoski97, Henning99, OOC99, Schmidt99].

DCOM is unsuitable since it is only implemented within Microsoft Windows and JAVA/RMI is unsuitable because it can only be used with software written in JAVA. CORBA was chosen because it is platform independent; language independent and a number of robust implementations are freely available on many platforms for non-commercial use.

6.1.1. The CORBA Architecture

The Common Object Request Broker Architecture (CORBA) is a very flexible means of allowing distributed objects to be integrated in a co-operative manner.

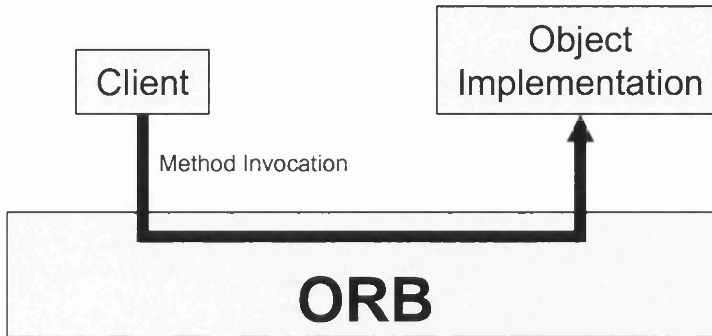


Figure 143 – Request being sent via CORBA

Figure 143 shows a request being sent by a client to an object implementation, both of which may reside in separate processes on geographically disparate computers. The client is the entity that wishes the object to perform an operation and the object implementation is the entity containing the code and data that actually performs that operation. A simple analogy is a subroutine call in a simple program. Here, the client is the part of the program performing the subroutine call and passing arguments, and the server is the actual subroutine that receives the arguments and performs the actual operation.

In order for these two steps (the subroutine call and the subroutine) to be located on separate computers in a transparent manner, there is a need for a system by which the arguments in the subroutine call are transmitted to the process containing the subroutine automatically. The ORB (Object Request Broker) performs this task by managing all of the communication mechanisms that:

- Find the object implementation (*subroutine*) to which the request (*subroutine call and arguments)* should be sent,
- Package up and send the request,
- Prepare the object implementation to receive the request and
- Ensure the object implementation performs the operations associated with that request.

The interface the client sees is independent of the location of the object, the language the object is written in, the computer architecture on which the object is run or any other details not specifically linked with the definition of the interface.

This is achieved using an Interface Definition Language (IDL). The interface for each object is defined in an object-oriented manner using the IDL. An IDL compiler is then used to create the source code in the required language for both the client and the object implementation. This is shown in Figure 144.

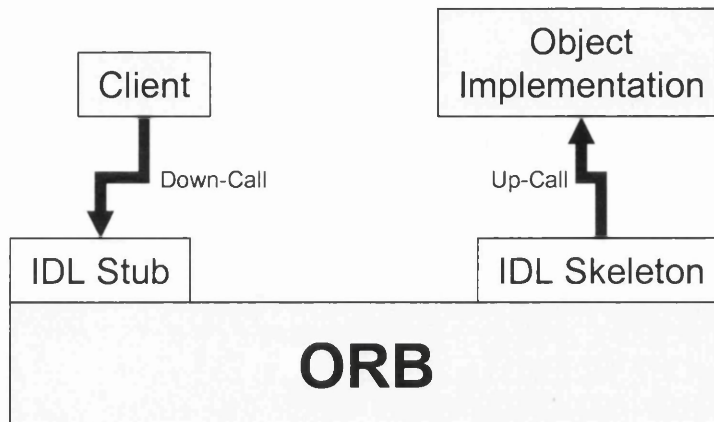


Figure 144 – The Structure of the Request Broker Interfaces

Here, the client requests an object implementation to perform an operation. This request is passed through an 'IDL stub', as a down-call, to the ORB. The ORB then locates the required object implementation and transfers the request. The request is then received by the 'IDL skeleton', which then calls the appropriate method in the object implementation as an up-call.

As a concrete C++ example, a very simple Database object could have an interface definition as follows:

```
class Database
{
private:
        Some private data here

public:
        void add_details( const string name, const int age );
        boolean find_age( const string name, int& age );
        boolean remove_details( const string name );
};
```

This defines the interface to a class that has three simple methods:

- *add_details* – Add a name and age to the database
- *find_age* – Return the age of a person with the given name in the argument *age* and returns the success of the operation.
- *remove_details* – Remove the given name from the database and return the success of the operation.

Implementing this object as a distributed object using CORBA involves three key steps:

Defining the Object Interface using IDL

In order for the client and the database object implementation to be able to communicate, the interface to the object needs to be defined using the IDL. An example interface for the Database object is shown below.

```
interface Database
{
    void add_details( in string name, in int age );
    boolean find_age( in string name, out int age );
    boolean remove_details( in string name );
};
```

This definition is very similar to the C++ definition but is actually independent of the language. The IDL definition would identical whether the target code was C, Java, Python, etc.

Compiling the IDL

Once the interface has been defined it should be compiled using the IDL compiler. This produces a 'stub' file and a 'skeleton' file in the required language. The 'stub' file is linked into the client executable and is responsible for packing and sending the request to the ORB. The 'skeleton' is linked with the object implementation and is responsible for receiving the request from the ORB and then calling the appropriate method in the object implementation.

In the case of the C++ example, the 'stub' file defines a class with exactly the same interface as the original C++ Database object. The methods in this class package the subroutine arguments and send them to the ORB. The 'skeleton' file implements a simple routine that waits for requests from the ORB and then calls the appropriate method in the Database object.

Performing a method on the Database Object

Once the client and the object implementation have been compiled, a client can invoke a method on the Database object using the following code snippets:

```
pMyDatabase->add_details( "A N Other", 45 );
success = pMyDatabase->find_age( "A N Other", age );
success = pMyDatabase->remove_details( "A N Other" );
```

As these snippets show, the location of the Database object is transparent to the client since if the Database object were a local C++ object, the syntax of the methods calls would be identical.

Figure 145 illustrates the entire sequence.

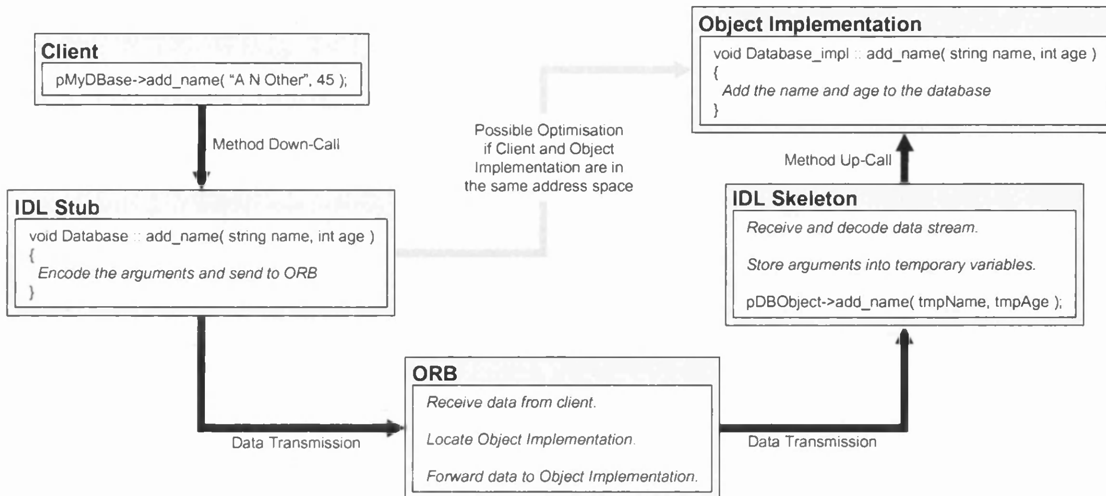


Figure 145 – Steps performed for a Method Invocation (using C++)

Although the logical model of CORBA is standardised, the way in which it is implemented is not. The ORB may be implemented as a library linked into the user's code, as a daemon running in the background on each computer or even as part of the service provided by the operating system. This is transparent to the application.

Conceptually, the code generated by the IDL compiler always packages up method invocations as streams of data and then transmits them across the network to the computer on which the object implementation is running. In reality, many implementations optimise this process by bypassing the encoding, transmission and decoding process if the object to which the client refers is located in the same program as the client. In this case, the method invocation is just passed to the object implementation as another method invocation with minimal overhead. These optimisations are also transparent to the application.

6.1.2. The Use of CORBA within the PSUE II v2.0

Due to its flexibility, the architecture of the PSUE II was modified to use CORBA as the communication link between the client and the server processes. As well as the obvious advantage that server and client processes could be connected and disconnected at will, it also brought the advantage that the communication model used by CORBA matched the communication model already used by the client-server link.

When PVM was being utilised, the method invocation model was emulated by encoding a function call as a unique integer followed by any supplied argument data. This was then sent via PVM to the server, which decoded the message and called the appropriate function with the supplied arguments. The same process was then repeated for any return values. With CORBA, the IDL compiler automatically generates the code that performs the encoding, transmission and decoding process in a transparent manner thus eliminating any coding errors.

However, the communication amongst the server processes was still performed by the PVM library. This was because:

- The PVM library matched the communication model of the server processes, i.e. message passing. This could be emulated in CORBA by passing data as arguments in a method invocation but this would produce unnecessarily complex code requiring the use of multi-threading to avoid any deadlocks.
- The PVM library was more performance oriented than most CORBA implementations.

As mentioned previously, the PVM library has specialised implementations that take advantage of any communication hardware available on a parallel computer to improve performance still further.

The final architecture of the PSUE II is shown in Figure 146.

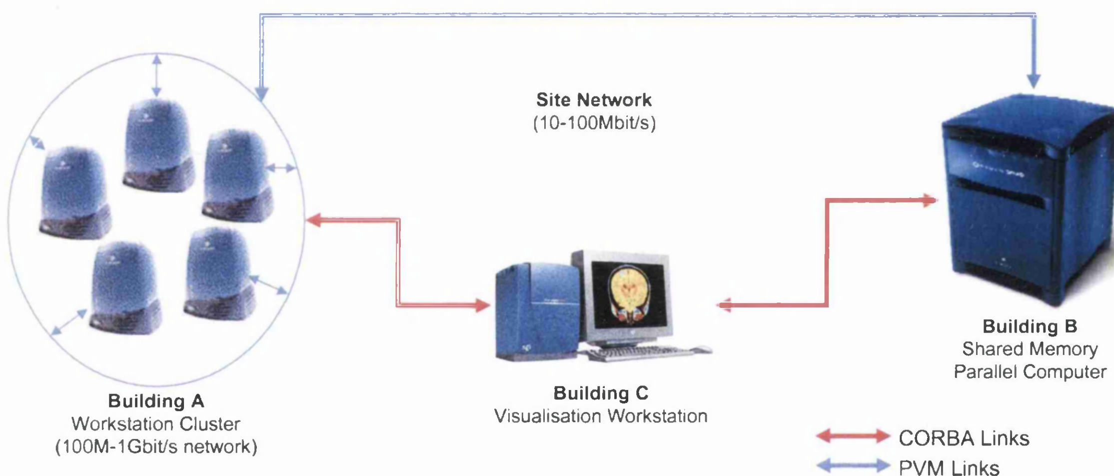


Figure 146 – The Final, CORBA-based Architecture of the PSUE II

6.2. Reducing Network Communication

The control structure of the PSUE II v1.0 has already been described in the previous section. The amount of data transferred from the client to the server processes as a control instruction is minimal, about 400-500 bytes, and so the time to communicate that data is insignificant. However, the amount of data returned as the result of that instruction, in the form of rendering primitives, could be quite considerable ranging from hundreds of kilobytes to many megabytes. For this size of data the transfer time can have a significant impact on the overall response of the environment. To minimise this impact, it is necessary to reduce this amount of data, and therefore the number of primitives that are returned.

A typical operation often performed on a finite element mesh is the cutting plane. This allows the user to examine the interior of a volume mesh for various features depending on the type of cutting plane. For investigating geometric features of the mesh elements, a rough cutting plane can be produced by selecting the set of elements that are wholly on one side of the plane. The others are then removed thus producing a jagged finish as shown in Figure 147. The second type of cutting plane is a perfectly smooth plane and is commonly used for investigating features of the solution such as shock waves or vortices. This is produced by actually intersecting the volume elements with the plane and producing a set of primitives that form a flat surface. This is shown in Figure 148 and forms the test case scenario for the rest of this section.

Figure 147 – A Rough Cutting Plane

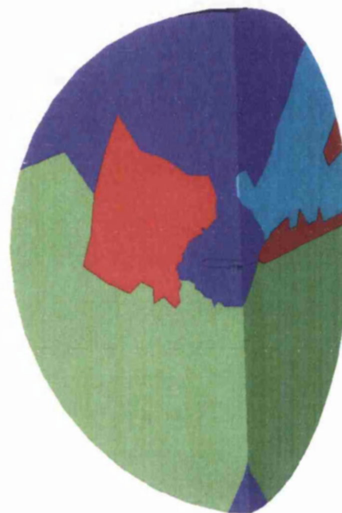


Figure 148 – A Smooth Cutting Plane

The method used by the PSUE II v1.0 to generate a smooth cutting plane was to treat both the volume data sets and the geometric primitives comprising the surfaces as simple, global, flat data structures. This meant that when a cutting plane was defined then the entire volume data set had to be searched to find which sets of elements were positioned on the correct side of the plane and which were to be discarded. After that, the set of geometric primitives that would form the surface of the remaining volume mesh had to be created after discarding the set of primitives present before the cutting plane was defined. The final step then entailed transmitting this new set of primitives across the network to the workstation to replace the previous set. This procedure is detailed below:

```
DESTROY rendering_primitive_list
FOR each element, e
    IF e intersects cutting plane THEN
        rendering_primitive = triangle/quad representing intersection
        ADD rendering_primitive to rendering_primitive_list
    ENDIF
ENDFOR
FOR each boundary face, f
```

```
        IF f is to the left of the cutting plane THEN
              ADD f to rendering_primitive_list
        ELSE IF f intersects cutting plane THEN
              rendering_primitive = triangle/quad representing intersection
              ADD rendering_primitive to rendering_primitive_list
        ENDIF
ENDFOR
TRANSMIT rendering_primitive_list to client for display
```

It is obvious that this procedure is rather naïve in that it does not make any use of the previous set of geometric primitives. This results in a large number of primitives being transmitted across the network that are exact duplicates of primitives already present.

An obvious improvement over the previous algorithm would be to only send the primitives that have actually been altered since the last set were generated. Figure 149 shows a coarse two-dimensional mesh through which a cutting plane has been defined. The primitives comprising this mesh have been coloured according to whether they remain unchanged (green); have been replaced with new primitives (red) or have been removed (blue).

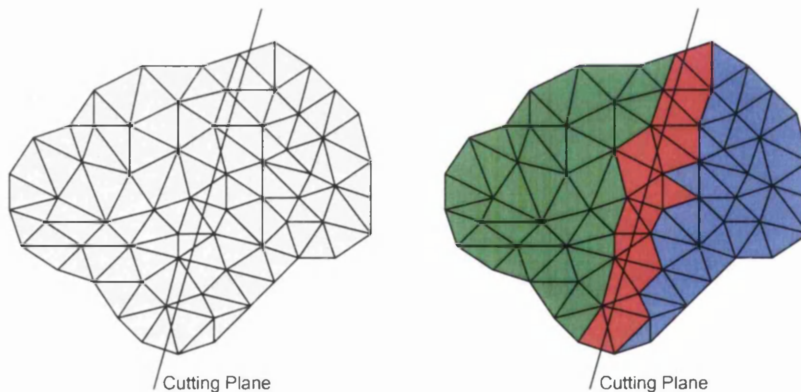


Figure 149 – The Geometric Primitives affected by a Cutting Plane

Now, instead of the entire set of primitives being sent back to the workstation as one long message the data is split into two sections. The red primitives, that need to have their details transmitted to the workstation, and the blue primitives for which flags need to be transmitted to the workstation in order for them to be removed. Obviously, in a three-dimensional case, the primitives making up the actual cutting plane are new and are thus sent to the workstation for rendering.

For a CFD mesh around an F16 Fighter Jet, consisting of 6.7 million elements, 1.1 million nodes and 0.3 million boundary faces, the sizes of the various data sets that need to be transmitted are shown in the table below. The position of the cutting plane (Figure 150) is across the aircraft wing roughly splitting the domain into two halves.

	Original Scenario	Improved Scenario
Nodes	2480298	90990
Triangles	1133778	46913
Primitive Deletion Flags	0	3204367
Total Data Size	14.12Mb	3.75Mb

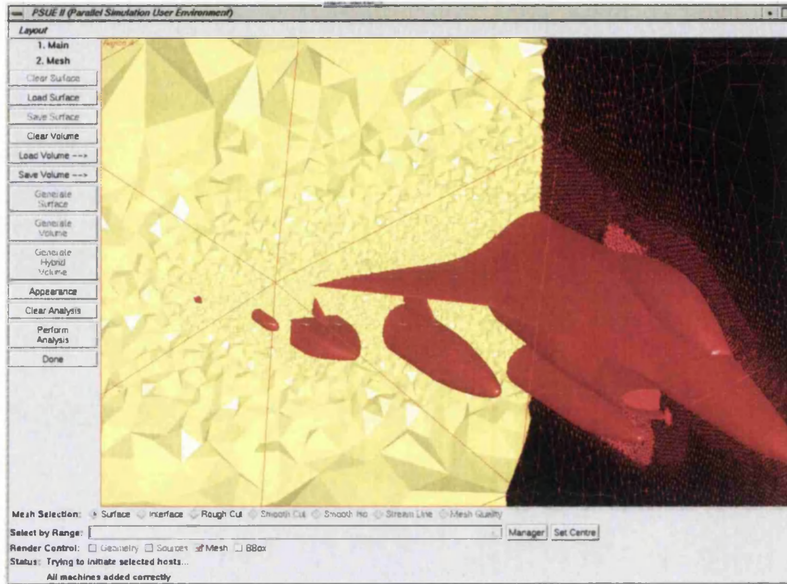


Figure 150 – Position of the Cutting Plane during for Collection of Statistics

As can be seen from the table, the actual amount of information that needs to be passed across the network has been significantly reduced by the second procedure, but the number of deletion flags is still quite considerable. If the time taken to traverse the array of primitives on the master process and remove the flagged primitives is taken into account then further improvements should be possible.

A good compromise between the two extremes of re-transmitting all of the primitives and transmitting a large array of primitive deletion flags is to use a modification of the second procedure. Here, the geometric primitives are combined into groups that have either all of their data re-transmitted if any member has been altered or all flagged for removal if all of their members are to be removed. One simple method is to group the primitives using a combination of the type of entity from which they are created and the identifier of that type. Examples of these may be geometric surface numbers, cutting plane numbers, etc. These groups can then be further sub-divided by treating the pair [*partition number, group number*] as a group.

If the same fighter jet example is used partitioned into 16 partitions, the number of groups becomes 8416. This comprises 16 groups for the cutting plane (1 for each sub-domain) and 8400 groups for the geometry surfaces sub-divided by the 16 partitions (i.e. 525 * 16 = 8400). Obviously some of the groups will be empty since not all sub-domains will

contain a section of every surface and the cutting plane. The statistics for the sizes of the data sets that need to be transmitted using this structure is shown in the table below.

	Final Scenario
Nodes	144867
Triangles	70913
Group Deletion Flags	2045
Total Data Size	1.00Mb

It is obvious from the table that the total amount of data that needs to be transmitted to the workstation has been reduced significantly, in fact from the first scenario to the last the amount of data that needs to be transmitted has been reduced by a factor of 14. The time for searching through the existing primitives to see which have been altered has also been reduced since the cutting plane can be compared to the bounding box cached for each group thus eliminating any unnecessary intersection tests between the individual primitives and the cutting plane.

This approach is more efficient, in terms of performance and network usage. In order to maximise these gains a more complex data management scheme is required to ensure both consistency between the geometric primitives on the workstation and the parallel server, and between the primitives on the server and the original volume mesh data sets.

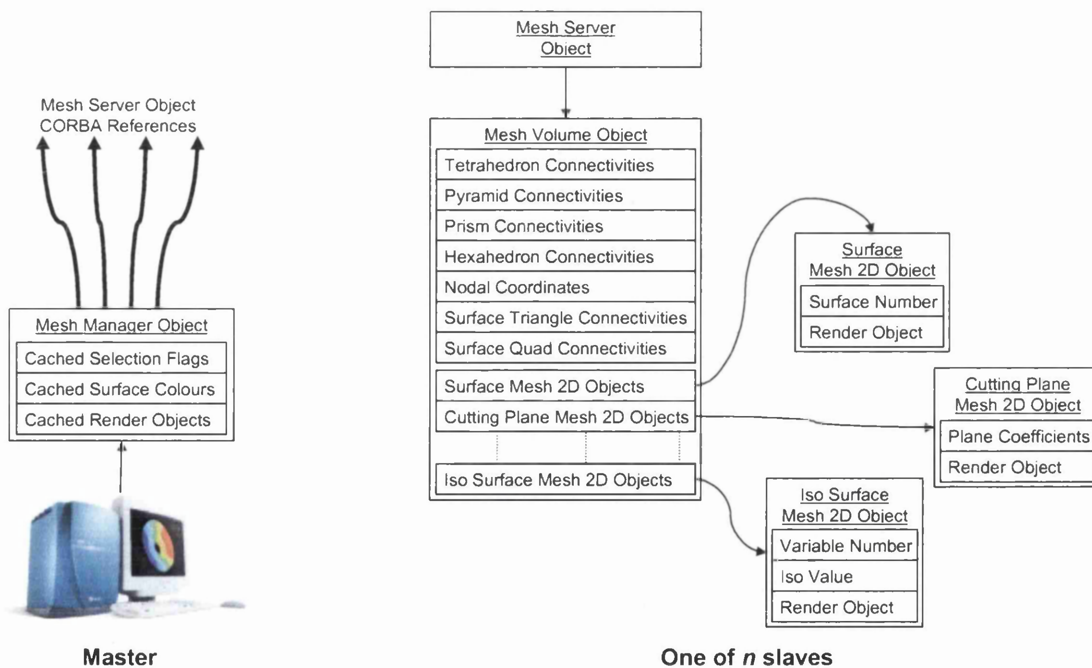


Figure 151 – The Hierarchy of Classes used to manage the Mesh Data Sets within PSUE II v2.0

6.2.1. The Mesh Management Class Hierarchy

To achieve this an object-oriented structure was developed consisting of a hierarchy of classes, with each object taking sole responsibility to ensure the consistency of the data stored both within the class itself and the objects that form its children. An overview of the hierarchy of classes for the management of the mesh data is shown in Figure 151.

6.2.2. Mesh Manager Object

The Mesh Manager object is responsible for the Master side of the Master-Slave CORBA link. Its purpose is to hide all of the complexities of interacting with the multiple mesh partitions distributed across many computers and maintain consistency between the information stored on the workstation and that stored on the parallel server. The external interface allows the rest of the routines within the master that interact with the user to ignore this distribution and treat the data sets as if they were combined into one partition and stored locally on the workstation.

Internally, each method call is converted into a number of CORBA method invocations that are then transmitted to each of the slave processes. Since calling a method on a CORBA object is analogous to calling a method on a local C++ object, the Mesh Manager initiates a multi-threaded environment in which each thread invokes a method on a slave and then waits for that method to finish. If the Mesh Manager was single-threaded each method would have to wait for its predecessor to complete. This would mean, at the least, the slaves performed their operations in a sequential manner thus reducing performance and, more probably, if the slaves needed to communicate with each other in order to complete their task, a deadlock would occur.

The last task performed by the Mesh Manager object is to cache small amounts of commonly used data kept on the slaves. This information includes details such as the number of each type of entity (e.g. surfaces, cutting planes) stored on each slave, which of them has been highlighted by the user, etc. This eliminates the need for frequently transferring very small packets of data across the network.

6.2.3. Mesh Server Object

The Mesh Server object forms the slave side of the CORBA link. Although this class has a large number of methods that can be invoked via CORBA, it is actually a very simple class. It essentially provides a PSUE II specific interface to the more general-purpose Mesh Volume object (described next). Most of the methods have direct one-one correspondences with methods in the Mesh Volume object. These methods simply invoke their respective methods in the Mesh Volume object passing in any necessary *in* parameters and passing back any *out* parameters whilst performing any necessary data conversions between the PSUE II data structures and the CORBA equivalents.

The remaining methods perform operations that are deemed too specific to one particular program to be included in the Mesh Volume object. These methods include the I/O routines that store and retrieve the mesh partitions in the various formats specific to the PSUE II.

6.2.4. Mesh Volume Object

The Mesh Volume object is probably the most important object within a slave process since it stores all of the data-sets associated with a given mesh partition. Examples of these data sets include the nodal coordinates, volume element connectivities, boundary element connectivities, inter-partition communication data and solution data.

It also performs all of the operations pertaining to these data sets including all geometric searching and analysis of the mesh partition. It calculates and stores a large number of extra data structures that are used by the various methods within the object. Common examples of these data structures include:

- *Element-based data structures*
  Element→Node and Element→Element connectivities.
- *Face-based data structures*
  Face→Left/Right Element + Node.
- *Edge-based data structures*
  Edge→Node and Edge→Element.
- *Node-based data structures*
  Elements around a node, Faces around a node, Edges around a node and Nodes connected to a node.

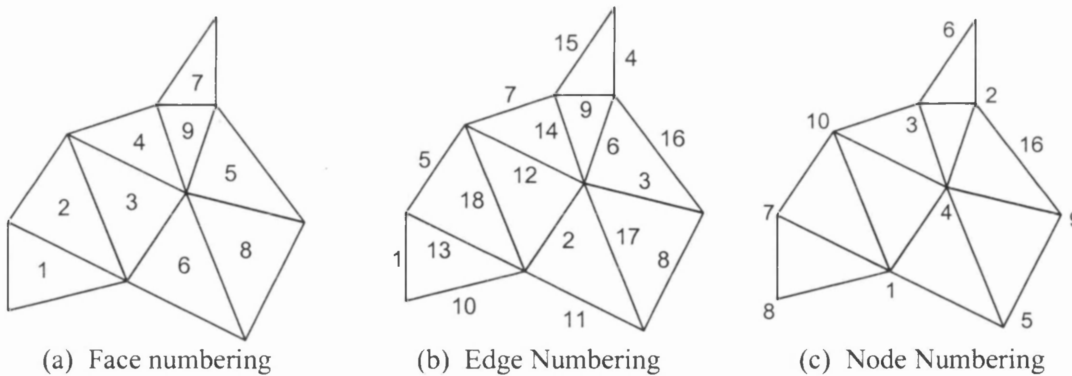


Figure 152 – Face, Edge and Node Numbering for a simple 2D Mesh

Figure 152 shows a small 2D mesh with the face, edge and node numbering and the following tables show examples of Element (Face)→Node and Face(Edge)→Nodes + Left/Right Element(Face).

Element(Face)→Node Connectivities				
1 {7, 1, 8}	2 {10, 1, 7}	3 {4, 1, 10}	4 {10, 3, 4}	5 {4, 2, 9}
6 {4, 5, 1}	7 {2, 3, 6}	8 {5, 4, 9}	9 {3, 2, 4}	

Face(Edge)→Left/Right Element(Face) + Node Connectivities				
1 {7, 8, 1, -1}	2 {1, 4, 3, 6 }	3 {4, 9, 5, 8}	4 {2, 6, 7, -1}	5 {7, 10, -1, 2 }
6 {4, 2, 9, 5}	7 {10, 3, -1, 4}	8 {5, 9, 8, -1}	9 {3, 2, 7, 9}	10 {8, 1, 1, -1}

11 {1, 5, 6, -1}	12 {4, 10, 3, 4}	13 {1, 7, 1, 2}	14 {3, 4, 9, 4}	15 {3, 6, -1, 7}
16 {2, 9, -1, 5 }	17 {4, 5, 8, 6}	18 {1, 10, 2, 3}		

For three-dimensional meshes, these data structures can all consume significant amounts of memory, especially for large test cases, and can also take a significant time to compute. Therefore, the Mesh Volume object caches these data structures in an intelligent manner releasing them only when the memory is required for another data structure.

The last task for the Mesh Volume object is to create, store and maintain the various 2D Mesh objects that, later, form the rendering data for the master. These are stored in a number of lists, one for each type of entity supported by the PSUE II. This forms the means by which the various 2D Mesh objects can obtain the data they require in order to ensure they are consistent with the volume data sets and each other.

6.2.5. 2D Mesh Object

A 2D Mesh object stores and maintains all of the information representing a given entity within the slave processes. These include mesh surfaces, cutting planes, iso-surfaces, etc. Each type of 2D Mesh object is specialised to perform the operations necessary for the given type of entity. For example, the 2D Mesh object representing a cutting plane includes the algorithms that can scan the volume data sets in order to produce the set of elements that form the cutting plane surface. Although each type is different, they all inherit a basic functionality from the same set of classes (described in the next section). This allows the Mesh Volume object, for the most part, to be able to treat them as the same object since they share a large number of their methods. This has the advantage of reducing unnecessary code duplication. Essentially, the set of 2D Mesh objects are the key to eliminating any of the unnecessary transmission of data sets that have not been altered since the last transmission.

6.2.6. Render Object

The Render object stores and maintains all of the geometric primitives and associated attributes that are necessary to render the object. There is a one-one correspondence between Render objects and 2D Mesh objects since each 2D Mesh object contains an instance of a Render object. Although the Render object stores data representing the same entity as the 2D Mesh object it is stored in a manner that allows efficient rendering rather than efficient searching and processing. This allows the structure of the two objects to be altered over time without having to be concerned that a change to increase the performance of processing the object might detract from its rendering performance. It is also the only object described here that has a duplicate in the master process running on the workstation and, thus, forms the means by which the rendering data is passed from the slave processes to the master.

6.2.7. Co-operation between the Objects

To illustrate how all of these objects fit together during a typical operation, the steps performed during a cutting plane definition will be described. As with most of the

operations performed within the PSUE II, defining a cutting plane is split into two main steps, the definition of the plane and then the updating of the Render Objects.

The operations performed in the first step are listed below:

1. The user interacts with the model on the display to define the position of the cutting plane.
2. The Mesh Manager Object then creates the appropriate number of threads and passes, via CORBA, the coefficients of the plane to each of the slave processes concurrently.
3. The Mesh Server Object in each slave simply passes these coefficients onto the Mesh Volume Object without performing any operations on them.
4. The Mesh Volume object checks to see if there is any Cutting Plane 2D Mesh Object already defined that matches these coefficients. If there is, then an error condition is returned. If there is no matching plane then a new Cutting Plane 2D Mesh Object is created and added to the list.
5. The newly created object is then passed the relevant volume data sets in order to produce the set of faces that represent the cutting plane.
6. The plane coefficients are then passed to all existing Surface 2D Mesh Objects. These objects then determine whether the cutting plane has affected their appearance. If not, then they return without changing anything. If the plane cuts them then their data sets are altered to reflect this.

At the end of this step, all of the data sets have been created and the only remaining operations to perform are to pass any new or recently changed Render Objects back to the Master. This is achieved in the second step:

1. The threads in the Mesh Manager Object created during the previous step request the new set of Render Objects back from the slave processes.
2. This request is passed from the Mesh Server Objects in each slave to the Mesh Volume Objects repeatedly until there are no more Render Objects to send back.
3. For each request, the Mesh Volume Object asks each of its 2D Mesh Objects, one by one, if they have been altered since the last request. The objects that have been changed then update their Render Objects to reflect these changes and pass them back to the Mesh Volume Object.
4. These Render Objects are then passed back to the Mesh Server object, which stores them in a list. When all Render Objects have been asked the entire list is then passed back, via CORBA, to the threads in the Mesh Manager object.
5. The Mesh Manager then collates all of the Render Objects from the various threads and terminates the threads. Any future requests to the Mesh Manager Object to draw the Render Objects uses the updated sets.

6.3. Improving Load Balancing

The previous implementation strategy for PSUE II was based around the use of the mesh partitions as created directly by the in-house parallel Delaunay mesh generator [Weatherill00b, Larwood01]. Although this was a perfectly valid approach, there were a number of disadvantages:

- The mesh partitions produced by the parallel generator were invariably not balanced in terms of the number of elements or the inter-partition communication.
- The number of slave processes used within the PSUE II was fixed to the number of partitions chosen when the mesh was generated. This obviously was not ideal since the mesh may have been generated on a large parallel computer in the past using 64 partitions, but the only computing hardware available at the current time might be a cluster of 4 workstations.
- If the environment were used on a mesh generated sequentially then the environment would operate sequentially due to their being only one partition (the global mesh).

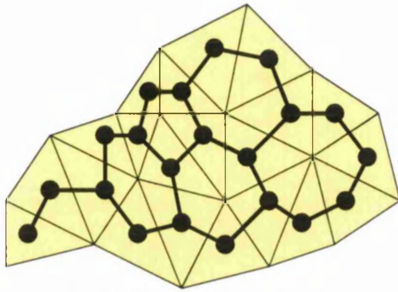
In order to overcome these deficiencies, the parallel mesh generator was modified to recombine the partitions into one global mesh during the output phase and a mesh partitioning algorithm was implemented that would allow the reading and partitioning of a global mesh into the required number of partitions at run-time.

