# Structure from Soup

Written by: John Beech
john.beech@student.manchester.ac.uk

Supervisor: Toby Howard

School of Computer Science
The University of Manchester

April 2008

## Title page

# Abstract

To break unstructured 3D polygon data into named structures, a practical software tool was built that allows a user to interact with model data, identify groups of polygons, and create a structured hierarchy. Following an agile software development process 3D visualisation and editing software can be developed. Using tools created with C++, OpenGL and OpenSceneGraph, unstructured model data is broken into a navigable hierarchy. Common features of 3D editors are analysed to implement and demonstrate alternative tools for 3D data manipulation. Possible uses of structured polygon data, for storage, search, semantic tagging, and sharing are considered.

# Acknowledgements

Thanks to Toby Howard for introducing and supervising this project, the support of the people in the OpenSceneGraph IRC channel, and to staff, students and friends at the University of Manchester.

Thanks to Professor Avideh Zakhor for permission to use *Figure 3. Source of polygon soup: Laser scanned street*, based on data from the UC Berkeley City Modelling Project[46].

# Table of Contents

Chapter title Page

Title page....1
Abstract....2
Acknowledgements....3
Table of Contents....4
List of figures....5
Chapter 1 Introduction....6
1.1 Report structure....7
Chapter 2 Background and initial research....8
2.1 Problem background....8
2.2 Existing tools and research....9
2.3 Chapter summary....11
Chapter 3 Design....12
3.1 Problem analysis....12
3.2 Goals of the project....13
3.3 Development toolbox....14
3.4 Scene graphs....15
3.5 Prototype driven development....18
3.6 User interface abstraction....19
3.7 Primitive data structures....20
3.8 Program design....22
3.9 Chapter summary....24
Chapter 4 Implementation....25
4.1 Algorithms for data manipulation....25
4.2 View interactions....27
4.3 Primitive interactions....32
4.4 Geometry selection....35
4.5 Scene graph interactions....37
4.6 A note on refactoring....40
4.7 Chapter summary....40
Chapter 5 Results....41
5.1 Using the editor....41
5.2 Cleaning primitive data....44
5.3 Loading data, persistent storage....44
5.4 Exporting knowledge....44
5.5 Chapter summary....45
Chapter 6 Testing and evaluation....46
6.1 Continuous testing and evaluation....46
6.2 Notes on build progression....47
6.3 Sample work flow example....47
6.4 Chapter summary....48
Chapter 7 Conclusions....49
7.1 Practicality of tool....49
7.2 Extensions to the editor....49
7.3 Development process....50
7.4 Success of the project....50
Chapter 8 References....52
Appendix A: Screenshots....54
Examples of 3D Editors....54
Using the Structure from Soup Editor....56
Appendix B: Program structure of final project....63
Appendix C: Key OpenSceneGraph class hierarchies....65

# List of figures

Except for *Figure 3.,* which has been used by permission of Professor Avideh Zakhor from the UC Berkeley City Modelling Project[46], all figures are drawn by John Beech for use in this report.

Figure Page


1. Primitives made up of vertices, line, triangle, triangle fan, triangle strip....6
2. Source of polygon soup: Unstructured warehouse model....8
3. Source of polygon soup: Laser scanned street, based on data from the UC Berkeley City Modelling Project [46]....9
4. Polygon data and rendered view for the letter “K”....9
5. Similar features of modern 3D editors....10
6. Requirements engineering - reasoning loop....12
7. Goal hierarchy structure....13
8. Scene graph of solar system....16
9. Split scene graph showing rendered root and data root....18
10. QT Designer UI files as part of a multiple-inheritance system....20
11. Relationships of data structures inside a geometry object ....21
12. Comparison of colour binding types....22
13. Class hierarchy and messages of the major program components....22
14. A comparison of the six available view modes....23
15. Illustration of finding vertices inside a sphere....26
16. Boundary box and radial distance search of vertex data....26
17. Difference of current and previous selections to locate boundary vertices....27
18. Summary of camera rotation, camera pan, and view scale in terms of camera position and camera target....28
19. Comparison of views after matrix rotations....29
20. Using the camera orbit tool to change the view....30
21. Illustration of dollying a camera along the view axis....31
22. Using the camera dolly tool to change the view....31
23. Using the camera pan tool to change the view....32
24. Polygon select interaction of volume select tool....33
25. Polygon select interaction of flood select tool....34
26. Creating a new node from a selection....35
27. Different view modes of a highlighted selection....36
28. Example of bounding box highlighting a selected cone (left)....36
29. Selecting an object using the select node tool....37
30. Features of the Scenegraph panel....37
31. Semantic soup displayed in the scene graph....38
32. Labelled screenshot of the structure from soup editor....41
33. User interaction work flow for adding structure to soup....42
34. A structured scene displayed in the editor....43
35. Complete sample work flow example of adding structure to soup....48

# Chapter 1 Introduction

The Structure from Soup Editor is an interactive tool designed for users to view 3D model data, select groups of polygons, and break an unstructured model into recognisable structures that humans can understand and computers can process.

The terms Structure, Soup, and Editor, are defined as follows:

- Structure is a meaningful grouping within a set of data. Structure can be as simple as a label that names a group of polygons, or a hierarchy of groups.
- Soup refers to polygon soup. Polygon soup is any set of unstructured 3D data containing vertices and primitives. A vertex is a 3D point in space, described as a vector with three numeric values. A primitive is a collection of vertices with instructions on how to connect those vertices, such as in a line, a triangle, a triangle fan, or a triangle strip *(See Figure 1)*.
- Editor refers to the software tool developed to allow a user to add structure to soup. The need for a software tool is justified later in the problem analysis section.

The following report describes the problem of polygon soup, the need for a software editor, the technical problems faced in producing the software, and the steps taken and tools used to solve these problems. The report also suggests other ways to deal with polygon soup and possible extensions to the project.

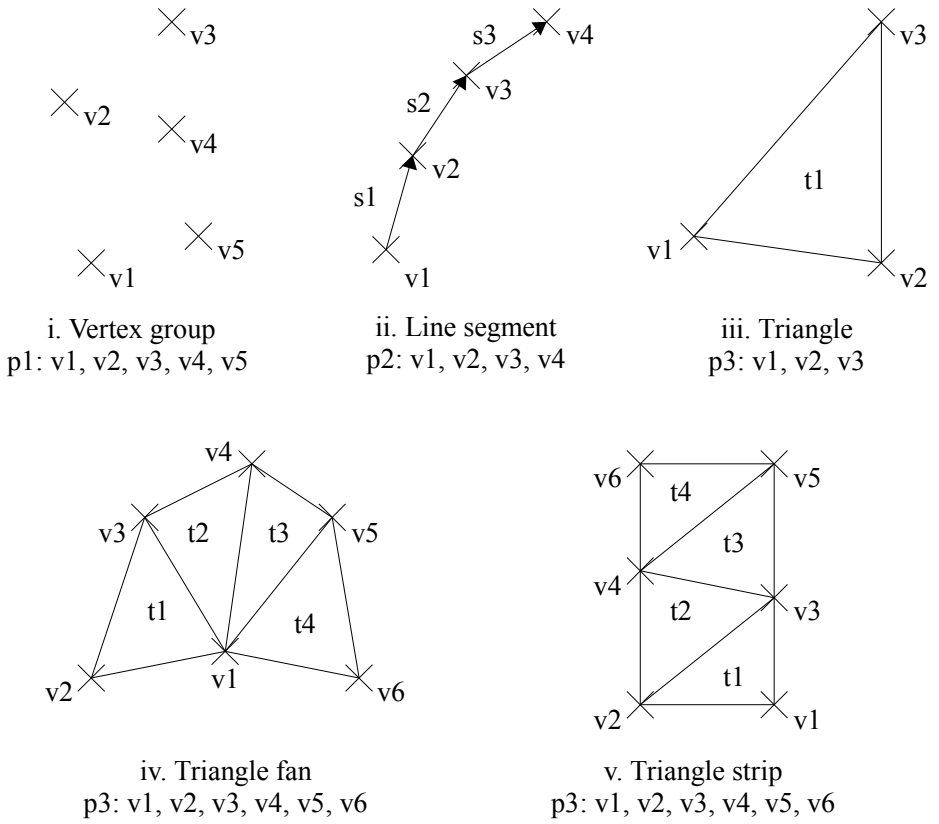



*Figure 1. Primitives made up of vertices, line, triangle, triangle fan, triangle strip*

## 1.1 Report structure

If you only need to know what the structure from soup editor is, *Chapter 5.1 Using the editor* explains its features, the available tools, and their intended use. *Chapter 2 Background and initial research* discusses the background of the problem and decisions made from the initial research. *Chapter 3 Design* contains information about software development techniques and the goals of the project. *Chapter 4 Implementation* contains in-depth descriptions of the algorithms and tools included in the editor. *Chapter 5 Results* summarises the implementation. *Chapter 6 Testing and evaluation* explains the merits of the chosen development methods, and *Chapter 7 Conclusions* summarises this report by concluding on the practicality of the tool, on possible extensions to the tool, and by criticising the development process.

The three appendices at the end of the report provide supplementary material, mostly diagrams and screenshots, useful for providing additional detail for some of the concepts referred to in this report.

# Chapter 2 Background and initial research

Large unstructured sets of polygon data are considered as polygon soup when there is no discernible grouping, naming, or hierarchy in place to describe the set of data. For example, the vertices that make up the wing of a plane are not known by simply looking at the data; either visual analysis of the rendered wing, or spatial analysis of the vertex data is needed to derive meaningful information from the raw data. This project looks primarily at the former option, using human ability to visually distinguish shapes to quickly build up a structured selection.

## 2.1 Problem background

Examples of polygon soup can be generated from laser scanning devices[1], optimized scenes with no source files, and unstructured VRML (Virtual Reality Modelling Language) files from the web. Optimized scenes and VRML are both examples of stripped information sources; when created, the structure of these datasets may have been included, but when published, structural data may have been stripped out. In some sense, optimization is necessary for gaining performance, but as data sources grow older, and source files are lost, the lack of structure hampers the ability for information managers to maintain these sources of data.

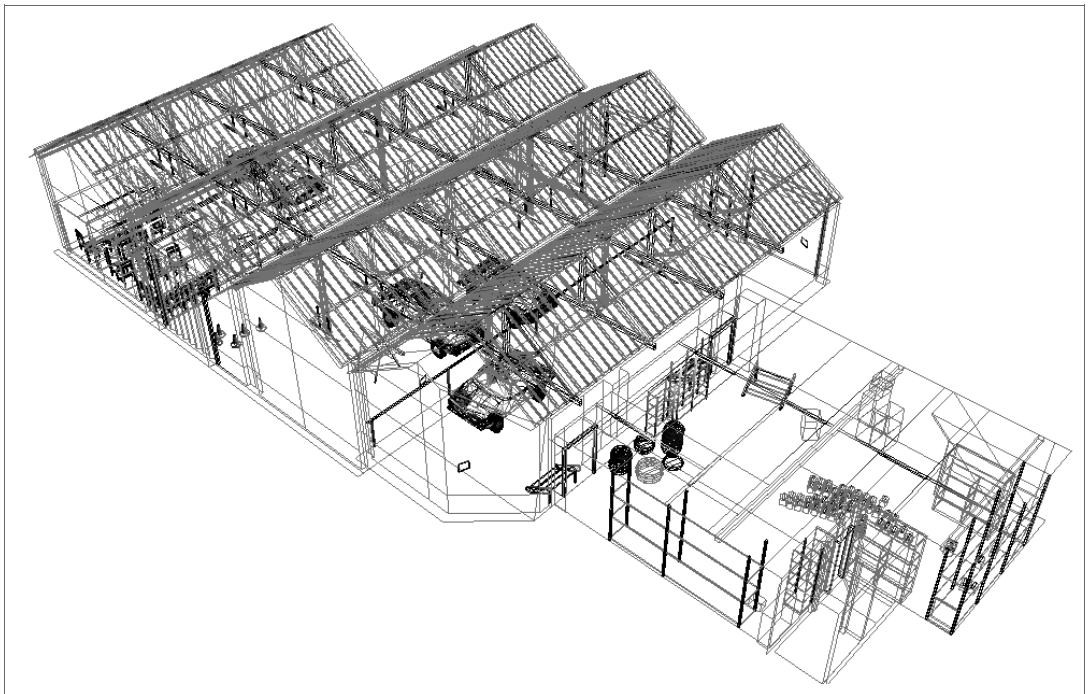


*Figure 2. Source of polygon soup: Unstructured warehouse model*

The unstructured warehouse model above is an optimized scene used in a crime scene investigation tool, but the source files were lost. This example of soup was provided at the start of the project. One desired result of the project was to produce a list of named polygon groups from this soup. Named polygon groups are data structures, perhaps lists of polygons, that are grouped by a label.

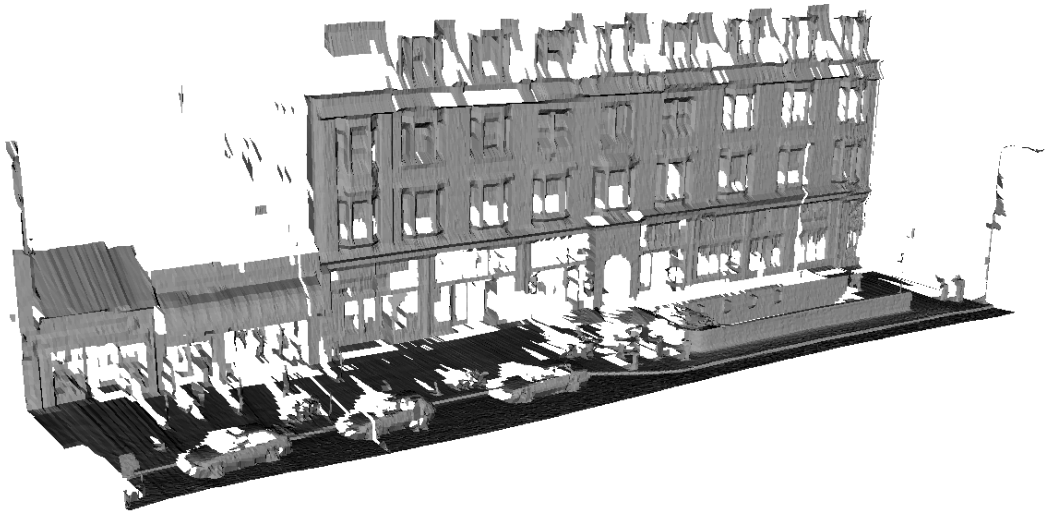


*Figure 3. Source of polygon soup: Laser scanned street, based on data from the UC Berkeley City Modelling Project* [46]

As a simpler example of soup, the figure below shows the vertex data for a few simple polygons as visible triangles. A rendered view of this data clearly shows the outline of the letter 'K'. From the source data, without a visual renderer, or the human cognitive ability for pattern matching, the source data is almost meaningless.

```
letter_k.txt
p 4
0.0 0.0 0.0
0.0 4.0 0.0
1.0 4.0 0.0
1.0 0.0 0.0
p 3
1.0 1.0 0.0
1.5 1.8 0.0
1.0 2.5 0.0
p 4
1.0 1.0 0.0
1.5 0.0 0.0
2.5 0.0 0.0
1.5 1.8 0.0
p 4
1.5 1.8 0.0
3.0 4.0 0.0
2.0 4.0 0.0
1.0 2.5 0.0
```

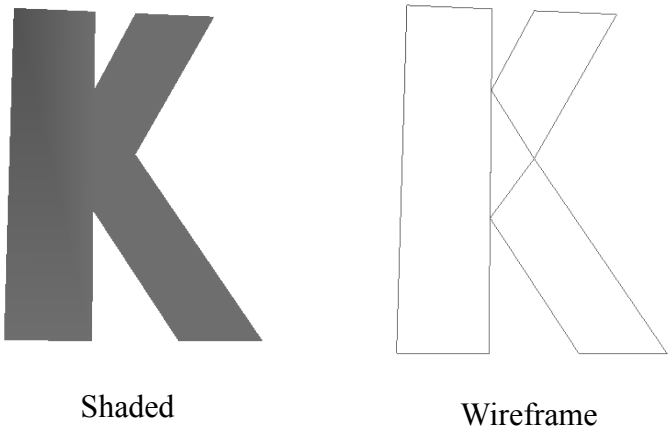



*Figure 4. Polygon data and rendered view for the letter "K"*

## 2.2 Existing tools and research

Prior to introducing the design of the structure from soup editor, research was performed on the commercial, amateur, and educational markets of existing tools as well as other reports, papers, and software relating to polygon soup.

### 2.2.1 Commonalities of existing 3D editors

Given the task of breaking unstructured polygon soup into more meaningful data you may be attracted to one of the many 3D editors available on the market. A 3D editor is a software tool that provides an interactive environment with tools for viewing and

modifying a scene containing 3D data, of which commercial, open source, and simple tools all share prominent similarities. The figure below demonstrates the common features that can be found across a handful of editors, including *Autodesk 3DSMax*[2], *Lightwave*[3], *Autodesk Maya*[4], *Blender*[5], *Google SketchUp*[6], *Moray*[7], and *Metasequoia*[8][*]. Notably modern computer games that require an overview of a large 3D area also share these features, such as *Emperor: Battle for Dune*[9], and *Warzone 2100*[10]. In summary, there are many common user interface trends between all these software applications as shown in *Figure 5. Similar features of modern 3d editors*.

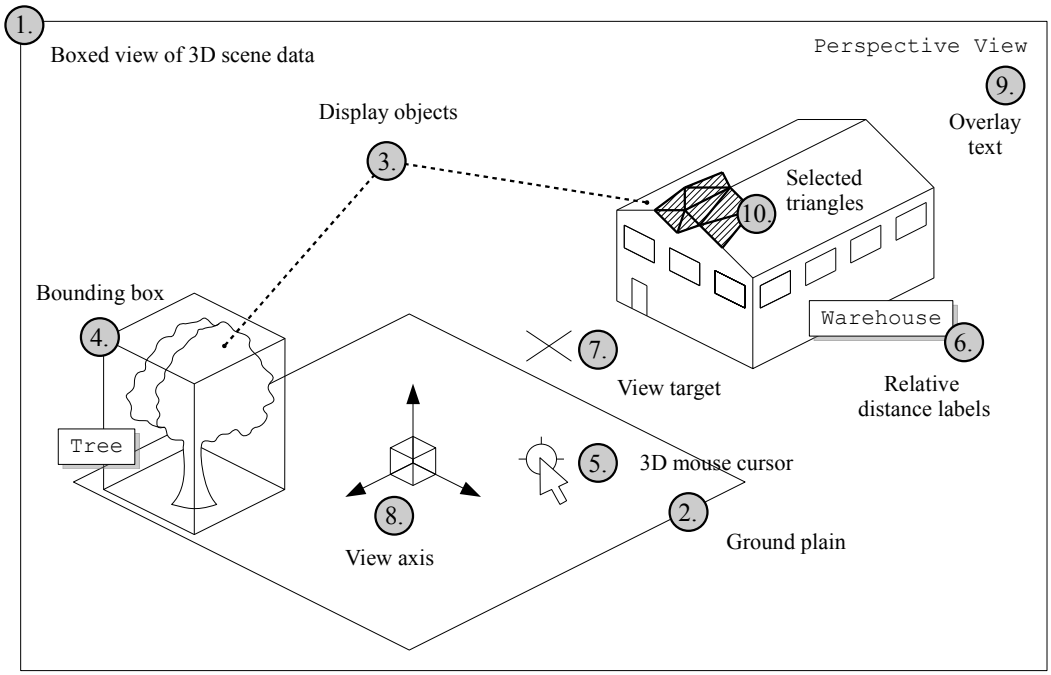



*Figure 5. Similar features of modern 3D editors*

1. A boxed view of the 3D data; the idea of a scene, and the concept of a user controlled camera by way of keyboard or mouse interaction
2. A ground plain, or reference grid; represents the default intersection for the on-screen mouse and the 3D scene
3. Display objects; things to look at, groups of polygons which have a position and size within the scene
4. Bounding boxes; a user interface cue that surrounds and identifies the current selection
5. 3D cursor position based off 2D screen mouse; a cue to help users understand how their mouse movements are being interpreted before making an action
6. Overlay labels; meta information, drawn on top of the scene without scale but in proximity to the associated item
7. View target; centre of the current view, may be above or below the ground plane
8. View axis; for 3D editors only, a guide, usually colour coded, that helps orient the user to the scene
9. Overlay / interface elements; overlay labels relative to the edges of the view independent of objects in the scene
10. Selected faces; for 3D editors only, a visual display of vertices or polygons that mark the current selection made by the user

[*] Example screenshots of other software tools can be found in *Appendix A: Screenshots*.

These commonalities demonstrate that 3D editors, and games alike, provide an easy way for users to navigate the presented 3D scene and choose appropriate actions to perform. Similarly the structure from soup editor needs help users navigate through different forms of polygon soup to make appropriate decisions about adding structure.

**Additional Reading**

The most useful articles researched were on the construction and features of modern modelling tools, as summarised in the *commonalities of 3D editors*, the most useful of which was an article on *UnrealEd*[11] for creation of 3D worlds that examined the features of the tool.

### 2.2.2 References to Polygon Soup

The term polygon soup is easy to understand once explained but is not widely used. Most dealings with polygon soup are short lived, since most unstructured data is either ignored or quickly processed into another form. While researching 'polygon soup' for this project, the following esoteric papers were found:

**Swept volume approximation of polygon soups**[12]

- Swept volume (SV) is the totality of points touched by a geometric entity while in motion. Useful in geometric modelling, collision detection, ergonomic studies.

**Geometry image matching for similarity estimation of 3D shapes**[13]

- Geometry images can be formed from irregular models into a regular structure capturing geometry as a simple set of geometric attributes. The paper suggests how geometry images could be used for 'geometry matching' now that their algorithm solves the issue of 'rotation invariance' on arbitrary model data.

**Shape similarity comparison of 3D models using alpha shapes**[14]

- A set of 3D Alpha Shapes is a family of 3D shapes computed from a 3D point set given various values from the data. Paper looks at a 'multi resolution' approach for indexing a 3D model so that the indexes can be searched against database of similar models. Different approach then Geometry images, demonstrated good search results.

**Mesh Parameterization: theory and practice**[15]

- Parameterization is introduced to map textures onto surfaces. Usable for morphing, mesh editing, mesh-completion, re-meshing, compression, surface filtering, shape analysis... and so on.

These articles apply to high level uses of polygon soup, the application of which is further explored in *Chapter 5.4 Exporting knowledge*, and *Chapter 7.2 Extensions to the editor*.

## 2.3 Chapter summary

The background and research performed provides an understanding the problem of polygon soup, and the approaches, tools, and papers related to this field. By identifying common features of 3D editors, existing knowledge about handling 3D data can be used to select appropriate approaches for adding structure to soup.

In the next chapter, the decisions about the design of the software, the goals, requirements, and tools used are discussed, based on this initial research.

# Chapter 3 Design

To design and build the structure from soup editor, a clear vision of purpose was required to set the functional boundaries of the software. The following chapter explains the processes and decisions taken to design the software prior to implementation.

## 3.1 Problem analysis

The design of the structure from soup editor stems from a single *super goal* agreed on at the start of the project:

Super goal: "Give the composite parts of an unstructured polygon soup semantic relevance"

This goal summaries the aim of the project and sets the first measure of what is in scope and what is out of scope. With the super goal in mind, a reasoning loop was applied to generate a goal hierarchy. The reasoning loop is a requirements engineering technique for breaking a goal down into sub goals by using hypotheses, arguments, and actions in a recurring loop. At each iteration of the loop, new goals are generated that support the parent goal. The reasoning loop is stopped when goals are detailed enough to write requirements from.

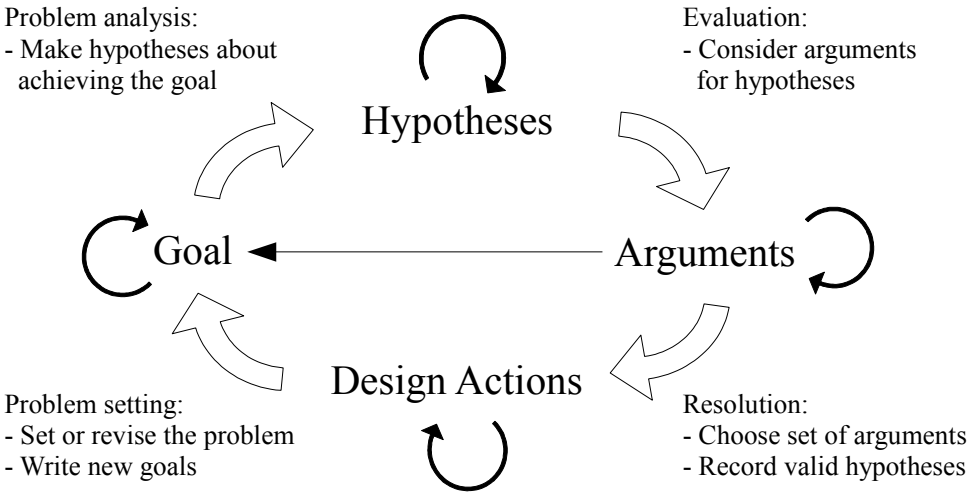



*Figure 6. Requirements engineering - reasoning loop*

The use of a hypothesis, and a supporting argument, are used to evaluate the positive merits of a goal so that assertions can be used to generate new goals. Arguments against a hypothesis provide reason to refocus on the goal and generate new hypotheses. For this project, six new goals were generated at a depth of three levels. At each stage the hypotheses, arguments, and actions were recorded. The results were stored in a document titled "Requirements with Reasoning".

### 3.1.1 Goal hierarchy

The result of applying the reasoning loop is a goal hierarchy. This is illustrated in the figure below. Descriptions of the goals generated from this loop can be found in *Section 3.2 Goals of the project*. The hierarchy records the parent and child relationships between super and sub-goals; this can be used to justify each goal by identifying its parent.

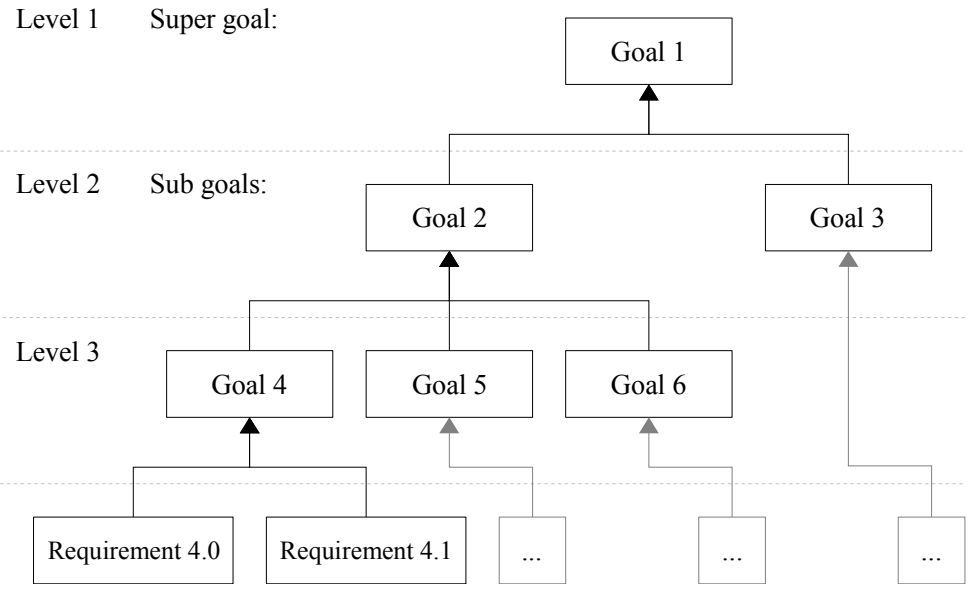



*Figure 7. Goal hierarchy structure*

### 3.1.2 Stage summary

By applying the reasoning loop to the super goal, a goal hierarchy, and a list of requirements was produced. This hierarchy could then be used to track each requirement back to a specific goal, and in turn to the super goal. This provided focus for the project and helped avoid 'feature creep' by requiring justification for each feature added.

## 3.2 Goals of the project

Each of the project goals, generated in the reasoning loop, are explained in terms of the requirements attached to them.

**Goal 1**

Give the composite parts of an unstructured polygon soup semantic relevance.

- As a super goal, no requirements were directly attached to this goal

**Goal 2**

Create a computer application that allows a user to add semantic information and structure to an unstructured polygon soup.

- Justifying the need for a computer application provides requirements for general program features such as loading, saving, and an undo-framework.

**Goal 3**

Create an algorithm to identify structures within a large set of polygon data.

- This goal justifies development time spent on algorithms that save the user time and effort by identifying structures on behalf of the user. This was eventually realised using the *volume select* and *flood select* tools.

**Goal 4**

Create an interactive visual display for the unstructured polygon data.

- An interactive visual display provides grounds for camera controls that allow the user to easily navigate around a loaded data set.

**Goal 5**

Provide tools that allow the user to add structure to the polygon data.

- Tools that allow the user to add structure tackles the core goal of adding structure to soup. The requirements included; tools for creating selection groups, ways of highlighting a selection, and adding structure to the soup (c.f. *Chapter 3.4 Scene Graphs*).

**Goal 6**

Provide an interactive visual display of the structured polygon data with tools to modify that structure

- Related to the structure produced as a result of *Goal 5* and how that structure should be viewed and managed.

Each goal was used as a foundation and reference point for development, providing justification for each action that followed in the software engineering process.

### 3.2.1 Example requirements

Each goal was used to justify detailed requirements that specified what the software should do. For example, **Goal 4** – Create an interactive visual display for unstructured polygon data, is used to justify requirements 4.0, 4.1 and 4.2 as below.

**Requirement 4.0**

Implement three orthographic views of the same scene covering the Left/Right (x axis), Front/Back (y axis) and Top/Bottom (z axis) views. Each orthographic view should respond to mouse actions and keyboard commands to support several interactions, including:

- Panning the view point and the view target vertically and horizontally
- Moving the view point towards and away from the view target

**Requirement 4.1**

Implement a fourth perspective view with a customisable camera. The camera should respond to mouse actions and keyboard commands to support several interactions, including:

- Orbiting the view point around the view target
- Moving the view point towards and away from the view target
- Panning the view target on the x, y plane
- Panning the view target on the z plane

**Requirement 4.2**

Each view should support switching between wire-frame and mesh-shaded rendering modes.

## 3.3 Development toolbox

The tools chosen developing the structure from soup editor were:

- *OpenGL*[16] – Low-level graphic manipulation and rendering language
- *Trolltech QT framework*[17] – C++ Library for user interfaces
- *OpenSceneGraph*[18] – C++ Library for OpenGL based scene graphs
- *HaiQ*[19] – Hand coders IDE for QT

**OpenGL**

OpenGL is a well established graphic manipulation language that has many cross-platform implementations. Its low-level nature means programmers have precise control over the graphics process within the target machine allowing highly optimized programs to be written for rendering 3D data. A competing framework might be Microsoft DirectX, but more material and expertise was available for learning OpenGL. There are many books and

tutorials available for OpenGL, from basic introductions to indepth manuals[20][21]. For this project, and in support for Goal 4, OpenGL provides a common language for dealing with 3D data.

**Trolltech QT**

In early research, Trolltech's QT proved itself to be a capable and well documented toolkit for creating user interfaces with support for OpenGL. Early tests showed that a graphical user interface (GUI) could be developed in QT that supported mouse and keyboard interactions with a 3D scene rendered in OpenGL. QT also comes packaged with a strong visual editor, QT Designer, for designing user interfaces. QT Designer can be included in a software development process where design decisions are separated from the functional code implementation. This is achieved through the QT *QAction* object and their *signals and slots* system. This process is described more fully in *Chapter 3.6 User interface abstraction*.

**OpenSceneGraph**

OpenSceneGraph is a wrapper library for OpenGL, providing a useful abstraction of common elements built on top of OpenGL functions. It's main attraction was the array of plug-in libraries for loading a variety of different file formats. This alone can speed up development by reuse, but OpenSceneGraph also simplifies other OpenGL based tasks such as wrapping camera and matrix functions, and encapsulating the idea of a *scene graph* (as described in *Chapter 3.4 Scene Graphs)*.

**HaiQ Handcoders IDE for QT**

HaiQ, a code-level integrated development environment (IDE) for QT provided project and file management, single click-to-compile, and an in-line debugger. HaiQ wraps the functions of the *MingW*[22] compiler and QT's *qmake* framework to manage consistent production of the executable software. There are many IDEs available, such as *Microsoft Visual Studio*[23] and *Eclipse*[24], each with support for QT, but HaiQ is straightforward to set up, it provides in-line help from QT Assistant, and it recognises the different project files found in a QT project environment.

**C++ Programming language**

Last to mention, C++ was chosen as the development language because of its strength as an object orientated programming language, and the ability to compile efficient machine code for a target machine. C++ is an *ISO standard*[25] and has implementations across many different computing architectures. Processing and rendering 3D data requires many intensive computations which can be highly optimized in C++. These speed benefits are better then the alternatives, such as Java or C# .NET. C++ provides strong inheritance support and type casting making it easy to implement complex class orientated designs. C++'s strongest and weakest point is the low-level access of memory management which can lead to unexpected bugs and memory leaks within a program. Using the QT and OpenSceneGraph frameworks can help mitigate memory issues by providing dynamic pointers which are self managing memory objects, simplifying the task of memory management for the developer.

## 3.4 Scene graphs

A **scene graph** is a *data structure* that holds information about a *scene*.

A scene graph is a tree structure, with branches and leaves called nodes. Group nodes, that contain one or more child nodes, are used for structuring and organising content- these are the branches in the tree. Leaf nodes, have no children; they hold distinct units of information such as geometry. Inter-relationships within a scene graph mean that a graph does not always form a clean hierarchical tree, but there is always a root node to start from.

Scene graphs are commonly found in 3D editors, computer games, and vector based graphical editors. Commercial examples include *Autodesk 3DSMax*, *Half-life 2*[26], and *Adobe Photoshop*[27]. A scene graph provides an interactive structure that makes it easy to

perform operations on different segments of the data. The simplest operation is 'show' and 'hide', where a branch can be completely hidden by hiding the group node that forms the branch. Effects, such as colour or drawing styles can also be applied at the group level, propagating down to the child nodes. Most importantly to this project, scene graphs are used to structure nodes within a scene. The relationship between a child and a parent infers ownership and belonging.

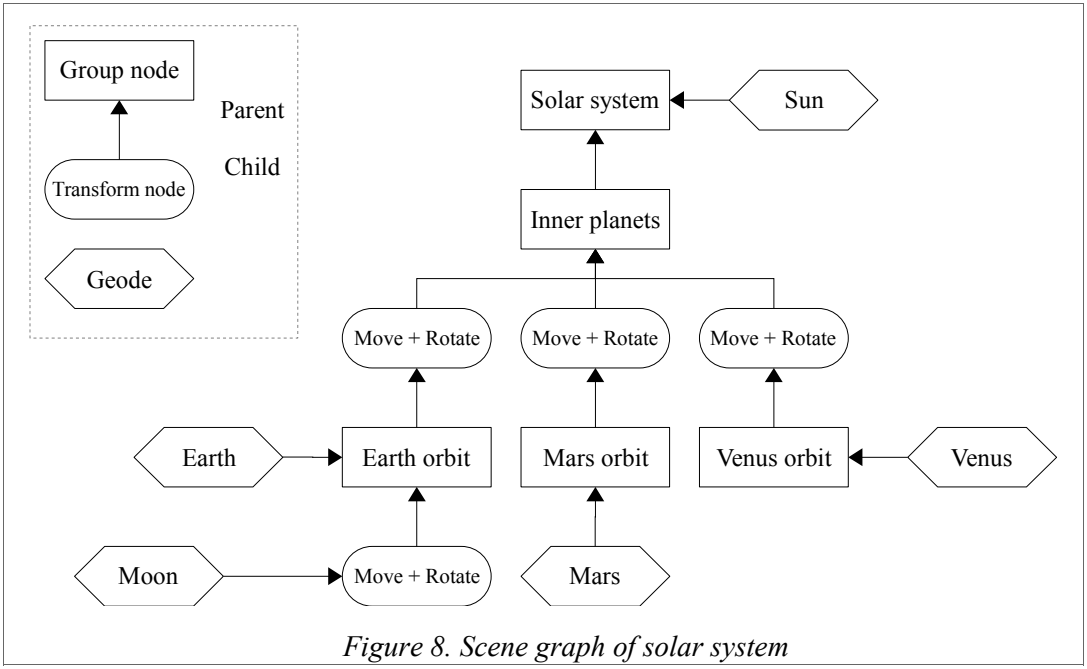



*Figure 8. Scene graph of solar system*

The figure above shows a scene graph constructed to represent parts of our solar system. The sun, the centre of the Solar System, is represented by a Sun *geometry node* (a geode). A *group node* named Inner Planets then contains *transformations* (move + rotate) and groups for three planet orbits. At the centre of each orbit is a planet geode. The Earth Orbit contains an Earth geode at its centre and a Moon geode is attached to a transform node which positions the Moon away from the Earth.