There are a number of different approaches available for serially decomposing a given unstructured mesh. However, for the purposes of the PSUE II it was envisaged that the mesh data sets would be too large to load onto one processor. Therefore, the partitioning process had to be parallelised and distributed amongst the processors at all times. The implementation utilised the ParMetis [Karypis98, Karypis02] library for the partitioning. This library produces high quality partitions in a fast, robust and parallel manner.

However, ParMetis operates exclusively on a graph data structure, which means that the mesh had to be represented as a set of edges. One method of doing this was to create the dual of the mesh, where the nodes represent the elements and the edge (ε_1, ε_2) is present if the two elements ε_1 and ε_2 are adjacent (Figure 153a). An alternative method was to create an edge-based representation of the original element edges (Figure 153b).

Using the dual of the mesh has the advantage of automatically producing an element based partitioning, whereas the edge-based representation of the mesh produces elements that are split across partitions. However, in the edge-based representation, the number of edges is approximately the same as the number of elements, whereas the mesh dual approach results in approximately twice the number edges as elements. For this reason, it was more efficient to use the edge-based representation in the partitioning process.

The output of the ParMetis library is a mapping from node number to partition number. For efficiency, a simple method of partitioning the elements was chosen in which the partition an element is placed in is governed by the partition of its first node.


(a) The dual of the elements

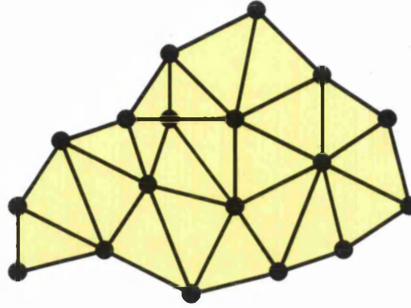

(b) The edges of the elements

Figure 153 – The two types of Edge-Based Data Structure for a mesh

In order to minimise memory usage during the partitioning process; the elements are read from disk twice. The first time is for the construction of the ParMetis edge-based data structure. This element information is then discarded before ParMetis is executed. When ParMetis has finished, the elements, vertices and boundary faces are read from disk again and placed in their respective partitions.

The final stage of the process involves the execution of a reverse Cuthill-McKee bandwidth minimisation algorithm [Cuthill69] on each partition independently in order to improve cache reuse, and hence, to improve the performance of the PSUE II and third party applications.

Figure 154 shows the performance of the partitioner (including I/O time) for various numbers of processors. The timings were obtained on a Cray T3E with 1024 processors and the mesh consisted of 100 million elements. The left graph shows the real time it took to partition the mesh and the right graph compares the speed up achieved compared with the ideal speed up, both of which are based on the result obtained with 64 processors.

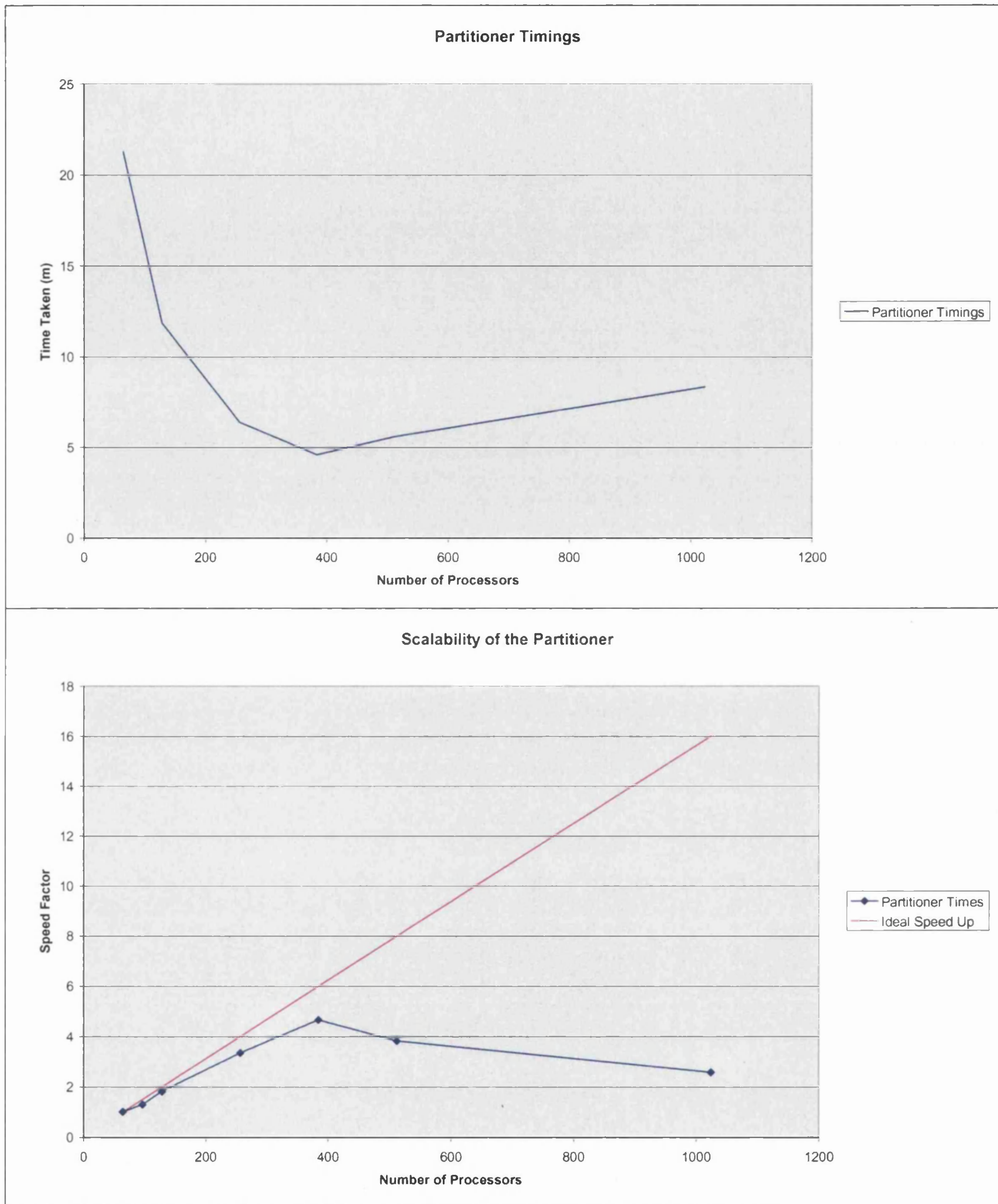


Figure 154 – Partitioner Performance Graphs

As can be seen, the speed up achieved only starts to degrade markedly when using over 256 processors. This was attributed to the fact that the I/O transfer rate of the computer does not scale well with the number of processors.

6.4. Increasing the Performance of Volume Data Set Traversal

The previous two sections have detailed how reducing network traffic and improving processor utilisation has enhanced the performance. Despite this, when the meshes reach the order of 100 million elements the time taken to traverse these large data sets to produce features, such as cutting planes, starts to become unacceptable. The only way to reduce this time is to reduce the number of volume elements the slaves must traverse.

In order to achieve this a more efficient data structure needs to be overlaid on top of the linear arrays of element connectivities. In the PSUE II v2.0, the data structure that was chosen was the Oct-tree. This is a very efficient data structure for many operations that require spatially searching for an object in three dimensions.

The Oct-tree is essentially a continuous, hierarchical sub-division of a cube into eight smaller cubes (or octants) using the central point of the parent as one of the corners of its children. This sub-division continues until a specified criterion is satisfied. For example, if an oct-tree is designed to spatially sort a collection of points, then this criterion might be to stop sub-division if an octant contains less than 100 points. When searching for a point this would allow the majority of points to be discarded very quickly finishing with a simple linear search through a maximum of 100 points.

As an example, Figure 155 shows an example of a mesh for which an Oct-tree data structure will be created. For purposes of clarity a 2D mesh has been chosen and a quad-tree data structure will be created. A quad-tree is a two dimensional equivalent of an Oct-tree where cubes (octants) are replaced by squares (quadrants) and each square is sub-divided into four children instead of eight. Figure 156 shows the quad-tree data structure using points for subdivision and the limit for further sub-division is five points.

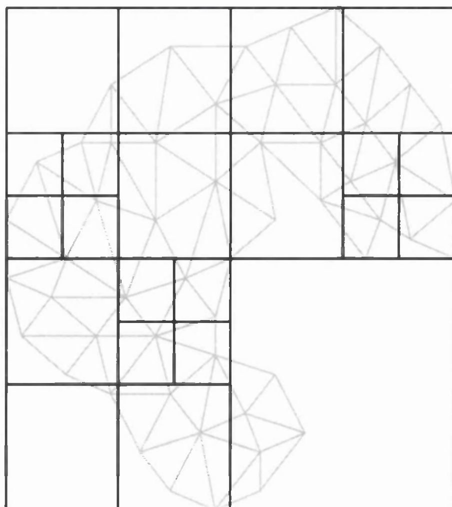


Figure 155 – The 2D Mesh

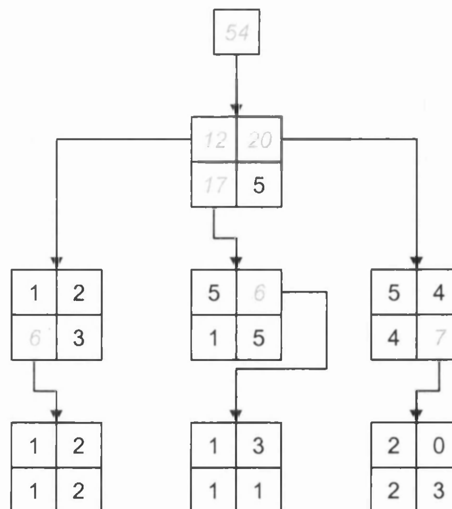


Figure 156 – The Quad-Tree Data Structure

In this example, it is assumed that the time taken to calculate the distance between two points and the time taken to decide which quadrant contains a point are approximately equal. Given that assumption, then even with this small data set it is obvious that the time required to search for the closest point to the point, *X*, has been improved. Searching linearly, the number of traversals is always 54, whereas with the quad-tree the worst case has been improved to just five traversals, and the best case to two.

If the Oct-tree is used to spatially sort a set of points then this method is adequate. However, in the PSUE II a more common set of entities to search for are the volume elements. This poses a problem for the Oct-tree since an element may span more than one octant. One way to overcome this problem is to assume an element is placed within an octant if any of its vertices are in that octant. This leads to a small amount of duplication when an element is present in more than one octant. However, using this procedure an element may be missed during a search. This is illustrated in Figure 157 that shows an element spanning four octants and having vertices in three of them. It is obvious that if the marked point is searched for then it will miss this element since it is not deemed to be in the fourth octant. In order to overcome this problem, a new way of determining whether an element is contained within an octant is required.

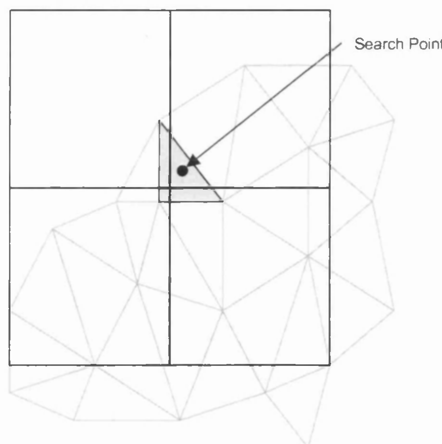


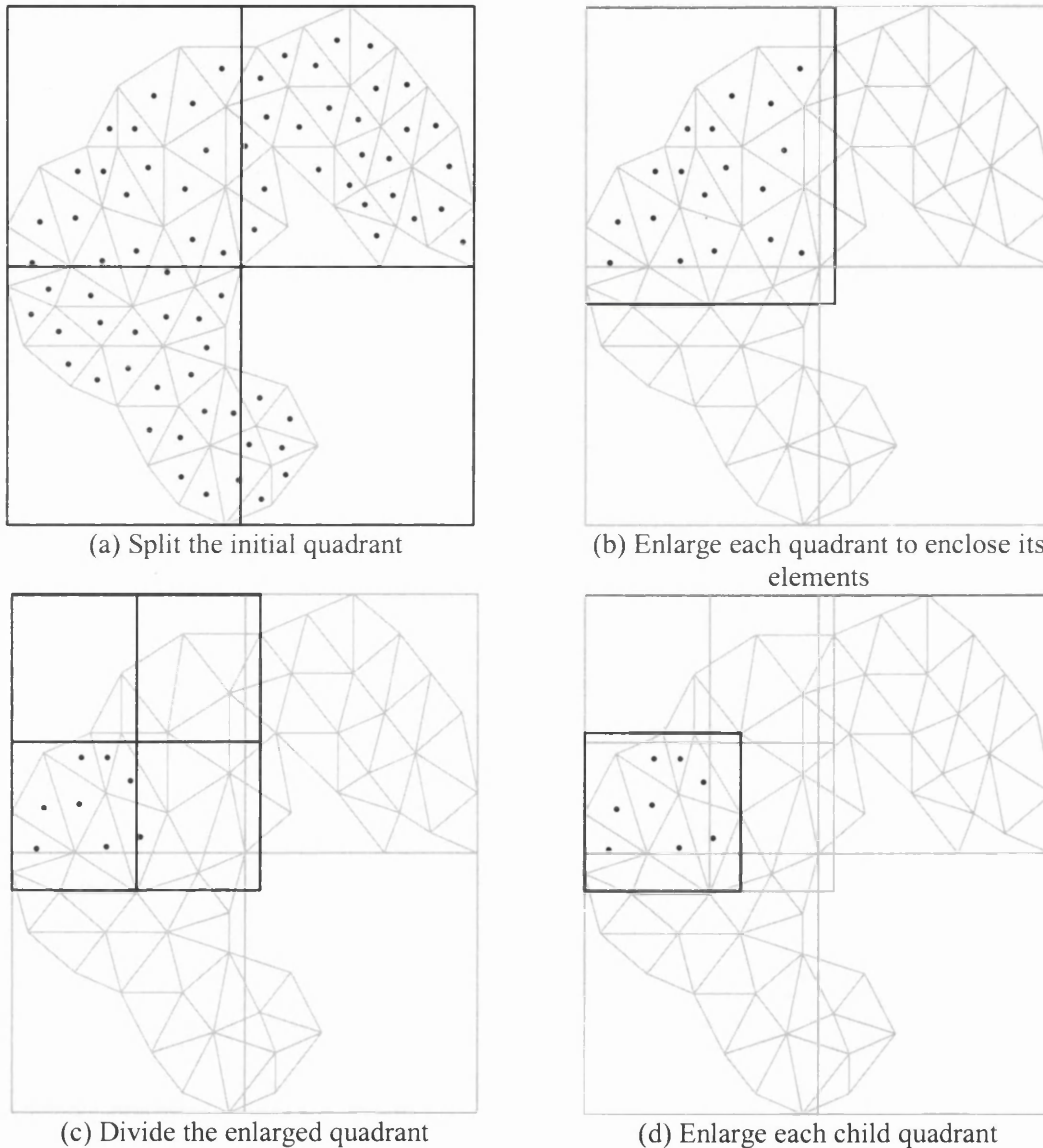
Figure 157 – Missing an Element in a Quad-Tree search

The method used within the PSUE II is to make three alterations to the algorithm that generates the Oct-tree data structure:

1. When an octant is being sub-divided into its eight children, the octant that is deemed to contain an element is the octant that contains the elements centre of gravity.
2. In order to ensure no elements are missed when performing any search operations, each octant's boundary is enlarged to fully enclose all of the elements deemed to be inside during the previous step.
3. These enlarged boundaries are then used when the octant is further sub-divided.

These additions to the basic algorithm ensure that an element cannot be missed during a search operation although it is likely that octants will overlap each other. If the search

procedure comes across two, or more, octants overlapping in the area of the search position then it has to choose one octant to follow and if that search is fruitless than it has to backtrack and follow the other possible routes. This makes the search procedure slightly more complicated but ensures that no elements can be missed. Figure 158 shows the same mesh as Figure 155 but with the three new rules applied when generating the quad-tree. For reasons of clarity, only one of the quadrants at each level is further sub-divided.



(a) Split the initial quadrant

(b) Enlarge each quadrant to enclose its elements

(c) Divide the enlarged quadrant

(d) Enlarge each child quadrant

Figure 158 – The Improved Quad-Tree Data Structure

As mentioned previously, using an Oct-tree during the creation of a cutting plane can significantly increase performance since the majority of the work involved when producing such a cut is the traversal through the data sets in order to find which elements have been cut by the plane. Without the use of a more advanced data structure, this search

involves traversing every element in the mesh, which for meshes of the order of 100 million elements can take an unacceptably long time.

With the use of the Oct-tree, the algorithm becomes:

```
for each octant do
    if Outer boundary is wholly on the correct side of the plane then
        Do nothing {All of the elements contained within remain unclipped}
    else if Outer boundary is wholly on the wrong side of the plane then
        Do nothing {All of the elements contained within are removed}
    else {Outer boundary of octant intersects the plane}
        if the octant has any children then
            Repeat algorithm recursively for each child octant
        else
            for each element in octant do
                if element intersects plane then
                    Produce intersecting primitive and add to cutting plane rendering list
                end if
            end for
        end if
    end if
end for
```

For two meshes over the same fighter aircraft, both partitioned into 16 sub-domains, using the same cutting plane position as shown above, the run-time of the two algorithms is shown in the table below. For this example, a octant is subdivided if it contains more than 1000 elements.

	Mesh Size (Tetrahedra)	Linear Search (seconds)	Oct-tree Search (seconds)
F16	6,725,979	10.38	1.91
F16	18,020,126	14.10	2.16

It should be noted that the time required for transmitting the geometric primitives to the master is not included in the measurements as it is the same for both cases; the timings presented are purely for searching through the volume data sets and creating the primitives.

As can be seen the performance improvements of the Oct-tree based algorithm are considerable even on reasonably small meshes of the order of 6 - 20 million elements. These improvements increase with mesh size as the $O(\log_8(n))$ oct-tree algorithm further diverges from the $O(n)$ linear search algorithm.

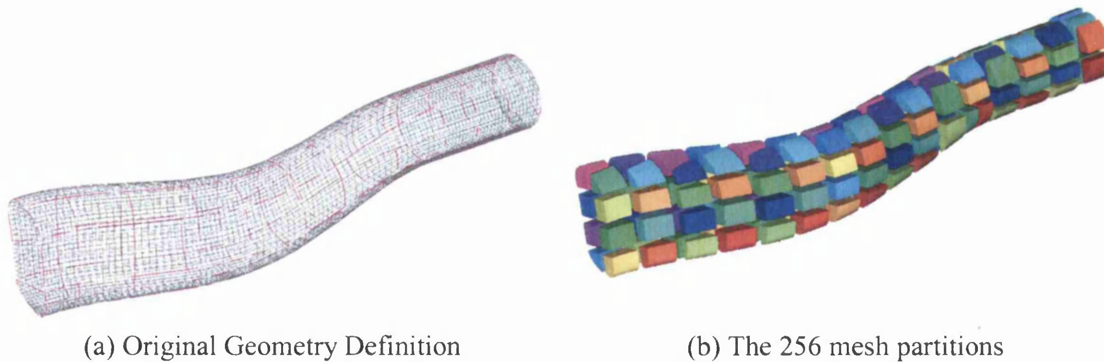
6.5. Increasing Rendering Speeds

As described previously, the PSUE II has always used the Vertex Array extension provided by Open-GL for the following reasons:

- It is faster than repeatedly sending individual vertices, colours and normals in Immediate Mode,
- It uses less memory than Display Lists and
- It can be altered more rapidly than Display Lists.

In PSUE II v1.0, this extension was used to render the geometric primitives contained within the Render Objects, which included points, lines, triangles and quadrilaterals. As already described this process involved creating the rendering data-sets on the slave processes, transmitting them to the master and then the master using them to produce the image on the display.

However, whilst testing the environment on a very large test case involving over 200 million elements (Figure 159), an unusual amount of memory usage was witnessed on the master running on the workstation. Further analysis revealed an unforeseen problem with the use of vertex arrays that had the potential to limit the size of the simulation that could be performed within the environment. Figure 160 shows the statistics of the mesh involved and Figure 161 shows the statistics for the rendering data.



(a) Original Geometry Definition (b) The 256 mesh partitions

Figure 159 – A 200 Million Element Mesh

	Global Mesh
Number of Tetrahedra	236,356,076
Number of Nodes	44,078,548
Number of Triangles	3,668,652

Figure 160 – Statistics for the 200 Million Element Mesh

	Num Items	Item Size	Data Size (Mb)
Coordinates (3 / triangle)	11,005,956	3*sizeof(real)	125.95
Normals (3 / triangle)	11,005,956	3*sizeof(real)	125.95
Colour Indices (3 / triangle)	11,005,956	sizeof(int)	41.98
	Total Data Size		293.89

Figure 161 – Statistics for the Rendering Data for the Mesh

As can be seen, although the memory usage of the slaves on the parallel server was reasonable small, the size of the rendering data sets on the workstation was quite alarming considering an average graphics workstation has between 256Mb and 512Mb of memory.

To solve this problem, an alternative method was found in the form of another set of geometric primitives provided by Open-GL, called strips. These could be used for lines, triangles and quadrilaterals. Figure 162 shows a comparison between the strips and their respective primitives specified individually.

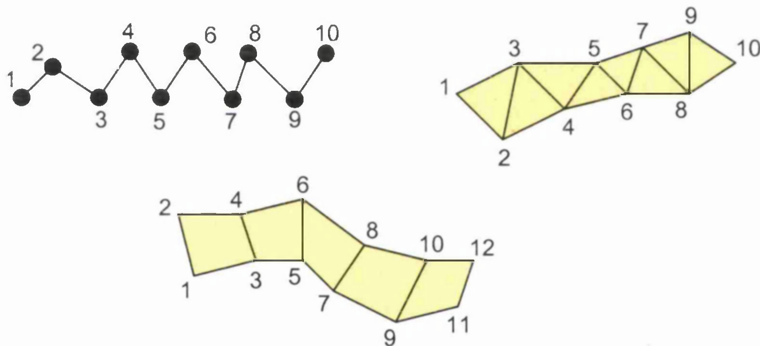


Figure 162 – Comparison between Single Primitives and Strips

For line strips consisting of n segments, the number of vertices needed is $n+1$ rather than $2n$ when specifying them as individual lines. Similarly, a triangle strip consisting of n triangles requires $n+2$ vertices rather then $3n$, and quadrilateral strips require $2n+2$ vertices rather than $4n$ for n quadrilaterals.

As can be seen, using strips of primitives can dramatically reduce the number of vertices. This reduction proves even greater when the fact that normals and colours are also specified at the vertices is taken into account. The use of strips also has a hidden benefit in terms of rendering performance since the graphics hardware in the workstation has far less vertices to pass through its pipeline to render the same number of triangles.

However, in order to be able to do this the individual primitives need to be transformed into a set of strips. Obtaining an optimal set of strips for a given set of primitives has been proved to be an NP-complete problem (i.e. it cannot be solved in polynomial time)

since it is a variation on an Hamiltonian Cycle [Dillencourt92]. However, obtaining a good (although not optimal) set of strips can be done in an efficient manner [Kornmann99, El-Sana00]. The algorithm used in the PSUE II v2.0 is based on a paper and source code by Kornmann. This is a fairly straightforward algorithm that works in a *greedy* manner with performance enhancements through the use of priority queues which provide good results for a small amount of computation.

In order to maintain the structure of the classes involved in the mesh storage, it was decided that this algorithm should be applied to the individual 2D Mesh objects and then the resultant data sets could be passed on to the Rendering Objects as before. Although this decision may produce slightly less optimal strips, it was deemed to be too complex and error-prone to maintain the local grouping of the primitives at the same time as producing triangle strips that spanned these groups. The new statistics for the rendering data are shown below in Figure 163. As can be seen, the memory required on the master process on the graphics workstation has been reduced by over 50%. This reduction also translates directly into the reduction into the amount of data that needs to be sent to the graphics hardware during the rendering of every frame thus improving the interactive performance.

	Num Items	Item Size	Data Size (Mb)
Coordinates	4,891,340	3*sizeof(real)	55.98
Normals	4,891,340	3*sizeof(real)	55.98
Colour Indices	4,891,340	sizeof(int)	18.66
	Total Data Size		130.61

Figure 163 – Statistics for the Rendering Data using Triangle-Strips

6.6. The Integration of Third-Party Applications

One of the objectives of the PSUE II was to allow the seamless integration of third party application software. The first stage of this work was to allow the user to execute these applications, which are possible parallel applications in their own right, on local or remote platforms. The second stage was to allow the transferral of data files between the application and the PSUE II. The third, and final stage, was to allow control and data information to flow between the application and the PSUE II whilst the application was running in order to support functionality such as monitoring the solution as a flow solver runs or computational steering.

6.6.1. Stage 1 – Application Execution

The initiation of third-party applications from the PSUE II without the need to modify any source code needed a method in which the user could add buttons and toolbars to the existing toolbars on the left of the PSUE II window (Figure 164). These are arranged in a hierarchical format (Figure 166) where clicking on a button on the top-level toolbar overlays the top-level toolbar with the appropriate one at the next level.

The means by which the user can add to this hierarchy is through a simple text file with the format:

```
Toolbar( "Button Title", "Toolbar Title" )
{
    Button( "Button Title" )
    {
        List of Operations
    }
    Toolbar( "Button Title", "Toolbar Title" )
    {
        Button( "Button Title" )
        {
            List of Operations
        }
    }
    Button( "Button Title" )
    {
        List of Operations
    }
}
```

Here, a hierarchical description of the user-defined toolbars and buttons can be specified. If the *button title* and *toolbar title* of a toolbar is the same as an existing toolbar then the items will be placed at the bottom of the existing toolbar. If it is preferred that a button is placed in an existing toolbar in a particular position then this can be achieved by the following line:

```
Button( "Button Title" ) before/after "Existing Button Name"
```

The *list of operations* for application execution consist of a list of one of the following operations:

execute_command("*command*")	Executes a shell command
define_platform()	Allows the user to define a parallel platform using a graphical panel (Figure 173).
save_platform("*filename*")	Saves the last parallel platform configuration defined using the *define_platform()* command as a simple text file.
execute_mpi_command("*command*")	Executes a remote *mpirun* on the first of the platforms defined using *define_platform()* using the remainder of the machines as the configuration file for the *mpirun* command.

The parsing of the text file is performed using the GNU implementations of the standard lexical analyser Lex [Aho86, Levine92] called FLEX [Paxson98], and the standard parser generator YACC [Aho86, Levine92] called Bison [Donnelly02].

The lexical analyser is used to scan arbitrary text files and return tokens that match defined patterns of characters, for example:

`[0-9]+`	Recognises a pattern of one or more digits (i.e. an integer)
`[a-zA-Z][_a-zA-Z0-9]*`	Recognises a string starting with a letter and continuing with one or more letters, numbers or an underscore (i.e. a standard C identifier)

FLEX parses the pattern definitions, such as those on the left, and generates a C subroutine that, when called from within a program, rapidly scans and matches patterns in any input text file.

The parser generator uses the tokens returned by the lexical analyser to perform a more sophisticated parsing of the text file. For example, the outline of a parser for a simple calculator might look like:

```
expression: NUMBER                          { $$ = number }
          | expression "+" expression       { $$ = $1 + $2 }
          | expression "-" expression       { $$ = $1 - $2 }
          | expression "*" expression       { $$ = $1 * $2 }
          | expression "/" expression       { $$ = $1 / $2 }
          | "-" expression                  { $$ = -($1) }
          | "sqrt" "(" expression ")"       { $$ = sqrt( $1 ) }
```

Here, an expression is recursively defined as a number or as a sub-expression that includes one or more expressions. The output of Bison is, again, a C subroutine that, when called with a given input text string parses the string based on the definitions given by the programmer.

The code in blue is actual C code that is placed in the generated C subroutine. The $$ symbol represents the return value of the expression and the $1, $2, ... represent the arguments of an expression. When the generated C subroutine is called with input text, the various portions of C code (in blue) are executed depending on which expression match the input text.

6.6.2. Stage 2 – Data File Transferral

When a third party application is executed via the commands described in the previous section it will often be necessary to transfer the data, in the form of files, to the application and then retrieve any output files generated by the application back to the PSUE II.

To perform these tasks, the following commands were added to the *list of operations* described above:

clear_#####()	Clears the ##### from the PSUE II
load/save_#####(*"platform name"*, *"filename"*)	Loads/saves a ##### data file on the specified remote platform using the specified filename.

Here '#####' can be one of geometry, sources, surface mesh, volume mesh, boundary conditions or solution. For file I/O on the local platform then the *platform_name* field would contain *localhost*. The remote I/O is performed using the FTP protocol [Stevens90].

6.6.3. Stage 3 – PSUE II and Application Interaction at run-time

Using the two previous stages, a remote application can be initiated, input files can be passed to it and any output files can be retrieved when the application has finished. For more advanced interaction between the application and the third party application, the above two stages are not sufficient.

In order to accommodate this extra functionality, two additional means of communicating with the PSUE II were developed:

- The ability to run Python programs from within the PSUE II using the configurable toolbars was added.
- The PSUE II allowed CORBA connections from the outside world.

Python Integration

Python [Rossum02a-h] is an object-oriented scripting language that has all of the programming constructs of many traditional programming languages such as C++. Integrating Python into the PSUE II has been achieved on two levels that are normally referred to as *extending* and *embedding* Python [Rossum02d].

Extending Python involved writing a series of modules, written in C, that are callable by a Python program. These modules allow the Python program to gain direct access to any information stored in the PSUE II and allow the Python program to control many aspects of the PSUE II functionality. They are implemented as a set of C subroutines that have a defined set of arguments. A simple example of an embedded Python module is the GUI module that controls the position of the model in the PSUE II window is shown below.

```
module gui
{
    translate( dx, dy, dz )          // Translates the model.
    rotate( angle, ax, ay, az )      // Rotates the model by angle degrees
                                     // about the axis [ax, ay, az]
    scale( s )                       // Magnifies the model by s.
    redraw()                         // Causes the model to be redrawn
```

```
}
```

Embedding Python involved initiating the Python interpreter from within the PSUE II. This allows the PSUE II to execute Python commands and programs at will. This functionality is accessed by adding the following command to the *list of operations* in the configurable toolbars:

execute_python_command ("*command*")	Executes a single python command. Multiple commands may be executed either by using multiple instances of *execute_python_command* or by including multiple Python commands within the quotes.
execute_python("*filename*")	Executes a Python program references by *filename*.

Outside CORBA Connections

This was achieved by allowing any external application to connect, via CORBA, to either the PSUE slave processes for parallel data transfer, or to the PSUE master process, for controlling its functionality. These extra connections allow third party applications to:

- *Update the PSUE II slaves with new data*
  This could be used to monitor the progress of a remote solver during its execution by updating convergence data or even the whole solution. This functionality could be combined with a separate GUI application that could control the solver remotely to perform computational steering.
- *Control the PSUE II master process operations*
  This could allow another application to control any or all of the functionality of the PSUE II. For example, a program could instruct the PSUE II to load in a set of solution data files, manipulate the position of the model on the screen and create a series of snapshot images in order to make a movie overnight.

Obviously, in order to be able to make use of this extra functionality, the external application would need to be modified so is only of use if the source code is available.

6.7. Summary

This chapter has described a number of improvements over the initial implementation of the PSUE II. As has been shown, these improvements had a significant improvement on the performance of the environment, and therefore enabled the use of larger data sets whilst still maintaining interactivity with the user. Chapters 7 and 9 complete the description of the PSUE II by presenting the user interface and some complete simulations performed using the environment.

Chapter 7. THE FUNCTIONALITY OF THE PSUE II v2.0

The aim of this chapter is to cover the various features available within the PSUE II. The first section introduces the PSUE II main GUI, with the various components and functionality detailed in later sections.

7.1. The Main Display

One of the aims of the PSUE II throughout its design and implementation was to bring all of the user interaction with the three-dimensional models together in one area rather than the user having to learn how to operate many different GUIs. Figure 164 shows a typical appearance of the main GUI of the PSUE II. This consists primarily of three sections:

- A set of nested toolbars on the left,
- Entity selection tools along the bottom and
- The main Drawing Canvas on the right.