A scene graph, like the model of the solar system, can be quickly and logically traversed to position and render each of the geodes in the scene. The structure of the scene graph lends itself to logical manipulation, such as rotating the position of the Earth orbit and automatically updating the earth and moon's position relative to the earth's orbit.

### 3.4.1 OpenSceneGraph

OpenSceneGraph is an open source toolkit implemented in C++ that uses OpenGL to construct and display 3D models. OpenSceneGraph implements the scene graph structure by providing methods for loading and rendering 3D polygon data. Its libraries provide support for managing a scene graph with nodes types such as groups, geometry (geodes), position and scale transforms, cameras, and lighting.

OpenSceneGraph is aimed at providing a professional level toolkit for creating visual applications and games. Examples include *ossimPlanet*[28] a virtual earth, and *FlightGear*[29] a flight simulator. The open source nature of OpenSceneGraph means that developers from around the world have contributed to and help maintain the toolkit. Low budget developers can use OpenSceneGraph to save effort when developing 3D visual applications.

Information about OpenSceneGraph, including documentation, binaries and sources can be found at: www.openscenegraph.org

### 3.4.2 Why a scene graph is needed

Super goal: "Give the composite parts of an unstructured polygon soup semantic relevance"

The super goal provides a requirement that by using the structure from soup editor it should be possible to take polygon soup (unstructured 3D data) and give the data structure and semantic relevance. Initially the data starts with no structure, but a structure can be formed by selecting parts of the polygon data, grouping the polygons, and labelling the groups. A scene graph provides the underlying structure that allows these semantics to be stored. A library such as OpenSceneGraph can also provide the methods to display and navigate the scene graph with programmatic ease.

The output of structuring and labelling the data actually produces a scene graph. This satisfies the super goal- the data starts off unstructured, structure is added building a scene graph, and then the structured data is saved as a scene graph. OpenSceneGraph provides the classes and methods to create, edit and display scene graphs.

### 3.4.3 Key terminology from OpenSceneGraph

The OpenSceneGraph **osg** namespace, the core group of libraries in the toolkit, provide basic scene graph classes such as Nodes, State and Drawables, as well as maths, and general helper classes. Each class represents different concepts that encapsulate information and operations that let developers create their own scene graphs. The names of the more common classes, used as terms in this report, are defined here:

**Structural classes**

Object, Node, Group, Transform, Switch

- The structural classes provide structure to the scene graph, creating parent->child relationships, positioning geometry, and switching visibility.

**Display classes**

SceneView, Camera, Lighting

- The display classes provide control over the display of the scene graph, including the viewing area of the camera, the lighting of the scene, and the rendering order of objects.
- SceneView provides access to the camera view port and the top-level of the scene graph. Instances of the SceneView class can provide a multi-view display of a scene.

**Geometry classes**

Geode, Drawable, Geometry, PrimitiveSet, VertexArray, ColorArray

- The geometry classes are used for representing 3D data loaded in the scene.
- *Geodes* contain *Drawables*, of which *Geometry* is a type of drawable. Geometry objects contain lists of *PrimitiveSets* that refer to *VertexArrays* and *ColorArrays*.

### 3.4.4 How to implement the scene graph

There is a direct correlation between a structured scene graph and a semantically structured set of polygon data. Modifying the structure of polygon data can be reflected by directly modifying the internal scene graph. The only caveat is that additional scene information, such as camera position and lighting that does not form part of the data, must be added at the top level of the scene graph, superseding the root of the data. From this perspective, two roots in the graph, or at least two starting points, can be considered. The first root is the actual renderable root of the scene graph; this is used for the display of the scene from multiple views. The second root is a model geometry group, a named group node that contains the loaded polygon data; this virtual root contains the data that the user intends to view or edit.

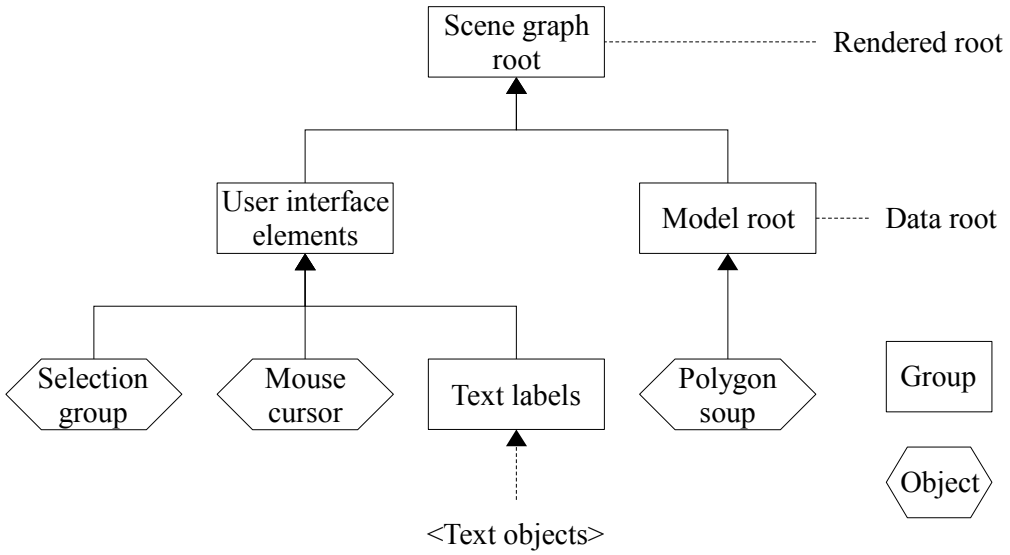



*Figure 9. Split scene graph showing rendered root and data root*

### 3.4.5 Scene graph summary

Scene graphs are often found in 3D editors, computer games, and other vector-based editing applications. Scene graphs are used interactively to quickly highlight branches of data, and make it easy to apply operations and effects to all of the children in a branch.
OpenSceneGraph is an open source set of libraries written in C++ that allows developers to construct their own scene graph based applications without 're-inventing the wheel'.

For the structure from soup editor, a scene graph is core to display and render 3D data. The scene graph is also needed to model the structure of the data that will be created by the user. OpenSceneGraph provides the necessary functionality to create and render the scene graph using OpenGL functions and its own internal libraries.

## 3.5 Prototype driven development

An agile development methodology is a process that produces a working version of the software at each stage of development. Agile methodologies[30] promote rapid cycles that produce testable results at the end of each cycle. This allows the quality of software to be validated regularly, and any problems with the process to be identified and corrected early. Prototype driven development is a specialised type of agile development that uses the latest working version of the software to decide on the next feature to implement. Whereas module-driven development, or requirements-driven development, might look to the next big contractual feature to implement, prototype driven development requires consultation with the customer to decide on the direction of the project.

The benefits of using a well defined process for software development mean that maximum value is gained from each iteration of the software. An agile incremental approach used in small recurring cycles allows a single feature to be written, compiled, and tested generating a working sub-version of the software. After several incremental additions a major version can be compiled, tested, and packaged for the next stage.

### 3.5.1 Chosen development process

Prototype driven development was decided on as the most appropriate methodology for the structure from soup editor in order to give regular feedback on the progress of the project and pick sensible features for implementation within the time constraints of the project.

The process used for the structure from soup editor was developed while researching suitable tool kits and test cases prior to requirements analysis. After each meeting with the

project supervisor, a set of features was agreed on for the next stage of development. These were then compared against the requirements document and goals to set the direction for the next week.

The development process was driven as follows:

1. A new working folder was generated by copying the previous working folder
2. A feature or bug was selected to be implemented, and coded as appropriate
3. The program was compiled and debugged
4. The feature was tested to see that it worked correctly
5. Features were added and tested (steps 3–5) until a functioning milestone was reached
6. The software was fully tested using all available data, any new non-critical bugs were noted for the next stage. Critical bugs were handled by returning to step 2.
7. The software was compiled, packaged for distribution, and backed up

A simple incrementing naming scheme (a1 – a9, b1 – b9, etc.) was used for each incremental copy of the folder structure. Zip archives provided compressed backups that could be easily moved between computers. Backups were made between a laptop and PC, and between PC and the university using FTP. Working versions of the HaiQ development environment with QT and OpenSceneGraph were installed on both the laptop and PC. Instructions on how to install and configure the development environment were kept in a notebook. These steps ensured that data was not lost at each stage of the development, and provided backup data and environments to work from.

Details of the testing plan can be found in *Chapter 6.1 Continuous testing and evaluation*.

## 3.6 User interface abstraction

The structure from soup editor was envisaged as a tool that the user would interact with in order to take unstructured polygon data and add structure. The actions involved start with loading polygon data; the user then needs to rotate and pan their view around the model before they can interact with the model data. In order to prevent actions becoming entwined with the functional code, a design pattern provided by QT was used that managed actions separately to the user interface specification. Using this design pattern a cohesive approach was taken to class design by writing functional classes to perform specific groups of actions.

QT provides the *QAction* class. Each action has a name, an optional icon, and a series of user interface features such as tooltip, “what's this”, and a short-cut key. This common grouping of information in the same place allows the same action to be used in several places across the user interface. In turn, QT also utilises a *signal and slots* framework, a tool incredibly useful for separating coupled classes. Instead of writing code directly to an action, code is written as a slot (a type of method) in an appropriate class. Objects in the QT framework then generate signals that can be connected to slots. The QAction class contains the signal *triggered()*, that can be connected to any slot method with the same method signature. Then, wherever a QAction is triggered the appropriate method in the code will be activated. This allows multiple QAction objects to be placed in the user interface, all pointing to the same code. Conversely, the signals and slots system means multiple actions, or signals, can be connected to the same slot, allowing for shared functionality to be reused between classes. Signals and slots provide an alternative to interfaces and inheritance taught in object orientated methodologies that is much easier to extend and allows better cohesion between classes.

The QT Designer application encourages users to create actions in the form of file menus. These actions are indexed in an actions panel to be reused elsewhere. This action system allows QT Designer to provide predefined components from the QT framework to build up

a modern user interface quickly with almost no coding. The designs can be previewed and tested in a live environment, but are ultimately stored as XML encoded *UI files* in the project directory. The UI files produced by QT Designer can then form part of a software process that means changes to the user interface can be made independent of the underlying functionality.

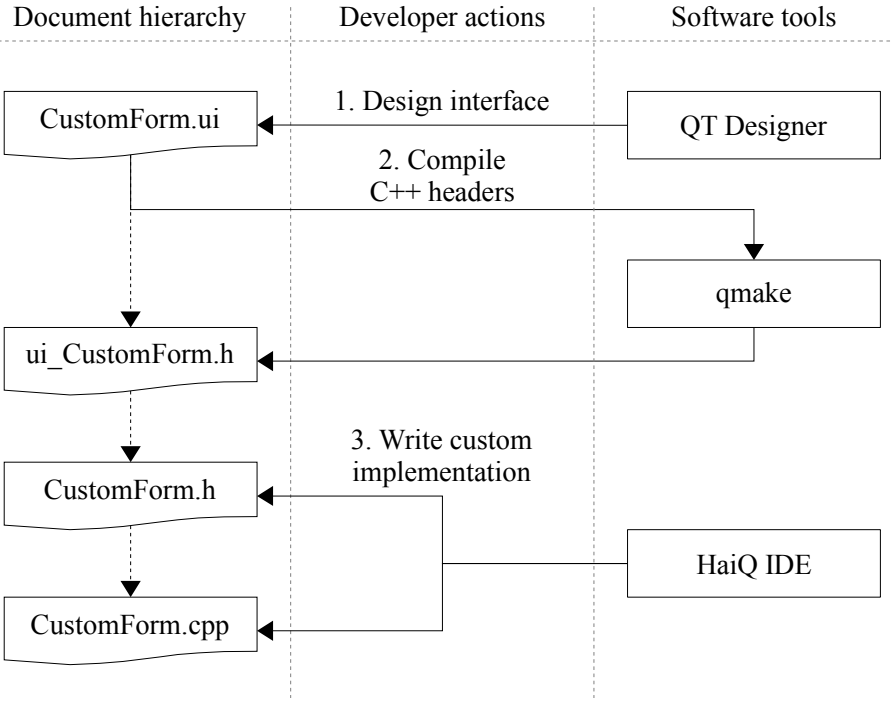



*Figure 10. QT Designer UI files as part of a multiple-inheritance system*

At compile time, the latest version of the UI file is used to generate a header file used by a named subclass. Instances of the subclass can then be created in the application, or the class can be sub-classed again to add specific functionality for the program. Subclasses of the user-interface header file gain all the attributes and fields defined in QT Designer while providing functionality to the main program. As long as named elements are not added, renamed, or removed, the functionality remains unchanged. Design decisions made at the QT Designer level such as text, positioning, and colours, can be made without breaking code in the subclasses. As part of a more structured design, code at the lower levels can specify named elements and actions at a higher level; actions included in a UI Design can be defined before the code is written, and vice versa. This helps to mitigate the need to embed user interface code within the main program.

## 3.7 Primitive data structures

A polygon is a shape described by a series of points connected in a given way. In OpenSceneGraph polygons are represented by PrimitiveSets, stored inside Geometry objects that use a shared array of Vertices (VertexArray) and Colors (ColorArray). This complexity optimizes the storage of the data by removing duplicate information. For example, for two polygons that share one or more vertices, rather then store the vertex twice, a single entry is stored in the VertexArray, and a reference to the vertex position in the array is used in the definition of the polygon.

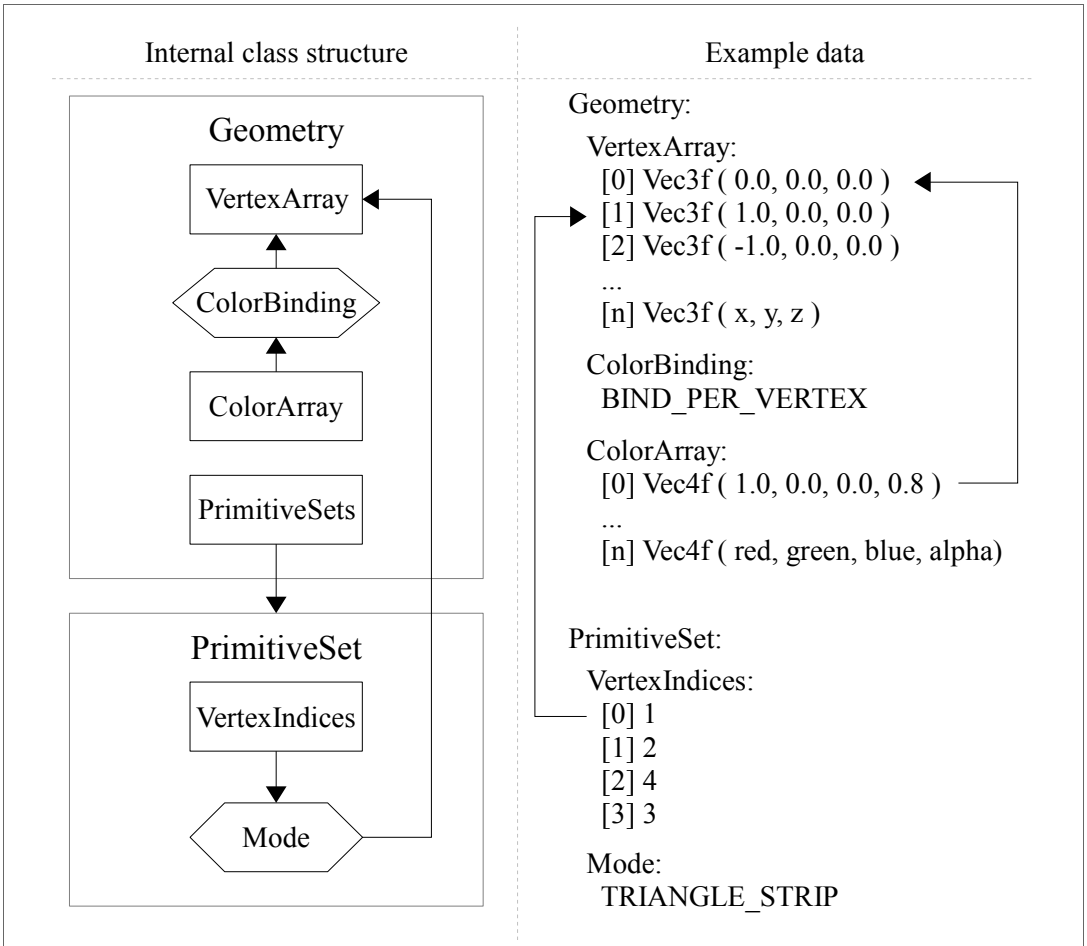



*Figure 11. Relationships of data structures inside a geometry object* [*]

This detail is important because any algorithms written must respect these data structures. Model data is loaded by default into this common geometry structure, and operations on model data must start and end with these structures in order to be rendered as part of the scene graph. Intermediary data structures such as maps and lists can be applied to reorganise this information as used in the following algorithms written for the structure from soup editor.