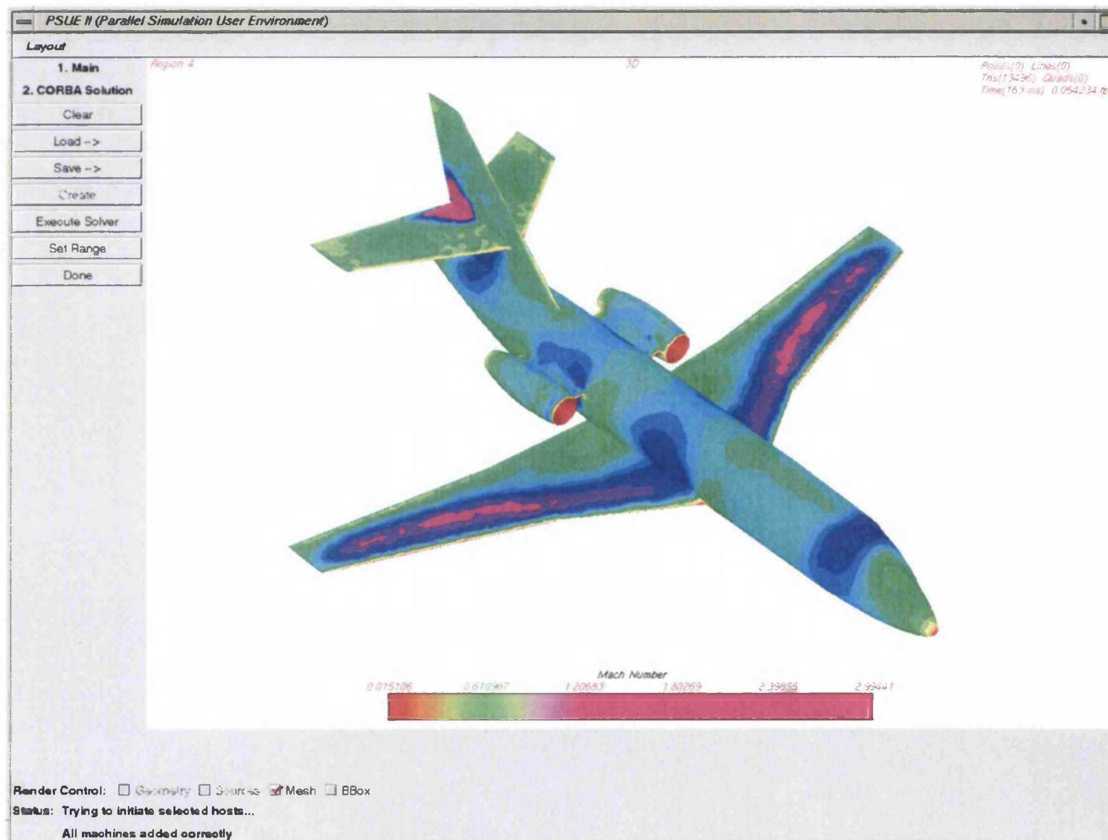


Figure 164 – The Main PSUE II GUI

The Toolbar contains buttons for all of the operations available within the PSUE II. This will be described in the next section.

The Drawing Canvas is used to display all of the data sets stored by the PSUE II. All of the manipulation and selection of these models is performed within this area. Regardless of the type of data currently residing within the environment, the manipulation is performed in a consistent manner:

- The Left Mouse Button is always used for selection / picking actions.
- The Middle Mouse Button is always used to manipulate the view of the data on the screen.
  - With *no keys pressed*, the model is **translated** in order to follow the mouse pointer.
  - With the *SHIFT key pressed*, a **zoom** operation is performed on the model. Moving the mouse right zooms in, and left zooms out.
  - With the *CTRL key pressed*, a **rotation** is performed on the model. Vertical motion of the mouse pointer causes the model to be rotated about the horizontal axis, with horizontal motion performing a rotation about the vertical axis.
- The Right Mouse Button is used for miscellaneous operations when required for some tasks.

7.2. The Nested Toolbars

During the design phase of the PSUE II there were three main means of allowing the user to perform operations within the environment. These were:

Microsoft Windows-style Graphical Toolbars

It was decided at an early stage that this style of toolbar would be too difficult to implement in a UNIX environment and the design of the graphical icons would consume too much valuable time.

Nested, Textual Toolbars

Nested toolbars have the advantages that they are straightforward to implement in any windowing environment under any Operating System, they are easy to maintain and adapt since they require no graphical design and they are a compact means of representing a large number of options to the user.

They also have the added advantage that only the options required for a particular operation are available to the user at any one time. This imposes a *modal* means of operation where the features of the model that are displayed and selected can be adjusted automatically by the environment to suit the particular operation. For example, if the user traverses to a toolbar that deals with mesh sources then the environment can determine the set of operations that the user wishes to perform. This allows the display of the model to be changed automatically so that the

sources are rendered in more detail, ready for editing, and any other features, not necessary during this operation, can be rendered in less detail or not at all. It also allows the environment to determine that during any selection operation it is the sources that are to be tested for selection rather than geometry curves or surfaces.

Pull-down Menus

Like the nested toolbars, these allow the environment to present the user with a large number of options in a textual and compact format. However, unlike the nested toolbars the style of operation within the environment is entirely *modeless*, i.e. any operation is available for selection at any time. This means that the environment has no means of knowing which operation is to be performed next and thus cannot adapt. Pull-down menus can also suffer from performance degradation, particularly on lower-end computers. This is due to the fact that as the user traverses the menus, any section of the drawing canvas that was previously obscured by a menu has to be redrawn before the next menu is displayed. This time can be quite lengthy if the model is complex. On higher-end computers this problem is eliminated since the menus are drawn into *overlay planes*. Overlay planes are a computer equivalent to overlaying a piece of paper with a sheet of transparency. Items drawn on the transparency obscure the paper beneath but do not actually change the image on the paper. Removing the items is simply a case of wiping the transparency without having to redraw anything on the paper.

7.2.1. The PSUE II Nested Toolbar

The hierarchical structure of the nested toolbar in the PSUE II is shown in Figure 166.

The top-level menu presents the user with the main stages of the computational simulation as would be performed within the environment as shown in Figure 165. Each of these toolbars is described in more detail in later sections.

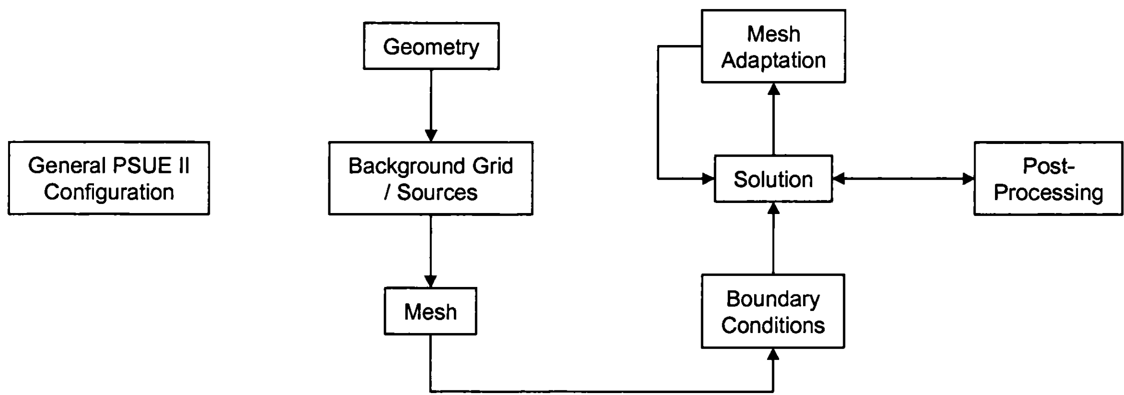


Figure 165 – Mapping of the PSUE II Toolbars to the Simulation Process

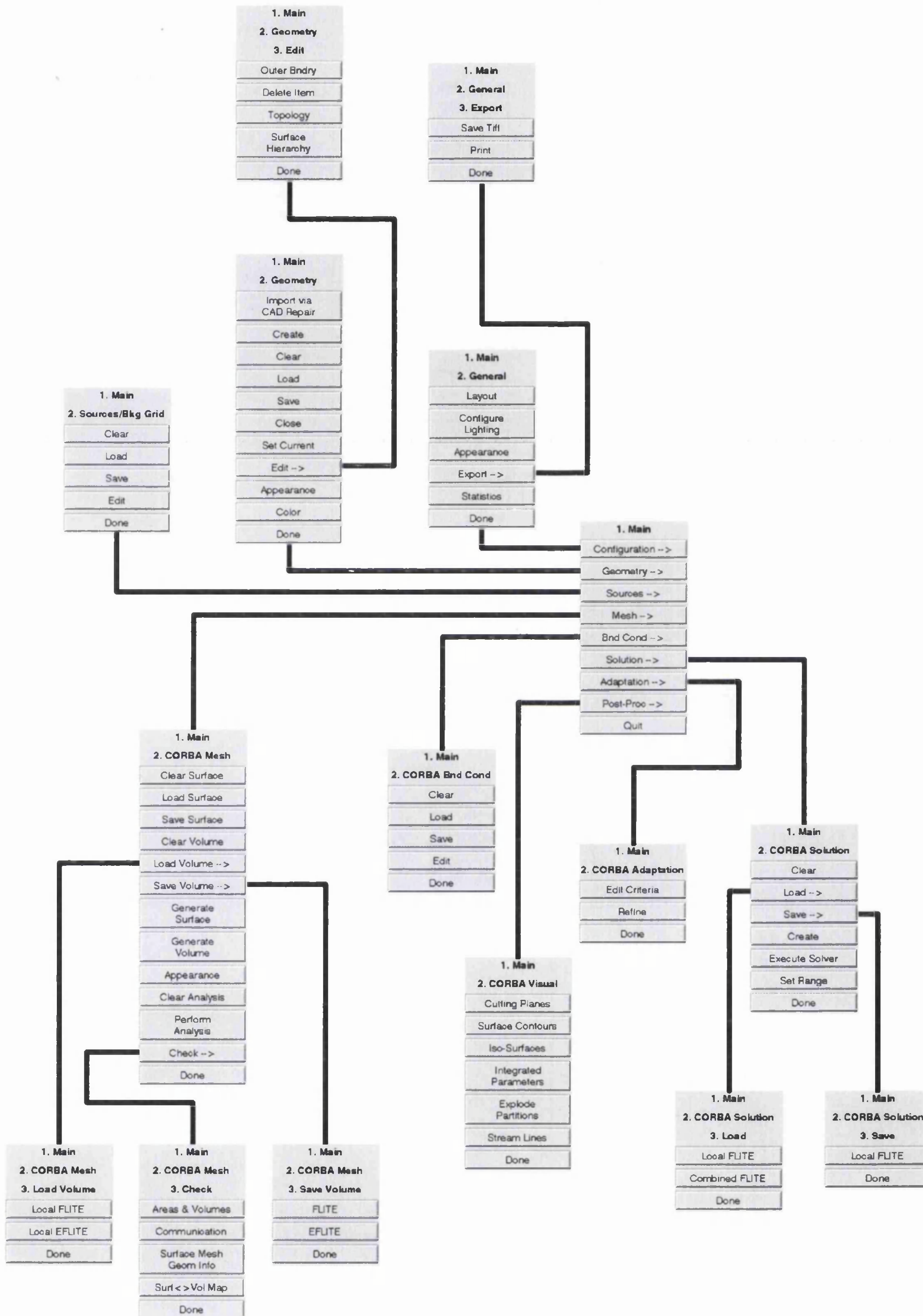


Figure 166 – The Structure of the Nested Toolbar in the PSUE II

7.3. The 'Configuration' Toolbar

The Configuration Toolbar contains the following options:

Layout

This option allows the user to change the general appearance of any graphics rendering on the main drawing canvas. As shown in Figure 167, the window is split, vertically, into two main sections. The top section governs which views of the model are displayed within the main drawing canvas. The top-left area represents the region layout of the main canvas and is shown here as being split into four regions displaying the front, left, top and a fully rotateable view of the model. This configuration can be changed by simply clicking on the buttons at the right of this area. These button cause the centre of the crosshair to be moved in the appropriate direction thus causing the number of regions to change from four, through three and two, to one. The toggle buttons below this area govern which views of the model each of the four regions' displays.

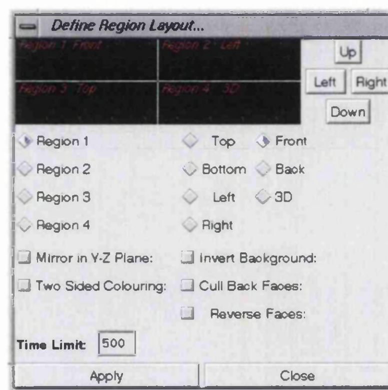


Figure 167 – Region Layout Panel

The lower half of the panel controls various other miscellaneous rendering options. These are:

- *Mirror in X-Z Plane*

  This allows the user to mirror any model with a symmetry plane so as to produce the full model. It should be noted that this doubling is only performed for rendering. The actual model remains the same for simulation purposes.
- *Invert Background*

  This is a quick and easy method of changing the background from black to white, with any white renderings changed to black. This is mostly used for producing screen dumps for printed material.
- *Two Sided Colouring*

  Normally the appearance and lighting of a surface is independent of the orientation of the rendering primitives. Selecting this option causes surfaces using different orientations to be displayed in different colours.

(This is most often used as a debugging aid during mesh generation development).

- *Cull Back Faces*

  Selecting this option causes any rearward facing polygons to be ignored during rendering. This can speed up rendering significantly and if the model is solid with no surfaces removed then its appearance is not changed. However, if any surfaces are removed then the missing rear surfaces become noticeable. This is also true when cutting planes are defined since they will only be visible from one side. When the model is rotated the cutting plane will disappear since it is now considered rearwards facing.
- *Reverse Faces*

  This option causes the orientation of the normals for all rendering primitives to be reversed. This affects the rendering when 'Two Sided Colouring' or 'Cull Back Faces' is enabled. (This is mostly used as a debugging aid).

The last item on the panel is the time limit box. This allows the user to fine-tune the automatic transition between the rendering modes used for objects when the model is still and the modes used when it is in motion. When the still rendering time of the model exceeds the number of milliseconds in the box the rendering mode for objects in motion is used. This is a simple means of maintaining an interactive rendering speed even for large models.

Configure Lighting

The 'Configure Lighting' panel allows the user to change the direction of the lighting and the material properties of the model. The top section of the panel represents the light source on a sphere constructed with a material using the current settings. The light source can be moved around on that sphere simply by clicking and dragging with the left mouse button.

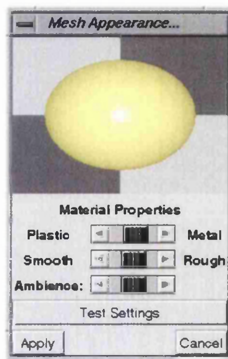
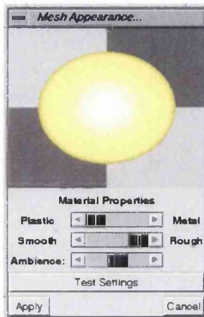


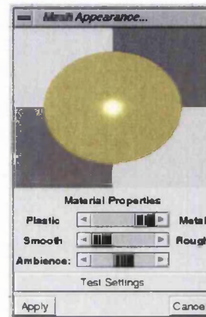
Figure 168 – Lighting and Material Panel

The material properties of the sphere can be adjusted using the Plastic-Metal and Smooth-Rough sliders. The ambient light can be altered using the Ambience slider. Figure 169 shows some typical effects along with their slider positions.

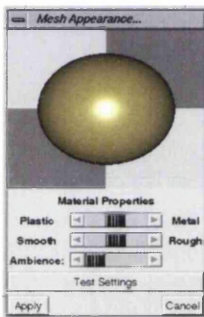
The Material sliders determine the width of the spot of specular reflection along with how quickly it degrades to having no reflection, and the Ambient slider determines how bright the object is outside the specular region.



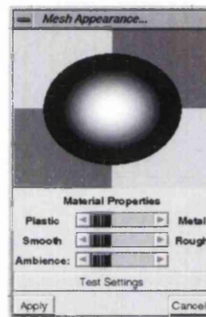
Rough Plastic Material
with 50% Ambient Light



Smooth Metal Material
with 50% Ambient Light



No Ambient Light



Smooth Plastic Material
with no Ambient Light

Figure 169 – Various Material Properties

General Appearance

The General Appearance Panel allows the user to customise the colours of the overall drawing canvas (not including the models). The background colour and the labelling colour are selected via the two tabs at the top of the panel. The colour can then be adjusted by either dragging the cursor across the colourful hexagon or moving the sliders. This allows the colours to be set using either the HSV (Hue, Saturation, Value) model or the RGB (Red, Green, Blue) model.

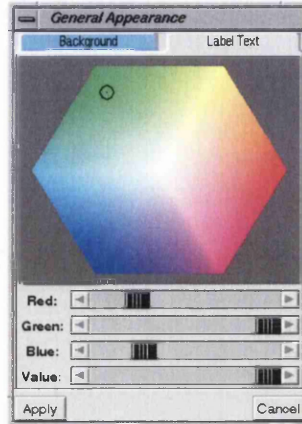


Figure 170 – General Appearance Panel

Export

The sub-toolbar under the Export item allow the user to save the current drawing canvas as either a TIFF image file or an EPS (Encapsulated Postscript) file. In either case the user is prompted for a filename under which the image is saved.

7.4. The 'Geometry' Toolbar

The Geometry Toolbar contains the following options:

Clear

This option causes the geometry to be removed from the environment. This disconnects and terminates the Geometry Servers and removes any Render Objects from the Master process.

Load

This option allows an existing geometry file to be loaded into the environment. A File Selection panel opens to allow the user to select the required file (Figure 171).

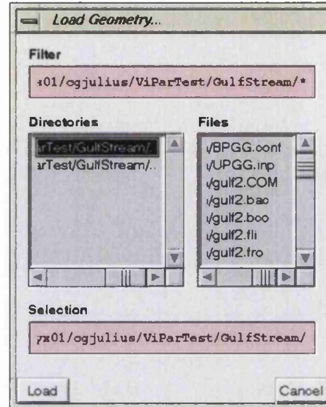


Figure 171 – A File Selection Panel

Once this has been done the panel will close and a second panel will open (Figure 172) asking the user to enter the number of servers to spread the geometry across. After this a third panel, the Parallel Platform Panel (Figure 173), is opened containing a list of computers on which the Geometry Servers may be executed. Once the set of computers has been chosen, the requested number of Geometry Servers is then initiated and connected to the environment. These servers then co-operate in order to load the geometry curves and surfaces and distribute them as evenly as possible across them.

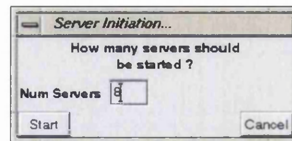


Figure 172 – Entering the number of Geometry Servers

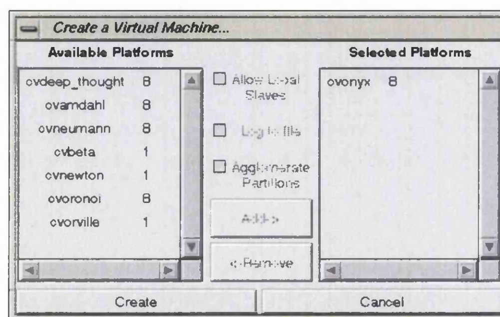


Figure 173 – The Parallel Platform Panel

Save

This option allows the user to save the geometry currently stored within the environment to a file on disk. The user is first presented with a File Selection

Panel (Figure 171) into which the file location should be selected and then the filename entered. The Geometry Servers then co-operate with each other to recombine the various geometry curves, surfaces and topology information back into one geometry file.

Edit

The 'Edit' option is actually the header of a sub-toolbar. Selecting this link causes the current toolbar to be replaced with the 'Geometry Edit' sub-toolbar. The options contained within this toolbar are described in detail in the next section.

Appearance

The 'Appearance' option allows the user to change the rendering style of the various geometry curves and surfaces on the workstation display through the opening of the Geometry Appearance Panel (Figure 174).

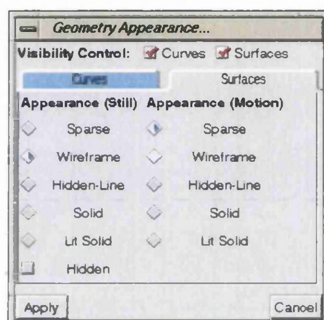


Figure 174 – The Geometry Appearance Panel

This panel is divided, vertically, into two sections. The top section contains two check boxes that control whether the geometry curves or surfaces are rendered. These options operate globally regardless of the settings in the lower section of the panel.

The lower section consists of two columns of toggles. These affect how the geometry curves and surfaces are rendered. The first column of toggles controls the rendering style of the geometry while it is still. The second column controls the rendering style when the geometry is being manipulated using the mouse (i.e. dragged, scaled or rotated).

At first these toggles will be disabled (ghosted) since no curves or surfaces have been selected on which to edit their appearance. Curves and surfaces may be selected using the Selection Bar in the Main Window (described above). The toggles in the Geometry Appearance Panel become enabled once one, or more, curves or surfaces have been selected. Each toggle always shows the current appearance settings for the selected entities, or blank if the selected entities have different settings for that particular toggle.

Selecting any toggle changes the appearance of all of the selected curves and surfaces to reflect the settings of the toggles. These changes are not reflected on the display until the 'Apply' button is selected. This closes the panel and updates the selected entities on the display. Selecting the 'Cancel' button also closes the panel and ignores any changes made by the user.

Colour

Selecting this option causes the Geometry Colour Panel to appear. The entity type for which the colour is to be altered is selected using the tabs at the top of the window. Once the curves or surfaces have been selected, the panel updates to show the current colour of the selected entities, or is blank and disabled if no entities are selected.

The user may then change the colour by either dragging the cursor in the coloured hexagon or by dragging the sliders below. The colours are then applied and the window closed by selecting the 'Apply' button.

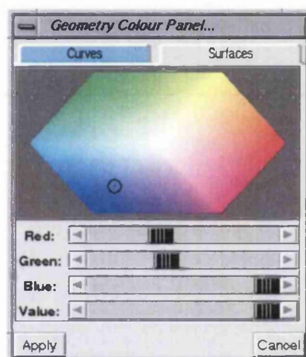


Figure 175 – Geometry Colour Panel

7.4.1. The 'Geometry Edit' Sub-Toolbar

The 'Geometry Edit' toolbar contains a number of items all connected with ensuring a geometry is in a form in which mesh generation can take place. It has the following two options:

Create Outer Boundary

This option allows the user to create a simple outer boundary for any geometry. A panel (Figure 176) opens to allow the user to choose between a number of standard outer boundary shapes comprising a sphere, cylinder or box with half-spheres and half-cylinders for geometries that require a symmetry plane.

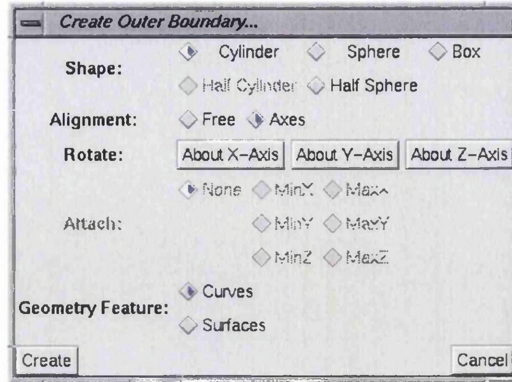


Figure 176 – Outer Boundary Editing Panel

Once the user has chosen a shape it appears on the main graphical display and can be manipulated in the same manner as the geometry. For outer boundaries that require a symmetry plane, the outer boundary shape can be attached to one of the boundaries of the geometry. When the outer boundary is of the correct size and in the correct position, it may be fixed by pressing the 'Create' button. This closes the panel and creates the relevant geometrical surfaces and intersection curves. For shapes with a symmetry plane, the appropriate bounding curves of the geometry are automatically attached to the symmetry plane to form a closed volume.

Edit Topology

This panel (Figure 177) allows the user to attach curves and surfaces to each other in order to form a topologically valid model. The panel contains a number of options depending on whether it is most suitable to connect curves to surfaces, surfaces to curves, disconnect curves or disconnect surfaces.

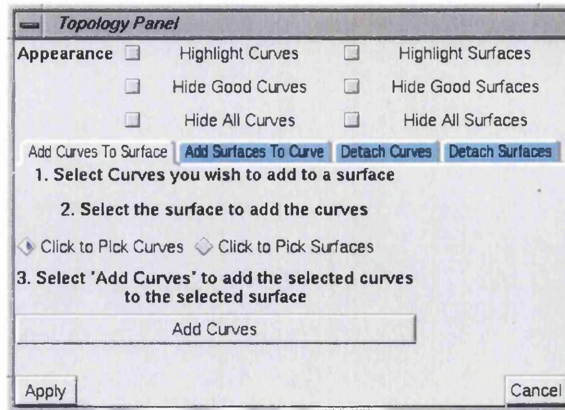


Figure 177 – The Topology Edit Panel

To aid the user in selecting the appropriate curves and surfaces, the two entities may be highlighted in red or green depending on whether they form part of a valid topology. To reduce any possible clutter on the display, any curves and surfaces

that are deemed to form part of a valid topology may be hidden. This allows the user to concentrate on areas that need repair.

7.5. The 'Sources' Toolbar

The Sources Toolbar contains the following options:

Clear

This option causes the sources to be removed from the environment.

Load

The option allows an existing set of sources to be loaded into the environment. A File Selection panel opens to allow the user to select the required file (Figure 171). The panel is then closed and the sources are loaded and displayed as a set of spheres in the main window.

Save

This option allows the user to save the current set of sources to a file. The user is presented with a File Selection Panel (Figure 171) into which the location and name of the file is chosen. The set of sources is then saved.

Edit

This option allows the user to create and/or edit the sources within the environment through the Edit Sources Panel (Figure 178).

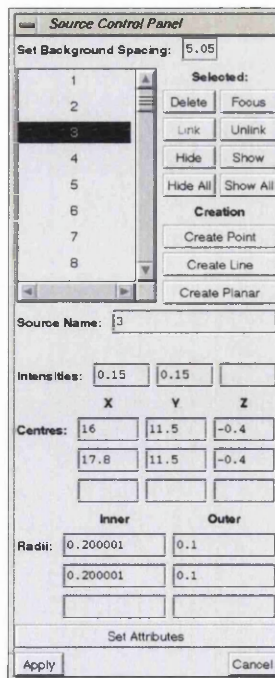
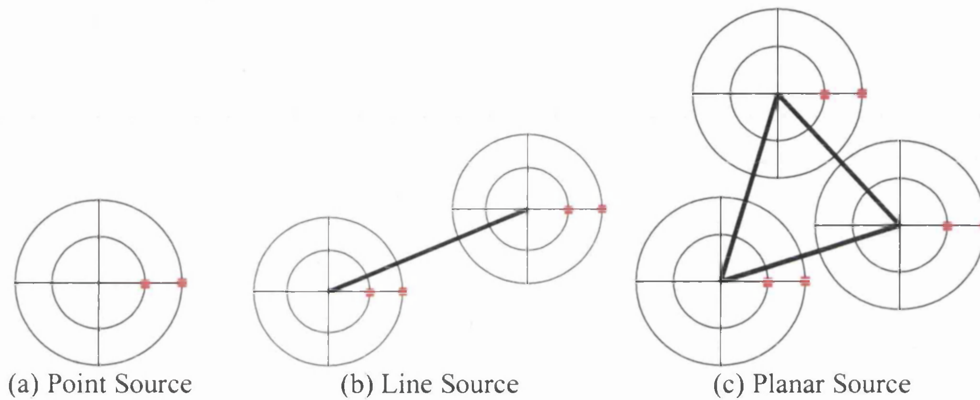


Figure 178 – The Edit Sources Panel

Using this panel, sources may be created, destroyed or manipulated. Selection of a source is performed by either selecting its entry in the scrollable list in the panel or by clicking on the source in the main graphical display.

Once a source has been selected, it maybe manipulated by either entering its details into the panel and clicking the 'Set Attributes' button or by simply clicking and dragging the highlighted source. A schematic of a selected point, line and planar source is shown in Figure 179 with the handles in red. Clicking and dragging the handles expands the source whereas dragging the source axes moves the entire source.



(a) Point Source (b) Line Source (c) Planar Source

Figure 179 – Dragging Handles for Sources

7.6. The 'Mesh' Toolbar

The Mesh Toolbar contains the following options:

Clear Surface

This option causes the surface mesh to be removed from the environment. This disconnects and terminates all of the Mesh Servers associated with the surface mesh. The Render Objects representing the mesh are then removed unless a volume mesh is also present, in which, no changes occur on the display.

Load Surface

This option allows an existing surface mesh to be loaded into the environment. A File Selection Panel (Figure 171) opens to allow the user to select the required surface mesh file. Once the file is chosen, the set of computers on which the Mesh Servers are executed is chosen using the Parallel Platform Panel (Figure 173). The Mesh Servers will then be initiated on the selected computers and the surface mesh will be loaded and distributed amongst them by their surface number (as described in Section 4.4.6).

Save Surface

This option allows the user to save the surface mesh currently stored within the environment to a file on disk. The user is asked to enter the file name and location via a File Selection Panel. The Mesh Servers then co-operate to save the combined surface mesh file.

Clear Volume

This option causes the volume mesh to be removed from the environment. This disconnects and terminates all of the Mesh Servers associated with the volume mesh. The Render Objects representing the mesh are then removed unless a surface mesh is also present, in which case, only the Render Objects representing features only in the volume mesh (e.g. cutting planes, iso-surfaces, etc.) are removed.

Load Volume

This option allows an existing set of volume mesh partitions to be loaded into the environment. The user is first presented with a File Selection Panel (Figure 171) to select any partition of the mesh. This panel is then replaced by a second panel (Figure 180) that asks the user to enter the number of partitions that comprise the mesh and place a '#' symbol in place of the partition number in the filename.

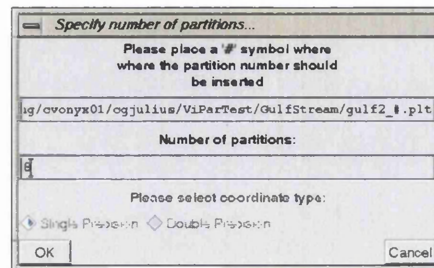


Figure 180 – Loading a Partitioned Volume Mesh

Finally, the Parallel Platform Panel (Figure 173) is opened to allow the user to choose the set of computers on which the Mesh Servers are executed. The number of Mesh Servers specified by the user are then initiated and connected to the environment. These servers are each given a filename representing the mesh partitions they should load. The files are loaded, in parallel, and the Render Objects representing the mesh surfaces are sent back to the Master process for display.

As an example, assuming the filename selected was '`/home/mymesh_1.plt`' with 4 partitions. Placing the '#' symbol changes the filename to '`/home/mymesh_#.plt`'. This causes four Mesh Servers to be initiated and passed the following filenames:

1. '`/home/mymesh_1.plt`' and '`/home/mymesh_1.com`',
2. '`/home/mymesh_2.plt`' and '`/home/mymesh_2.com`',

3. '/home/mymesh_3.plt' and '/home/mymesh_3.com', and
4. '/home/mymesh_4.plt' and '/home/mymesh_4.com'

The files with the '.plt' extension contain the volume mesh data sets and the '.com' files contain the communication information.

Save Volume

This option allows the user to save the mesh partitions currently stored in the environment to a set of files on disk. A File Selection Panel (Figure 171) opens to allow the user to select the name and location of the files. The chosen filename then has the various suffixes appended before being sent to the Mesh Servers for saving. As an example, if a filename '/home/mysave' was selected then the filenames sent to the four Mesh Servers are:

1. '/home/mysave_1.plt' and '/home/mysave_1.com',
2. '/home/mysave_2.plt' and '/home/mysave_2.com',
3. '/home/mysave_3.plt' and '/home/mysave_3.com', and
4. '/home/mysave_4.plt' and '/home/mysave_4.com'.

Save Combined Volume

This option allows the user to save the volume mesh partitions as a single volume mesh file. Although the aim throughout the environment is to operate on partitioned data sets without brining them back together, it is acknowledged that there maybe circumstances when a single mesh file is preferable for operations outside the environment. The user chooses a filename for the mesh in the normal manner and then the Mesh Servers co-operate in order to produce a valid single volume mesh.

Generate Surface

This option allows a surface mesh to be generated through the initiation of a CORBA-wrapped implementation of the FLITE Surface Mesh Generator. This module is then sent the necessary geometry data and sources in order for the mesh to be generated successfully. During the mesh generation process an information panel (Figure 181) appears showing the current progress of the generator as it passes over each geometry curve and then surface.

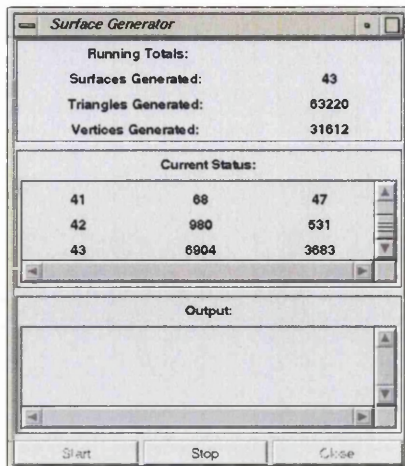


Figure 181 – The Surface Mesh Generation Information Panel

At any time during its execution, selecting the 'Stop' button terminates the execution of the generator and closes the panel. At the end of the generation the panel remains open until the 'Close' button is selected. This then causes the surface mesh to be passed directly back to the environment in a manner analogous to loading it from disk.

Generate Volume

Selecting this option causes a partitioned volume mesh to be generated in parallel. This is accomplished by initiating the Parallel Delaunay Mesh Generator on a specified set of computers, connecting it to the environment and then transmitting the necessary surface mesh and source data to it.