A Geometry object can store any number of PrimitiveSet objects, but it can only have one array of vertices, and one array of colours. There are several classes of PrimitiveSet defined by OpenSceneGraph, but to reduce complexity only one type, DrawElementsUInt, was considered by the algorithms. DrawElementsUInt contains a list of vertex indices stored as unsigned integers (UInt) that reference the vertex array in the parent geometry. This means that PrimitiveSets on their own are not complete sets of data, and cannot be moved around easily without re-indexing their values, or the values of the vertex array that they reference. Additionally, depending on the colour binding property of the geometry, the ColorArray is bound to either the vertices or the polygons. Two colour binding settings, BIND_PER_PRIMITIVE and BIND_PER_VERTEX, help illustrate the difference in data structures implied, or at least possibly contained, in the geometry object. For simplicity, model data was flattened to BIND_PER_VERTEX during file loading so that data was presented in a consistent manner to the algorithms for data manipulation.

[*] Note: Figure 11. is a grossly simplified example of the osg::Geometry and osg::PrimitiveSet classes; osg::Geometry contains separate ColorIndexArray and ColorArray lists for storing colour data, and PrimitiveSet is a virtual super-class of data structures such as DrawElementsUInt.

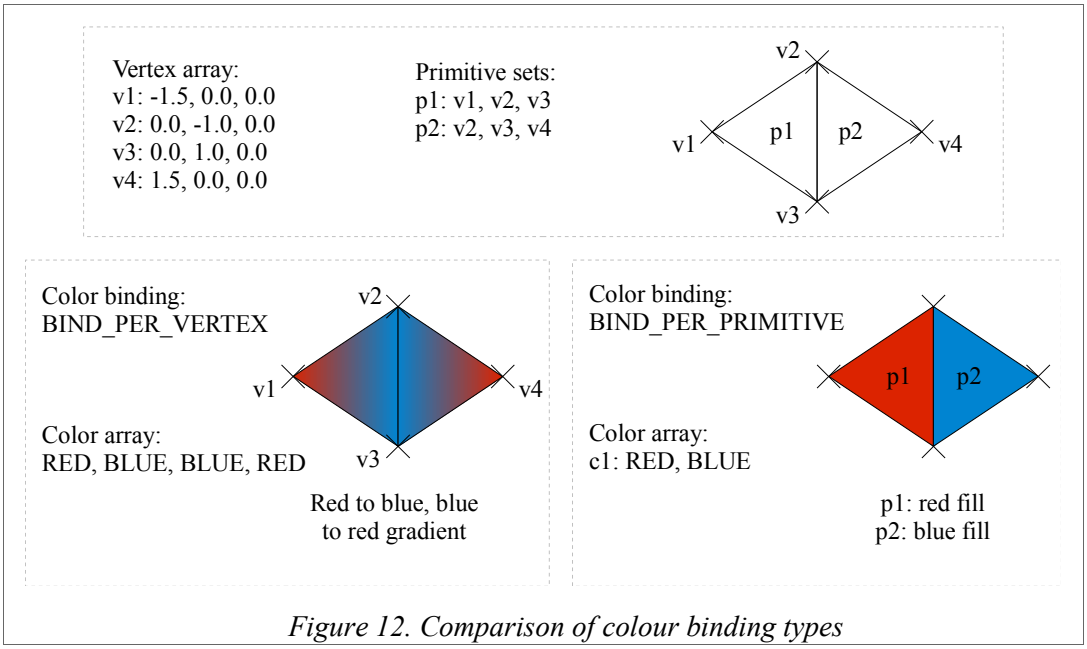



*Figure 12. Comparison of colour binding types*

## 3.8 Program design

To bring all of the researched elements together, the design of the system was evolved to the point described in this section. Because of the prototype driven development of this system, many parts of the design were not finalised until late on. Instead, the design of classes was made within an outlook for extension, generalising functions where possible and keeping signal and slot interfaces open for use by other parts of the program structure.

The *main window controller* is the main control class that brings all the elements of the structure from soup editor together. This is illustrated in the following class diagram by indicating some of the signals that are passed between the major program components.

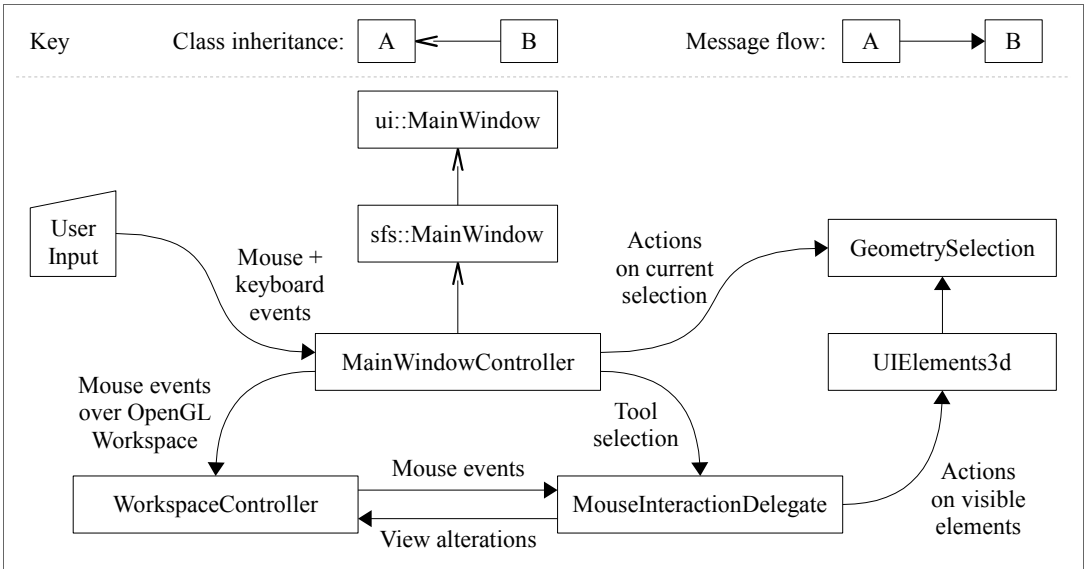



*Figure 13. Class hierarchy and messages of the major program components*

At the top of the application structure, the MainWindow UI class is sub-classed by MainWindowController. MainWindowController creates the major instances of the program, including the *WorkspaceController*, *MouseInteractionDelegate*, *UIElements3D*, and *GeometrySelection*. These classes are described as follows:

**WorkspaceController**

The WorkspaceController class defines how BorderedSceneViews work as OpenGL viewports into the OpenSceneGraph scene. It has definitions for four basic views (top, side, front, custom) and six display modes for organising those views in different positions on the screen (four view, full screen top, full screen side, full screen front, full screen custom, and top-side-front view). The workspace controller, by inheritance from QOsgWidget, connects both OpenSceneGraph and QT mouse and keyboard interaction events, making it a common interface to communicate signals between the 3D views and the rest of the application.

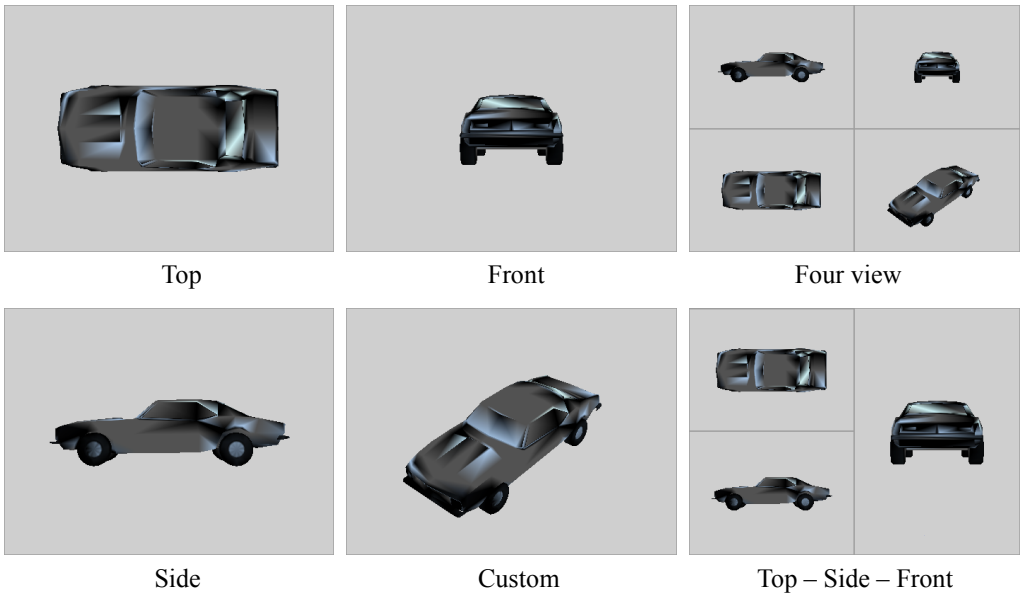



*Figure 14. A comparison of the six available view modes*

**MouseInteractionDelegate**

The MouseInteractionDelegate class is used to track the current tool selected by the user and selects an appropriate method to call for each mouse interaction. This class contains several switch statements; these structures contain the program logic for delegating actions from the WorkspaceController to functioning parts of the code, such as making a polygon selection or changing the the camera view.

**UIElements3D**

The UIElements3D class is used to create and maintain a scene graph of user interface elements. Most elements are hidden by default using *osg::Switch* node objects, becoming visible as signals are received from the MouseInteractionDelegate.

**GeometrySelection**

The GeometrySelection class defines methods for building up and displaying the current selection. GeometrySelection is also connected up to the MouseInteractionDelegate to receive signals containing new primitives that are to be added to, or removed from, the current selection.

**MainWindowController**

The MainWindowController defines how the different elements of the program tie together. By inheritance it has access to all the actions defined in the UI file from the MainWindow class. In turn, it provides action slots (methods) for every action in the user interface to connect to. The MainWindowController handles loading and saving of files, the manipulation of the scene graph, and serves to update the user interface elements when appropriate. This class provides a central place to find every named program action executable through the code.

## 3.9 Chapter summary

The design phase of this project started with the planning and evaluation of the goals. By producing a goal hierarchy, the boundaries of the project were set and agreed on, allowing suitable tools and technologies to be selected, along with a suitable development process, and finally to produce a set of requirements. The knowledge and research of scene graphs, prototype driven development, and user interface tools, performed at the earliest stage of the project, helped plan out the features of the software and decide what might be possible in the 6 month time scale.

Additionally, the design decisions were taken on the internal program structure. This was motivated heavily by the limitations of the OpenSceneGraph toolkit and inspired by the design patterns introduced by the Trolltech QT Framework.

In the next chapter, the implementation of the core features of the editor are described, making reference to the goals, requirements, and design decisions identified here.

# Chapter 4 Implementation

The implementation of the structure from soup editor occurred in an agile manner, meaning that regular working releases of the software were produced with incremental changes to the set of features implemented. This chapter highlights the final features of the software by describing the technical details settled on after several iterations of design and implementation.

## 4.1 Algorithms for data manipulation

The following algorithms were used in the structure from soup editor for searching and manipulating OpenSceneGraph geometry structures. Their form was that of static methods in a class named *sfs::PrimitiveSearch*. *(A full class hierarchy can be found in Appendix B: Program structure of final project.)*

### 4.1.1 FindPrimitive

```
osg::PrimitiveSet* FindPrimitive(
     osg::Geometry* geom, IndexList vertexIndex
);
```

The find primitive method identifies a unique primitive (PrimitiveSet) in the supplied geometry object based on a list of indexes (IndexList). The method compares each primitive set in the geometry data against all of the vertex indices to find an exact match.

Using *Big O notation*[31], the algorithm is $O(N^3)$ complex based on the number of primitive sets in the geometry, the number of vertices in each primitive ($V_p$), and the number of vertices ($V_{il}$) in the index list. In most cases however $V_p$ and $V_{il}$ are of a predictable size, as the number of vertices in a primitive are usually 3, and don't usually exceed 10. Arguably this reduces the complexity to O(3N) because the deciding variable is the size of the geometry primitive set list, whereas the other values of the search have a constant size.

The method is used by the *Select Polygon* tool *(c.f. Chapter 4.3 Primitive interactions)* to identify a single primitive based on a list off vertices. This is required to interpret the data generated when creating a linear intersection with the model. Once a primitive is found there is little need to call this method again unless the reference to the original primitive is lost.

### 4.1.2 FindPrimitives

```
PrimitiveSetList* FindPrimitives(
     osg::Geometry* geom, IndexList vertexIndex
);
```

The find primitives method, plural, performs a more general match on the supplied geometry by matching primitives that have *at least one* vertex index in the supplied index list. The method returns a list of primitives that satisfy the search conditions.This method is very inefficient for large data sets because at worse it runs at $O(N^2)$ computations, meaning that as the number of primitives increase linearly the computation time increases exponentially. Unlike *FindPrimitive,* which performs an exact match on a specific primitive, this method matches on partial finds to reduce the maximum complexity from $O(N^3)$ to $O(N^2)$; in early revisions two or more vertices were required to match edges rather then corners, but this heavily impacted the performance for large sets of data.

Another way to reduce the computational complexity of this search could be to pre-index the available data. A binary space partition of the geometry, generated based on regions of

the model, would greatly improve algorithmic load by selecting a subset of the geometry prior to running the algorithm.

### 4.1.3 FindVertices

```
IndexList FindVertices(
     osg::Geometry* geom, osg::Vec3f point, float _distance
);
```

The find vertices method returns a list of vertex indices from the geometry that lie within a radial distance of the specified point; this is illustrated in the figure below.

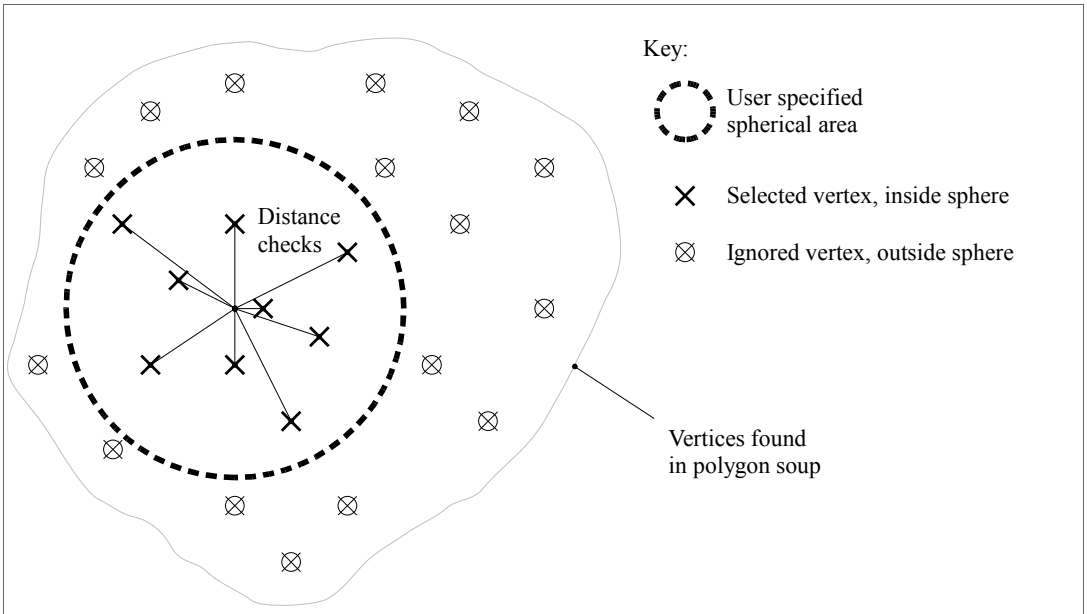



*Figure 15. Illustration of finding vertices inside a sphere*

The method has been optimized to first check the bounding box area of a vertex prior to checking the radial distance. This means simple comparison checks (greater than, less than) can be used before performing a costly floating-point distance calculation. This works well on small selections of points within a large data set. The method could be further improved to check the internal bounding box of the sphere to reduce the number of distance calculations, but the performance of this algorithm is suitable for the tool it supports.

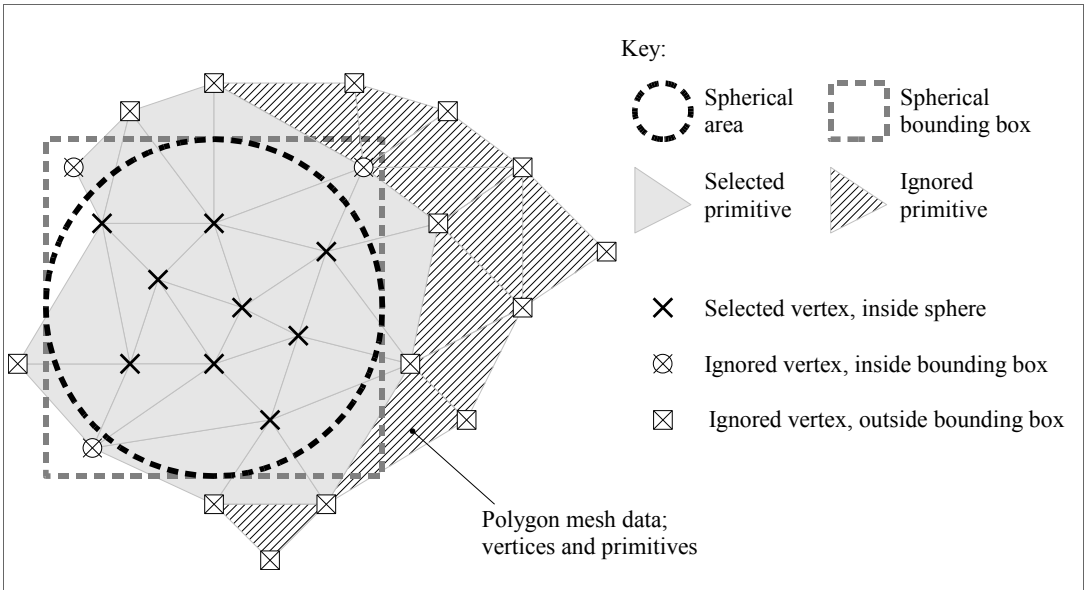



*Figure 16. Boundary box and radial distance search of vertex data*

The method is used in the *Volume Select* tool in conjunction with the *FindPrimitives* method to create a selection based on user input.