Once the volume mesh partitions have been generated, a set of Mesh Server objects are initiated on the same set of computers, connected to the environment and the volume mesh data sets passed to them in parallel. Once this has completed, the Mesh Generator processes are disconnected and terminated.

The input parameters to the generator are specified using the Volume Generation Control Panel (Figure 182). Once these have been specified, the panel is closed and the Parallel Platform Panel is opened to allow the user to select the computers on which the generator is executed. Once this has been done then the generator is initiated and a window opened to show the output during its execution.

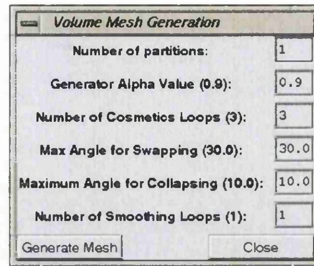


Figure 182 – Specifying the Parameters for the Parallel Delaunay Mesh Generator

Appearance

This option allows the user to change the rendering appearance of the various entities associated with the surface and volume mesh data sets. The appearance and operation of the Mesh Appearance Panel (Figure 183) is almost identical to that of the Geometry Appearance Panel described in Section 7.4. The only difference is that the curve and surface entities are replaced by the various mesh entities, e.g. surfaces, interfaces, cutting planes, iso-surfaces, etc.

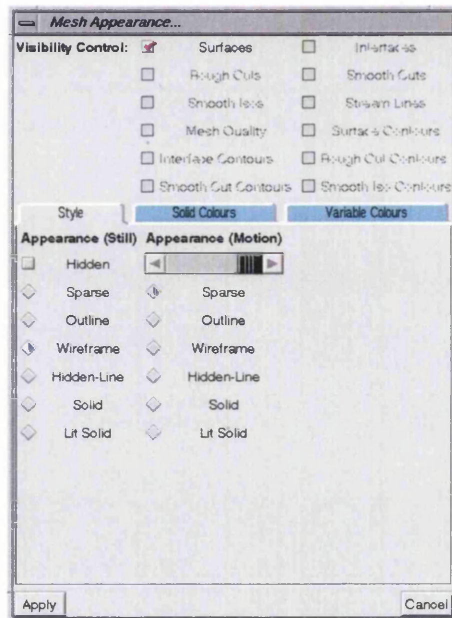


Figure 183 – The Mesh Appearance Panel

Colour

This option allows the user to change the colours of the various entities associated with the surface and volume mesh data sets. The appearance and operation of the Mesh Colour Panel (Figure 184) is almost identical to that of the Geometry

Colour Panel described in Section 7.4. The only difference is that the curve and surface entities are replaced by the various mesh entities, e.g. surfaces, interfaces, cutting planes, iso-surfaces, etc.

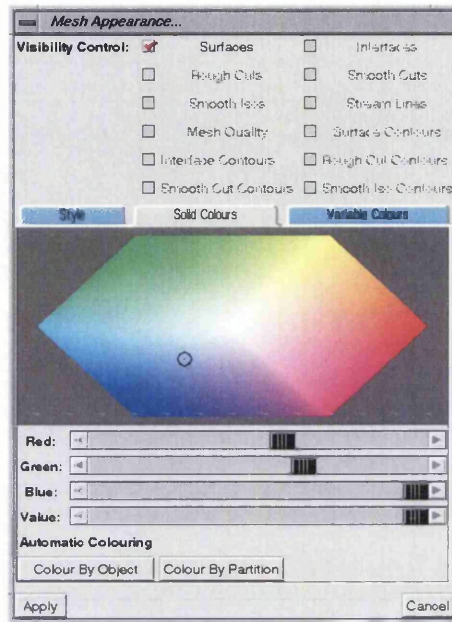


Figure 184 – The Mesh Colour Panel

Quality Analysis

This option allows the user to analyse the quality of a surface or volume mesh in parallel. When selected the user is presented with the Mesh Quality Analysis Panel (Figure 185).

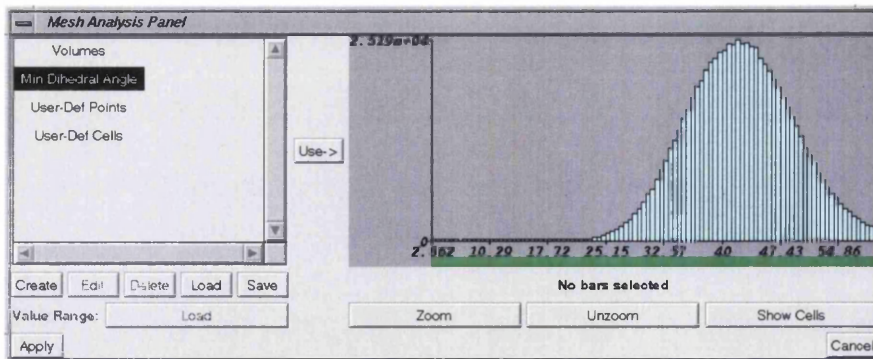


Figure 185 – The Mesh Quality Analysis Panel

This panel is divided vertically into two sections. The left section lists the available mesh quality measures. These are all either geometric or topological

measures that, together, give a good indication of how well a mesh will perform within a typical equation solver.

To analyse a particular metric, the user simple selects it from the list and then selects the 'Use' button. This instructs the Mesh Servers to compute the specified metric, in parallel, and return the values used to fill the histogram on the right. The histogram shows the range of values of the specified metric along the horizontal axis. This range is divided into 100 bars, whose heights are determined by the number of nodes/edges/faces/cells that fit within that sub-range.

Using this histogram, the user can perform one of three functions:

- Select a range of bars and then zoom into that range (Figure 186). This causes the histogram values to be recomputed by the Mesh Servers. The histogram is then updated to show the selected range of values represented by all of the 100 bars (Figure 187). This operation can be performed repeatedly in order to zoom in an ever-decreasing section of the histogram.

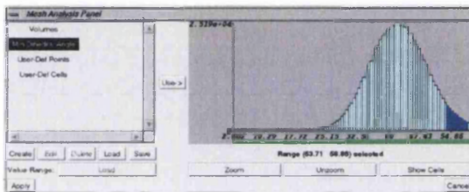


Figure 186 – Selecting a range of Histogram Bars

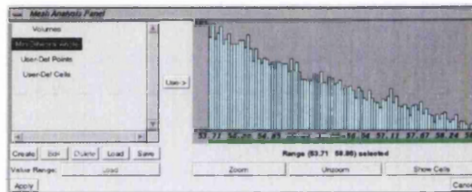


Figure 187 – The same Histogram zoomed into the selected range

- Unzoom the histogram back to a previous level. This effectively undoes the effects of the most recent zoom. This can be performed repeatedly until the histogram, once again, covers the full range of metric values.
- Highlight the individual nodes/edges/faces/cells whose metric value falls within the range of the selected histogram bars. This causes the selected entities to be highlighted within the actual mesh in the Main Window (). This allows the user to examine whether any elements of poor quality are in regions where the solution may be affected.

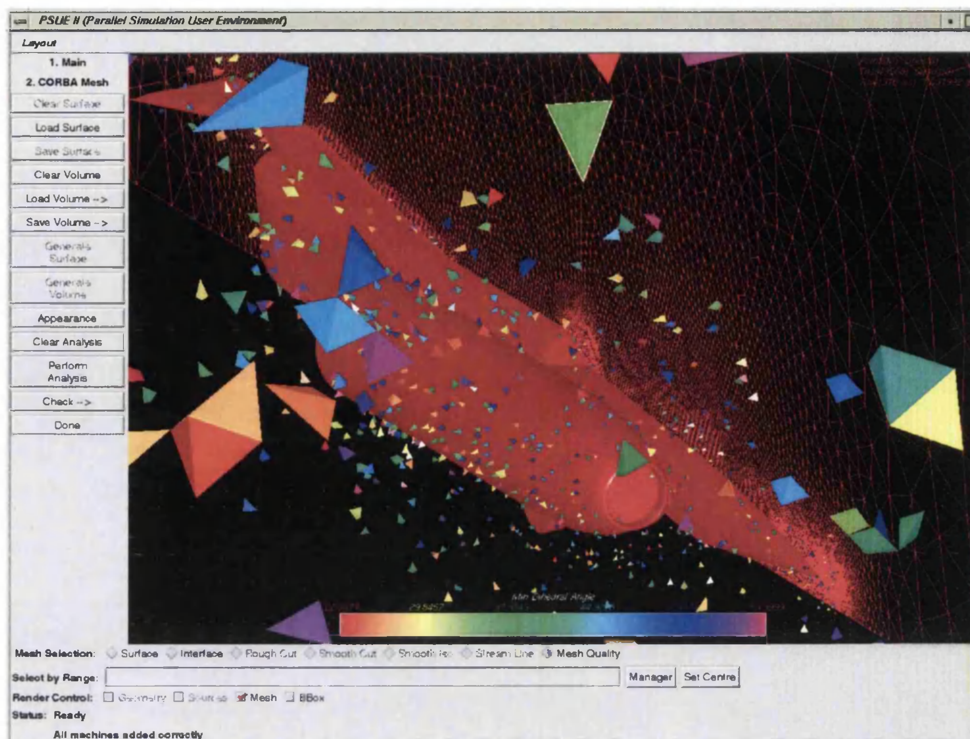


Figure 188 – Some Volume Elements highlighted in the Mesh

7.7. The 'Boundary Conditions' Toolbar

The Boundary Conditions Toolbar contains the following options:

Clear

The Clear option causes any Boundary Conditions currently stored within the environment to be removed.

Load

The Load option allows the user to load a pre-defined set of boundary conditions from a file selected from a File Selection Panel.

Save

The Save option allows the set of boundary conditions contained within the environment to be saved to a file selected via the File Selection Panel.

Edit

This option allows the user to set up / create a set of boundary conditions for a particular geometry through the use of the Boundary Condition Editor Panel (Figure 189).

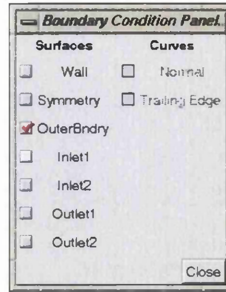


Figure 189 – The Boundary Condition Editor Panel

This panel is divided horizontally into two sections. The left section allows the user to apply boundary conditions to the various geometrical surfaces and the right second allows the user to flag certain geometry curves as sharp edges.

When the panel is opened and no boundary conditions exist then a complete set are created, with a *solid* boundary condition being applied to all surfaces and flagging all curves as not being sharp edges. These defaults were chosen since in most cases the majority of the geometry will have these features applied with only a few curves and surfaces being flagged differently.

To alter the boundary conditions, the user must select the required geometrical surfaces in the main window. This then enables the boundary condition toggles to allow the user to select a boundary condition to apply. If all of the selected surfaces currently have the same boundary condition then the toggles in the panel reflect this, otherwise they remain blank to show that surfaces with different boundary conditions are currently selected.

Flagging geometry curves as sharp edges (often referred to as trailing edges) is performed in a similar manner. The user, in the main window selects the curves, and then the required toggle is chosen.

The geometry and mesh can also be coloured according to their boundary condition settings if the appropriate toggles are set in the Geometry and/or Mesh Appearance Panels.

7.8. The 'Solution' Toolbar

The Solution Toolbar contains the following options:

Clear

This option causes the solution to be removed from the environment. This instructs the Mesh Servers to free the solution values and then recreate the Render Objects of the mesh. These are then passed to the Master process for display.

Load

This option allows an existing solution to be loaded into the environment. A File Selection Panel (Figure 171) appears to allow the user to select one of the partitions solution files. This is then replaced by a second panel to allow the user to place a '#' character in place of the partition number in a manner identical to loading a set of volume mesh partitions. These filenames are then passed to the set of Mesh Servers, which then load the solution files. These servers then recreate the Render Objects and send them back to the Master process for display.

Save

This allows the solution currently stored within the environment to be written to disk as a set of partition solution files. The filename is chosen via a File Selection Panel (Figure 171) and then sent to each of the Mesh Servers, in parallel, to save the files.

Execute Solver

This option allows the user to execute a parallel CFD solver on the volume mesh data sets using the specified boundary conditions.

The input parameters to the solver are specified using the Solver Control Panel (Figure 190). Once these have been specified, the panel is closed and the Parallel Platform Panel is opened to allow the user to select the computers on which the solver is executed. Once this has been done then the solver is initiated on the specified computers and connected to the environment. The required mesh and boundary condition data sets are then transmitted to it in parallel. During its execution, depending on the users choice, the output from the solver processes are either logged in a file or echoed into a set of windows opened on the desktop.

When the solver has the specified number of time-steps, the solution data is then transmitted back to the environment and the solver processes are disconnected and terminated. The Mesh Servers then recreate the Render Objects to include the solution values and send them back to the Master Process for display.

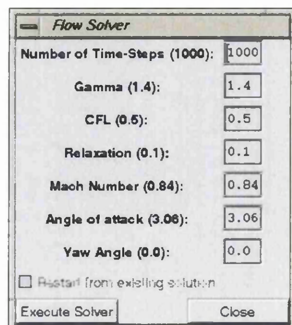


Figure 190 – The Solver Control Panel

Change Variable

This option allows the user to choose which of the solution variables is used to colour the mesh. This is achieved by selecting the required variable in the list in the Variable Selection Panel (Figure 191).

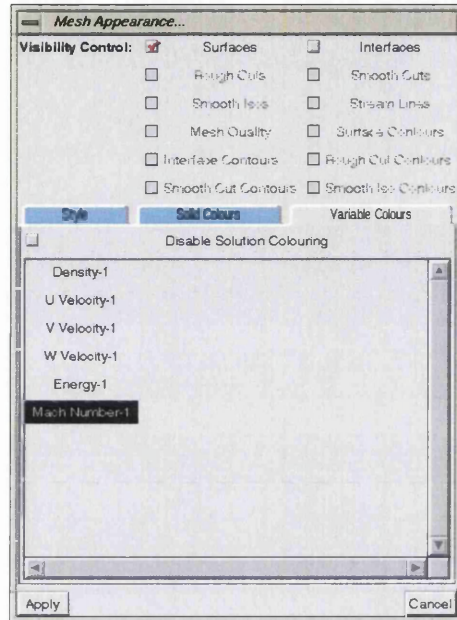


Figure 191 – The Variable Selection Panel

Selecting the 'Apply' button causes the Mesh Servers to recreate the Render Objects with the new variables' values and then send them back to the Master process for display.

The Variable Selection Panel also allows the creation of user-defined variables based upon a combination of the generic variables produced by the solver and geometric features of the mesh. Selecting the 'Create' button opens a sub-panel that contains three fields to be filled by the user:

- The new variables name,
- A short mnemonic for the new variable. This would be used when referencing this variable during the creation of another, higher-level variable.
- A mathematical expression describing how this variable is to be computed from the generic variables. This mathematical expression is entered in a similar style to the C programming language. For details, see Appendix A.

Set Range

This option allows the user to alter the range of solution values that are mapped to the colour scale. By default, the minimum solution value is mapped to the red end

of the scale and the maximum mapped to the magenta end. However, it is often the case that a few rogue solution values cause the rest to be mapped into a small section of this scale causing a loss of detail and, generally, a washed out appearance. Using this option, the range of values mapped to the colour scale can be narrowed thus providing much more information. The solution values that then fall outside this range are clamped to the appropriate end.

The user achieves this by sliding the bars representing the minimum and maximum solution values until the required range has been achieved. Selecting the 'Apply' button then causes the colours on the mesh in the Main Window to change to reflect the new settings. Figure 192 shows the Mach number on an aircraft using the default solution range of 0.015 – 2.99. Figure 193 shows the same variable but with the range reduced to 0.31 – 1.06.

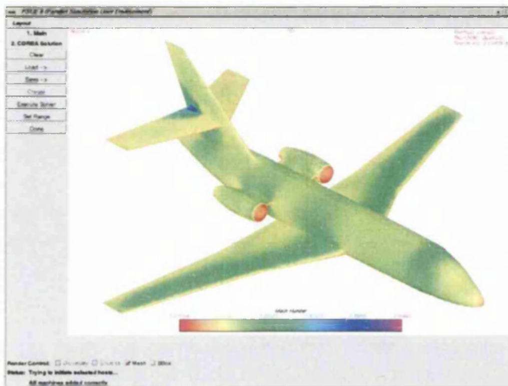


Figure 192 – The Default Solution-Colour Mapping

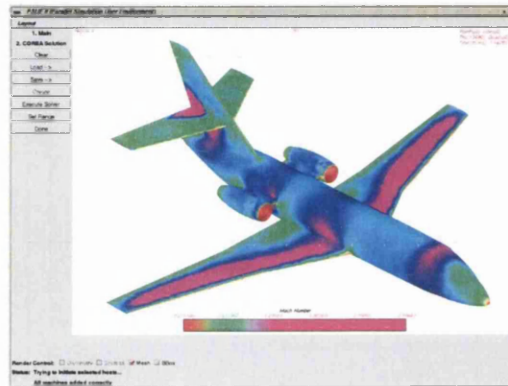


Figure 193 – The User-Defined Solution-Colour Mapping

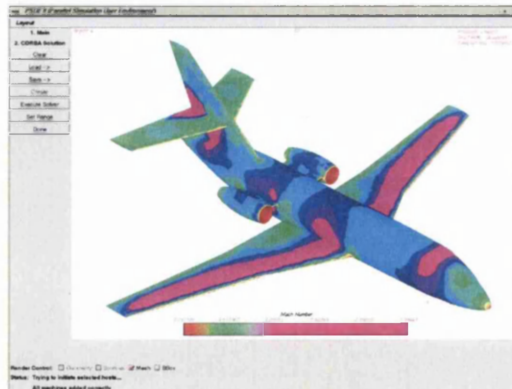


Figure 194 – The User-Defined Solution-Colour Mapping with Contouring

7.9. The 'Post-Processing' Toolbar

The Post-Processing Toolbar contains the following options:

Cutting Planes

Selecting this option opens the Cutting Plane Panel (Figure 195) to allow the user to define/edit cutting planes through the mesh.

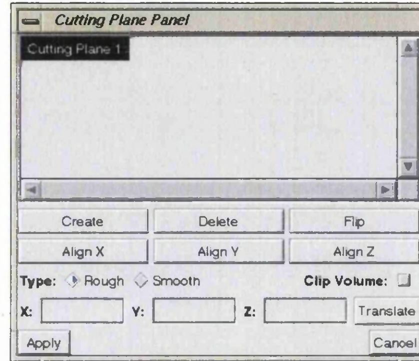


Figure 195 – The Cutting Plane Panel

Using this panel the user is able to define either rough or smooth cutting planes. Rough cutting planes are defined as the set of faces bounding the set of elements placed wholly on the correct side of the plane. Smooth cutting planes are defined as the faces created through intersecting each element by the cutting plane. These are shown in Figure 196 and Figure 197.

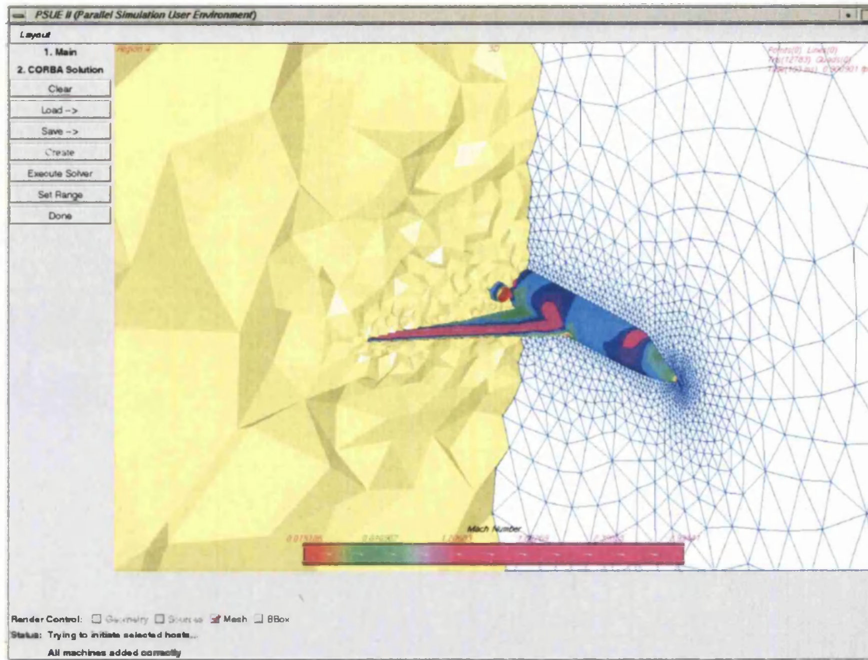


Figure 196 – A Rough Cutting Plane

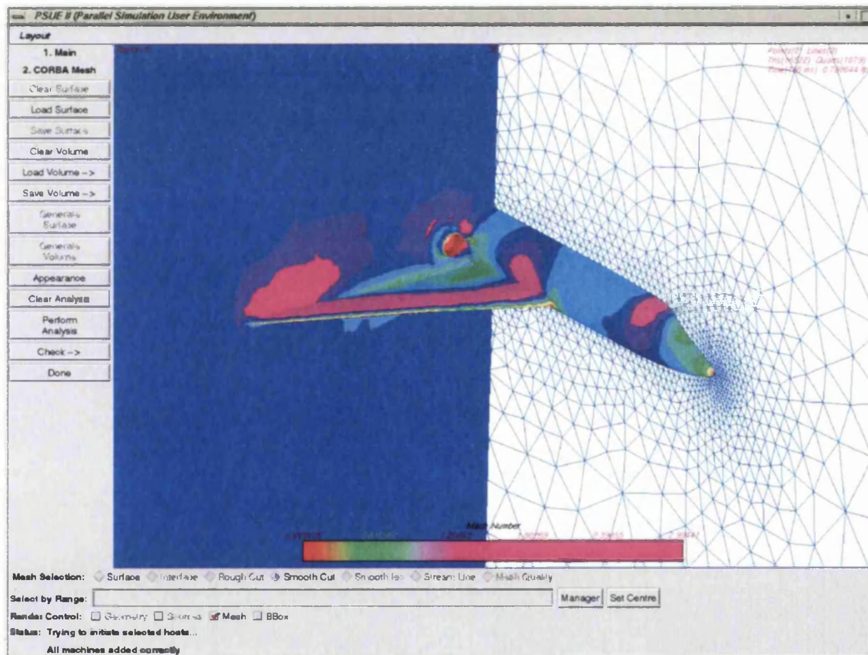


Figure 197 – A Smooth Cutting Plane

The cutting planes currently defined are listed in the panel and shown in the main window as rectangles at various orientations. Selecting one of these causes the associated rectangle to be highlighted. This can then be manipulated with the mouse (translated, scaled and rotated) in the same manner as the mesh. Whilst this

is being performed the mesh is clipped in real-time using the Open-GL clipping mechanism. Selecting the 'Apply' button causes the cutting plane to be fixed and the various Mesh Server objects work, in parallel, to produce the set of faces representing that cutting plane. When they have finished these faces are then displayed.

Selecting one of the 'Create X', 'Create Y' or 'Create Z' buttons creates a new cutting plane. These create a new cutting plane that is aligned with the constant *x*-, *y*- or *z*-axis. This alignment is purely a starting point since the user may subsequently manipulate the plane.

Iso-Surfaces

This option opens the Iso-Surface Panel (Figure 198) through which the user may create any number of iso-surfaces.

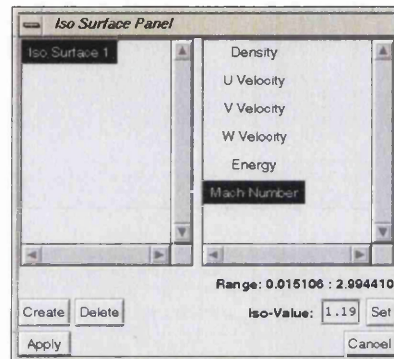


Figure 198 – The Iso-Surface Panel

This panel contains a list of currently defined iso-surfaces, each of which may be selected for editing. Selecting the 'Create' button creates a new iso-surface based on the first solution variable with a value placed at the centre of the range of that variable.

An iso-surface may be edited by simply selecting the solution variable on which the iso-surface should be based and then entering the value that the iso-surface should represent. To aid the user, the range of values of the selected solution variable is also displayed in the panel.

Selecting the 'Apply' button causes the Mesh Server objects to recreate, in parallel, all of the necessary primitives used to render the current set of iso-surfaces. These are then passed back to the Master process to be displayed. Figure 199 shows an example of rendering an iso-surface representing Mach 1.0 over a small business aircraft.

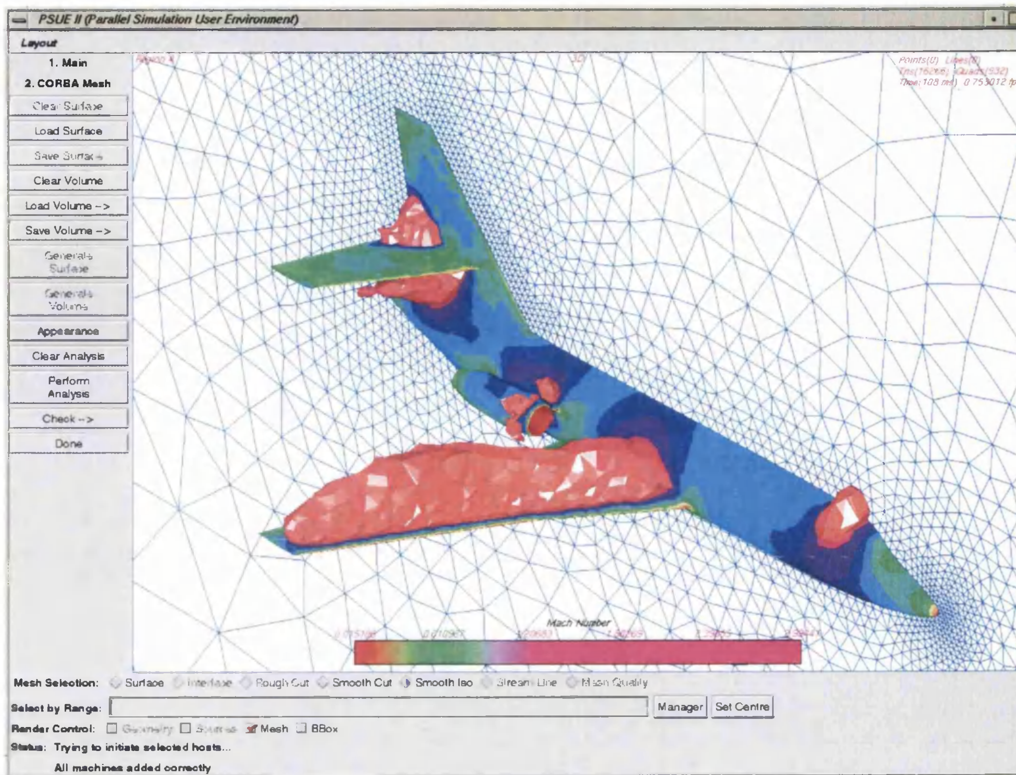


Figure 199 – An Iso-Surface of Mach 1.0 over the Gulf-Stream

Chapter 8. ADDRESSING THE ISSUES OF SOFTWARE PORTABILITY

As computational simulation moves from the traditional research environment into the commercial design and manufacturing environment, the issues of software portability become fundamentally important.

For tools that are designed to run on high-performance computing platforms, the range of computers on which the codes may be used is quite extensive ranging from traditional parallel super-computers, through traditional UNIX workstations and Linux based PCs to Microsoft Windows based PCs.

For small Fortran 77 based codes the issues of portability are usually quite simple to overcome. This is, in most parts, due to two factors:

- The Fortran 77 language has been stable for a long number of years,
- Most Fortran 77 programs remain completely within the environment of the language itself. They often do not interface with the underlying Operating System in any way.

These two factors mean that any differences between the various underlying Operating Systems are hidden from the program completely.

This can also be true when programs use the ANSI C language. Again, the language has been stable for a number of years and, for simple tasks, the program can stay within the confines of the language itself and, thus, be protected from the underlying Operating System.

However, many programs written using C are not able to stay within the confines of the language and must interact directly with the libraries of subroutines provided by the Operating System. Examples of these types of programs include:

- Programs that create Graphical User Interfaces (GUIs),
- Programs that perform any graphical rendering (two-dimensional or three-dimensional),
- Programs that need to perform any other operations that are not provided within the C language.

The difficulties involved in ensuring these types of programs remain portable across different platforms depend heavily on the range of platforms required. If the portability of programs can be limited to computers based around a UNIX Operating System then this can be achieved with a little effort and forward planning. This is because modern versions of UNIX are based on one of two standards, BSD from Berkeley and System V from AT&T, with most incorporating the functionality of both. However, some vendor

specific functionality is still present in almost all versions of UNIX and care must be taken when this functionality is used.

If the range of computers on which the programs must run include the Microsoft Windows range of Operating Systems then the issues of portability become much more difficult since there are almost no common sets of subroutines with UNIX.

For programs coded using the C++ language then the problems of ensuring portability include all those found when using the C language. However, since the C++ language is relatively new there tend to be incompatibilities between the various vendors' implementations of the actual language itself. This means that even if a program stays within the confines of the language, portability is in no way guaranteed.

For a large suite of programs such as PROMPT and the PSUE II, which include GUI's, three-dimensional graphics and frequently interface with the underlying Operating System, the difficulties of ensuring portability are numerous. The rest of this chapter describes the various incompatibilities that were encountered during development. These are ordered according to the difficulty of overcoming them:

- Language features,
- Graphics,
- Threading Interface,
- Input / Output (I / O),
- Inter Process Communication (IPC) and
- Graphical User Interfaces.

8.1. Language Features

During the development of both PROMPT and the PSUE II, by far the most portable language out of the three was C, where no problems were encountered on any platform. Fortran 77 had a number of portability issues, especially when mixing it with other languages in the same program. The two most common ones were dynamic memory allocation and subroutine name mangling.

The C++ language has only recently adopted a standard and thus has the most portability issues, which are due mainly to the different degrees with which the various compilers have managed to adapt at this stage. The two main features of C++ that have the most portability problems are templates and run-time type identification (RTTI). Of these, the functionality of RTTI can be emulated simply within the code and thus can be safely ignored. However, the functionality inherent with templates is a very powerful feature and, thus was considered too worthy not to use.

8.1.1. Fortran 77 Pointers

Fortran 77 has no intrinsic mechanism for dynamic memory management. This feature was considered essential in user-friendly codes such as PROMPT and the PSUE II. In order to overcome this limitation, an extension to the language, originally conceived by

CRAY, allowed the use of 'C' like pointers. These could then be used to allocate memory at any point in the program and then use this memory as if it was a normal, static array. This extension has since been widely adopted by all of the commercial compilers for both UNIX and Windows. The only exception to this is the GNU implementation of Fortran, g77, which is widely used under the Linux OS. However, this limitation could be overcome through the use of commercial Linux compilers that do support the extension.

8.1.2. Fortran 77 Subroutine and Variable Names

Unlike C and C++, the Fortran 77 compiler performs some *name mangling* on subroutine and variable names during the compilation process. This is normally completely transparent to the user unless the Fortran routines are mixed with routines from C or C++. The way a name is mangled is dependent on the compiler. Typical examples are given in the table below.

Compiler	Description of Name Mangling	F77 Label	C Label
SGI (f77) / Intel (ifc) / Solaris (f77) / DEC (f77)	Converted to lower-case and underscore appended.	SubRoutine1	subroutine1_
CRAY (f77)	Converted to upper-case	SubRoutine1	SUBROUTINE1
HP (f77) / IBM (f77)	Converted to lower-case	SubRoutine1	subroutine1
GNU (g77)	Converted to lower-case and two underscores appended	SubRoutine1	subroutine1__

As can be seen, most of the name mangling is reasonably trivial to overcome. However, there is an exception to this rule. When using the combination of Microsoft Visual Studio and Compaq Visual Fortran, the name mangling is somewhat more complex. For example, a Fortran 77 subroutine declared as:

```
subroutine DoThis( j, a )
```

needs to be written as:

```
subroutine __syscall DoThis( j, a )
```

and is called from C or C++ as:

```
dothis@8( &j, &a )
```

Passing a pointer to `j` and `a` is standard practise due to Fortran always passing variables by reference. However, the '`@8`' sequence is derived from the fact the two arguments to that function take a total of eight bytes.

In order to overcome these difference in a neat manner a Macro Processor, called m4, was used during the compilation process to pre-process the C, C++ and Fortran 77 source code in order to ensure any subroutine and variable names were correct.

8.2. Graphics

During the early development of PROMPT, a standard for three-dimensional graphics was emerging called Open-GL. This was supported to varying degrees by most of computer vendors. For those that had not supplied a native implementation for their hardware, a free software-based implementation was available called Mesa.

The core of Open-GL was designed to be portable across all implementations on all platforms and during the development of both PROMPT and the PSUE II no portability problems were encountered. Any extensions added to the library by a particular vendor were clearly marked as such by appending an acronym identifying the vendor. For example, `glBegin()` is a standard, core subroutine whereas `glTexImage4DSGIS()` is identified by the SGIS suffix as an SGI extension. This system easily identifies all non-standard functionality, so as not to be used if portability is intended.

However, the Open-GL standard does not encompass the interaction between the three-dimensional graphics functionality and the underlying windowing system since this is inherently non-portable. These differences will be covered under the 'Graphical User Interface' section below.

8.3. Threading Interface

It is often desirable to have the ability to execute more than one thread of execution in a given executable. These threads would execute concurrently[17], all being able to access the same memory spaces if only one thread was executing.

During the development of PROMPT creating such threads was vendor specific. For example, SGI used a feature called Shared Processes (or *sproc*) whereas Sun used a feature called Solaris Threads. Each of these thread variants had a number of features in common, such as sharing the memory of the process, being initiated by specifying a subroutine name that is to be run as a separate thread. However, a number of important differences remained. For example, threads under the SGI variant had unique process identifiers whereas the other vendors did not, some variants allowed threads to be suspended and resumed at any time and the API to each of the vendors thread implementations was unique.

Soon after the PROMPT project was completed, most vendors adopted a standard called POSIX threads. This provided a simple and portable interface to the multi-threading capability of the OS. This has been used throughout the PSUE II development where no portability issues have arisen. A simple example of its use is shown below.

[17] As with normal multi-tasking, a single processor computer would emulate concurrency via time-sharing whereas on a multiple processor computer each thread may, indeed, operate concurrently.

```
void my_subroutine( void *pArgument )
{
  /* Do something as another thread */
}

int main( int argc, char *argv[] )
{
  pthread_t  threadID;

  /* Start subroutine as another thread */
  error = pthread_create( &threadID, my_subroutine, NULL,
NULL );

  /* Do some other processing while thread is running */

  /* Wait until thread finishes */
  pthread_join( &threadID, NULL );

  /* End program */
}
```

8.4. Input / Output

Performing formatted I/O in any of the three languages is completely standard with no portability problems either between computers or between languages. The only issue arising is when reading and writing floating point numbers since the conversion between binary and decimal representations ultimately leads to small losses of accuracy. However, this usually only affects the first or second least significant digit.

For programs, such as the PSUE II, which deals with large files, formatted I/O[18] is too slow and produces files that are very large. For these operations, unformatted I/O[19] is necessary. However, unformatted I/O has some portability issues due mainly to the differences between how numbers are stored internally inside different computers.

8.4.1. Unformatted I/O between Fortran 77 and C

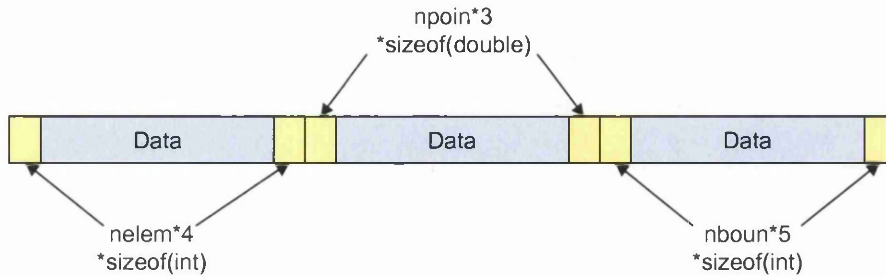
When performing unformatted I/O in Fortran 77, the file is transparently sub-divided into records. For example, given the following Fortran routine:

```
write(20) ((ielem(j,i),  i=1,nelem),j=1,4)
write(20) ((coorp(j,i),  i=1,npoin),j=1,3)
write(20) ((iboun(j,i),  i=1,nface),j=1,5)
```

[18] Formatted I/O reads and writes files using ASCII files that are human readable.

[19] Unformatted I/O reads and writes files using the computers native binary format that is not human readable.

the following file is produced:



This shows that the data is written out surrounded by record delimiters. These record delimiters are four bytes long and store the length, in bytes, of the data.

When reading and writing in Fortran 77, these delimiters are dealt with transparently. However, in C these records need to be read and written explicitly. These record delimiters also cause a problem when individual records are longer than 2GBs (2^{31}-1 bytes) since the ability to store the data size in an integer is no longer possible. This problem has yet to be overcome by many compiler vendors.

8.4.2. Portability Issues due to Big and Little Endian Computers

Although modern computers mostly use the standard 2's complement format for storing integers and the IEEE standard for storing floating point numbers, there is still an incompatibility between how computers represent any multi-byte quantity. Two standards encompass all common computers: big endian and little endian.

The big endian representation stores numbers in the intuitive manner with the left-most bytes being most significant, whereas the little endian representation stores numbers with the right most bytes being most significant. Figure 200 illustrates both representations. Most modern RISC computers store numbers using the big endian format with the little endian format being used almost exclusively by the DEC Alpha and Intel processors.

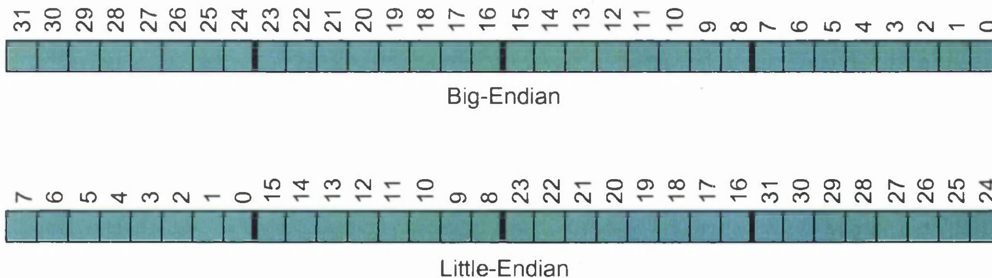


Figure 200 – Byte layout for a 32-bit quantity

This problem was overcome by always maintaining any unformatted files using the big-endian representation. Byte swapping routines were then defined as C++ inline functions so as not to introduce any function call overhead. For platforms using the little-endian

representation, these routines would swap the bytes during the I/O operations. For the platforms using the big-endian representation, these routines were defined to be empty subroutines with no operations and would therefore be removed during the optimisation stage of the compilation process.

8.5. Inter-Process Communication

As mentioned previously, the inter-process communication system used during the development of PROMPT was based around UNIX sockets. This communication mechanism is supported in a portable fashion by all of the UNIX vendors. The only portability issue on UNIX platforms was when two processes communicate on two computer platforms with different endian representations, but this could be overcome using the same method as above. Although Microsoft Windows does not support UNIX sockets as a native functionality a third-party library, called WinSock (available from Microsoft and the public-domain), does provide the same functionality and interface thus eliminating potential portability problems.

During the development of the PSUE II, it was decided to use communication mechanisms that abstracted away from the native UNIX sockets, and thus hid any portability issues for all platforms. For communication between the slave processes the MPI library was used and for master-slave communication, CORBA was used. Both of these libraries have, as one of their major requirements, portability across the full range of UNIX and Windows platforms.

8.6. Summary

The sections above have described the individual difficulties encountered whilst porting both PROMPT and the PSUE II to a number of UNIX platforms. In addition to these points, decisions were made throughout the implementation of both environments to restrict the use of system subroutine calls to those that are generally regarded as being standard.

Unlike with UNIX systems, where OSF/Motif is fully supported, when Microsoft Windows is considered, then portability issues involving the actual graphical user interfaces (i.e. windows, menus, buttons, etc.) become very complex. In order to provide this functionality the use of third-party solutions is recommended. These include:

- Exceed from UniPress Software Inc. that provides a means by which UNIX applications can be displayed on a Windows platform.
- WxWindows (www.wxWindows.org) which provides a freely available, cross platform GUI library that provides a portable interface to the native GUI libraries of the various platforms. GUI libraries supported by wxWindows include MS Windows 95/98/Me/NT/2000/XP, Linux GTK, OSF/Motif and Apple Mac.

Chapter 9. EXAMPLE TEST-CASES

9.1. Explanation of Test-Cases

In order to show the operation and functionality of the PSUE II, three test cases were chosen:

- *CFD Simulation over a Dassault Falcon*
  The emphasis with this test case was to illustrate the majority of the functionality of the PSUE II, from geometry repair operations and creation of sources, through to mesh generation, quality analysis, flow simulation and post-processing.
- *CFD Simulation over a complete F16 configuration*
  The purpose of this test case was to illustrate the typical sequence of operations a user may perform if presented with a topologically valid geometry but with no outer boundary. The actual geometry is far more complex than the Falcon and the meshes involved are over six million elements.
- *Pre-processing of a Grand Challenge Simulation over a Dassault Falcon*
  The purpose of this test case is to show how the parallel architecture of the PSUE II enables the user to manipulate very large meshes; in this case, approximately half a billion elements.

For each of the test cases, the process is divided into a number of stages, each dealing with one of the major data sets involved. For each stage, the operations performed are described along with the approximate times taken which include all user interactions. For example, the time taken to generate a surface mesh begins when the button on the toolbar in the PSUE II is selected and ends when the mesh appears in the main display window. For this reason, all of the times are approximate since, for smaller test cases, the speed of the user could have a significant effect when compared to the execution time of the algorithm.

All parallel computations, including the slave processes of the PSUE II were performed on an SGI Onyx 3800 with 64Gb of memory and 32 R14000 processors running at 500MHz.

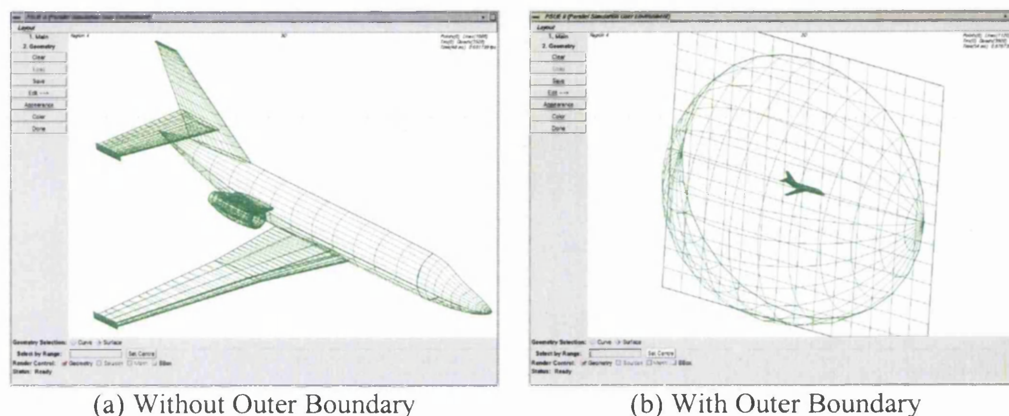
9.2. CFD Simulation over a Dassault Falcon

This test case involved performing a CFD simulation using a reasonably small mesh of approximately half a million elements. The geometry, although known to be topologically valid, has had all of the topological information removed, and its outer boundary removed. The purpose of this was to illustrate the pre-processing functionality of the PSUE II in setting up the geometry, background grid and sources for mesh generation and flow simulation.

Geometry

The main pre-processing options that were necessary in order to prepare this geometry for meshing were to recreate the topological information (i.e. which curves are attached to which surfaces) and the outer boundary.

The first operation was to use the Outer Boundary Creation panel (described in Section 7.4.1) to create a half-sphere outer boundary and symmetry plane, with the symmetry plane attached to the correct side of the aircraft. Figure 201 shows the before and after appearance of the geometry.



(a) Without Outer Boundary (b) With Outer Boundary

Figure 201 – Creation of the Outer Boundary

This operation automatically attached the relevant geometry curves to the symmetry plane so this would not need to be manually done later.

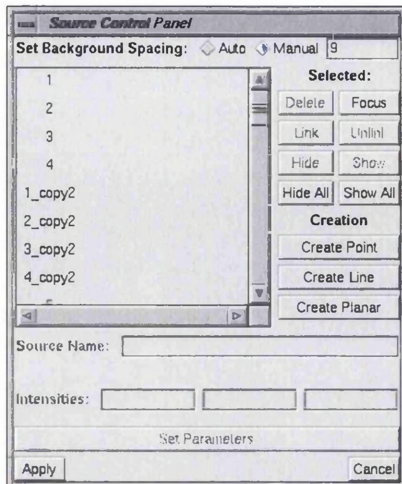
The second operation was to use the Edit Topology panel (also described in Section 7.4.1) to attached the remaining curves to the surfaces.

These two operations took approximately 1.5 hours to complete.

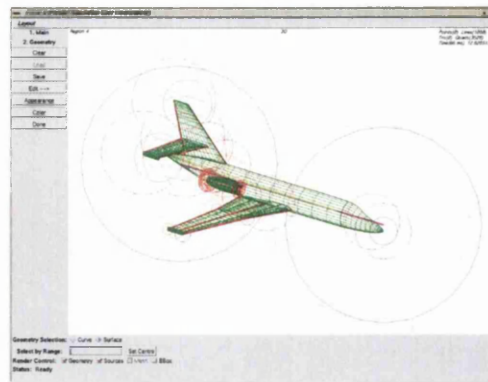
Background Grid and Sources

Once the geometry was valid, the next operation was to create the sources that would increase the mesh density in certain key areas of the geometry, such as the leading and trailing edges of the wings, and the fuselage.

This was performed using the Edit Sources panel (described in Section 7.5). The panel and resultant sources are shown in Figure 202. Although, there is feedback on how the source strength will affect the mesh density, it was still necessary to generate two test surface meshes in order to be satisfied with the result.



(a) The 'Edit Sources' Panel



(b) The resultant sources

Figure 202 – Creation of the Sources

This operation, including the generation of the two test surface meshes, took approximately two hours.

Surface and Volume Meshes

During the construction of the sources, the surface mesh was generated three times with the last mesh being used for the volume generation. The final surface mesh consisted of 12,366 triangles and 6183 nodes and took under one minute to generate. The volume mesh was generated using eight processors in less than 2 minutes and comprised 304,374 tetrahedra and 55,216 volume nodes. Figure 203 shows the surface mesh and Figure 204 shows the interface surfaces of the volume mesh as produced by the parallel Delaunay mesh generator.

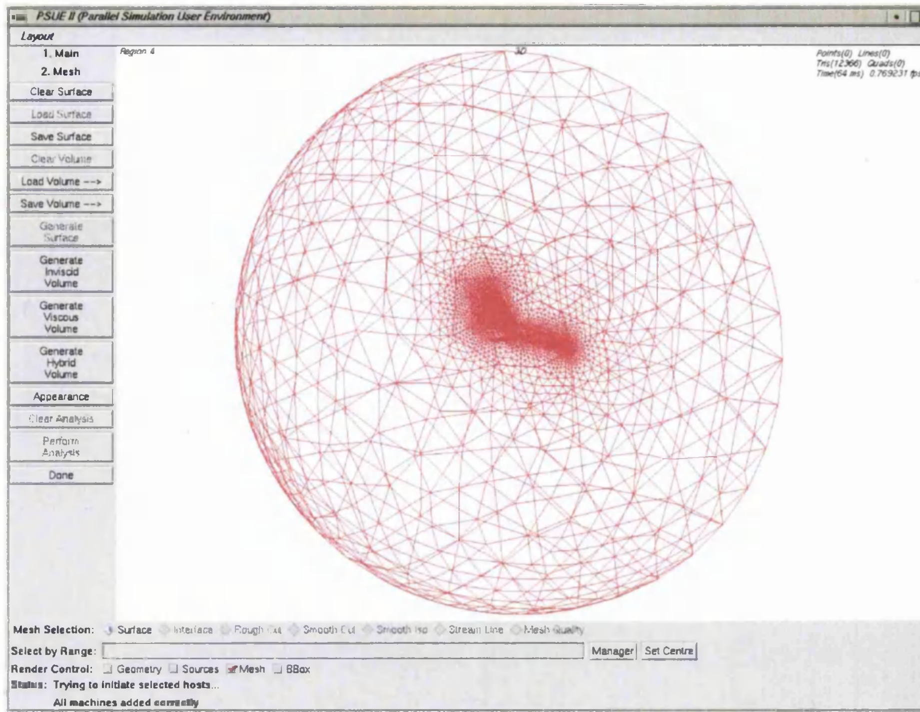


Figure 203 – The Surface Mesh of the Dassault Falcon

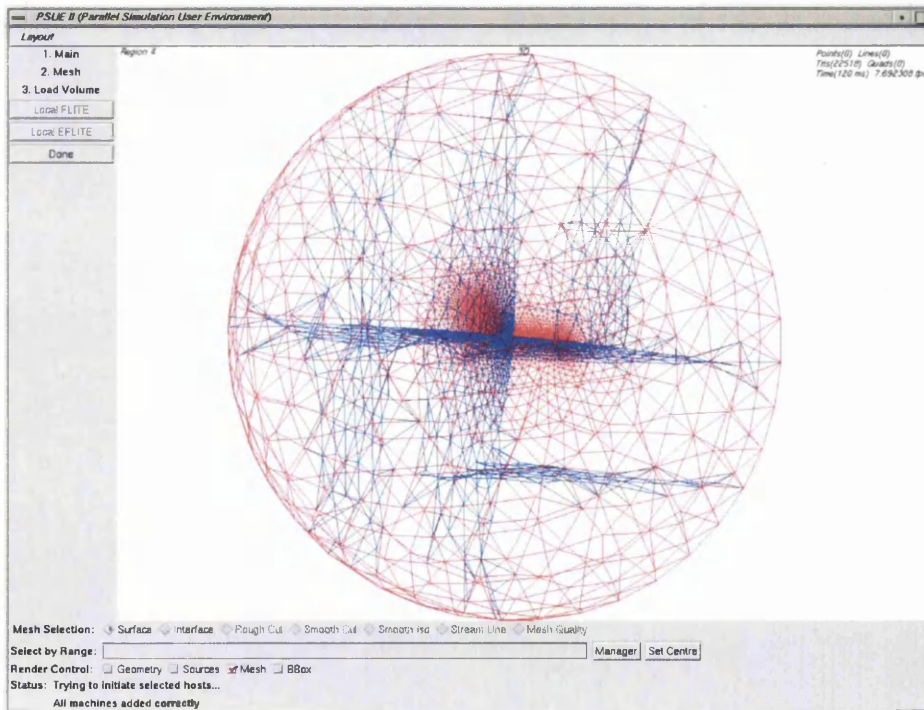


Figure 204 – The Volume Mesh (with interface surfaces)

Mesh Quality Analysis

The quality of the volume mesh was analysed by looking at the minimum dihedral angle for each element. The histogram, shown in Figure 205, shows the angle along the x-axis and the number of elements that fall within each range along the y-axis.

Elements with angles up to approximately 12° were then selected and highlighted in the mesh. As can be seen in Figure 206, most of the flatter elements were produced on the ends of the wings. Since this is due to the fact that the geometry surfaces in these regions meet at an acute angle, the mesh is deemed suitable for the solver. This process took under one minute to complete, although this does not include the time for the user to analyse the highlighted elements once displayed.

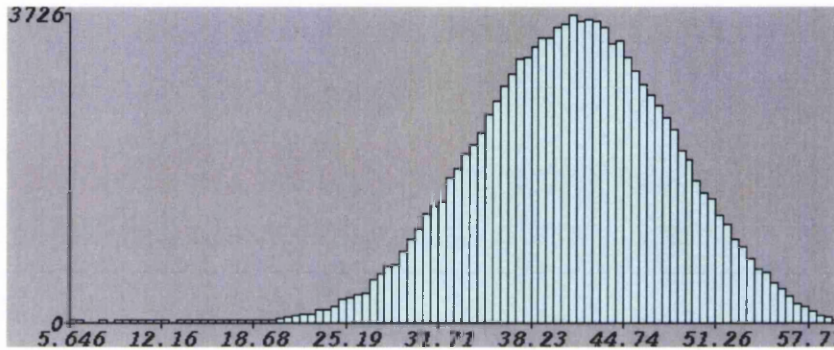


Figure 205 – The Mesh Quality Graph of the Falcon Mesh

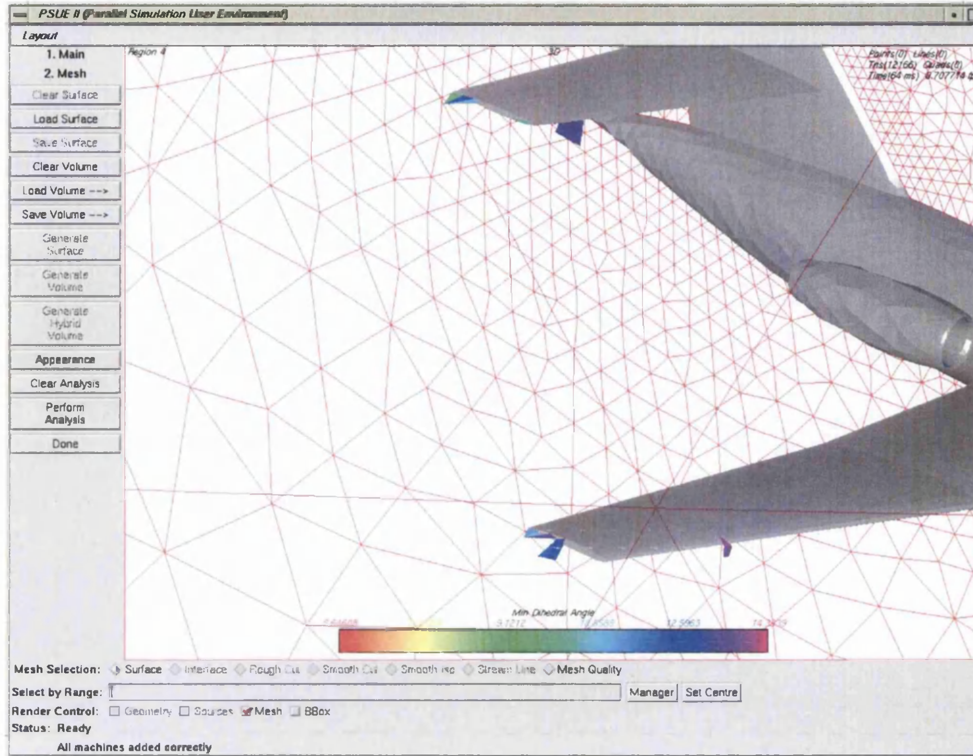


Figure 206 – Highlighting the *flat* elements within the mesh

Boundary Conditions

The boundary conditions were trivial to apply. Firstly, all of the surfaces were selected and a wall boundary condition was applied. Then the symmetry plane was selected and the boundary condition changed to be a symmetry. The same was then done for the outer boundary. The last operation involved the selection of the intersection curves on the trailing edges of the two wings and the fin. These were then flagged as being trailing edges in order that the intended solver would treat them accordingly.

This operation took approximately ten minutes.

Flow Solution

The flow simulation was performed using eight processors and converged in less than five minutes. The resultant solution is shown using colour plots (Figure 207) and contours (Figure 208).

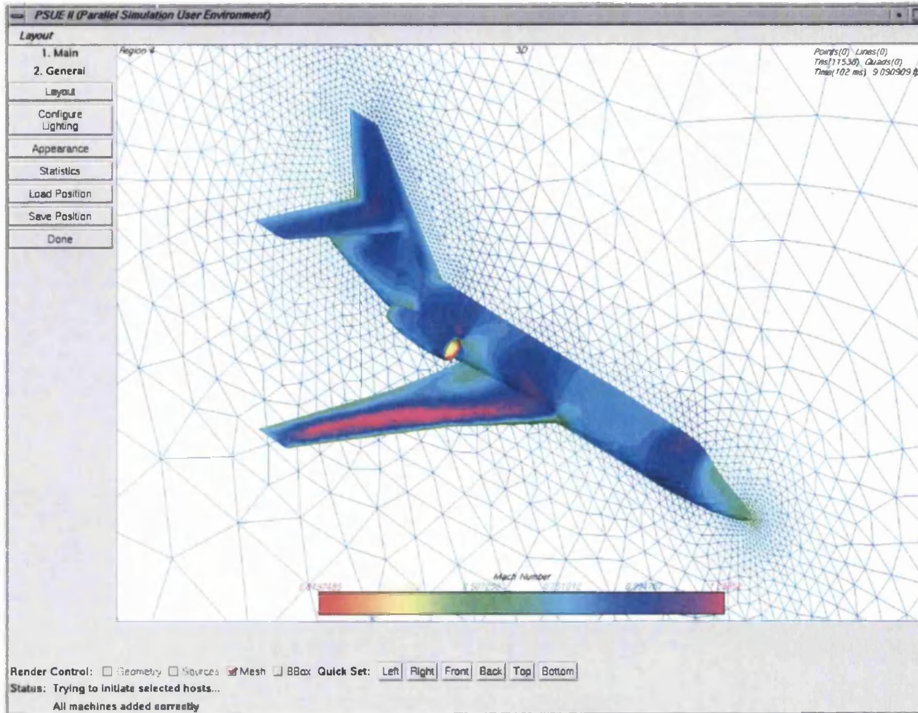


Figure 207 – Solution Colours of Mach Number

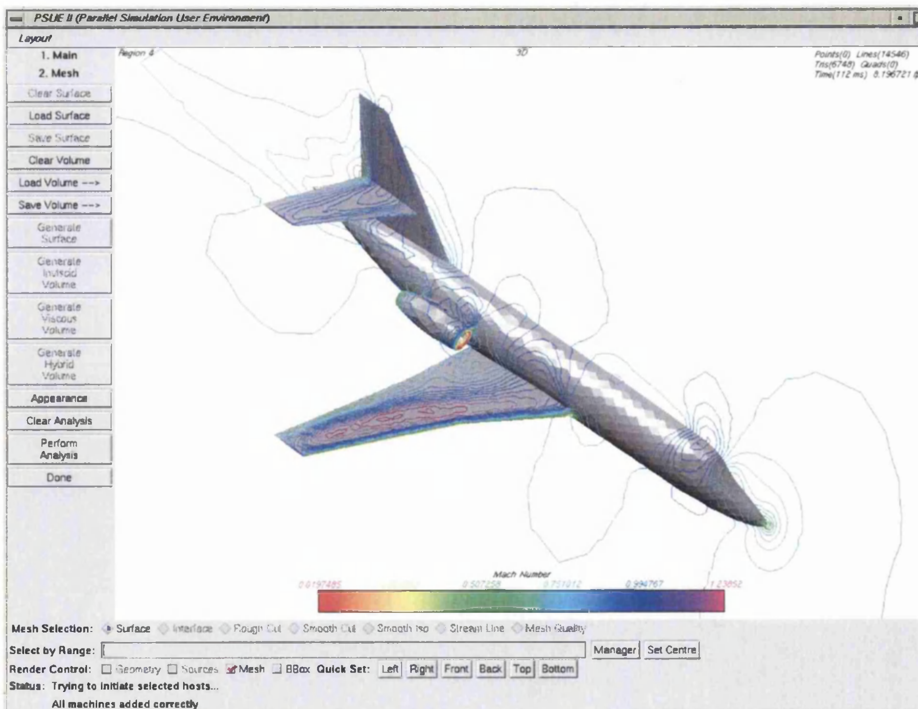


Figure 208 – Solution Contours of Mach Number

Post-Processing

Figure 209 shows a cutting plane across the wing of the aircraft and Figure 210 shows an iso-surface of Mach 1.0. The time taken to produce these features was under 30 seconds in both cases.

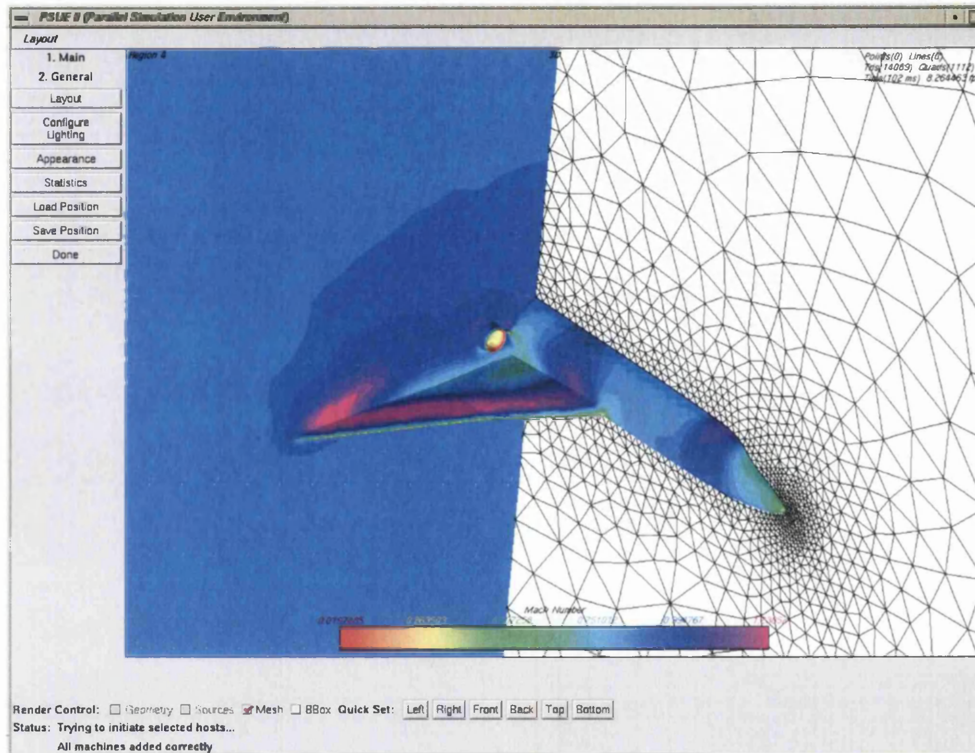


Figure 209 – A Cutting Plane through the Falcon

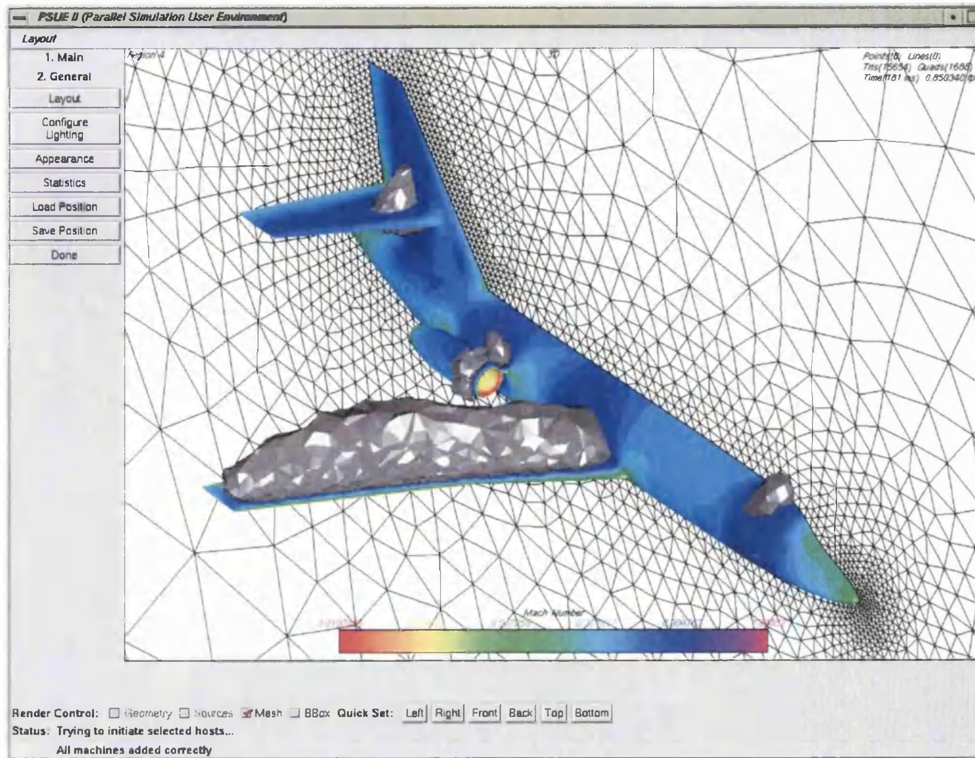
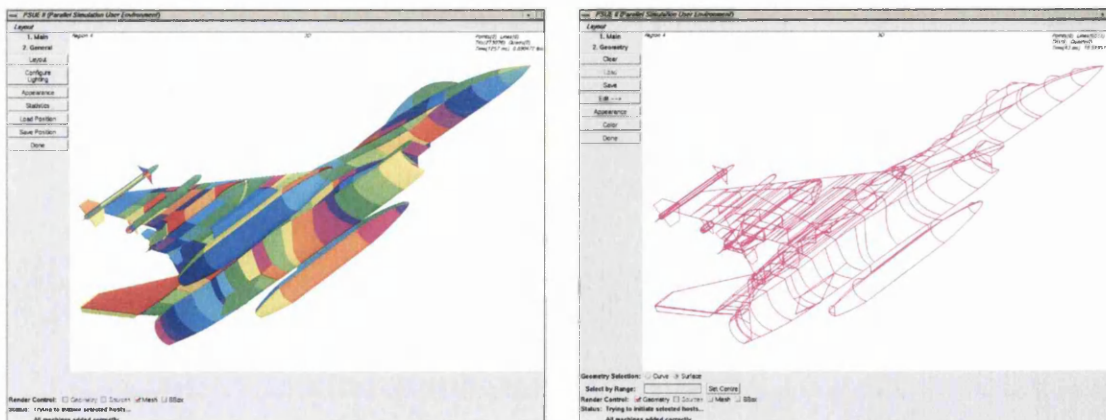


Figure 210 – An Iso-Surface of Mach 1.0

9.3. CFD Simulation over a complete F16 configuration

The purpose of this test case is to show a simulation over a complex geometry being performed within the PSUE II. The F16 (Figure 211) configuration comprises over 500 geometrical surfaces and over 1000 intersection curves. This geometry was made available courtesy of EADS in Munich, Germany.