### 4.1.4 ExpandSelection

```
PrimitiveSetList* ExpandSelection(
     osg::Geometry* geom, PrimitiveSetList* _currentSelection,
     PrimitiveSetList* previousSelection
);
```

ExpandSelection is an algorithm that returns a new selection of primitives from the geometry based on the supplied arguments *current selection* and *previous selection*. If no new primitives are found then the returned selection will be the same as the current selection.

This method performs initial optimizations based on the supplied selections to minimise the search space of the algorithm. It does this by creating two lists of vertices for both the current and previous selections. The difference of these two lists produces the boundary vertices; these are vertices that lie on the edge of the current selection *(See Figure 17. below)*. A search is then performed by passing these edge vertices through the *FindPrimitives* method. The new primitives are then appended to the set of the current selection primitives and returned. The complexity of this method is kept at $O(N^2)$ by reducing the search space of the current selection, this becomes more apparent when the method is used in conjunction with the *Flood Select* tool in a recurring iteration.

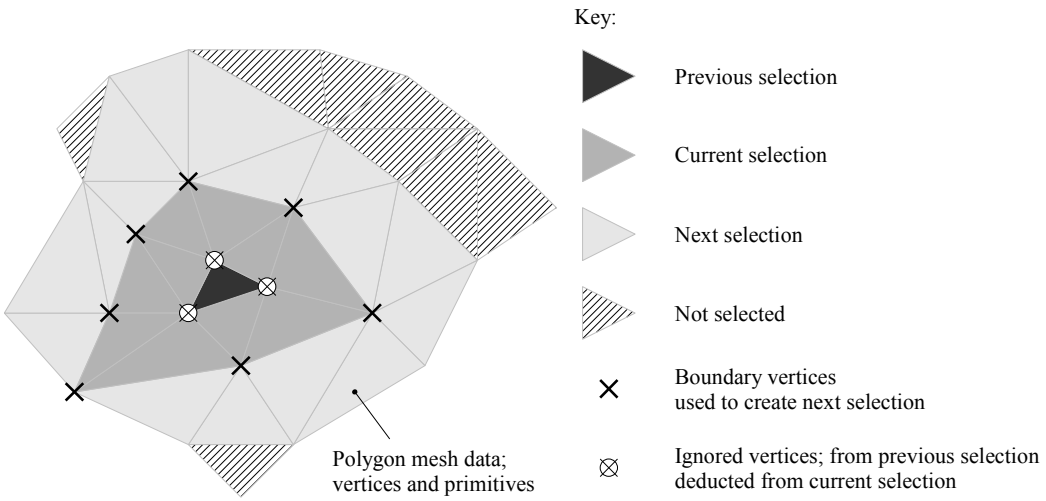



*Figure 17. Difference of current and previous selections to locate boundary vertices*

As mentioned, the method is used by the *Flood Select* tool as explained in *Chapter 4.3 Primitive interactions*.

## 4.2 View interactions

Visualising data is key to allowing users to make decisions on their next action. By viewing the scene they form an understanding of the data. This understanding is reinforced by being able to customise the view of the data, similar to looking around a room, or rotating an object in our hands to get a better understanding of the object's shape. As such, three common view interactions were identified to help the user visualise the data; these enable the user to customise their view. The interactions are *Camera Orbit*, *Camera Dolly (Zoom)*, and *Camera Pan*. This section describes in detail the implementation of these methods as

they are a key part of the strength of the structure from soup editor in providing a usable interface.

### 4.2.1 Camera control class

To de-couple the implementation of the view actions from OpenSceneGraph's *osg::SceneView* class, the custom camera control class *sfs::Camera* was written to store the abstract position of the camera in terms of *camera rotation*, *camera pan*, and *view scale*. These values could then be used to calculate the view matrices used by the OpenGL method *gluLookAt*[32] wrapped by *osg::SceneView*.

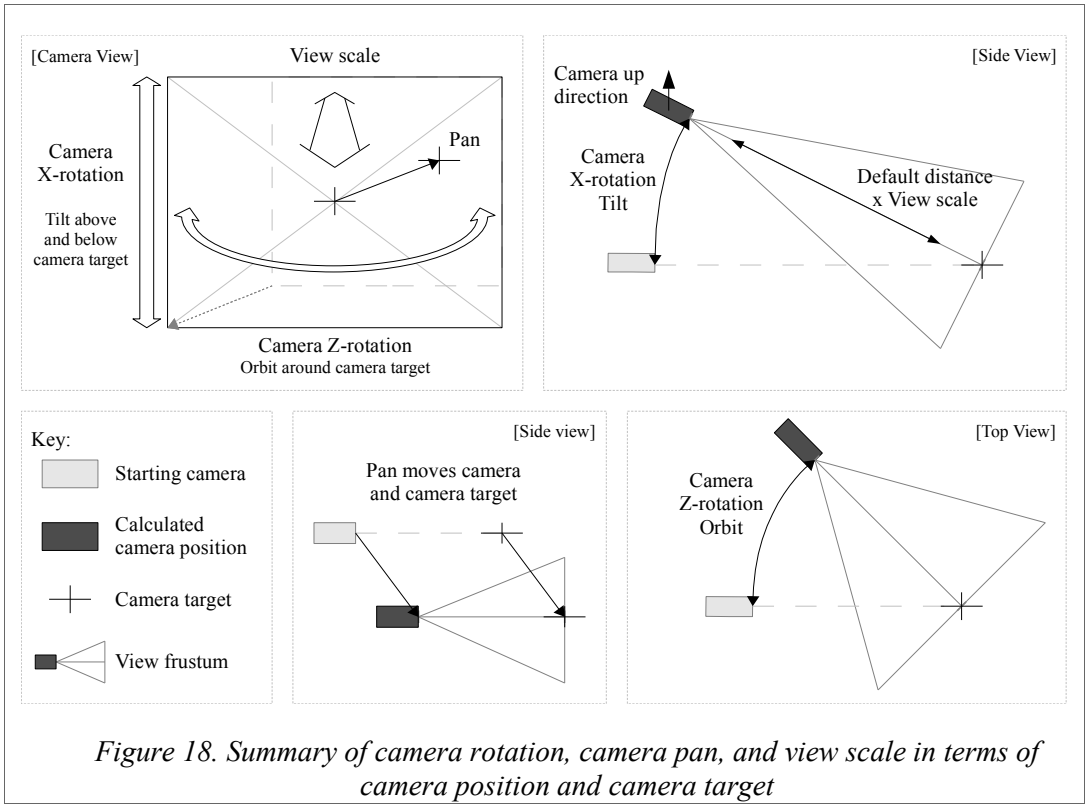


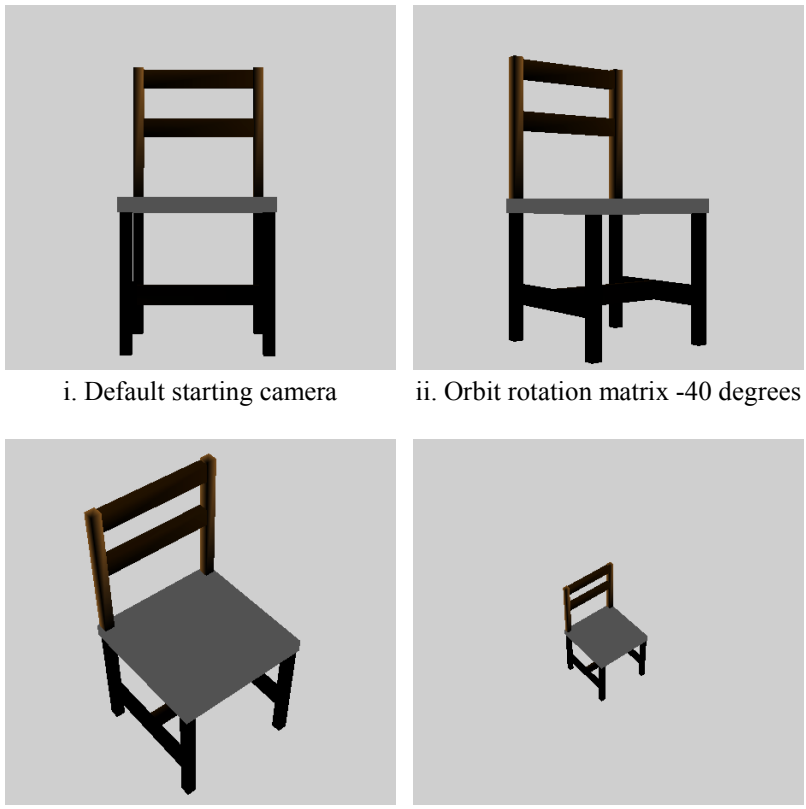

*Figure 18. Summary of camera rotation, camera pan, and view scale in terms of camera position and camera target*

The camera control class calculates a view matrix compatible with OpenGL methods using the following process:

1. Create temporary matrix variables for orbitRotation and tiltRotation
2. Make an orbit rotation matrix for the Z axis using the rotationOffset z property
3. Make a tilt rotation matrix for the X axis using the rotationOffset x property
4. Calculate temporary viewMatrix = (orbitRotation * tiltRotation) * startingMatrix
5. Extract the vectors cameraPosition, cameraTarget, and cameraUp from the values in viewMatrix using the *osg::Matrixd::getLookAt(eye, target, up)* function
6. Scale the cameraPosition by multiplying the viewScale
7. Pan the cameraPosition and cameraTarget in parallel by adding panOffset vector
8. Recalculate the viewMatrix with final values using a method equivalent to gluLookAt; *osg::Matrixd::lookAt(cameraPosition, cameraTarget, cameraUp)*

Step 4 is the most important; it multiplies the rotation matrices against the starting matrix for the camera. The order of these multiplications is very important because in matrix theory A * B is not equivalent to B * A. The order of the multiplications ensures the orbit values from the perspective of the model are applied before applying the tilt rotation from the perspective of the camera view. The final multiplication maps the camera relative to the

default distance of the camera, this sets the camera an arbitrary distance of 10 units away from the view target.



i. Default starting camera ii. Orbit rotation matrix -40 degrees

iii. Tilt rotation matrix -35 degrees iv. View scale applied 0.5 zoom

*Figure 19. Comparison of views after matrix rotations*

### 4.2.2 Camera Orbit

The orbit tool provides rotation around the model's Z-axis and tilt along the camera X-axis. When this tool is selected, clicking and dragging on the screen X-axis modifies the orbit, and dragging on the Y-axis modifies the tilt. This ability to orbit around an object is very convenient and was identified by research (*c.f. Chapter 2.2 Existing research and tools*) as the most useful way to navigate around a 3D object.

The strength of this tool is enhanced by the *Centre view on selection* action which positions the *view target* central to the current selection and calculates an appropriate v*iew scale*. With a simple system in place for positioning the camera, the user can easily rotate their view around an object before selecting another tool such as *Select polygon* or *Flood select*.

CameraOrbit is implemented simply by mapping the mouse interactions on the screen X-axis and the screen Y-axis to calculate camera offset values of view rotation in the Z-axis and X-axis. The matrix transformations in the camera update method are stable enough to allow positive and negative view rotations above and below the ground plane. In some camera view systems the view can become unstable as the tilt angle approaches ± 90 degrees; this is not the case with this tool because the tilt calculation is relative the camera view and is not dependent on the orbit rotation *(See Figure 20. Using the camera orbit tool to change the view)*.

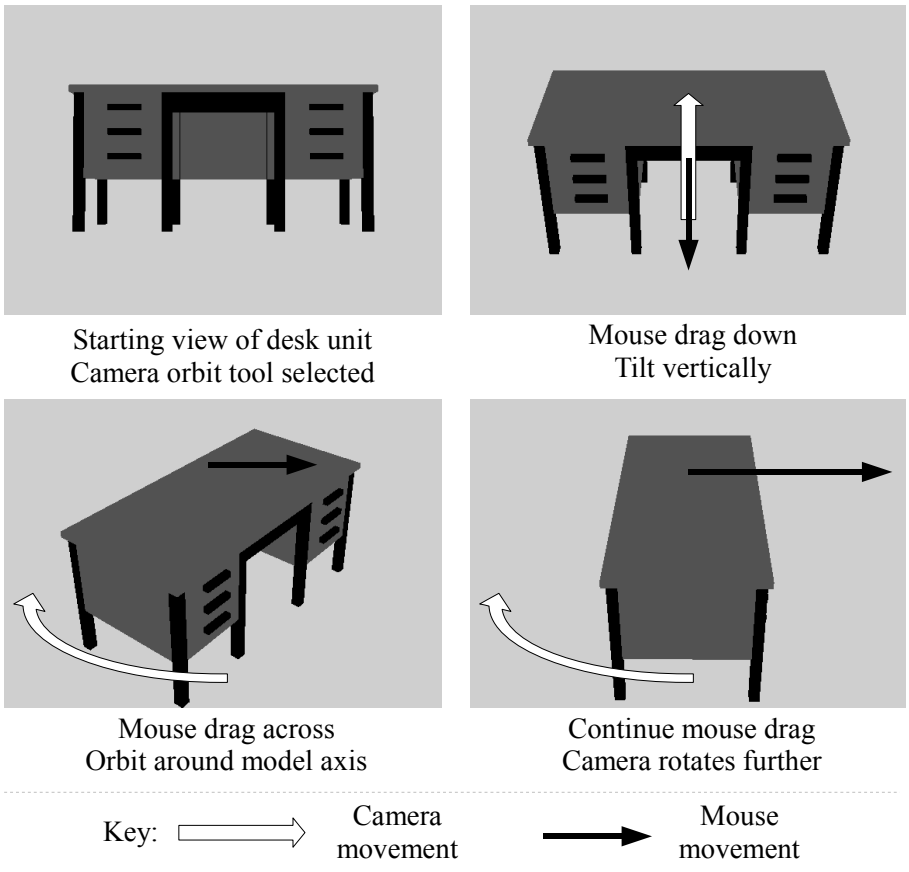



*Figure 20. Using the camera orbit tool to change the view*

### 4.2.3 Camera Dolly (Zoom)

The dolly tool is accurately named by its function, but is described in brackets as (Zoom) for those unfamiliar with the term. In a zoom function, the viewing angle of the camera is changed, creating a view of the scene that is closer or further away, in the same way as a telescopic lens is used to enlarge the image of a distant object, or a fish-eye lens is used to get a wide view of near by objects. In a dolly function, instead of changing the view angle, the camera position is physically moved towards and away from the target. This provides a much more consistent view of the scene and prevents distortion caused by a very wide (fish-eye) or very tight (telescopic) viewing angle.

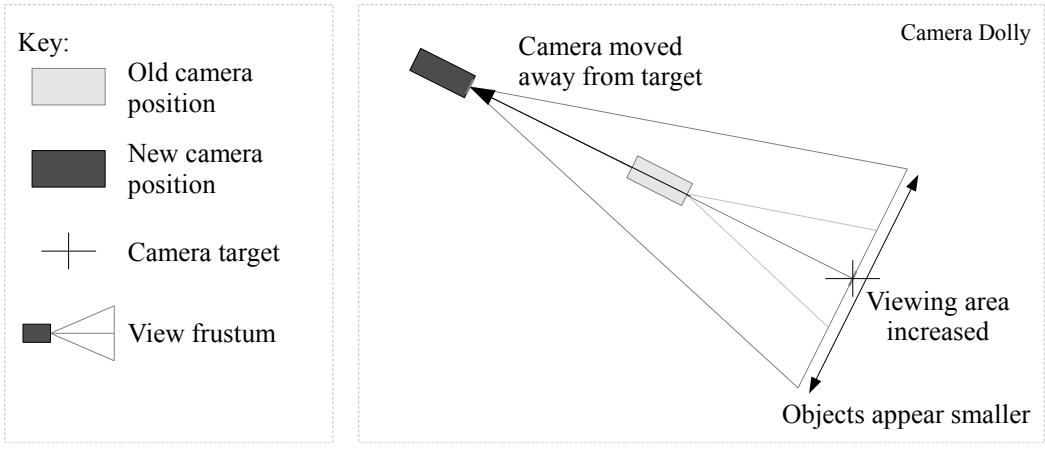



*Figure 21. Illustration of dollying a camera along the view axis*

The tool is implemented by modifying the view scale property of the custom camera control. The view scale affects the distance of the calculated camera position after taking the orbit and tilt properties into account. From the user interface, when this tool is selected, clicking and dragging along the screen Y-Axis produces a non-linear scale value relative to the starting view scale stored when the user begins the interaction. This means as the user drags further along the Y-Axis the scale of the interaction is multiplied exponentially. This allows the user to quickly zoom in and out at the expense of accuracy. This can be rectified by zooming out to an approximate distance, then restarting the interaction by releasing the mouse and clicking and dragging again to more accurately set the resolution. This works because the exponential method is more accurate at smaller distances where as it tends to jump ahead as the mouse is dragged further away from the starting point.

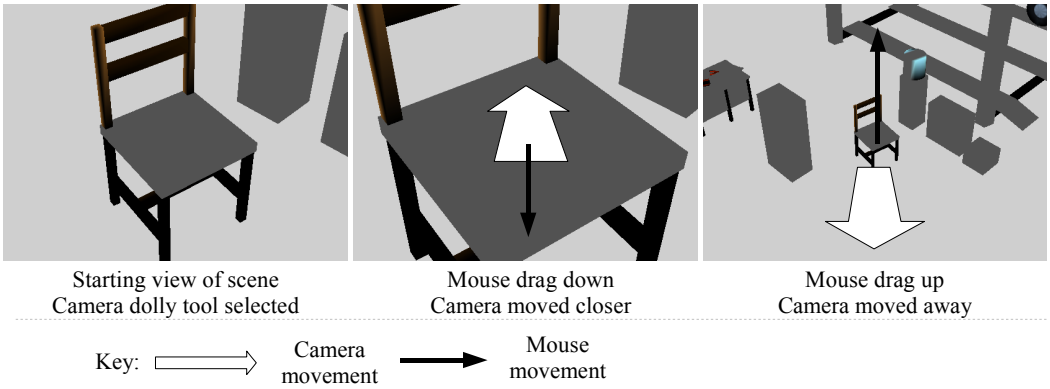



*Figure 22. Using the camera dolly tool to change the view*

### 4.2.4 Camera Pan

The camera pan tool allows the user to shift the view and target of the current view relative to the current viewing angle. In conjunction with the camera orbit tool this is a powerful way to recentre the view on an unstructured part of the model. By selecting this tool and clicking and dragging along the screen X-Axis and screen Y-Axis a *pan offset* is calculated based on the current view matrix of the camera. After the camera is updated the view angle of the view matrix stays the same, meaning a reverse action can be performed without losing position within the scene.