(a) F16 Surfaces (colour coded) (b) F16 Intersection Curves

Figure 211 – Illustration of the Complexity of the F16 Configuration

Geometry

As the above figure shows, the geometry consisted of half an aircraft with no outer boundary attached. The topology of the curves and surfaces of the geometry was valid so the only operation that had to be performed was the creation of the outer boundary. As with the previous example, a half-sphere was created with symmetry plane and attached to the aircraft. This automatically attached the appropriate curves of the geometry to the symmetry plane and so created a watertight model. The resultant geometry is shown in Figure 212. This operation took less than ten minutes.

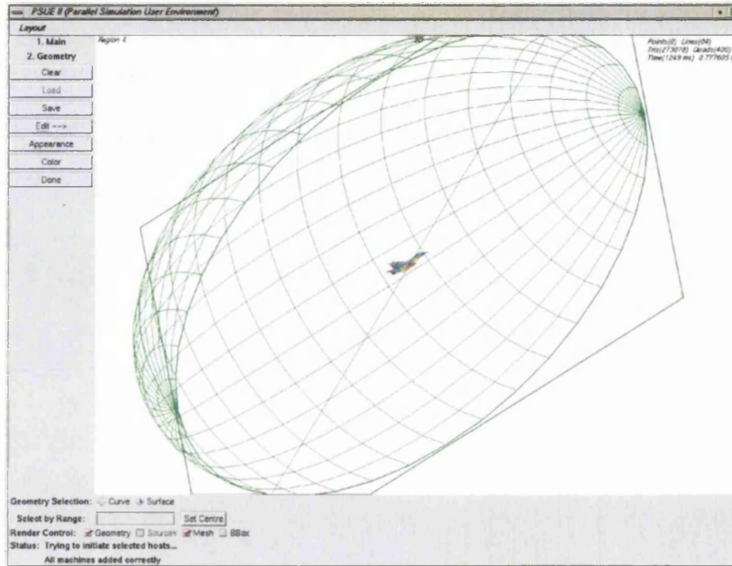
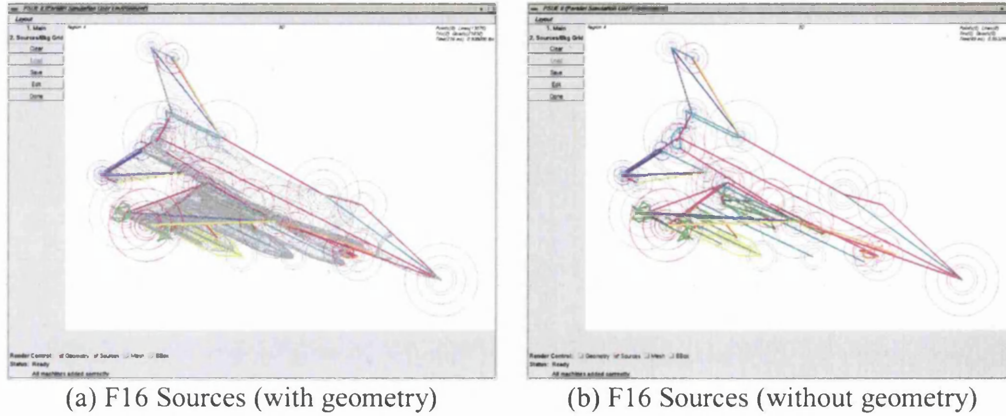


Figure 212 – The F16 Configuration with Outer Boundary

Background Grid and Sources

The team in EADS, using the PSUE, had already positioned a number of sources at the key features of the geometry. The exact time for this was not available but it was estimated about three days. The resultant sources are shown in Figure 213.



(a) F16 Sources (with geometry) (b) F16 Sources (without geometry)

Figure 213 – The Sources used for the F16

Surface and Volume Meshes

The surface mesh was generated in under two minutes and comprised 310,030 triangles and 155,025 nodes. The volume mesh was then generated using eight processors in under one hour and comprised 6,725,979 tetrahedra and 1,117,320 volume nodes. Figure 214 and Figure 215 show the surface and Figure 216 shows a cut through the volume mesh.

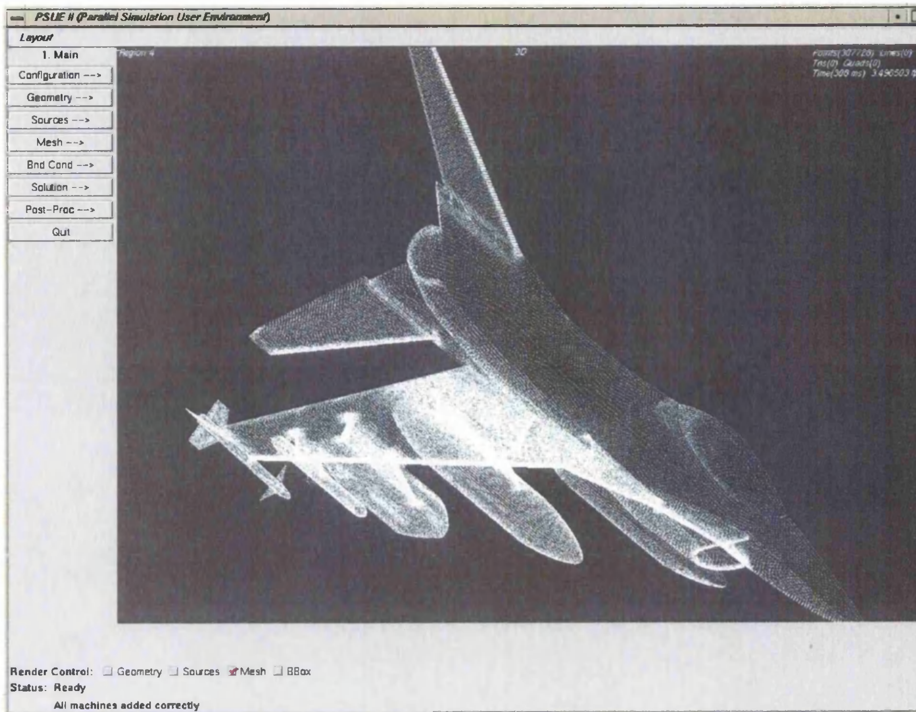


Figure 214 – The Surface Mesh of the F16

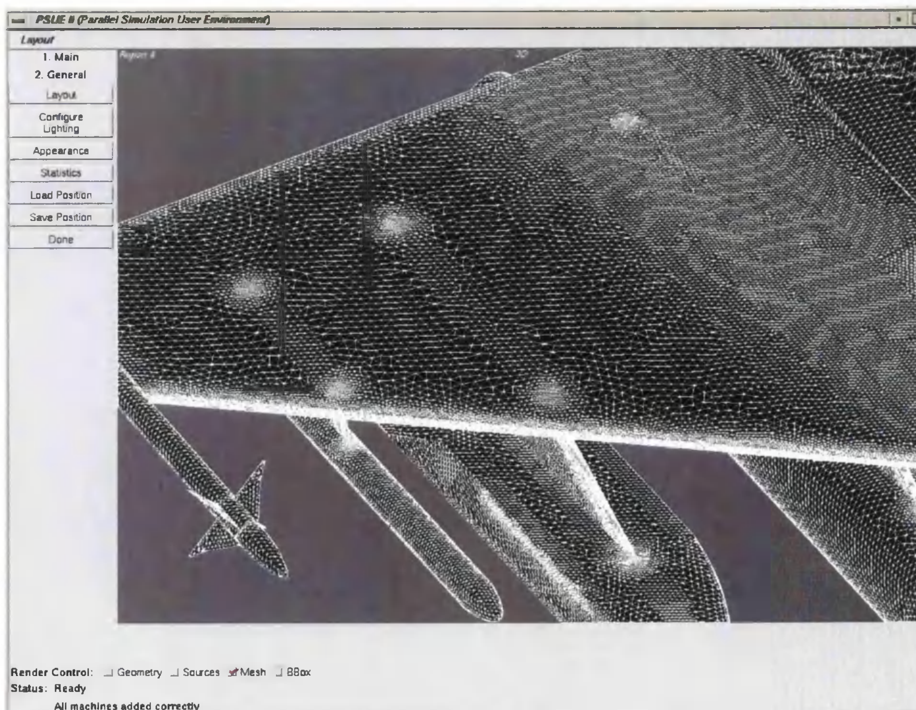


Figure 215 – A zoomed view of the F16 Surface Mesh

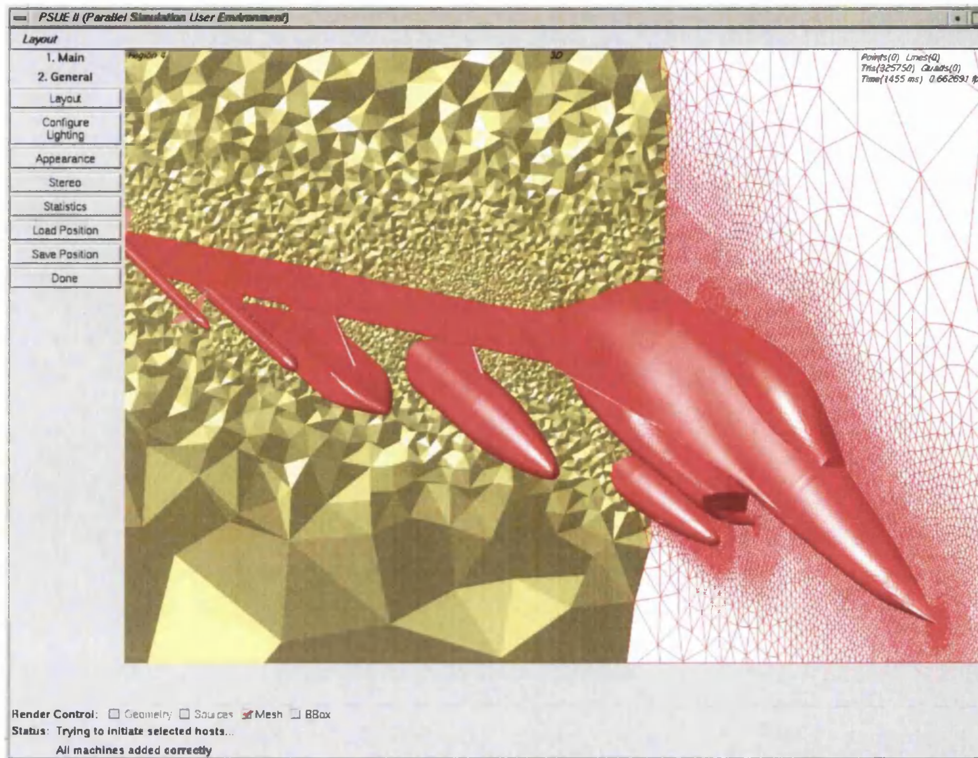


Figure 216 – A Cut through the Volume Mesh of the F16

Mesh Quality Analysis

As with the previous test case, the quality of this mesh was tested using the minimum dihedral angle (Figure 217) and elements whose angle was less than 7° were highlighted (Figure 218). As can be seen, like the previous test case the elements with the small angles tend to cluster around ends of wings and fins due to the shape of the geometry surfaces.

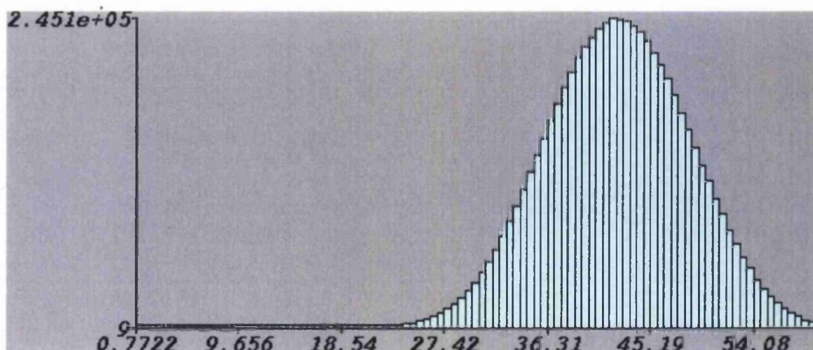


Figure 217 – The Mesh Quality Graph for the F16

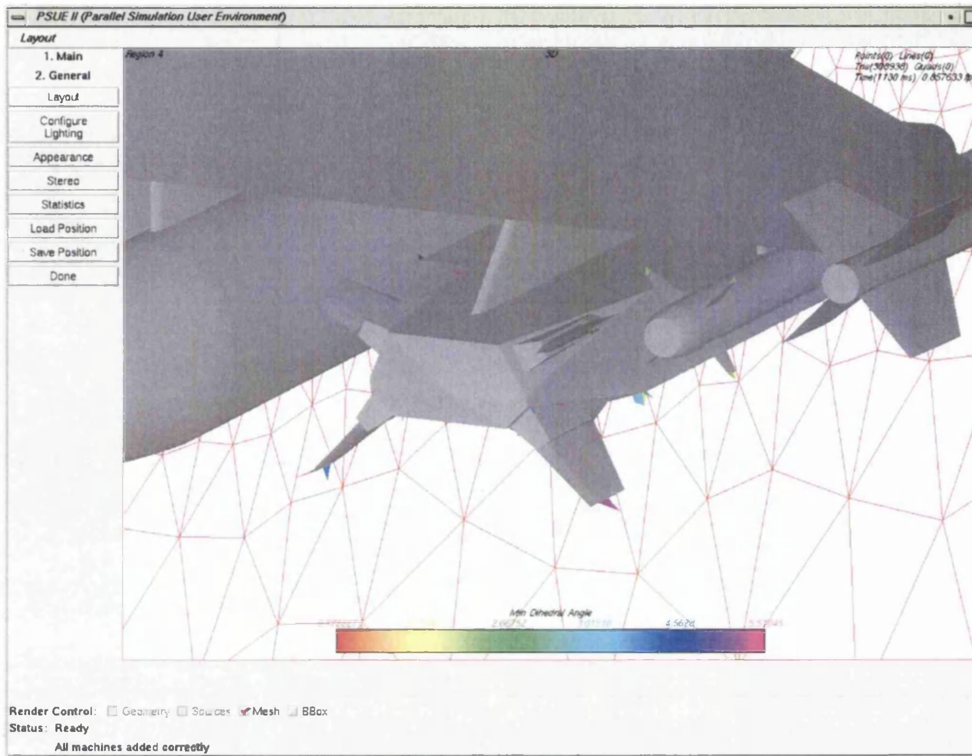


Figure 218 – Highlighting the *poorer* quality elements

Boundary Conditions

The boundary conditions were created within the PSUE II by selecting all the surfaces and applying a wall boundary condition. The individual symmetry, outer boundary and engine inlets and outlets were then selected and the appropriate boundary condition applied. The intersection curves that formed trailing edges on the wings, aircraft fin and missile fins were then selected and flagged. This whole operation took approximately 30 minutes.

Flow Solution

The flow solver ran using eight processors and converged in less than two hours to the flow solution shown in Figure 219.

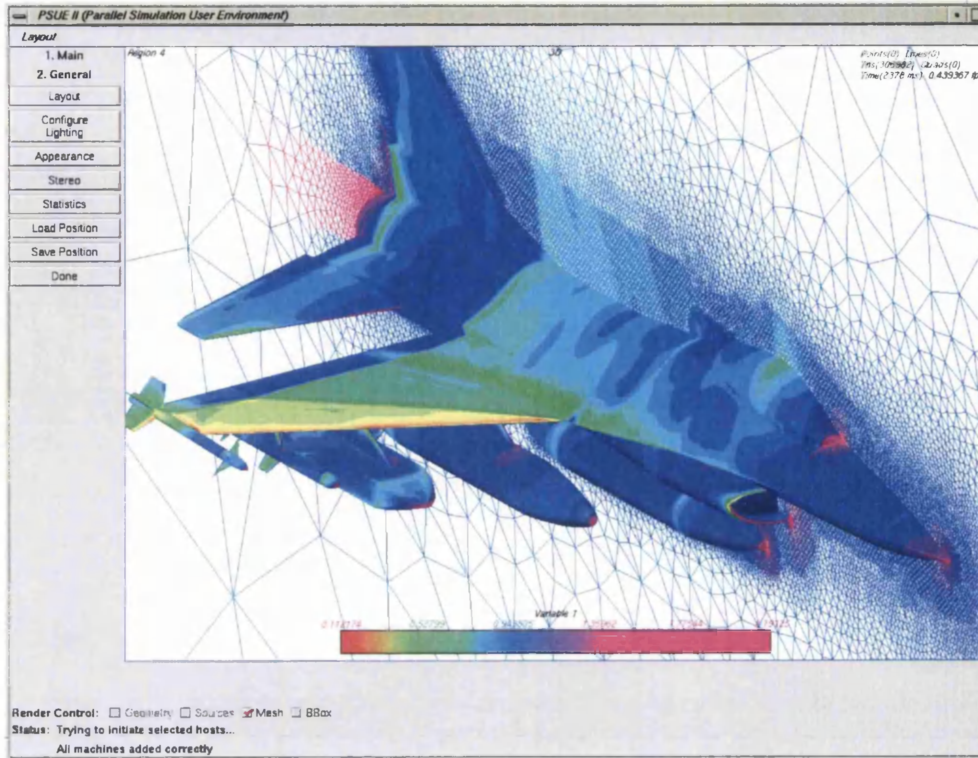


Figure 219 – The Flow Solution over the F16

Post-Processing

Figure 220 shows a cutting plane over the wing and Figure 221 shows an iso-surface of Mach 1.0. These operations were performed in less than 2 minutes each.

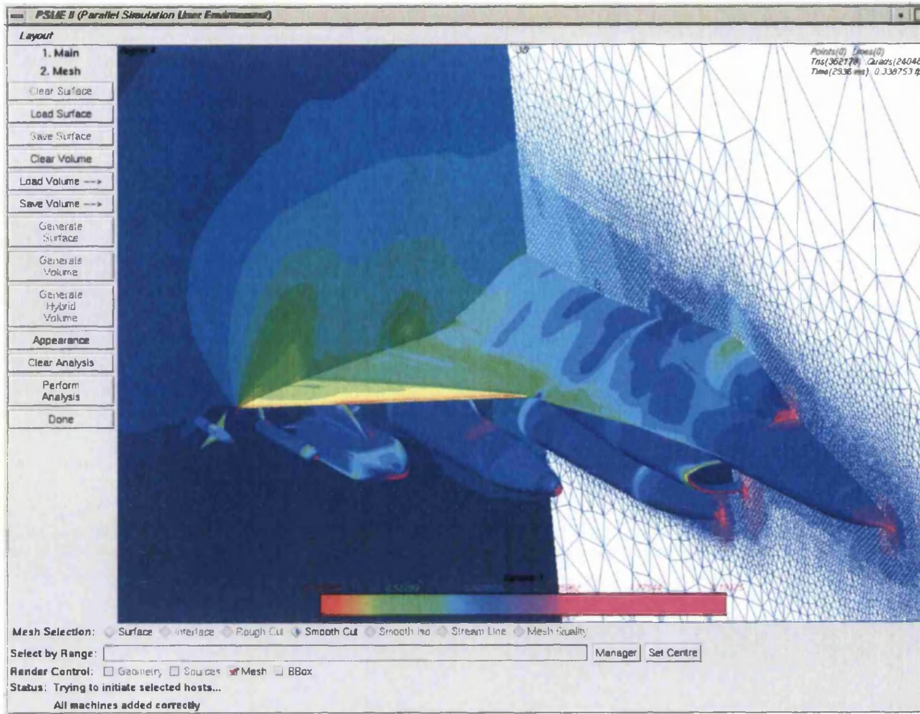


Figure 220 – A Cutting Plane over the F16 Wing

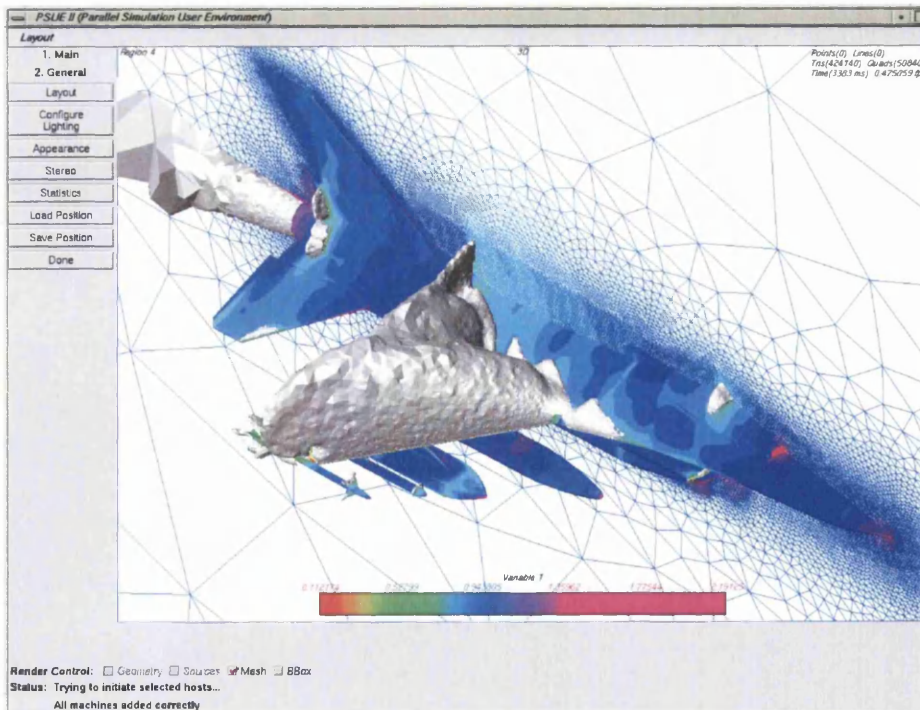


Figure 221 – An Iso-Surface of Mach 1.0 over the F16

9.4. Pre-processing and Post-processing of a Grand-Challenge Simulation over a Dassault Falcon

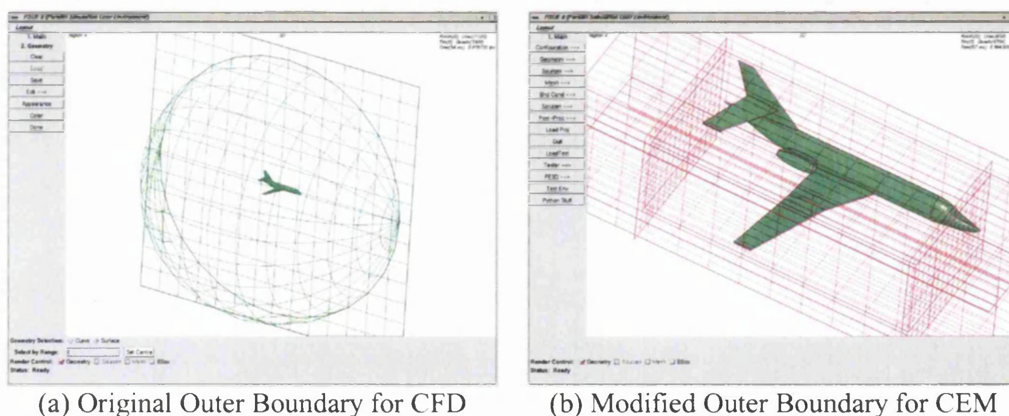
The purpose of this test case was to perform a simulation of a high-frequency electro-magnetic wave over a complete civil aircraft.

Geometry

Since the Dassault Falcon geometry has already been used within the environment for various types of simulations, no geometry repair processes had to be carried out, as the model was already topologically valid. The only editing processes required were to:

- Remove the outer boundary and symmetry plane surfaces and curves.
- Create a reflection of the aircraft geometry and join the two halves to form a complete aircraft.
- Create a closer outer boundary in the shape of a box.

This process was completed in less than five minutes; the resulting geometry is shown in Figure 222.

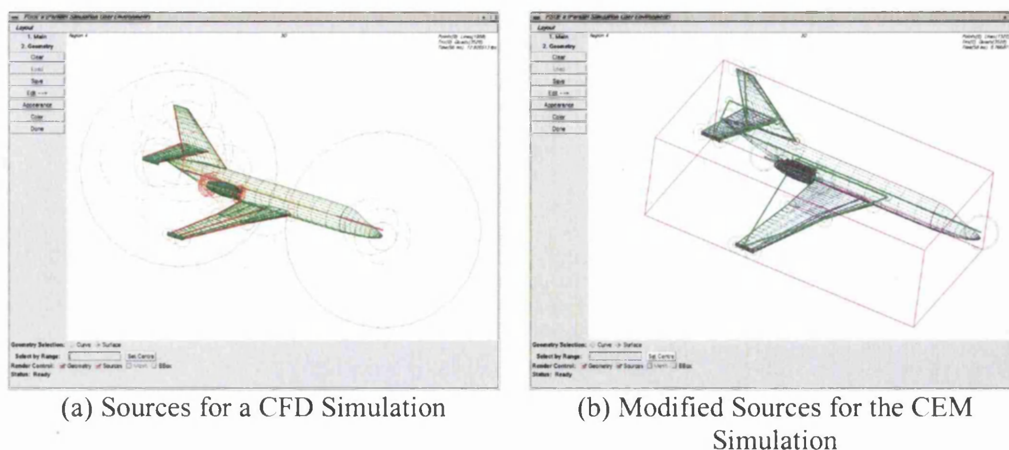


(a) Original Outer Boundary for CFD (b) Modified Outer Boundary for CEM

Figure 222 – Modification of the Outer Boundary for the CEM Simulation

Background Grid and Sources

The setting of sources was a trivial process as sources had already been defined for previous CFD simulations. Since CEM simulations require a reasonably constant mesh spacing, most of the sources were removed. The remaining sources were re-scaled, along with the background spacing, to produce the required number of elements. A coarser mesh was quickly generated in order to approximate the scaling required for the background spacing and source strengths in order to get the required number of elements. This process was completed in approximately 30 minutes. Figure 223 shows the sources before and after this process.



(a) Sources for a CFD Simulation

(b) Modified Sources for the CEM Simulation

Figure 223 – Modifying the Sources for the CEM Simulation

Surface Mesh and Volume Mesh

The surface mesh generation process took approximately five hours and produced a surface mesh consisting of 4,190,720 triangles and 2,095,360 nodes.

The volume mesh was generated using the parallel Delaunay generator using 32 processors taking under 36 hours. The generated mesh consisted of 488,370,760 tetrahedra and 81,612,618 volume nodes. This size of mesh was chosen because it was estimated to be the largest CEM simulation that could be performed on the given platform.

Figure 224 shows an overall view of the surface mesh of the aircraft. As can be seen, the density of the mesh means that it looks like a solid model so Figure 225 shows a zoomed in area of the front of the engine in which the individual triangles can clearly be seen.

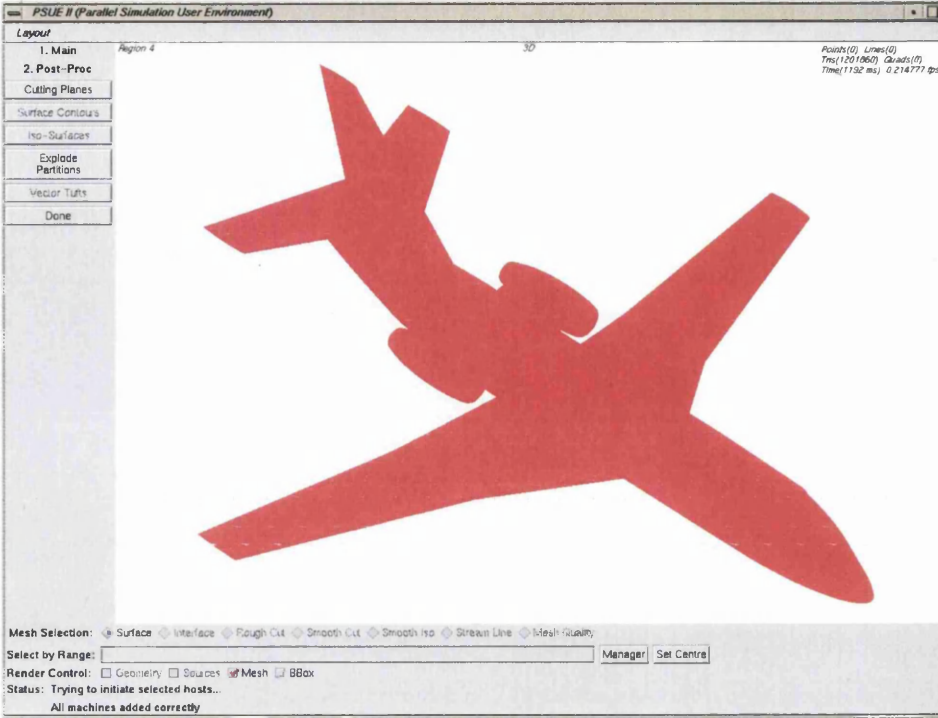


Figure 224 – The Surface Mesh for the CEM Simulation

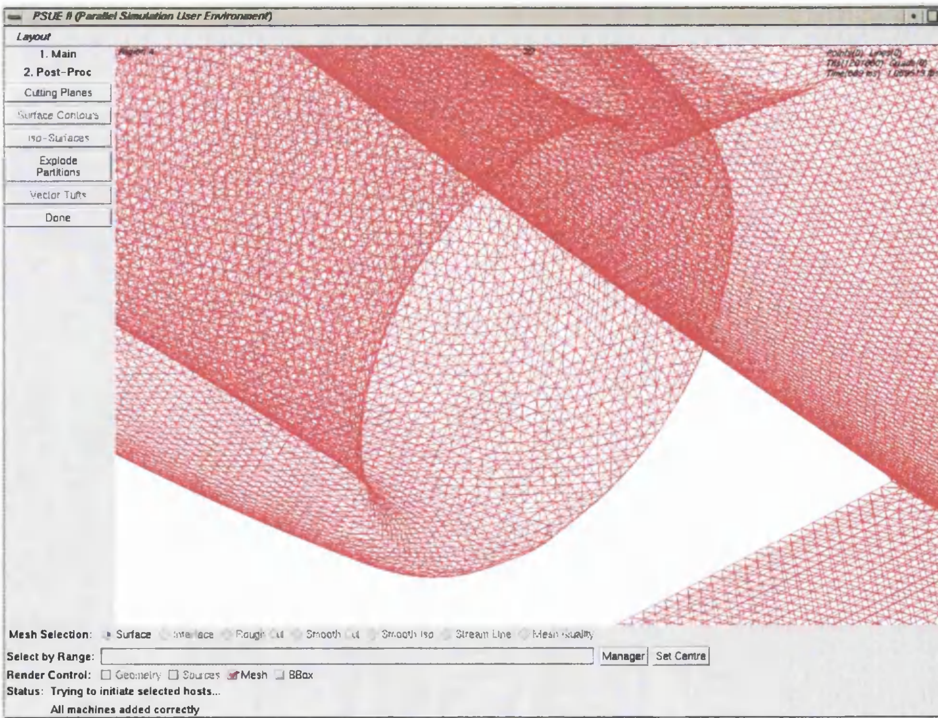


Figure 225 – The Surface Mesh zoomed in on the front of the engine

Mesh Post-Processing

Figure 226 shows the surface of the aircraft with a cutting plane through the volume of the mesh. The cutting plane took approximately 5 minutes to produce.

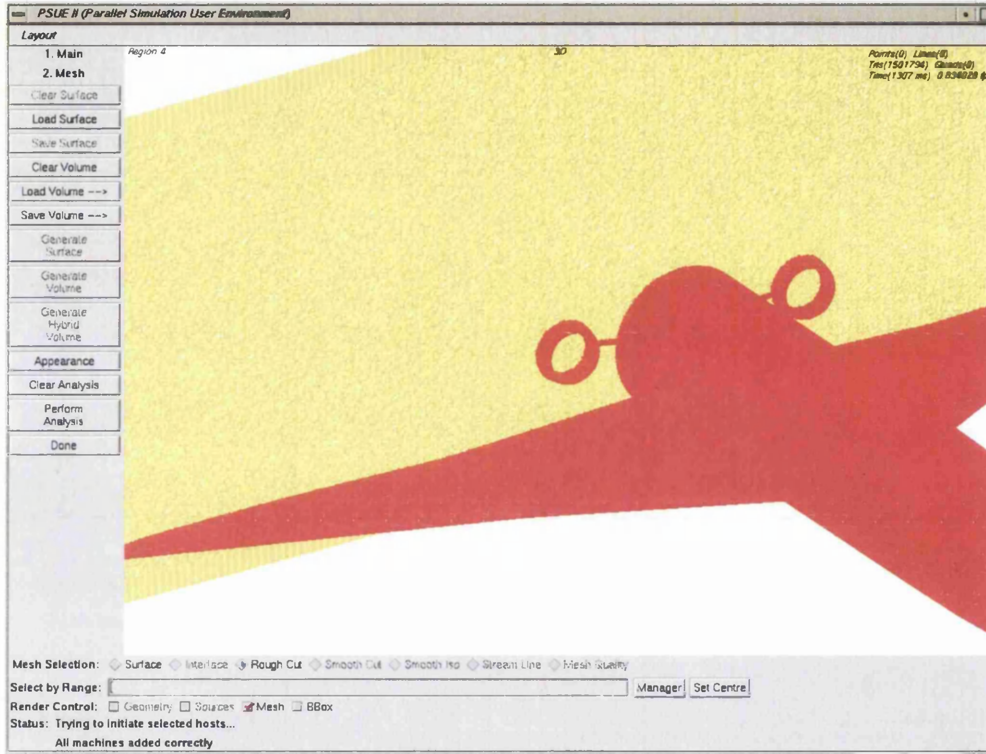


Figure 226 – A cut through the Volume Mesh

Figure 227 – A zoomed view of the cut around the engine

9.5. Summary of Test Cases

The two full simulations shown in the previous sections have shown that through the combined use of the PSUE II, and the various parallel modules, complete simulations can be performed over complex geometries in a matter of 4 – 5 days, with simpler configurations being possible within a day.

The third test case, although not a full simulation, illustrates that the PSUE II can successfully allow the user to interact with and manipulate very large meshes for state-of-the-art calculations.

A summary of the times involved whilst performing these test cases is summarised in the table below.

Test Case 1: CFD Simulation over a Dassault Falcon

Action	Approx. Time Taken
Geometry Preparation	1.5 hours
Background Grid and Sources	2 hours
Surface and Volume Mesh Generation	< 4 minutes
Mesh Quality Analysis (not including time for user to view analysis)	< 1 minute
Boundary Condition Definition	10 minutes

Flow Simulation	< 5 minutes
Post-Processing (Cutting Plane and Iso-Surface)	< 1 minute
Total	**< 3 hours**

Test Case 2: CFD Simulation over a complete F16 Configuration

Action	Approx. Time Taken
Geometry Preparation	< 10 minutes
Background Grid and Sources	~3 days
Surface and Volume Mesh Generation	< 1 hour
Mesh Quality Analysis (not including time for user to view analysis)	< 10 minutes
Boundary Condition Definition	~30 minutes
Flow Simulation	< 2 hours
Post-Processing (Cutting Plane and Iso-Surface)	< 4 minutes
Total	**~3.5 days**

Test Case 3: Pre- and Post-Processing of a Grand Challenge Simulation over a Dassault Falcon

Action	Approx. Time Taken
Geometry Preparation	< 5 minutes
Background Grid and Sources	~30 minutes
Surface and Volume Mesh Generation	< 2 days
Post-Processing (Cutting Plane)	~5 minutes
Total	**~2.5 days**

Chapter 10. CONCLUSIONS AND FUTURE RESEARCH

10.1. Conclusions

In Chapter 1, a number of requirements were listed for which a *successful* Problem Solving Environment would have to meet. These were:

- Problem set-up time must be reduced.
- The user must be guided through the simulation process.
- The details of the execution of tasks on remote parallel computers must be hidden from the user.

In order to achieve these requirements, a number of challenges were listed that the PSE must overcome in order to satisfy these requirements:

- The User Interface must remain intuitive throughout the simulation.
- All invalid routes through the environment should be disabled.
- All three-dimensional rendering must be real-time.
- All interaction must remain as close to real-time as possible.
- The use of parallel computers should be (semi-) transparent.

In the context of the JULIUS project, a number of more specific requirements were added:

- To develop an integrated functional HPCN environment for simulation in multiple disciplines,
- To provide and demonstrate HPCN tools and engineering simulation tools that efficiently work together in this environment,
- To put developments in place to remove the major limitations and bottlenecks in engineering simulations and
- To demonstrate the entire system working with embedded applications software for realistic, industrial problems.

It is obvious from the above three lists that, although expressed using different words, a number of these requirements and challenges overlap significantly. In this section, the key points in the lists above will be tackled one by one in order to show that the PSUE II has been a *successful* Problem Solving Environment.

The user must be guided through the simulation process

This requirement has been a constant factor during the design of the PSUE II and has been manifested in the design and implementation of the toolbars through which all of the functionality of the environment is accessed. The organisation of

the top-level toolbar is designed to match the traditional route through a typical computational simulation as described in Chapter 1. The hierarchical structure of the toolbars ensures the only options that are visible to the user are those that are relevant to that stage of the simulation. Other means of presenting the functionality of the environment, such as pull-down menus, present all of the operations to the user all of the time, which could be confusing.

Disabling any options that are inappropriate at that time, for example an operation that needs a mesh, further enforces guidance through the simulation but there is no mesh present. This functionality is extended to any user-defined toolbars through the toolbar configuration files. For example, if any commands to save data sets are present for a user-defined button, then that button will be disabled unless all of the required data sets are present within the environment. For example, if a button was defined as:

```
Button( "Generate Mesh" )
{
  save_geometry( "/tmp/geom" )
  save_bkg_grid( "/tmp/sources" )
  execute_command( "mesh_generator" )
  load_surf_mesh( "/tmp/mesh" )
}
```

then this button will only be enabled if the geometry and background grid are currently present within the environment.

The PSUE II should be capable of dealing with realistic, industrial problems using complex geometries and large meshes

The capability to cope with large and complex data sets was the main driving force behind the PSUE II. As the testcases in Chapter 9 show, realistic geometries can be used (e.g. F16 with 525 geometrical surfaces and over 1000 intersection curves) and very large scale meshes (e.g. CEM calculation on a Dassault Falcon comprising over 0.5 billion elements).

The use of parallel computing should be as transparent as possible to the user

Obviously, in order to be able to interact with data sets of the order of 100's of millions of elements, the use of parallel computing technology is fundamental. As shown in Chapters 7 and 9, the only extra interaction needed by the user is to choose which parallel computers and/or workstations are used to form a parallel computer. Every other aspect of parallel computing is hidden from the user. Even this extra user interaction could be easily avoided if the organisation in which the PSUE II is used has only one parallel computer. In this scenario, the use of parallel computing would be completely hidden from the user.

All interaction with the PSUE II should be as real-time as possible

In order for a graphical environment, such as the PSUE II, to be deemed a success in the eyes of a user it must provide as much feedback to the user as possible. One of the key elements to this is the time is takes for the environment to respond to a user's action. If the user is continuously waiting for the environment to catch up then frustration can quickly build dramatically increasing the chance of errors.

In order to maintain rapid response times, the performance of a number of key areas is paramount:

- All rendering of the three-dimensional data sets must be real-time. Any delays whilst a user is manipulating an object on the display or selecting a feature of an object is intolerable.
- Any operation performed on the model, such as cutting plane, iso-surface, etc. should be as rapid as possible. A typical user will appreciate that some operations cannot be performed instantaneously but it should be remembered that even 30 seconds can seem like hours when waiting for an operation to complete.

In order to satisfy these requirements, a number of features have been described in previous chapters that enable the PSUE II to maintain reasonable response times at all times. These are:

- The use of parallel computing to perform all operations on the large volume data sets.
- The use of an oct-tree data structure to dramatically increase the performance of any operations that involve traversing the volume data sets.
- The use of *intelligent* techniques to minimise the network traffic between the slave processes and the master.
- The use of Open GL strips in order to reduce the size and increase the performance of the rendering in the master.

The PSUE II should be flexible and configurable enough to meet the demands of a multi-disciplinary environment

Throughout the design of the PSUE II, care has been taken to ensure that no component of the environment is hard-wired to any particular type of simulation. For example:

- Meshes can comprise any combination of the common linear element types (e.g. hexahedra, prisms, pyramids, tetrahedra, quadrilaterals and triangles).
- The definitions of the boundary conditions is completely generic and can be customised to any particular solver through a simple text file that maps integer identifiers with boundary condition types.
- The types of variables in the solution can be any combination of scalar or vector. The names of these variables are defined in the solution data file itself.

- The use of parallel algorithms throughout the environment ensures that almost any size of simulation can be performed.
- The configurable toolbars allows any type of mesh generator, solver or any other code to be integrated within the environment.

Chapter 9 illustrates the use of the environment for both CFD and CEM calculations.

Problem set-up time must be reduced.

As has been shown in Chapter 9, the PSUE II has reduced problem set-up time considerably when compared with performing the same simulations using a standard shell window and command-line tools, especially for the non-computer specialist. The reasons for this have been outlined in the points above.

Use of the PSUE II by non-specialist personnel

Probably the best indicator of the success of an interactive environment is its day to day use by the people for which it was intended. In the case of the PSUE II, it is currently being used by:

- A number of research students throughout the Centre for Computation and Simulation within the School of Engineering in Swansea.
- A number of aerospace companies in Europe such as BAE Systems (UK) and EADS (Germany).

The Army Research Labs in the US are also evaluating its use as an environment in which their high-performance parallel algorithms could be integrated.

10.2. Future Research

Although the PSUE II has been shown to be a successful Problem Solving Environment, there are still a number of areas in which big improvements can be made:

Improve User Guidance

Although the user is guided through the simulation process by the PSUE II disabling any options that would cause the user to stray, much more could be achieved in terms of actually helping the user to create suitable data sets. For example, at the current time the user needs to define an outer boundary, sources and a background spacing in order to get a suitable mesh for a particular CEM calculation. However, the size of the outer boundary and the mesh spacing could all be determined automatically by the frequency of the wave to be simulated. This means that instead of the user defining low-level parameters, the interaction could be at a much higher level. Similarly, the environment could warn the user if the generated mesh is too coarse to pick up certain small geometric features.

Development of a Visual Programming Environment

There are a number of features already implemented within the PSUE II that enable some configuration by the user. These are described in Section 6.6 and

include configurable toolbars, run-time data link to third party applications and the integration of the Python scripting language.

However, these are all features that require some level of programming experience in order to utilise them. For the design engineer, who wishes to combine a number of operations in order to perform a task, the Visual Programming Environment is the most appropriate tool.

A simple VPE could be implemented within the PSUE II with relative ease. This could represent all of the functionality of the PSUE II as the traditional boxes of a VPE. These boxes could then be linked together graphically in order to form a map which could then be executed either interactively or in a batch mode overnight.

A simple prototype of a VPE has been implemented within the PSUE (Figure 228). However, it is purely to show the possible appearance and has no actual functionality to date.

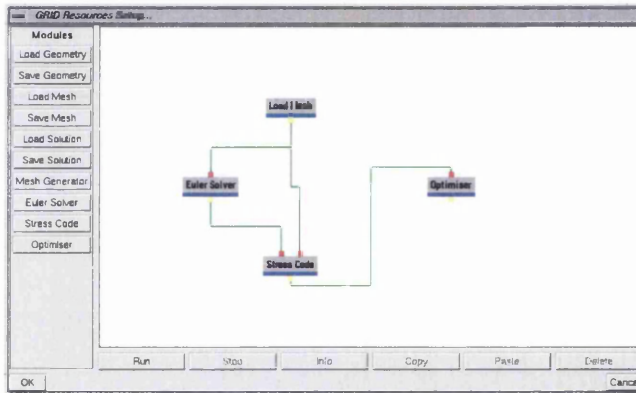


Figure 228 – A simple prototype of a VPE within the PSUE II

Geometry Editing and NURBS

Currently, the PSUE II uses a natural spline and Ferguson patch representation for geometries. This decision was due to the native representation the in-house mesh generation algorithms used. However, in the industrial environment, the use of NURBS is prevalent with standard file formats such as IGES and STEP being used for geometry file transfer.

Conversion between the NURBS representation and the Ferguson representation is a difficult and very time consuming process, often involving days (or possibly weeks) of user time, which also can add a significant degree of error. In order to be able to operate directly on models obtained from typical CAD systems, a number of modifications/additions are required:

- The PSUE II needs to be modified to operate directly with NURBS curves and surfaces.
- The Mesh Generation algorithms need a similar modification to be able to generate meshes directly on NURBS geometries.
- A means by which an IGES (or STEP) file can be read directly into the PSUE II.
- The geometry editing facilities need to be improved in order to be able to produce topologically valid models from the IGES files. These enhancements would, invariable, be driven by the user's needs as and when they occurred.

There has been some recent work undertaken in this area using the DT Nurbs library [http://ocean.dt.navy.mil/dtnurbs] for the underlying geometry operations.

Feature Detection

As simulations involving hundreds of millions of elements becomes more common, the need for the environment to automatically detect and display relevant information becomes paramount. Some simple feature detection algorithms are currently implemented in the PSUE II, such as iso-surfaces, contour lines and vector tufts. However, these need to be extended to encompass features such as stream lines and vortex detection [Darmofal92, Haimes99, Cebral01].

Virtual Reality

As the cost of high-performance graphics hardware decreases and the complexity of the simulations being performed increases, the use of virtual reality as a means of interacting with the data is increasing in popularity. Virtual reality technology is available in many different forms depending on the users requirements and the available budget.

The lowest end technology involves the use of shutter glasses with a high-end PC. This can be used to produce stereoscopic images on the display where the model appears to either pop out of the screen or move further into the depths of the monitor. A simple extension to this is to use a projector and screen in a darkened room instead of the computer monitor. This uses the same software but can produce a more convincing effect since the screen takes up more of the users peripheral vision and is matt so eliminating distracting reflections.

Several screens can then be combined with several projectors to form a small cube shaped room (approximately ten feet in each direction). This produces a very convincing three-dimensional image of the model in the centre of the cube. This type of display was first developed in University of Chicago and is often referred to as a CAVE [Cruz-Neira93]. However, the computer power required to drive such a display increases well beyond the scope of a PC since the model needs to be rendered once for each screen many times a second with the time for each frame being perfectly synchronised. A typical computer often used for this type of application is the Onyx supercomputer developed by SGI.

Further enhancements can be made to this type of environment such as:

- Head tracking so the computer can draw the model taking into consideration the position of the user's head.
- Three-dimensional mice for interacting within the three-dimensional world.
- Gloves for allowing the user to grasp objects and move them directly.

The use of such virtual reality techniques within the PSUE II would enable the user to interact with the various data sets in a much more intuitive manner than is possible with a flat, two-dimensional screen.

Since this could be such a benefit to the user, a preliminary effort has been undertaken to implement, within the PSUE II, the ability to render a stereoscopic image that with the use of shutter glasses can be made to either pop out or move back into the monitor.

Meta-Computing and the Grid

The previous points for improvement have concentrated on individual sections of the PSUE II. A more fundamental change to the design of the PSUE II would be the integration of meta-computing and the Grid.

The term 'the Grid' was coined in the mid-1990s to describe a distributed computing infrastructure for advanced science and engineering. This, essentially, entails the co-ordinated use of geographically disparate super-computers in order to solve a problem. When using such an infrastructure several difficulties arise, such as; authentication and authorisation of users; controlling resource access; and, even, discovering resources that are available.

Significant progress has been made in the development of the underlying infrastructure [Foster97b, Barnard99, Allcock01, Keahey02], in particular the GLOBUS toolkit [Foster97a], which attempts to alleviate some of the difficulties described above.

In order to advance Grid technology further, a lot of funding is available for research into this area, both in the UK, and around the world. For example, in the UK, over £150 million pounds has been allocated over three years to support research into e-Science and the Grid. This has enabled thirteen e-Science centres to be set up in order to form a focus for Grid related research. As can be seen, from this expenditure alone, the area of Grid technology is seen as being of fundamental importance in the future.

The distributed, and parallel, nature of the PSUE II with its utilisation of CORBA technology is in a prime position to form a basis for such research and development.

Chapter 11. BIBLIOGRAPHY

[Adobe90] Adobe Systems Inc. 1990. Postscript Language Reference Manual. Addison-Wesley Publishing Company.

[Aho86] Aho, AV. Sethi, R. Ullman, JD. 1986. Compilers – Principles, Techniques and Tools. Addison-Wesley Publishing Company.

[Allcock01] Allcock, B. Foster, I. Nefedova, V. Chervenak, A. Deelman, E. Kesselman, C. Leigh, J. Sim, A. Shashani, A. Drach, B. Williams, D. 2001. High-Performance Remote Access to Climate Simulation Data: A Challenge Problem for Data Grid Technologies. *Supercomputing 2001*, November 2001.

[Amdahl67] Amdahl, G. 1967. Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities. *AFIPS Conference Proceedings*, (30), pp. 483-385, 1967.

[ANSI85] ANSI (American National Standards Institute). 1985. American National Standard for Human Factors Engineering of Visual Display Terminal Workstations, *ANSI, Washington, DC.*1985

[ANSI88] ANSI (American National Standards Institute). 1988. American National Standard for Information Processing Systems – Programmer's Hierarchical Interactive Graphics System (PHIGS) Functional Description. *ANSI, X3.144-1988, ANSI, New York.* 1988.

[AVS] Advanced Visualisation Systems. AVS. www.avs.com.

[Barnard99] Barnard, S. Biswas, R. Saini, S. Van der Wijngaart, R. Yarrow, M. Zechter, L. Foster, I. Larsson, O. 1999. Large-Scale Distributed Computational Fluid Dynamics on the Information Power Grid using Globus. *Proceedings of Frontiers '99*, 1999.

[Beguelin94] Beguelin, A. Dongarra, J. Geist, A. Manchek, R. Sunderam, V.S. 1994. Recent Enhancements to PVM. *International Journal for Supercomputer Applications*, **9** (2).

[Belytschko84] Belytschko, T. Ong, J.S-J. 1984. Hourglass Control in Linear and Nonlinear Problems. *Computer Methods in Applied Mechanics and Engineering*, **43**, pp. 251 – 276.

[Berthou97] Berthou, J-Y. Colombet, L. 1997. Which approach to parallelizing scientific codes – That is the question. *Parallel Computing*, **23**, pp. 165 – 179. 1997.

[Bradley91a] Bradley, C. 1991. SWIPE: Monitor Program. *Rolls-Royce Internal Report CUGJH04*. Dec 1991.

[Bradley91b] Bradley, C. 1991. SWIPE: Structured Workstation Input Preparation Environment. *Rolls-Royce Internal Report CUGSWIPE.* Dec 1991.

[Bramley98] Bramley, R. Gannon, D. Stuckey, T. Villacis, J. Balasubramanian, J. Akman, E. Breg, F. Diwan, S, Govindaraju, M. 1998. The Linear System Analyzer. *Submitted for IEEE book on PSEs*, March 1998.

[Brockschmidt95] Brockschmidt, K. 1995. Inside OLE, 2nd edition. *Microsoft Press.* 1995

[Brodersen98] Brodersen, O. Ronzheimer, A. Ziegler, R. Kunert, T. Wild, J. Hepperle, M. 1998. Aerodynamic Applications using MegaCads. *Proceedings of the 6th International Conference on Numerical Grid Generation in Computational Field Simulations, Greenwich, London.* pp. 793 - 802. July 1998.

[Buschelman00] Bushelman, K.R. Gropp, W.D. McInnes, L.C. Smith, B.F. 2000. PETSc and Overture: Lessons Learned Developing an Interface between Components. ANL/MCS-P858-1100. *Proceedings of the International Federation for Information Processing Working Conference on Software Architectures for Scientific Computing, Kluwer, 2000.*

[Cebral01] Cebral, J.R. Löhner, R. 2001. Flow Visualization On Unstructured Grids Using Geometrical Cuts, Vortex Detection and Shock Surfaces. *AIAA Paper 2001-0915, Reno, NV.* January 2001.

[Cignoni98] Cignoni, P. Montani, C. Scopigno, R. 1998. A Comparison of Mesh Simplification Algorithms. *Computers & Graphics*, **22** (1), pp. 37 – 54.

[CrayInc01] Cray Inc. 2001. Cray T3E 1350 Brochure. http://www.cray.com/products/systems/t3e/t3e.pdf.

[Cuthill69] Cuthill, E. McKee, J.M. 1969. Reducing the bandwidth of sparse symmetric matrices. *ACM Publication P69: Proceedings of the 24th National Conference of the Association of Computing Machines*, pp. 157 – 172, New York, 1969.

[Cruz-Neira93] Cruz-Neira, C. Sandin, D. DeFanti, T. 1993. Virtual Reality: The Design and Implementation of the CAVE. *Proceedings of SIGGRAPH 93 Computer Graphics Conference, ACM SIGGRAPH*, pp. 135 – 142, 1993.

[Daconta96] Daconta, M.C. 1996. JAVA for C/C++ Programmers. *Wiley Computer Publishing, John Wiley & Sons, Inc.* 1996.

[Darmofal92] Darmofal, D. Haimes, R. Visualisation of 3-D Vector Fields: Variations on a Stream. *AIAA Paper 92-0074, Reno, NV.* Jan1992

[Dillencourt92] Dillencourt, M. 1992. Finding Hamiltonian cycles in Delaunay triangulations is NP-complete. *Canadian Conference on Computational Geometry*, pp. 223 – 228, 1992.

[Dongarra95] Dongarra, J. Otto, S.W. Snir, M. Walker, D. 1995. An Introduction to the MPI Standard. *Internal Report CS-95-274*. http://www.netlib.org/ncwn/mpi-cacm.ps. January 1995.

[Donnelly02] Donnelly, C. Stallman, R. 2002. Bison – The YACC-compatible Parser Generator. http://www.fsf.org/manual/bison-1.35/bison.html. May 2002.

[El-Sana00] El-Sana, J. Evans, F. Kalaith, A. Varshney, A. Skiena, S. Azanli, E. 2000. Efficiently Computing and Updating Triangle Strips for Real-Time Rendering. *Computer-Aided Design*, **32**, pp. 753 – 772.

[Fol91] Fol, T. Katossky, V. 1991. Brite-Euram Euromesh Sub-Task 1.2 : Basic Metrics for Mesh Quality. *Aérospatiale Division Avions No. 443.558/91.*

[Foley90] Foley, J. van Damn, A. Feiner, S. Hughes, J. 1990. Computer Graphics: Principles and Practice. Addison-Wesley Publishing Company.

[Foster97a] Foster, I. Kesselman, C. 1997. Globus: A Metacomputing Infrastructure Toolkit. *International Journal of Supercomputer Applications*, **11** (2), pp. 115 – 128.

[Foster97b] Foster, I. Geisler, J. Nickless, W. Smith, W. Tuecke, S. 1997. Software Infrastructure for the I_Way High Performance Distributed Computing Environment. *Proceddings of the 5th IEEE Symposium on High Performance Distributed Computing*, pp. 562 – 571, 1997.

[Foster01] Foster, I. Kesselman, C. Tuecke, S. 2001. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *International J. Supercomputer Applications*, **15** (3), 2001.

[Gaither96] Gaither, A. Jean, B. Remotigue, M. Whitmire, J. 1996. NGP: Defining a Grid Generation Paradigm on NURBS and Solid Modeling Topology. *Proceedings of the 5th International Conference on Numerical Grid Generation in Computational Field Simulations, Mississippi, MS.* April 1996.

[Gaither00] Gaither, A. Marcum, D. Mitchell, B. 2000. SolidMesh: A Solid Modelling approach to Unstructured Mesh Generation. *Proceedings of the 7th International Conference on Numerical Grid Generation in Computational Field Simulations, Whistler, BC.* September 23-28, 2000.

[Gallopoulis94] Gallopoulos, S. Houstis, E. Rice, J. 1994. Computer as Thinker/Doer: Problem-Solving Environments for Computational Science. *IEEE Computational Science and Engineering, Summer 1994.*

[Goel99] Goel, A. Baker, C.A. Shaffer, C.A. Grossman, B. Mason, W.H. Watson, L.T. Haftka, R.T. 1999. VizCraft: A Problem Solving Environment for Configuration Design of High Speed Civil Transport. Internal Report TR-99-02 (http://people.cs.vt.edu/~shaffer/TR.html). September 24, 1999

[Gropp99a] Gropp, W. Huss-Lederman, S. Lumsdaine, A. Lusk, E. Nitzberg, B. Saphir, W. Snir, M. 1999. MPI – The Complete Reference Volume 1. The MIT Press.

[Gropp99b] Gropp, W. Huss-Lederman, S. Lumsdaine, A. Lusk, E. Nitzberg, B. Saphir, W. Snir, M. 1999. MPI – The Complete Reference Volume 2. The MIT Press.

[Haimes91a] Haines, R. Giles, M. 1991. Visual3 : Interactive Unsteady Unstructured 3D Visualisation. *AIAA Paper 91-0794, Reno, NV.* Jan 1991.

[Haimes91b] Haimes, R. Darmofal, D. 1991. Visualisation in Computational Fluid Dynamics: A Case Study. *IEEE Computer Society, Visualization '91.* October 1991.

[Haimes91c] Haimes, R. Giles, M. Darmofal, D. Visual3 – A Software Environment for Flow Visualization. 1991. *Computer Graphics and Flow Visualisation in Computational Fluid Dynamics, VKI Lecture Series #10, Brussels.* 16-20 Sept 1991.

[Haimes93] Haimes, R. Connell, S. Vermeersch, S. 1993. Visual Grid Quality Assessment for 3D Unstructured Meshes. *AIAA Paper 93-3352, Orlando, Florida.* July 1993.

[Haimes94] Haimes, R. 1994. pV3: A Distributed System for Large-Scale Unsteady CFD Visualisation. *AIAA Paper 94-0321, Reno, NV.* January 1994.

[Haimes95] Haimes, R. Sondak, D. 1995. Visualization Math Library User's Guide. http://raphael.mit.edu/pv3/vizmath.ps. April 1995.

[Haimes97] Haimes, R. Edwards, D. 1997. Visualization in a Parallel Processing Environment. *AIAA paper 97-0348, Reno, NV.* January 1997.

[Haimes98a] Haimes, R. 1998. Visual 3 Advanced Programmers Guide. http://raphael.mit.edu/visual3/advpro.ps

[Haimes98b] Haimes, R. 1998. Visual 3 Users and Programmers Manual. ttp://raphael.mit.edu/visual3/user3.ps

[Haimes99] Haimes, R. Kenwright, D. 1999. On the Velocity Gradient Tensor and Fluid Feature Extraction. 1999. *AIAA Paper 99-3288, Norfolk, VA.* June 1999.

[Hancock97] Hancock, D.J. Hubbold, R.J. 1997. Distributed parallel volume rendering on shared memory systems. *Future Generation Computer Systems*, **13**, pp. 251 – 259. 1997/98.

[Hanrahan90] Hanrahan, Pat. Haeberli, Paul. 1990. Direct WYSIWYG Painting and Texturing on 3D Shapes. *Proceedings of SIGGRAPH 90*, **24** (4), pp 215 – 223, August 1990.

[Hassan96] Hassan, O. Probert, E.J. Morgan, K. Peraire, J. 1996. Unstructured tetrahedral mesh generation for three dimensional viscous flows. *International Journal of Numerical Methods in Fluids*, **39**, pp. 549 – 567.

[Hassan99a] Hassan, O. Weatherill, N.P. Morgan, K. 1999. FLITE System Reference Manual, Part 1 – Basic Theory. *Internal Report.*

[Hassan99b] Hassan, O. Weatherill, N.P. Morgan, K. 1999. FLITE System Reference Manual, Part 1 – User Manual. *Internal Report.*

[Hassan99c] Hassan, O. Probert, E.J. 1999. Handbook of Grid Generation (ed. Joe Thompson, Bharat Soni and Nigel Weatherill). pp. 35-6 – 35-9. CRC Press. 1999.

[Hassan01] Hassan, O. Bayne, L.B. Morgan, K. Weatherill, N.P. 2001. Improving the efficiency of explicit schemes for 3D transient compressible flows with moving boundaries on unstructured meshes. *Computational Fluid Dynamics Journal*, Special No. 2001, pp. 380 – 389.

[Hassan02] Hassan, O. Jones, J. Larwood, B. Morgan, K. Weatherill, N.P. 2002. A fully parallel approach for the simulation of electromagnetic scattering using unstructured meshes. *Book of Abstracts of 5th World Congress on Computational Mechanics*, 262, Vienna, 2002.

[Henning99] Henning, M. Vinoski, S. 1999. Advanced CORBA Programming with C++. Addison-Wesley Publishing Company.

[Hirsch94] Hirsch, C. Torreele, J. Keymeulen, D. Vucinic, D. Decuyper, J. 1994. Distributed Visualization in CFD. *SPEEDUP Journal*, **8** (1).

[Hoppe98] Hoppe, H. 1998. Efficient Implementation of Progressive Meshes. *Computers & Graphics*, **22** (1), pp. 27 – 36.

[Hsieh95] Hsieh, S-H. Paulino, G.H. Abel, J.F. 1995. Recursive Spectral Algorithms for Automatic Domain Partitioning in Parallel Finite Element Analysis. *Computer Methods in Applied Mechanics and Engineering*, **121**, pp. 137 – 162.

[ISO88] International Standards Organisation. 1988. International Standard Information Processing Systems – Computer Graphics – Graphical Kernel System for Three

Dimensions (GKS-3D) Functional Description. *ISO Document Number 8805:1988(E), American National Standards Institute, New York.* 1988.

[Johnson98] Johnson, C. Berzins, M. Zhukov, L. Coffey, R. 1998. SCIRun: Application to atmospheric dispersion problems using unstructured meshes. *Numerical Methods for Fluid Dynamics VI, M.J. Baines, ed.*, 1998.

[Johnson00] Johnson, C. Parker, S. Weinstein, D. 2000. Large-scale Computational Science Applications using the SCIRun Problem Solving Environment. *Supercomputer 2000.*

[Jones98a] Jones, J.W. Weatherill, N.P. 1998. A flexible approach to expression evaluation within a computational engineering environment. *International Journal of Numerical Methods in Fluids*, **28**, pp. 1183 - 1197, 1998.

[Jones98b] Jones, J. W. Weatherill, N.P. 1998. Parallel visualisation of computational engineering data. *Advances in Computational Mechanics with High Performance Computing, (ed, B H V Topping), Civil-Comp Press, Edinburgh*, pp. 1 – 9. 1998.

[Jones98c] Jones, J. Weatherill, N.P. 1998. The visualisation of large unstructured grid data sets. *Proceedings of the 6th International Conference on Numerical Grid Generation in Computational Field Simulation, (ed. M. Cross, P. Eiseman, J. Hauser, B. K. Soni and J. F. Thompson), pub. NSF Research Center, Mississippi State University, USA*, pp. 899 – 914. July 1998.

[Jones99] Jones, J.W. Weatherill, N.P. 1999. Visualisation of large unstructured grids within a parallel framework. *AIAA Paper 99-3289. Presented at the AIAA CFD Conference, Hampton, Virginia, USA*. June 1999.

[Jones00] Jones, J.W. Weatherill, N.P. 2000. A parallel environment for large scale computational engineering simulation. *Proceedings of the 7th International Conference on Numerical Grid Generation in Computational Field Simulation, British Columbia, Canada*. September 2000.

[Jones02] Jones, J. Weatherill, N.P. 2002. Techniques for visualising large unstructured grid data-sets. *Applied Mathematical Modelling, 2002 (submitted).*

[Karypis98] Karypis, G. Kumar, V. 1998. Metis: A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reduced orderings of sparse matrices. http://www-users.cs.umn.edu/~karypis/metis/metis/files/manual.pdf.

[Karypis02] Karypis, G. Schloegel, K. Kumar, V. 2002. ParMetis: Parallel graph partitioning and sparse matrix ordering library. http://www-users.cs.umn.edu/~karypis/metis/parmetis/files/manual.pdf.

[Keahey02] Keahey, K. Fredian, T. Peng, Q. Schissel, D.P. Thompson, M. Foster, I. Greenwald, M. McCube, D. 2002. Computational Grids in Action: The National Fusion Collaboratory. *Future Generation Computer Systems*, **18** (8), pp. 1005 – 1015, October 2002.

[Kornmann99] Kornmann, D. 1999. Fast and Simple Triangle Strip Generation. *VMS Finland, Espoo, Finland.* 1999

[Larwood01] Larwood, B.G. Weatherill, N.P. Hassan, O. Morgan, K. 2001. The generation of large unstructured meshes on parallel platforms. *ACME 2001 9th Annual Conference of the Association for Computational Mechanics in Engineering, Univesity of Birmingham*, pp. 45 – 48.

[Levine92] Levine, J.R. Mason, T. Brown, D. 1992. Lex & Yacc. O'Reilly & Associates. October 1992.

[Maplesoft] MAPLE. Waterloo Maple Inc. http://www.maplesoft.com/main.shtml.

[Marchant96] Marchant, M.J. Weatherill, N.P. Turner-Smith, E. Zheng, Y. Sotirakos, M. 1996. A Parallel Simulation User Environment for Computational Engineering. *Proceedings of the 5th Intrnational Conference on Numerical Grid Generation in Computational Field Simulation, Mississippi State University, MS.* April 1996.

[MathWorks] MATLAB. The MathWorks. http://www.mathworks.com.

[Mehrotra98] Mehrotra, P. van Rosendale, J. Zima, H. 1998. High Performance Fortran: History, status and future. *Parallel Computing*, **24**, pp. 325 – 354. 1998.

[Mezentsev00] Mezentsev, A. Hassan, O. Weatherill, N.P. 2000. Geometric model analyses for unstructured surface mesh generation. *Proceedings of the 7th Internation Conference on Numerical Grid Generation in Computational Field Simulation, September 2000, Whistler, BC.*

[Microsoft95] Microsoft Corporation. 1995. The Component Object Model Specification Version 0.9. http://www.microsoft.com/Com/resources/comdocs.asp. October 24, 1995.