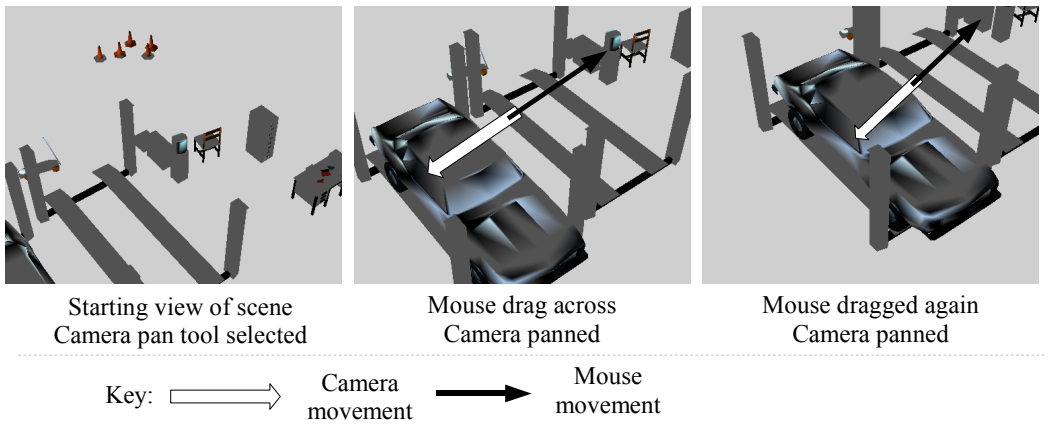



*Figure 23. Using the camera pan tool to change the view*

The pan action is calculated by scaling the screen coordinates relative to the view scale of the camera, and creating a translation matrix using the function *osg::Matrixd::makeTranslate(x, y, z)*, where x and y are the screen coordinates, and z, in this case the screen depth, remains at 0. The translation matrix is then multiplied against the camera's viewMatrix, in the order viewMatrix * translationMatrix, producing a pan offset

vector in terms of abstract camera coordinates. This is a somewhat round-about way to update the view matrix, but abstract control of the camera is maintained at the cost of a few repetitive calculations in the camera update method. There is no performance hit in this process as interactions with the camera happen in sequence with primitive interactions meaning there is no parallel overlap with multiple calculations going on at the same time. In a real time, animated, camera control system more optimization would be possible if required.

## 4.3 Primitive interactions

In order to add structure to soup, the structure from soup editor provides three tools for creating geometry selections: *Select polygon*, *Volume select*, and *Flood select*. These tools allow the user to build up a selection within the geometry data. The *Create new node* from selection tool then allows the user to break the selected geometry out of the soup and give it structure by way of a label. The actions, with the camera control actions, form the basic work flow that allows a user to add structure to soup. This section describes in detail the function, intended purpose, and implementation of these tools.

### 4.3.1 Select polygon

The select polygon tool allows a user to click on the model data to build up a selection of individual primitives, usually triangles. This is the simplest way to build up a selection, but it requires the most effort from the user in terms of identifying and clicking on individual primitives. It is also the most precise, allowing edges to be accurately set, such as detaching an object connected to a floor, or smoothing the corner selection between two surfaces. It is also useful for selecting small groups of polygons in tight places.

The way this tool is implemented is extended by the *Volume select* and *Flood select* tools. The OpenSceneGraph utility class *osgUtil::LineSegmentIntersector* was used to convert x, y coordinates from the screen into a selected polygon within 3D space. Objects of this class use the objects of the *osg::IntersectVisitor* class to traverse the scene graph and calculate intersections between two vector positions; a start position and an end position. These start and end positions are calculated from the view matrix of the currently selected view. In turn these values are used to calculate an intersection with the model data to identify the selected primitive.

The flow of this tool is as follows:

1. Catch the mouse press event
2. Select the view under the mouse as the starting view
3. Use the *osg::SceneView::projectWindowXYIntoObject(...)* function to project mouse coordinates into 3D object space
4. Retrieve start and end vectors from the near and far planes set by the scene view (The near and far planes are calculated from the boundaries of the model data)
5. Create a LineSegmentIntersector object using the start and end vectors
6. Run the LineSegmentIntersector through the IntersectVisitor on the root of the model data
7. Retreive the first hit, if any, record the selected geometry, and the list of vertices of the intersected triangle
8. Use the *sfs::PrimitiveSearch::FindPrimitive* method to identify the selected primitive from the list of vertices
9. Add the identified primitive to the current geometry selection

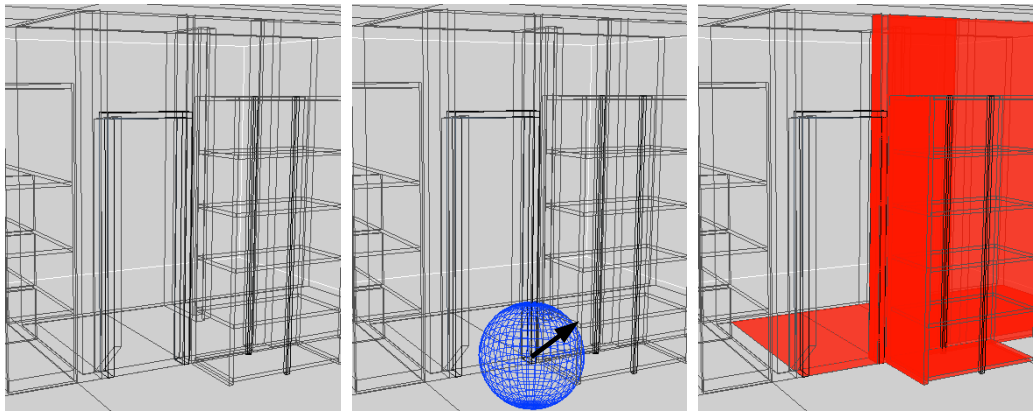
The workings of the current geometry selection object can be found in *Chapter 4.4 Geometry selection*. Using the polygon select tool builds up a selection of primitives that

are highlighted to the user in a bright red colour. This tool, and the other primitive selection tools, also support removing primitives by use of the keyboard control key to reverse the action of the tool. This modifier is an important part in reversing a user action, such as correcting a mistake, or trimming an over ambitious selection. This feature does not however replace the need for a undo/redo framework (*c.f. Chapter 7.2 Extensions to the editor*).

### 4.3.2 Volume select

The volume select tool lets the user draw a sphere on the surface of the loaded geometry, vertices inside the sphere are then used to find polygons to add to the current selection. This is a simple broad selection tool for building up selections faster then the click-by-click approach of the select polygon tool.

The tool uses the same interaction framework as the select polygon and view manipulation tools. When the tool is selected, and the user clicks the mouse on a view, the start coordinates of the interaction are tracked to calculate the distance of the mouse from the starting point. The distance variable is used to set the size of a visible sphere. The centre of the sphere is calculated by the same *LineSegmentIntersection* object used in the select polygon tool; the intersection returns the surface hit point as a vector.



Wireframe view of soup
Volume select tool selected

Click and drag on corner of
door frame: Sphere is drawn

Mouse released: Polygons
around door frame are selected

*Figure 24. Polygon select interaction of volume select tool*

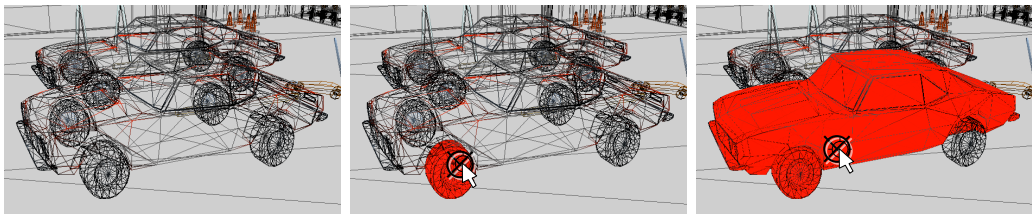
The visible sphere is a 3D user interface element that does not form part of the model scene graph. This is an example of a structure rendered by OpenSceneGraph that lies outside of the document data. In the original implementation a transparent flat-shaded sphere was used, but in the brief user testing performed it was identified that the depth of the sphere relative to the model was difficult to discern. Instead, a wireframe sphere was used. This change was made in the UI elements class independent of the calculations of the mouse interaction class. This was possible by connecting signals from the mouse interaction class to method slots in the UI elements class (*c.f. Chapter 3.8 Program design*).

### 4.3.3 Flood select

The flood select tool performs an expanding search on the model based on a selected polygon. It starts using the same steps as the polygon select tool, but then uses this selection to run the *ExpandSelection* algorithm exhaustively until the selection does not grow any bigger.

The tool allows the user to very quickly build up a selection of enclosed structures within a scene. Clean, well modelled data is ideal for this tool, as enclosed spaces such as disconnected wheels, cones, and other independent groups of polygons are easily identifiable. The tool works less well for large interconnected sets of data, for example, it would not be suitable for selecting the wing on a plane if the wing was connected to the

fuselage by adjacent polygons. In such situations, variations of this tool that require some user input to limit the size of the expansion would make it more useful. Using the same algorithm, an *expand selection* action, and a complimentary *contract selection* action could be added to allow a user to increase and decrease the size of a selection.



Starting view of cars in scene
Flood select tool selected

Mouse clicked on triangle
Primitives in wheel selected

Mouse clicked on triangle
Primitives in chassis selected
Disjoint second wheel not selected

*Figure 25. Polygon select interaction of flood select tool*

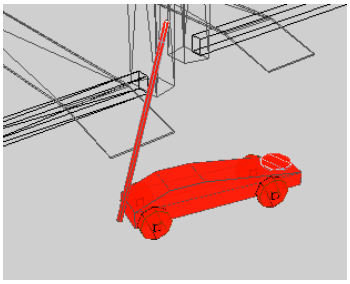
To correctly implement this tool, the model parser *sfs::GarageLoader*, a custom loader for the sample data set, had to be modified to optimize the geometry. The first implementation of the flood select tool failed because the vertices of the loaded data were duplicated, meaning the indices stored in the PrimitiveSet data structures were different for the same the vertex. Once the data was reindexed at load, the tool worked as intended, and the file size of the model data was reduced. This optimization could be generalised for any type of model data, allowing the editor to be more flexible about the formats of soup it can handle.

### 4.3.4 Create new node from selection

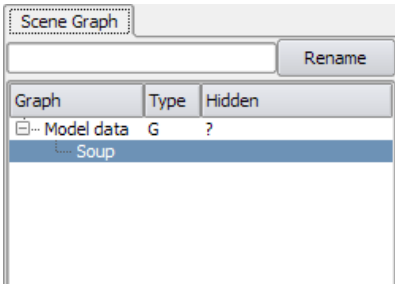
This action relies on a selection existing, if the current selection geometry is empty a warning is displayed to the user. If a selection has been made, the user is prompted to enter in a descriptive name for the selection, which is then used to label a new node created in the scene graph tree.

This is a core action in the process of adding structure to soup, and is supported by the short-cut Ctrl+N to make the user interface easier to use.

The implementation of this tool was trivial enough to be defined in the action method of the MainWindowController. The method takes the geometry selection object, copies it, and attaches it to the scene graph.



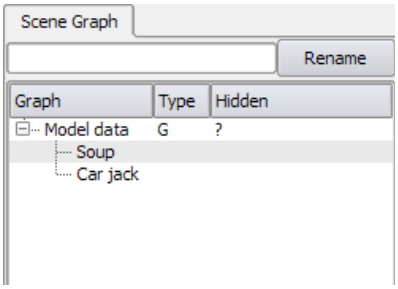
Selection of primitives is made from scene




Scene graph panel contains no structure, only soup

Create new node from selection action selected

User asked for descriptive name
Text: "Car jack" entered




"Car jack" added as new node in Scene graph panel

*Figure 26. Creating a new node from a selection*

## 4.4 Geometry selection

The geometry selection object represents the major visual aid, and structural assistant in adding structure to soup. As the user builds up a selection using the polygon select tools, a copy of the selected polygon data is added to a geometry selection object, defined by the *sfs::GeometrySelection* class. The geometry selection object is a child of the UI Elements portion of program's scene graph. The object provides methods for adding and removing individual primitives, and sets of primitives. The GeometrySelection class is limited to one geometry object reference at a time, meaning that only a subset of the current model data can be created. If model data from two separate geometry objects is required, there is a *Merge Geometry* action available.

Selected primitives are rendered as red triangles above the model. The structure from soup editor provides a series of actions regarding the display of these triangles, such as wireframe, outlined, or flat-shaded highlights to help the user understand their current selection and modify the view.

### 4.4.1 Highlighted selection

The current selection is highlighted by a bright red colour. This red colour overrides the actual colour of the selected primitives referenced by the geometry selection object. This bright visual indicator provides feedback to the user about actions performed with the polygon select tools. Depending on the view mode, highlighting selected faces can be changed to a wireframe outline mode, where only the edges of triangles are visible. This is useful for maintaining a view of individual triangles within a large selection, as demonstrated in the figure below.

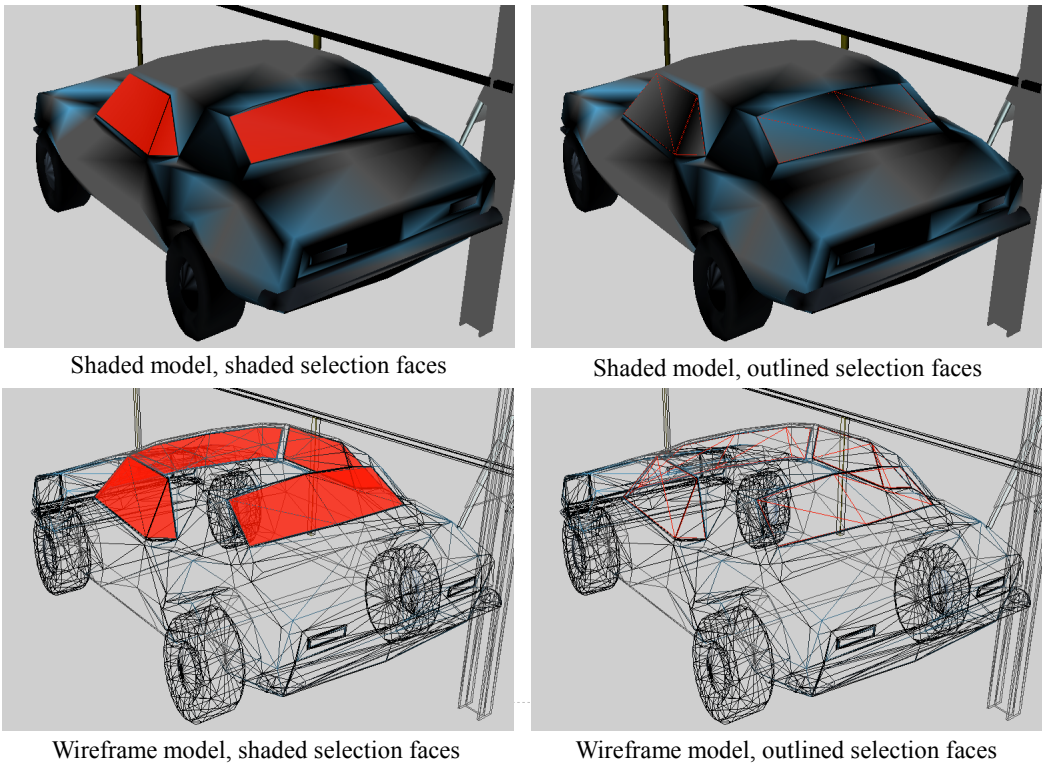



*Figure 27. Different view modes of a highlighted selection*

Red was chosen as a bright contrasting colour to normal model data. Preferably, this colour would be configurable by the user as a setting within the program.

### 4.4.2 Bounding box

The bounding box is a white wireframe box that provides a visual cue to show which geometry is currently being selected. The bounding box is also updated by the select node tool to identify the currently selected node.

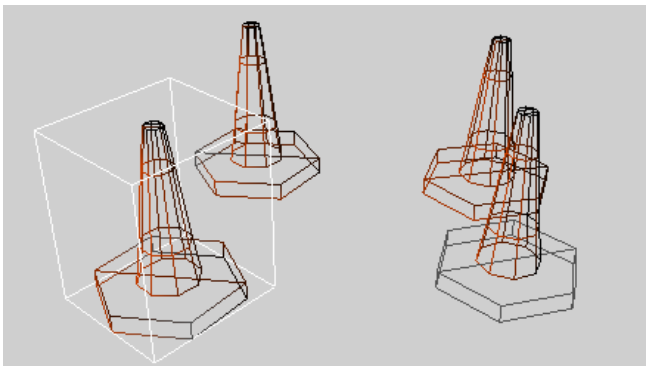


*Figure 28. Example of bounding box highlighting a selected cone (left)*

Alternatives to a bounding box could have been to brighten the current object, or grey out deselected objects. Bounding boxes are commonly found in 3D editors and computer games alike as a lightweight method of showing the boundaries of a selected object.

### 4.4.3 Select node

The select node tool is a simplified version of the select polygon tool. When this action is selected, the user can click on the view of a scene to select a node. A selected node is indicated by a white bounding box and by highlighting the associated entry on the scene graph panel.

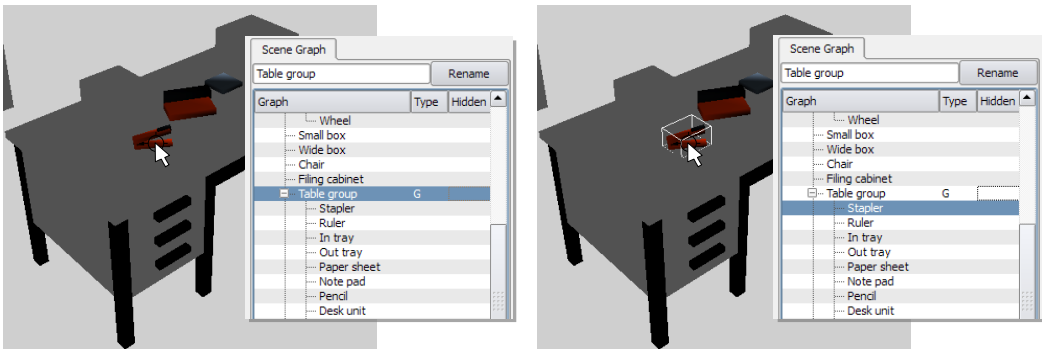



*Figure 29. Selecting an object using the select node tool*

Ideally, the select node tool would work passively to display the name of an object underneath the mouse without requiring the user to click. This would allow the user to identify objects without making a new selection. This feature was considered non-critical and left out during the remaining time-scale of the project.

## 4.5 Scene graph interactions

The scene graph of the loaded model is represented by the scene graph panel. This panel displays a hierarchical tree list of the model data, identifying groups and geometry nodes.

The scene graph panel was implemented early on to provide a visual indicator of the underlying model structure. The actual scene graph interactions weren't implemented until the end of the project.

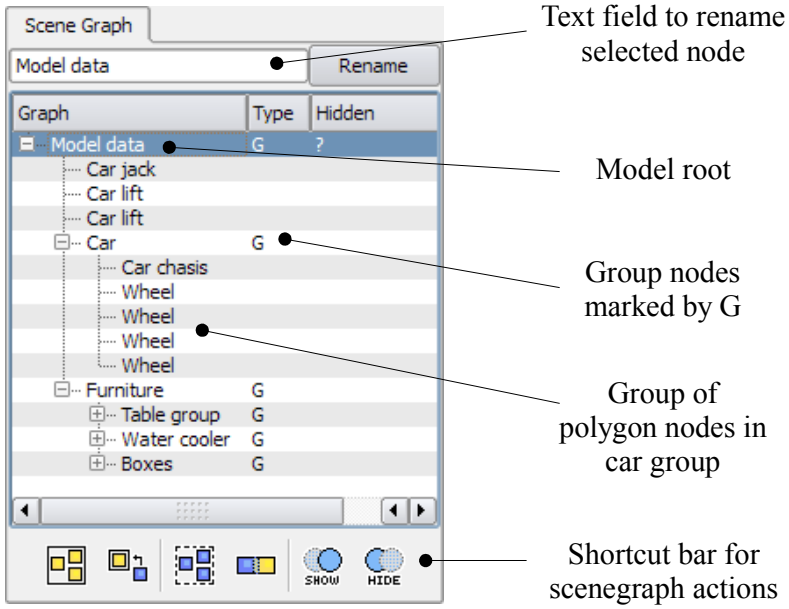
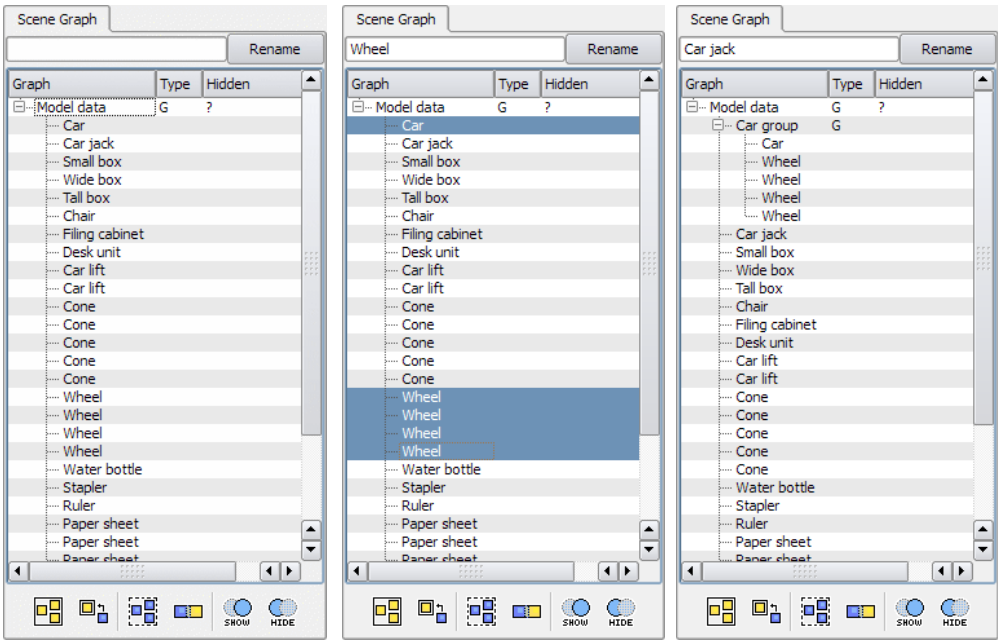



*Figure 30. Features of the Scenegraph panel*

### 4.5.1 Semantic soup

Once structure could be created using the *select polygon* and *create new node from selection* tools, a new problem was realised: semantic soup. Where polygon soup was the initial problem, after selecting and identifying all the structures within a scene, this then created a new set of data, an unstructured list of objects. This was not unexpected, but illustrated the need for a second set of tools for managing the structural complexity within the data. The following tools: *new group from selected nodes*, *move selected nodes to*

*group*, *merge selected nodes*, *ungroup branch*, and *remove selected nodes*, all provide methods to manage the scene graph and turn semantic soup into real structure.




Semantic soup
Unstructured list of nodes

Selection of car and wheels made from list

New "Car group" created adding structure to list

*Figure 31. Semantic soup displayed in the scene graph*

### 4.5.2 New group from selected nodes

The *new group from selected nodes tool* asks the user to make a selection of nodes. A selection of nodes can be made by holding down the keyboard control key, or pressing the keyboard shift key and clicking on nodes in the scene graph. This interaction is provided by the QT framework as part of the TreeList UI object. The user is then prompted for a new name for the selection. Similar to prompting the user to name a group of polygons, this method asks for a descriptive name to describe the group of nodes that they want to form. Finally, the user is shown an abridged scene graph of group nodes, and is prompted to choose a parent node to add the group to.

The implementation of this tool is quite long, and not described in full here. The user is required to choose a parent node because nodes can be selected from all parts of the scene graph, and as such a common parent cannot be automatically identified, at least not in a consistent or obvious way to the user. The method provides multiple stages of checks to ensure a valid selection before performing any actions. For instance, the method will not allow a node, and its selected parent to be moved at the same time. The method will also prevent a group node being moved into itself. These checks are essential to maintain the consistency of the document structure, and prevent the user from making accidental or illogical relationships within the data. OpenSceneGraph allows complicated scene graphs to be constructed that may contain duplicate or copied structures, this however is neither required nor desirable. As such, this method is cautious about the data it is given, and will warn the user at several points if the selection is not as expected.

The icon for this tool represents a set of nodes in yellow, that are grouped by a black box. This signifies the creation of a new group.

### 4.5.3 Move selected nodes to group

The *Move selected nodes to group* tool allows a structure to be reorganised. It is unreasonable to expect a user to create a perfect scene graph using the *new group from selected nodes* tool alone. As such, the move selected nodes to group method asks the

user to make a selection of nodes, and then prompts the user for a new parent to add the nodes to.

In a similar implementation to the group nodes tool, this method performs several checks on the validity of the selection and the target node. These checks ensure illogical structures cannot be accidentally created.

The tool is a supporting tool that allows the user to reorganise an existing structure, or modify a newly generated one. In the absence of a full undo/redo framework, this method allows mistakes or oversights to be corrected.

The icon for this tool represents an ungrouped node, in blue, being added to a grouped node, in yellow, surrounded by a black box. This signifies adding an ungrouped node to a group node.

### 4.5.4 Merge selected nodes



The *merge selected node tool* is a multi-stage tool which performs one of two opposite reactions to either the the *create new node from selection* tool, or the *create new group from selected nodes* tool. In some senses this is the anti-function of the structure from soup editor, as it removes structure, creating soup.

Depending on the selection, a decision is made on the type of merge to make. If all the nodes are group nodes, then a new group named 'merged nodes' is created, and the contents of the selected groups are moved into the new group. If all the nodes are geometry nodes, then the contents of the nodes will be removed, merged, and a new node called 'merged geometry' will be created. A mix of nodes, or a selection bridging multiple parents is not allowed, and will generate a warning displayed to the user. This tool is an example of a context sensitive action. Depending on the selection made by the user, the action of the tool changes slightly to accommodate the type data chosen.

This tool is important in the absence of an undo framework to roll back an action. It can also be used to reorganise the structure of a document. The method provides an atomic roll back to turn a structured document back into its original form.

The icon for this tool represents two different nodes being joined together by a broken dotted edge. This signifies the joining of two nodes into one.

### 4.5.5 Ungroup branch

Similar to the function of the *merge selected nodes* tool for ungrouping branches, this tool allows a user to remove a group while preserving the nodes contained within. This tool consolidates the actions of moving all the child nodes out of a group node into the parent group, and removing the group node.

The icon for this tool represents a set of nodes, in blue, inside a broken group, represented by a dotted black box. This signifies the nodes being removed from a group, in contrast to the group node icon, where yellow nodes and a solid line are used to indicate the grouping of nodes.

### 4.5.6 Remove selected nodes

The *remove selected nodes* tool allows pruning of data by deleting a selection of nodes from the document. This method has no undo, so is quite risky to use. It is however necessary to crop a set of data in order to clean up the structure of a model. Say for instance a user only wanted a car, and not the warehouse that the car was in, this tool could be used to delete everything except for the car.

Because of the problematic lack of undo, this method is not presented on the front of the user interface like the other actions, but instead hidden away on the tools menu, and the tree menu. Due to the strength of the QT Action framework, the same commands can be easily copied to different places in the user interface without modifying the underlying code. As

such, although the tool is not obvious from the user interface, it can be found on two relevant menus, increasing the chance of a user finding this action if desired.

The icon for this tool uses the colour red, the text DEL (delete), and a cross to signify danger in an attempt to warn the user of its potency.

## 4.6 A note on refactoring

Using QT's signals and slots system encourages the design of decoupled classes that communicate general purpose messages about their state. During the prototyping of the structure from soup editor the design was modified in incremental stages. Refactoring was used at several stages to simplify the growing complexity of the class hierarchy. Refactoring helped reduce the complexity of classes by abstracting common functionality to separate reusable objects. The signals and slots system provided a straightforward way to split functions apart during refactoring, while keeping the interfaces open, and without needing to generate interface classes to generalise functionality. Where more specific functionality was required, sub-classing was used to simplify the code base and reduce complexity of special case objects; a good example is the *sfs::BorderedSceneView* class which encapsulates the properties of *osg::SceneView* and the *sfs::Camera* object, as well as providing a border and a named text label. Prior to the BorderedSceneView class, the relationships between these properties were loosely managed by QOsgMultipleViewWidget, which became too complicated to extend. Refactoring made the QOsgMultipleViewWidget a more flexible and useful class, enabling the subclass *sfs::WorkspaceController* to be written very quickly and cleanly. This refactoring had minimal impact on the functionality of the application but cleaned up the interface, allowed for customisable view layouts, and allowed text labels to be added to the scene.

Further information on refactoring techniques is available through a number of book sources; this report recommends Martin Fowler's book on *Refactoring*[33], sub titled *Improving the Design of Existing Code*.

## 4.7 Chapter summary

The structure from soup editor, based on the goals and requirements identified in the design phase, was a non-trivial project to implement. The full set of features, most of which have been described in this chapter, illustrate the technical challenges involved. Fortunately, the concepts involved have been well researched and implemented by industry, providing good measures to check the quality of the software. The underlying problem of rendering and structuring data was largely solved thanks to the OpenSceneGraph framework, meaning the implementation of the view and selection tools could become the focus of the project.

The structure of the software was inspired by object orientated practices, and design patterns for the QT Framework[34]. The concept of an agile, prototype driven process for software development was executed, and the process of refactoring code was used to evolve the code through the different stages of release.

In the next chapter, a review of the final software, and how the implemented tools can be used as part of a work flow are described.

# Chapter 5 Results

The aim of the project was to add structure to polygon soup; this aim was achieved through the design and implementation of the structure from soup editor. The following chapter introduces the final tool, evaluates how it can be used to add structure to soup, how data needs to be cleaned before it can be used, what format the resulting structure is saved as, and how this structured data could be used by other tools.

## 5.1 Using the editor

This section introduces the structure from editor and explains the how the tools are used in conjunction with each other to add structure to an unstructured warehouse model. A visual guide to this material is included in *Appendix A: Screenshots*.

### 5.1.1 An introduction to the editor

The core areas of the structure from soup editor are named the *Workspace*, the *Shortcut toolbar*, the *Action menus*, and the *Scene graph panel*.

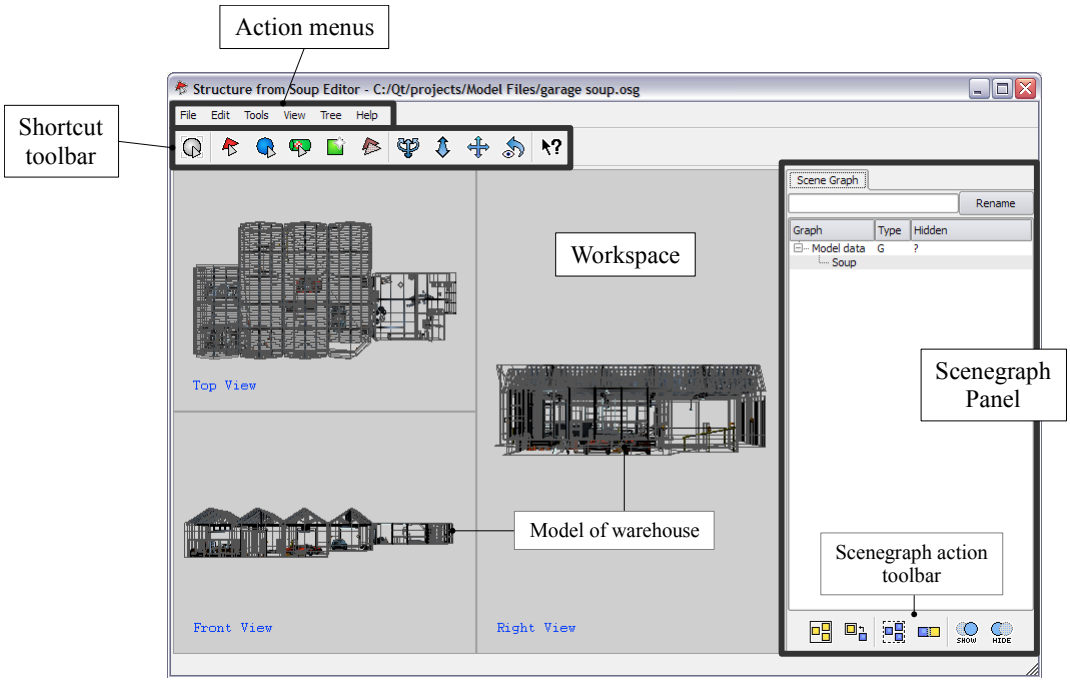



*Figure 32. Labelled screenshot of the structure from soup editor*

**Workspace**

The workspace displays the currently loaded model. It contains an OpenGL (3D rendered) view that lets the user interact with the data.

**Shortcut toolbar**

The shortcut toolbar provides shortcuts to the most common tools available in the editor. The actions here can also be found in the *Tools* action menu, but this abridged list omits the actions *delete current selection* and *remove selected nodes* to prevent accidental interactions.

**Action menus**

Stretching across the top of the editor are the action menus. These menus provide full lists of all the available actions in the program. Actions are grouped by common features found in traditional editors; File actions, Edit actions, Tools, View actions, Tree actions, and Help actions. The Tools action menu lists all of the first stage actions involved with manipulation of the view and of loaded data. The Tree action menu provides the second stage actions for

hierarchical and semantic manipulation. The View actions menu lists actions that can be used to configure the current view, such as choosing wireframe or flat-shaded modes, and hiding and showing selection indicators.

**Scene graph panel**

The scene graph is a tree list that displays the structure, or hierarchy, of the currently loaded model. Below the scene graph is another toolbar that provides shortcuts to tree manipulation methods.

### 5.1.2 Stages of adding structure to soup

The process in summary: the user loads model data as polygon soup, customises their view of the scene, creates a geometry selection, creates structure based off this selection, and then manipulates the tree structure to add additional semantics. The figure below illustrates these actions and shows how the stages of customising the view and creating a geometry selection are repeated until a structure can be identified. This work flow is evaluated in more detail in *Chapter 6.3 Sample work flow example*.

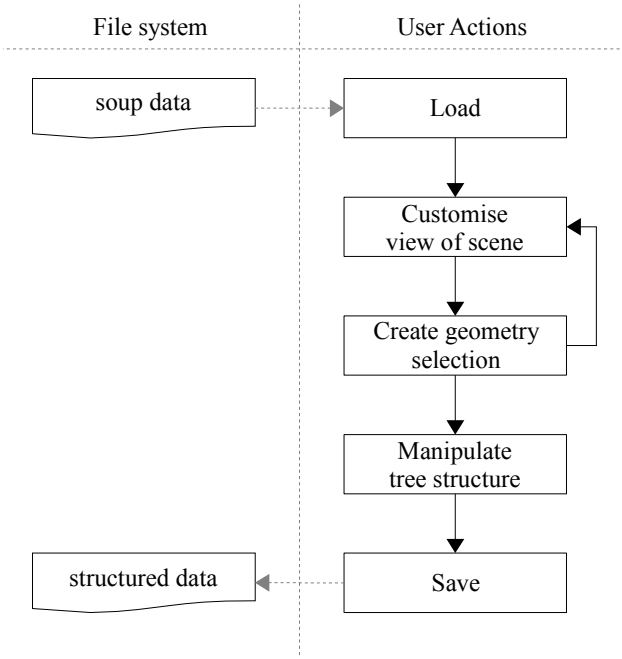



*Figure 33. User interaction work flow for adding structure to soup*

**Loading soup**

Files are loaded using either the *File > Open file...* or *File > Recent files* command. A system based file browser is provided to help users locate a file. Models are displayed straight away in the different views provided by the editor.