[Microsoft97] Microsoft Corporation. 1997. Distributed Component Object Model Protocol-DCOM/1.0 draft. http://www.microsoft.com/Com/resources/comdocs.asp. November 1996.

[Microsoft95] Microsoft Corporation. 1995. Direct-X. http://www.microsoft.com/windows/directx. 2002.

[Morgan00] Morgan, K. Hassan, O. Pegg, N.E. Weatherill, N.P. 2000. The simulation of electromagnetic scattering in piecewise homogenous media using unstructured grids. *Computational Mechanics*, **25**, pp. 438 – 447.

[Nag] Numerical Algorithms Group. IRIS Explorer. http://www.nag.co.uk/Welcome_IEC.html.

[Neider93] Neider, J. Davis, T. Woo, M. 1993. Open-GL Programming Guide. Addison-Wesley Publishing Company.

[Northall02] Northall, J.D. 2002. Overview of the SWIPE System. *Rolls-Royce Internal Memo*. September 2002.

[Nye88] Nye, A. 1988. Volume One: Xlib Programming Manual for Version 11. O'Reilly & Associates, Inc. November 1988.

[Nye90] Nye, A. 1990. Volume Two: Xlib Reference Manual for Version 11. O'Reilly & Associates, Inc. April 1990.

[Nye93] Nye, A. O'Reailly. T. 1993. Volume Four : X Toolkit Intrinsics Programming Manual. O'Reilly & Associates Inc. April 1993.

[OGL-ARB92] OpenGL Architecture Review Board. 1992. Open-GL Reference Manual. Addison Wesley Publishing Company.

[OOC99] Object Oriented Concepts, Inc. 1999. ORBacus for C++ and Java Version 3.1.3. Object Oriented Concepts Inc. 1999.

[OSF93] Open Software Foundation. 1993. OSF/Motif Programmers Guide Release 1.2. Prentice Hall PTR. 1993.

[OSF95] Open Software Foundation. 1995. OSF/Motif Programmers Reference Release 2.0. Prentice Hall PTR. 1995.

[Paxson98] Paxson, V. 1998. Flex, version 2.5.4 – A Fast Scanner Generator. http://www.fsf.org/manual/flex-2.5.4/flex.html. November 1998.

[PHIG88] PHIGS+ Committee (chaired by Andries van Damn). 1988. PHIGS+ Functional Description, Revision 3.0. *Computer Graphics,* **22** (3), pp. 125 – 218. July 1988.

[Plansky95] Plansky, D. 1995. Particle3 User's Guide. http://raphael.mit.edu/visual3/P3manual.ps. January 1995.

[Purdue02a] ELLPACK : Software for Solving Elliptic Problems. www.cs.purdue.edu/ellpack.

[Purdue02b] PDELab. http://www.webpdelab.org.

[Ramakrishnan01] Ramakrishnan, N. Watson, L.T. Kafura, D.G. Ribbens, C.J. Shaffer, C.A. 2001. Programming Environments for Multidisciplinary Grid Communities. *Concurrency – Practice and Experience*, **14**, pp. 1 – 35. 2002.

[Ravinshankar00] Ravinshankar, L. Singh, K.P. 2000. Grid-View – An interactive software for the visualization and analysis of 3D multi-block structured grids and flow field. *Proceedings of the 7th International Conference on Numerical Grid Generation in Computational Field Simulations, Whistler, BC.* pp. 839 – 850. September 23-28, 2000.

[Reinhard98] Reinhard, K. 1998. Multiresolution Representations for Surfaces Meshes based on the Vertex Decimation Method. *Computers & Graphics*, **22** (1), pp. 13 – 26.

[Rice86] Rice, J.R. Boisvert, R.F. 1986. Solving elliptic problems using ELLPACK. *Springer, New York, 1985*. 1986.

[Risk96] Risk, I. 1996. CAESAR Final Report. *BAE Systems Reference: CAESAR/TR/BAE/IR9610l6/1*. October 1996.

[Robb99] Robb, R.A. 1999. Visualization in biomedical computing. *Parallel Computing*, **25**, pp. 2067 – 2110. 1999.

[Rossum02a] van Rossum, G. 2002. Python Tutorial. http://www.python.org/doc/current/download.html. 2002.

[Rossum02b] van Rossum, G. 2002. Python Reference Manual. http://www.python.org/doc/current/download.html. 2002.

[Rossum02c] van Rossum, G. 2002. Python Library Reference. http://www.python.org/doc/current/download.html. 2002.

[Rossum02d] van Rossum, G. 2002. Extending and Embedding the Python Interpreter. http://www.python.org/doc/current/download.html. 2002.

[Rossum02e] van Rossum, G. 2002. Python/C API Reference manual. http://www.python.org/doc/current/download.html. 2002.

[Rossum02f] van Rossum, G. 2002. Installing Python Modules. http://www.python.org/doc/current/download.html. 2002.

[Rossum02g] van Rossum, G. 2002. Distributing Python Modules. http://www.python.org/doc/current/download.html. 2002.

[Rossum02h] van Rossum, G. 2002. Documenting Python. http://www.python.org/doc/current/download.html. 2002.

[Rowse00] Rowse, D. 2000. JULIUS Final Report – D5.1.4.6. Internal Report – BAE Systems ATC-S Project Number 076126. June 2000.

[Schmidt95a] Schmidt, D.C. Vinoski, S. 1995. Object Interconnections: Introduction to Distributed Object Computing (Column 1). *SIGS C++ Report Magazine.* January 1995.

[Schmidt95b] Schmidt, D.C. Vinoski, S. 1995. Object Interconnections: Modelling Distributed Object Applications (Column 2). *SIGS C++ Report Magazine.* February 1995.

[Schmidt99] Schmidt, D.C. Wang, N. Vinoski, S. 1999. Object Interconnections: Collocation Optimisations for CORBA (Column 18). *SIGS C++ Report Magazine.* September 1999.

[Scott-McRae00] Scott McRae, D. 2000. r-Refinement Grid Adaptation Algorithms and Issues. *Computer Methods in Applied Mechanics and Engineering*, **189**, pp. 1161 – 1182.

[Shevare00] Shevare, G.R. Bhagat, N. Kadam, N. Bakre, S. 2000. IITZeus: A Versatile Geometric Modeling and Grid Generation Software. *Proceedings of the 7th International Conference on Numerical Grid Generation in Computational Field Simulations, Whistler, BC.* pp. 861 - 869. September 23-28, 2000.

[Shimrat62] Simrat, M. 1962. Algorithm 112, Position of Point Relative to Polygon. *CACM*, p. 434. August 1962.

[Sommer99] Sommer, O. Dietz, A. Westermann, R. Ertl, T. 1999. An interactive visualization and navigation tool for medical volume data. *Computers & Graphics*, **23**, pp. 233 – 244. 1999.

[Sorenson01] Sorenson, K.A. Hassan, O. Morgan, K. Weatherill, N.P. 2002. An agglomerated unstructured hybrid mesh methopd for turbulent compressible flows. *Computational Fluid Dynamics Journal*, Special No. 2001, pp. 690 – 699, 2001.

[Stevens90] Stevens, WR. 1990. UNIX Network Programming. Prentice-Hall Inc.

[Sturler97] de Sturler, E. Loher, D. 1997. Parallel iterative solvers for irregular sparse matrices in High Performance Fortran. *Future Generation Computer Systems*, **13**, pp. 315 – 325. 1997.

[Sunderam90] Sunderam, V.S. 1990. PVM: A Framework for Parallel Distributed Computing. *Concurrency: Practice and Experience*, **2** (4), pp. 315 – 339. December, 1990.

[Turner-Smith96a] Turner-Smith, E.A. Zheng, Y. Sotirakos, M. Parallel Simulation User Environment (PSUE) User Documentation. *Internal Report.* September 1996.

[Turner-Smith96b] Turner-Smith, E.A. Zheng, Y. Sotirakos, M. Parallel Simulation User Environment (PSUE) System Documentation. *Internal Report*. September 1996.

[Turner-Smith98] Turner-Smith, E.A. Weatherill, N.P. Marchant, M.J. Jones, J. A computer environment for unstructured grid generation. *Proceedings of the 6th International Conference on Numerical Grid Generation in Computational Field Simulation, (ed. M. Cross, P. Eiseman, J. Hauser, B. K. Soni and J. F. Thompson), pub. NSF Research Center, Mississippi State University, USA*, pp. 889 - 898. July 1998.

[Vilacis99] Vilacis, J. Govindaraju, M. Whitaker, A. Breg, F. Deuskar, P. Temko, B. Gannon, D. Bramley, R. 1999. CAT: A High Performance, Distributed Component Architecture Tookit for the Grid. *Proceedings of the HPDC'99*. 1999.

[Vinoski97] Vinoski, S. 1997. CORBA: Integrating Diverse Applications Within Distributed Heterogenous Environments. *IEEE Communications Magazine,* **14** (2). February 1997.

[Walshaw97] Walshaw, C. Cross, M. Everett, M.G. 1997. Parallel Dynamic Graph Partitioning for Adaptive Unstructured Meshes. *Journal of Parallel and Distributed Computing*, **47**, pp. 102 – 108.

[Walshaw01] Walshaw, C. Cross, M. 2001. Multilevel Mesh Partitioning for Heterogenous Communications Networks. *Future Generation Computer Systems*, **17**, pp. 601 – 623.

[Walshaw02a] Walshaw, C. 2002. The serial JOSTLE library user guide : Version 3.0. http://www.gre.ac.uk/~c.walshaw/jostle/jostleslib.ps.gz.

[Walshaw02b] Walshaw, C. 2002. The parallel JOSTLE library user guide : Version 3.0. http://www.gre.ac.uk/~c.walshaw/jostle/jostleplib.ps.gz.

[Weatherill94a] Weatherill, N.P. Hassan, O. 1994. Efficient three-dimensional Delaunay triangulation with automatic point creation and imposed boundary constraints. *International Journal of Numerical Methods in Fluids*, **37**, pp. 2005 – 2039.

[Weatherill94b] Weatherill, N.P. Hassan, O. Marchant, M.J. Marcum, D.L. 1994. Grid adaptation using a distribution of sources applied to inviscid compressible flow simulation. *Internation Journal of Numerical Methods in Fluids*, **19**, pp. 739 – 764.

[Weatherill99] Weatherill, N.P. Turner-Smith, E.A. Marchant, M.J. Hassan, O. Morgan, K. 1999. An integrated software environment for multi-disciplinary computational engineering. *Engineering Computations*, **16** (8), pp. 913 - 933, 1999.

[Weatherill00a] Weatherill, N.P. Morgan, K. Hassan, O. Jones, J.W. 2000. Large-Scale aerospace simulations using unstructured grids. *Computational Mechanics for the 21st Century, Chapter 12*. Saxe-Coburg Publications, Editor B.H.V. Topping, 2000.

[Weatherill00b] Weatherill, N.P. Hassan, O. Morgan, K. Said, R. 2000. The generation of large unstructured meshes by parallel Delaunay. *ECCOMAS 2000: European Congress on Computational Methods in Applied Sciences and Engineering, Barcelona, 2000.*

[Weatherill01a] Weatherill, N.P. Hassan, O. Morgan, K. Jones, J.W. Larwood, B.G. Sorenson, K. 2001. Next generation large-scale aerospace simulations on unstructured grids. *ICFD Conference, Oxford, UK, March 2001.*

[Weatherill01b] Weatherill, N.P. Hassan, O. Morgan, K. Jones, J.W. Larwood, B. 2001. Towards fully parallel aerospace simulations on unstructured meshes. *Engineering Computations,* **18**, (3/4), pp. 347 - 373, 2001.

[Weatherill02] Weatherill, N.P. Hassan, O. Morgan, K. Jones, J.W. Larwood, B.G. Sorenson, K. 2002. Aerospace Simulations on Parallel Computers using Unstructured Grids. *To appear, Special Issue of IJNMF*, 2002.

[Wolfram] Mathematica. Wolfram Research. http://www.wolfram.com.

[Wright01] Wright, R. 2001. Computing with MAPLE. Chapman & Hall / CRC. 2001.

[Xiao00] Xiao, Y. Ziebarth, J.P. 2000. FEM-based scattered data modelling and visualization. *Computers & Graphics*, **24**, pp. 775 – 789. 2000.

[Yang92] Yang, C.C. 1992. Effects of Coordinate Systems on Color Image Processing. *M.S. Thesis , Department of Electrical and Computer Engineering, the University of Arizona, Tucson, AZ.* 1992.

[Yang96] Yang, Z. Duddy, K. 1996 CORBA: A Platform for Distributed Object Computing. *ACM Operating Systems Review*, **30**, 1996.

[Yang97] Yang, C.C. Rodriguez, J.J. 1997. Efficient Luminance and Saturation Processing Techniques for Color Images, *Journal of Visual Communication and Image Representation*, **8** (3), pp 263 – 277.

[Zheng00] Zheng, Y. Weatherill, N.P. Turner-Smith, E.A. Sotirakos, M.I. Marchant, M.J. HASSAN, O. 2000. Visual steering of grid generation in a parallel simulation user environment. *Enabling Technologies for Computational Sciences; Frameworks, Middleware and Environments, Chapter 27.* Kluwer Academic Publishers, 2000.

Appendix A. EQUATION EDITOR – EQUATE

During the development of PROMPT and PSUE II, it became evident that there was a requirement to allow the user to define his/her own mathematical expressions for data such as solution variables, mesh analysis criteria and error estimators for mesh refinement.

This would require an expression evaluation system that could compute expressions based on edge-based, face-based, element-based, as well as the usual nodal values. To meet these requirements a module, called EQUATE (EQUATion Editor) was developed.

During its development a number of key requirements were highlighted:

- *The ability to cope with different data-types*
  To allow EQUATE to be utilised in these three different stages it must be able to evaluate equations that are based on variables computed at nodes, edges, faces and cells, even though the input data to EQUATE is always node based. This is achieved by grouping the individual nodal values into tuples to represent the higher level entities. For example, grouping nodes at either end of an edge into pairs creates a pair (2-tuple) which represents an edge value. This would be useful in mesh adaptation, where the rate of change of solution values along an edge could easily by calculated by dividing the difference between the two end nodal values by the distance between them.
- *Completeness*
  To enable the user to quickly and easily define new variables EQUATE contains all of the usual mathematical operators (e.g. $+$, $-$, $*$, etc..) and functions (e.g. log(x), e^x, sin(x), etc.). A number of other functions which could be defined using the generic operators and functions are included in EQUATE for efficiency purposes due to their frequent usage (e.g. edge length, face area, cell volume, dot product, etc.).
- *Speed*
  Evaluating a particular variable in EQUATE on a large mesh will require the same expression to be evaluated many millions of times. Obviously the time to evaluate an expression must be as quick as possible in order to provide a reasonable response time to the user.

A.1. Definition of a Generic Mathematical Expression in EQUATE

In EQUATE, a mathematical expression consists of classes of constants, variables, operators and functions, i.e. a generic mathematical expression, E, is expressed as:

$E = f(C, V, O, G)$ where

$C = (c_1, c_2, \ldots, c_n)$ is the class of constants,

e.g. 1, 5, λ, π, -1.5.

$V = (v_1, v_2, \ldots, v_n)$ is the class of variables,

e.g. p (pressure), ρ (density).

$O = (o_1, o_2, \ldots, o_n)$ is the class of operators,

e.g. $\alpha + \beta, \alpha - \beta, \alpha \times \beta, \alpha / \beta, \alpha^{\beta}$

$G = (g_1, g_2, \ldots, g_n)$ is the class of functions,

e.g. $\sin(\phi), \log(\phi), e^{\phi}, \min(\phi, \varphi)$

For example, the equation:

$$\left(\frac{\rho^{\lambda} + 2}{p^{\log(\lambda)} - 1} \right)^{3.5}$$

is expressed in EQUATE as:

$E = f(C, V, O, G)$ where

$$C = \left(\underset{c_1}{1}, \underset{c_2}{2}, \underset{c_3}{3.5}, \underset{c_4}{\lambda} \right)$$

$$V = \left(\underset{v_1}{p}, \underset{v_2}{\rho} \right)$$

$$O = \left(\underset{o_1}{\alpha + \beta}, \underset{o_2}{\alpha - \beta}, \underset{o_3}{\alpha / \beta}, \underset{o_4}{\alpha^{\beta}} \right)$$

$$G = \left(\underset{g_1}{\log(\alpha)} \right)$$

where

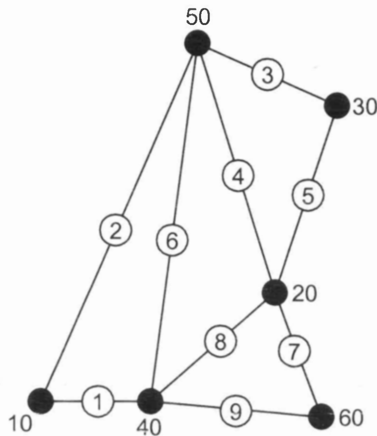
$$E = \left(\frac{(v_2^j) \wedge_{o_4} (c_4^j) +_{o_1} (c_2^j)}{(v_1^j) \wedge_{o_4} g_1(c_4^j) -_{o_2} (c_1^j)} \right) \wedge_{o_4} (c_3^j) \text{ where } j \text{ is the cardinality of the tuple}$$

A.2. EQUATE Syntax and Semantics

As mentioned above, EQUATE can evaluate expressions based on data defined at nodes, edges, faces or elements. The input to EQUATE consists entirely of nodal values; either solution values produced by the equation solver or geometric values based on the mesh itself.

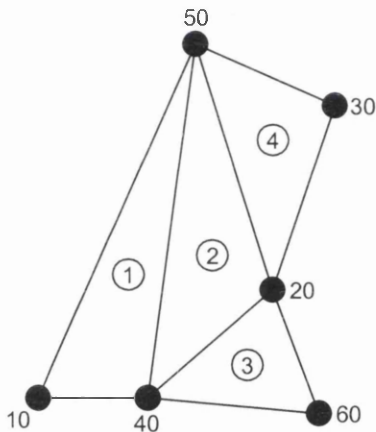
In order for EQUATE to evaluate expressions based on entities other than nodes, the nodal values are grouped into tuples of varying cardinality to represent the desired entity. An example of the tuples formed to allow edge-based expressions to be evaluated is

shown in Figure 229. Figure 230 shows a similar grouping of nodes into 3-tuples for triangular faces. Other face and element types are constructed in a similar manner.



Edge	Node 1	Node 2
1	10	40
2	10	50
3	50	30
4	50	20
5	20	30
6	50	40
7	20	60
8	20	40
9	40	60

Figure 229 – Example of representing edges by forming node pairs



Face	Node 1	Node 2	Node 3
1	10	50	40
2	40	50	20
3	60	40	20
4	30	20	50

Figure 230 – Example of representing triangular faces by forming 3-tuples of nodes

New variables can be defined using any valid combination of constants, built-in variables (both geometric and generic solver variables), operators and functions.

A.3. Syntax and Semantics of Expressions

An expression can be built from a number of sub-expressions separated by semi-colons. Each sub-expression (except the last) has an assignment section at the front. This causes the value of the sub-expression to be assigned to the specified local variable. The last sub-expression has no assignment section since the result of the entire expression is automatically assigned to the resultant value.

The purpose of the sub-expressions is to mimic the normal method of constructing mathematical equations, with the complete expression split into a number of more

manageable components. It also means that is any sub-expression is used more then once in the global expression then it needs to be calculated once only with the values being stored for later use.

A simple example of this is shown below. The edge-based expression:

```
( ( Density.n2 - Density.n1 ) / Length )
/ ( ( Density.n2 - Density.n1 ) / Length - 1 )
```

can be simplified to:

```
a = ( Density.n2 - Density.n1 ) / Length

a / ( a - 1 )
```

An expression containing sub-expressions is semantically valid if, and only if, the expression would be valid if the sub-expressions were inserted into the global expressions to form one expression.

A.4. Syntax and Semantics of Operators

In order to be able to build expressions using variables and constants, it is necessary to combine them using mathematical operators and functions. The operators included in EQUATE consist of the usual mathematical operators; $a + b$, $a - b$, $a * b$, etc.

Operators are like functions (in fact operators could be replaced with an equivalent function) except that they have the restriction that the operands must be n-tuples of the same cardinality with the result of the operator being an n-tuple with the same cardinality.

The operators broadly fall into one of two categories, unary and binary. The unary operators, in EQUATE, take one n-tuple as an argument and return one n-tuple as a result. The only unary operator included within EQUATE is the negation operator which is defined as:

$$\text{Let } A^n = (a_1, a_2, \ldots, a_n)$$

then

$$-\left(A^n\right) \rightarrow B^n \text{ where } B^n = \left(-a_1, -a_2, \ldots, -a_n\right)$$

EQUATE also contains five binary operators, $a + b$, $a - b$, $a * b$, a / b and a^b. These have similar definitions to the unary operator,

$$\text{Let } A^n = (a_1, a_2, \ldots, a_n), \text{ and } B^n = (b_1, b_2, \ldots, b_n)$$

then

$$A^n \circ B^n \rightarrow c^n \text{ where } C^n = (a_1 \circ b_1, a_2 \circ b_2, \ldots, a_n \circ b_n)$$

where

$$x \circ y = x + y, x - y, x \times y, x \div y, x^y$$

As shown above, the two arguments of each of the operators must have the same cardinality with the result of the operator having the same cardinality as the arguments.

A.5. Syntax and Semantics of Functions

There are a substantial number of functions built into EQUATE. These include the usual exponential, logarithmic and trigonometric functions. The exponential and logarithmic functions are e^x, $\log_e(x)$ and $\log_{10}(x)$. The trigonometric functions are $\sin(x)$, $\cos(x)$, $\tan(x)$, $\sin^{-1}(x)$, $\cos^{-1}(x)$ and $\tan^{-1}(x)$. The other functions do not fit into a particular category so are classed as miscellaneous functions. These are $\sqrt{x}$, $\min(x, y)$ and $\max(x, y)$.

These functions are either unary or binary functions and have the same properties as unary and binary operators. The unary functions are defined as follows:

$$\text{Let } A^n = (a_1, a_2, \ldots, a_n)$$

then

$$f(A^n) \rightarrow B^n \text{ where } B^n = (f(a_1), f(a_2), \ldots, f(a_n))$$

where

$$f(x) = e^x, \log_e(x), \log_{10}(x), \sin(x), \cos(x), \tan(x), \sin^{-1}(x), \cos^{-1}(x), \tan^{-1}(x), \sqrt{x}$$

The binary functions are defined as:

$$\text{Let } A^n = (a_1, a_2, \ldots, a_n) \text{ and } B^n = (b_1, b_2, \ldots, b_n)$$

then

$$f(A^n, B^n) \rightarrow C^n \text{ where } C^n = (f(a_1, b_1), f(a_2, b_2), \ldots, f(a_n, b_n))$$

where

$$f(x, y) = \min(x, y), \max(x, y)$$

There are also a number of functions in EQUATE that operate on an n-tuple and reduce it to a single number (1-tuple). These are tmin(x), tmax(x), tavg(x) and trms(x). These functions could be performed using a combination of EQUATE's more basic functions but since they are common operations they have been built in both for convenience and efficiency. They are defined as follows:

$$\text{Let } A^n = (a_1, a_2, \ldots, a_n) \text{ and } C^1 = (c_1)$$

then

$$\text{tmin}(x) \rightarrow C^1 \text{ where } C^1 = \min_{i=1}^{n}(a_i)$$

$$\text{tmax}(x) \rightarrow C^1 \text{ where } C^1 = \max_{i=1}^{n}(a_i)$$

$$\text{tavg}(x) \rightarrow C^1 \text{ where } C^1 = \frac{\sum_{i=1}^{n} a_i}{n}$$

$$\text{trms}(x) \rightarrow C^1 \text{ where } C^1 = \sqrt{\frac{\sum_{i=1}^{n} a_i^2}{n}}$$

The last group of functions in EQUATE compress n single numbers (1-tuples) into one n-tuple. These are given the names *tuple2*, *tuple3*, ..., *tuple8* depending on how many arguments they take. These are defined as:

$$\text{Let } c_1, c_2, \ldots, c_8 \text{ be single numbers (1 - tuples)}$$

then

$$\text{tuple2}(c_1, c_2) \rightarrow A^2 \text{ where } A^2 = (c_1, c_2)$$
$$\text{tuple3}(c_1, c_2, c_3) \rightarrow A^3 \text{ where } A^3 = (c_1, c_2, c_3)$$
$$\vdots$$
$$\text{tuple8}(c_1, c_2, c_3, c_4, c_5, c_6, c_7, c_8) \rightarrow A^8 \text{ where } A^8 = (c_1, c_2, c_3, c_4, c_5, c_6, c_7, c_8)$$

The cardinality of the tuple that may be generated is dependent upon the semantic requirements at that point in the expression. Any attempt to generate a tuple that is not compatible results in an error.

A.6. Syntax and Semantics of Variables

There are two types of variable that may be used inside an expression:

Built-in Solver Variables

For node based expressions, these variables take on the solution value at each mesh node in turn. For edge based expressions, the variables take on the form of a pair of values at either end of each edge in the mesh. Face and element variables are constructed in a similar manner.

Geometric Variables

These variables are constructed in the same way as the built-in solver variables but instead of containing solution values they contain geometric quantities. In EQUATE, there are three such nodal variables:

X – The x-coordinate of the node
Y – The y-coordinate of the node
Z – The z-coordinate of the node

A small number of higher-order variables were also included in EQUATE as generic variables due to their frequency of use. These are:

Length(Edge)	– The length of an edge
Area(Face)	– The area of a face.
Volume(Cell)	– The volume of a cell.
CentreX(*All Types*)	– The x-coordinate of the centre of the edge, face or cell.
CentreY(*All Types*)	– The y-coordinate of the centre of the edge, face or cell.
CentreZ(*All Types*)	– The z-coordinate of the centre of the edge, face or cell.

Syntactically the definition of a variable in EQUATE is the same as an identifier in the language 'C', i.e. a lower or upper case letter or underscore followed by any combination of lower or upper case characters, underscores or digits.

Like a constant, the semantic definition of a variable is an n-tuple of numbers. Unlike a constant, the cardinality of the tuple is fixed and if it is does not match the requirements of the expression then an error is generated. The cardinality of a variable depends on the data type it represents, as shown below:

$$V^n = (e_1, e_2, \ldots, e_n) \text{ where } n = \begin{cases} 1 \text{ (node)} \\ 2 \text{ (edge)} \\ 3 \text{ (triangular face)} \\ 4 \text{ (quadrilateral face / tetrahdral cell)} \\ 5 \text{ (pyramidal cell)} \\ 6 \text{ (prismatic cell)} \\ 8 \text{ (hexahedral cell)} \end{cases}$$

Accessing Variable Sub-components

The sub-components of any variable with a tuple of cardinality greater than 1 may also be accessed and used in expressions. This is achieved by adding a suffix `.n`, `.e` or `.f` followed by the node, edge or face number to the end of the variable name. An example of accessing the sub-components of an edge variable is computing the gradient of the density along each edge in the mesh:

```
(Density.n2 - Density.n1)
/sqrt( ( X.n2 - X.n1 ) ^ 2
     + ( Y.n2 - Y.n1 ) ^ 2
     + ( Z.n2 - Z.n1 ) ^ 2 )
```

which could also be simplified to:

```
( Density.n2 - Density.n1 ) / Length
```

A.7. Syntax and Semantics of Constants

To the user, constants are simply floating-point (real) numbers. However, in EQUATE, constants are treated as a tuple consisting of one or more copies of the same value. The cardinality of the tuple is automatically determined by EQUATE so as to satisfy the semantic requirements of the expression. For example, when defining a node-based expression a constant is stored as a single number (1-tuple); in an edge-based expression a constant is stored as a 2-tuple consisting of 2 copies of the given constant.

In general an EQUATE constant is defined as an n-tuple:

$$C^n = \left(\underbrace{c, c, \ldots, c}_{n \text{ times}} \right)$$

where c is the constant value and n is chosen to satisfy the semantic requirements of the expression.

Constants may also be defined directly as a tuple by surrounding a list of comma-separated constants with curly brackets. The number of entries in the tuple must satisfy the semantic requirements of the expression. Examples of constants are:

```
12, 12.45, -1.2e-5, {3.0,-1,5e2,-1E0.5}
```

A.8. Some Implementation Details

In order to maximise efficiency, the input expression is parsed once using a combination of Flex [Paxson98] and Bison [Donnelly02] to produce a tokenised form of the expression which is then stored in a tree-like structure. This structure is then traversed for each set of data items to compute the results. Input data is stored in flat arrays dimensioned n by m, where n is the number of values in the tuple and m is the number of data items.

A.9. Performance Figures

As mentioned previously, one of the key requirements for EQUATE was performance. In order to evaluate the performance of EQUATE a number of test expressions were created ranging from simple expressions (with simple operators) through to more complex expressions (with functions that are known to be more compute intensive). These were than evaluated on a mesh containing four million nodes and 11.9 million edges using both EQUATE and a hard-wired C code.

The results, along with the relative speed comparisons are shown below. The tests were performed on a Silicon Graphics Challenge with 512Mb of memory.

Simple Expressions

The following sets of expressions are very simple using only the basic arithmetic operations. These expressions were chosen in order to highlight what was to be

expected the worst-case scenario for EQUATE where the overhead for the tree traversal was expected to be significant.

	Equation	EQUATE time (s)	Hard-wired C-code time (s)	Relative Speed
1	`Density + Temperature`	8.6	2.8	3.1
2	`Density.n1 * Temperature.n2 + Pressure.n2`	47.3	22.7	2.1
3	`( Density.n2 - Density.n1 ) / Length`	50.6	25.6	2.0

Complex Expressions

These expressions are somewhat more complex than the previous examples in the sense that the functions that are used are known to be more compute intensive. These expressions were expected to show EQUATE fairing a little better against the C code since the overhead for the tree traversal should be less significant compared with the time computing the functions.

	Equation	EQUATE time (s)	Hard-wired C-code time (s)	Relative Speed
1	`sin( Density ) ^ 2 + cos( Density ) ^ 2`	40.9	29.5	1.4
2	`( log( Temperature.n2^4 ) + log( Temperature.n1^2 ) )^2`	201.2	161.2	1.25
3	`Density.n32 ^ Density.n1 / ( Temperature.n1 - Temperature.n2 )`	128.6	98.6	1.3

Explanation of Times

As can be seen from the timings, the hard-wired C code was quicker in every case than EQUATE, but this was to be expected. For the simple expressions EQUATE was approximately 2 – 3 times slower, but this dropped to only 1.2-1.5 times slower for the more complex expressions.

The reason for this behaviour is easily explained if the total execution time is split into two parts:

$$T = C + F$$

where T is the total execution time,

C is the time evaluating actual operators and functions and
F is the time spent executing the surrounding flow constructs.

We shall use $T_{Equte}^{Complex}$ to denote the total execution time for the *complex* expressions using EQUATE; similarly for F and C.

We shall represent the relative speed of EQUATE to the hard-wired C code as,

$$S = \frac{T_{Hard}}{T_{Equate}}$$

It is obvious that $C_{Equate} = C_{Hard}$ for the same expression whether simple or complex since the same operators and functions are being evaluated. Therefore the relative speed, S, is directly dependent on the difference between F_{Equate} and F_{Hard}. It is also obvious from the results that as the expressions gets more complex (but still has the same number of operands) the relative speed of EQUATE to the hard-wired C code increases, i.e. $S^{Complex} < S^{Simple}$.

Expanding the above we have,

$$\frac{F_{Hard}^{Complex} + C_{Hard}^{Complex}}{F_{Equate}^{Complex} + C_{Equate}^{Complex}} < \frac{F_{Hard}^{Simple} + C_{Hard}^{Simple}}{F_{Equate}^{Simple} + C_{Equate}^{Simple}}$$

Now $C_{Equate} = C_{Hard}$, therefore substituting λ for C_{Equate} and C_{Hard}, we have;

$$\frac{F_{Hard}^{Complex} + \lambda^{Complex}}{F_{Equate}^{Complex} + \lambda^{Complex}} < \frac{F_{Hard}^{Simple} + \lambda^{Simple}}{F_{Equate}^{Simple} + \lambda^{Simple}}$$

Rearranging, we have;

$$\frac{F_{Hard}^{Complex} + \lambda^{Complex}}{F_{Hard}^{Simple} + \lambda^{Simple}} < \frac{F_{Equate}^{Complex} + \lambda^{Complex}}{F_{Eauate}^{Simple} + \lambda^{Simple}}$$

Since the expression has the same number of operands, $F_{Simple} = F_{Complex}$ we can perform a similar substitution to the one above, leaving us with;

$$\frac{\phi_{Hard} + \lambda^{Complex}}{\phi_{Hard} + \lambda^{Simple}} < \frac{\phi_{Equate} + \lambda^{Complex}}{\phi_{Equate} + \lambda^{Simple}}$$

Now, ϕ_{Hard} and ϕ_{Equate} are constant regardless of the complexity of the expression; therefore it can be seen that as the complexity of the expression increases the two sides of the relation converge to equality.