**Visually interacting with the data**

The camera tools, *camera orbit*, *camera pan*, and *camera dolly (zoom)* are available from a toolbar along the top of the screen underneath the menu bar. Alternatively these tools can be found in the *Tools* menu. Using the camera tools allows the user to position the view on any part of the model. The view mode can be switched between a flat-shaded and a wireframe view of the model data. The view layouts can be switched between six pre-set views, for example to a single fullscreen view of the scene, or a multi-view providing top, side, and custom views.

**Creating geometry selections**

The polygon select tools provide flexible alternatives for selecting groups of polygons. By manipulating the view, and clicking on polygons in the scene, selection groups can be formed. Selected polygons are highlighted in red, and polygons can be added to and

removed from the selection. These selection groups can then be stored as named nodes in the scene graph by using the *create new node from selection* tool.

### Adding structure to soup

Newly named structures are listed in the scene graph. By default, new nodes are attached to the model root node- the root of the document data. Once a list of structures has been identified from the soup further semantics can be added. The *create new group from selected nodes* tool allows nodes to be grouped into a hierarchy. This helps solve the problem of semantic soup, where multiple named elements share the same name, requiring additional information to tell them apart.

### Manipulating tree structures

Structured trees can be reorganised using the tree manipulation tools; these include *move selected nodes to group*, *ungroup branch*, *merge selected nodes*, and *remove selected nodes*. The *remove selected nodes* tool allows a data set to be pruned, allowing useless data to be cropped from a scene. The other tools allow a structure to be created in a non-linear and reversible fashion, allowing for reorganisation and to correct mistakes.

### Saving data

The save feature is implemented using OpenSceneGraph's native file format. This allows the newly structured data to be preserved and made available to other programs. If the original file was not an OSG file, the user is warned that the file will need saving into the OSG file format. Saving provides persistent storage, and a common output format for structured model data. The file format is an open standard, text based format, with open-source, freely available loaders available.

### Navigating structures

Models which already have structure, or have had structure added to them can be easily navigated using the *scene graph panel* and by using the *select node* tool. Clicking on an entry in the scene graph highlights the selected node with a white bounding box in the model view panes. Double clicking on an entry adjusts the view pane cameras to centre on and auto-fit the selected node. Reversely, the *select node* tool can be used to click on an object in the view pane to select and identify a node from the scene graph.

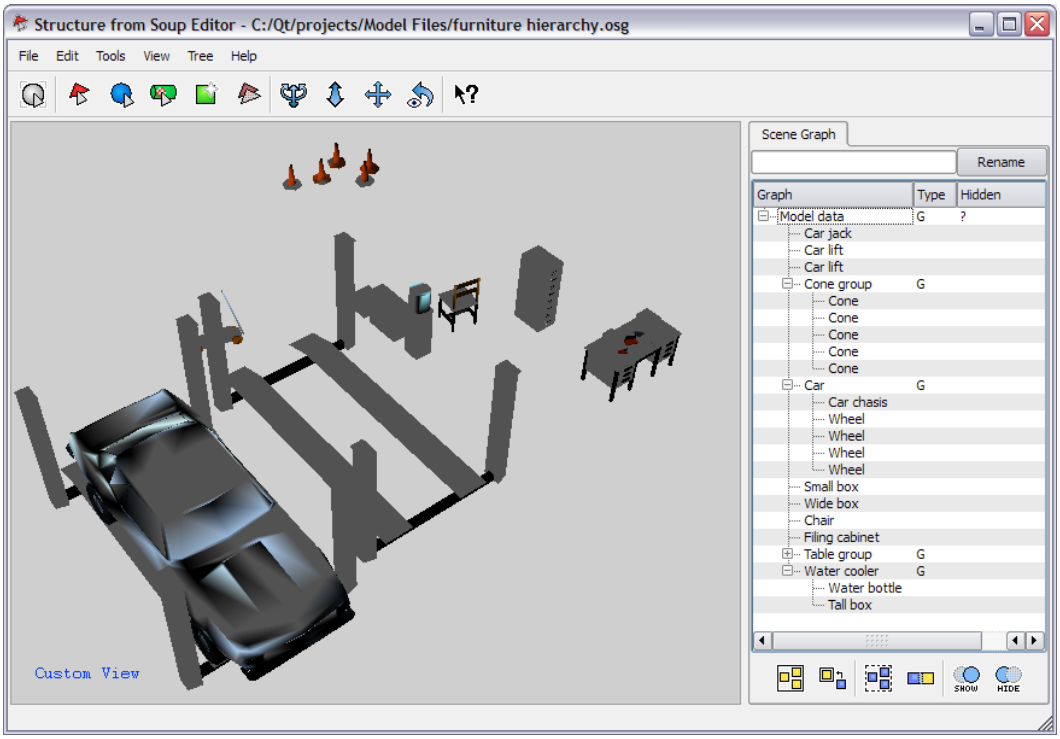



*Figure 34. A structured scene displayed in the editor*

## 5.2 Cleaning primitive data

The use of the tool is demonstrated on the sample warehouse scene which is a large set of polygon soup. Arguably, this data set is clean data, the data has easily identifiable structures that were once created in a 3D editor that have lost their names and groupings; but even though the data looked neat visually, its actual structure was incompatible with the flood select algorithm. The first test of the flood select tool resulted in only single triangles being selected. It was quickly deduced that the format of the loaded data was full of repetition, and even triangles at the lowest level were separated from each other. The model loader was rewritten to create a set of unique vertex points. This was a very memory intensive process that required checking of each new vertex against vertices already in the set. As the set of vertices was constructed, primitive sets that referenced the vertices were also created. The resulting optimised data set, loaded into memory, could then be saved out in the OSG file format. The flood select tool, unmodified, worked as expected on this clean set of data.

This example illustrates that the strength of the flood select tool is heavily reliant on the structure of the loaded data set. This suggests that the structure of other sets of data may be incompatible with the tool. The process used in the loader to restructure the warehouse data could be made available as an additional tool within the program, or as an automated process after identifying loaded soup.

## 5.3 Loading data, persistent storage

The OpenSceneGraph libraries provide support for several different file formats through its osgDB plug-in framework. Plug-ins are searched for as the program is loaded, meaning users can add support for specific file formats as new libraries become available. Additionally, a specialised loader for loading a model of a warehouse is available. Loaded data is represented in memory by OpenSceneGraph data structures and attached to the root of the model scene graph.

Not all plug-ins in the osgDB framework support saving of files. For simplicity, the OSG file format, native to OpenSceneGraph was chosen as the single output format from the editor. This format is a text based, logical descriptive notation that serialises OpenSceneGraph structures. It is very easy for a human reader to follow, and of course can be read back into the scene graph framework. The OSG file format means data can be loaded, modified, and saved with convenience for the user, and additionally provides reliability and consistency for programmers.

Ideally, the data would be available in a range of export formats, allowing compatibility with other 3D editing tools. These possibilities are discussed in *Chapter 5.4 Exporting knowledge*.

## 5.4 Exporting knowledge

The boundaries of this project were to develop a software editor to add structure to soup. Once structure has been added to soup, the information contained within the data could then be used by other programs to index and search through the 3D data structures. Although the structure from soup editor provides persistent storage for structured data, no goals or restraints were placed on the output format of the data. As such, no additional output formats were specified or implemented for exporting this knowledge to other systems.

For indexing purposes an XML format such as *X3D*[35] could be used to output the semantic structure of a scene with reference to the 3D data. Such integration would simplify the process of exporting knowledge about 3D scenes to the internet and other to databases of information. For reference purposes the scene graph could be outputted in an XML or HTML format without the 3D data to provide a summary of the model data. This

could be used to understand the contents of a 3D data file without needing specialised 3D viewing software.

From the initial research, the identified structures could also be exported with model indexes. The report on *shape similarity comparison of 3D models using alpha shapes* describes ways of indexing 3D models spatially in order to compare them against other models by their shape. For each structure identified using the structure from soup editor, an index could be generated to perform this comparison. Additionally, a model browser could be used to perform a search to present similar shapes back to the user. An advanced algorithm might be able to identify and index shapes within a supplied model and perform a search against an existing database of models to automatically name structures within a scene.

All these ideas suggest possible extensions to the the editor by making use of the information created.

## 5.5 Chapter summary

This project delivered on the initial project goals of presenting 3D model data to a user in such a way that new structures within the data could be identified, named, and stored. This process is well documented, and the tools provided allow the task to be quickly performed.

The tool, as a prototype, is sensitive to the types of data it can effectively handle. Although it works well for the test data, additional filters are required to clean and prepare different types of data. The work already done on the software could be readily extended to support more general types of data.

The loading and saving framework, provided by OpenSceneGraph, provides a persistent storage system that allows useful work performed in the editor to be saved and used in other programs. The possibilities of exporting knowledge in extension to the editor are wide reaching, and integration of some of these features might be desirable in order to enhance the tools already provided by the editor.

# Chapter 6 Testing and evaluation

The use of agile based, prototype driven software development methods in the construction of the structure from soup editor implied a continuous testing environment. The description of an agile methodology, that requires regular, fully working versions of the software, means that testing needs to be performed at each build stage of the project. This chapter describes the merits of continuous testing and evaluation, with notes on build progression and a sample work flow example.

## 6.1 Continuous testing and evaluation

For each cycle of the software implementation, a feature was added and fully tested before implementing the next feature. This process of continuous testing and evaluation provided regular feedback to the developer and to the project supervisor on the working state of the software. The process of testing each new feature with all available data ensures that features added to the program do not impact other parts of the program.

As discussed in *Chapter 3.5 Prototype driven development*, the software development process can be summarised as:

Select feature > Implement feature > Compile and debug program > Test all features

If a feature caused an error, or prevented another feature from working, then this was noted as a bug and the process was restarted with the bug in mind. Rather then a formal test plan, all implemented features were tested by repeatedly following the intended user process of loading and interacting with model data. The most regular test, to test the *Open File...* and *Close File* actions, was performed by repeating: open file and close file a number of times. This test proved that data was being cleared out properly when closing a file. Early on in development it was a useful test for identifying memory leaks. Later, these tests were wrapped around a set of actions, such as *open file, orbit camera, pan camera, zoom camera, close file*. As new features were implemented, the stages of loading files, changing the view, selecting each of the tools, building a selection, and creating structure were frequently repeated.

A series of automated tests performed after each compiled version of the software would have been desirable. Much of the implementation was trivial processing of input and output conditions. A test harness could have been written to evaluate these functions, but instead methods were written and tested from the prototype. If they worked, and did not interfere with other parts of the program, they stayed. If features caused problems, the actions were disabled from the user interface, and a bug noted. If a core feature needed debugging, the output panel, added to the user interface, was used to trace test variables and run methods in an arbitrary fashion to examine the results.

The majority of the bugs found were related to user interactions. The type of data produced from these interactions could not be produced by writing a program. For example, a visual bug discovered mid-way through the project was when the user had zoomed in multiple times, and could not zoom back out. It would be very difficult to write a test case for such an interaction without first understanding what data was being generated by the user interaction. In tracking this input data, the problem was often revealed, and fixed, without needing to write a programmatic test case. As such, running tests from the user interface on a regular basis highlighted the most problematic bugs (*See Chapter 6.3 Sample work flow example*).

The idea of test-driven-development (TDD) was considered as an option for the software development process, but investigation into TDD for C++ found little evidence of preferred or well established frameworks. Test driven development is the idea that tests are written for individual units of code before writing the code. This creates an environment where all code is thoroughly tested, creating better, less error-prone software. It seems TDD for C++ is still in its infancy, or at least not in widespread adoption. *CppUnit*[36] was an example of

one TDD framework looked at. There is however mitigating opinion that TDD is not suitable for developing GUI frameworks[37], or for developing objects that have complex containers. Trying to write test cases prior to implementing GUIs requires simulation of more elements then is worthwhile, and in many cases this results in a harness-style approach to testing; this can lead to bugs somewhere else in the system breaking tests that have nothing to do with functionality of the test framework.

In summary, although no formal testing methodology was applied, regular testing of all the features was part of the software development process to provide continual feedback and to identify bugs at an early stage. For prototype driven development, this was suitable for creating stable, presentable software at regular intervals. This chosen methodology would not however be suitable for producing production quality programs aimed at commercial or public release.

## 6.2 Notes on build progression

In order to preserve the working state of the software, while implementing new functionality, regular backups were made of the project's folder structure. This ensured that new functionality, which often involved modifying existing code that might break or damage the program, could easily be rolled back. On a few occasions code was reimplemented from backups to correct errors in newly implemented methods.

A version naming scheme in the form “SFS Editor a1” was used at the start of the project to contain the working files. At each major, demonstratable release of the software, a copy of the working folder was made and renamed, for example “SFS Editor a2”. After version a9, a new release, “SFS Editor b1” was created. This simple backup process meant old versions were readily available for reference. Additionally, in communications with the project supervisor, versions could be easily denoted.

At the start of the project the use of an archiving tool such as *CVS*[38] or *SVN*[39] was considered. For the size of the project, and because there would be only a single developer, it was decided that this was unnecessary. Source control systems are much more important in team development environments as part of a well defined, professional quality, software development process- the benefits of which would be largely lost on a single developer.

The data from each backup was regularly copied across three separate machines. A PC, a laptop, and a remote file store at *The University of Manchester - School of Computer Science*[40]. Synchronising software was used to compare each of the folders and *contribute* files between sources. This ensured that no files were deleted, and the latest files were made available across all servers. This additionally allowed the developer to work from multiple locations, and provided alternative development environments should one of the systems fail.

## 6.3 Sample work flow example

The purpose of the software was to add structure to soup. The following work flow diagram summarises the order of common tasks that can be performed by users.

The sample work flow indicates how users might interact with polygon soup by firstly identifying individual structures, and then creating a hierarchy. Alternatively, groups could be created, and placed into a structure before identifying individual structures, but this process would require a priori knowledge of the model data not deductible from the data itself.

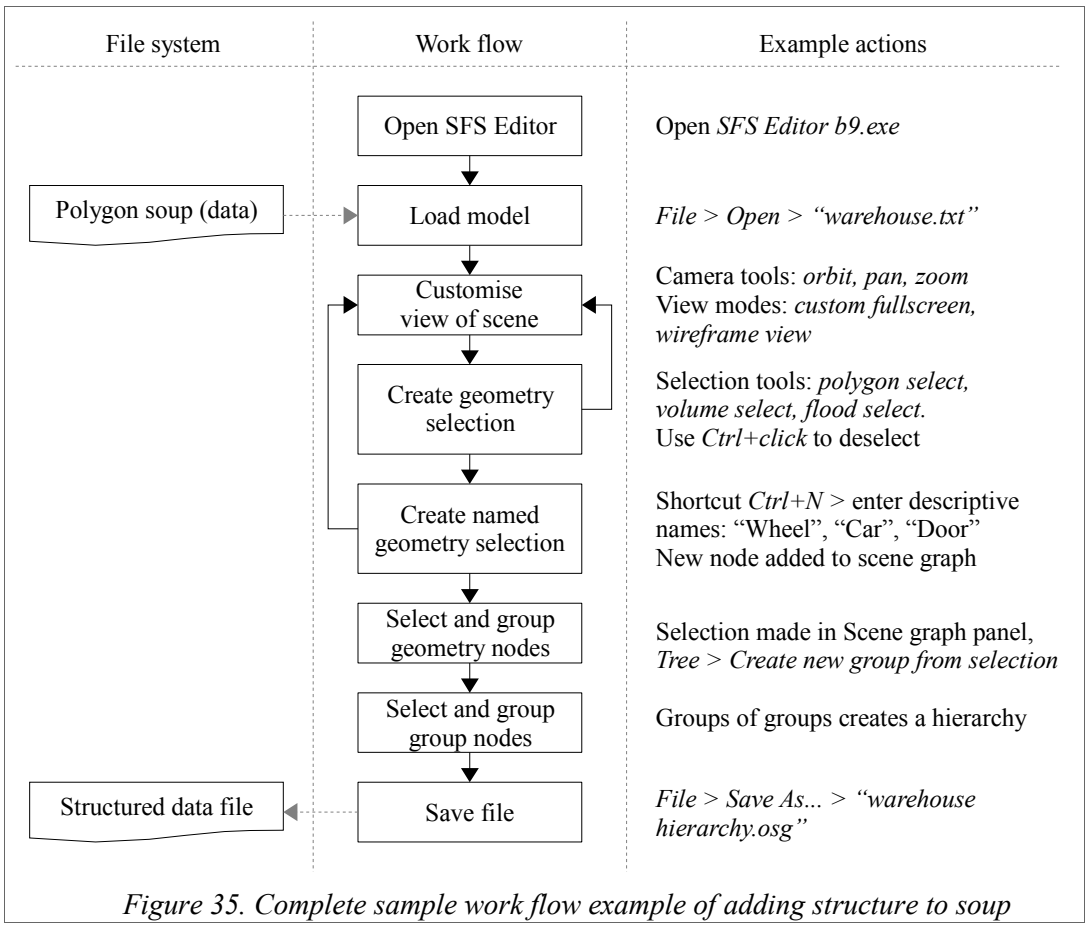



*Figure 35. Complete sample work flow example of adding structure to soup*

The work flow maps closely on to the groups of tools provided by the editor; it was evolved through continual testing of the program. As such, this is the recommended process for adding structure to soup, using the editor, based on the experience of structuring several sets of sample data.

## 6.4 Chapter summary

By choosing an appropriate development process, and agreeing suitable source control and delivery mechanisms at the start of the project, a testing and evaluation methodology was executed that produced regular demonstrations of the software. This helped communicate the working state of the software, indicate problems and limitations encountered, and allow decisions to be made on the direction of the project.

# Chapter 7 Conclusions

The project goals set out at the start of the project provide a tangible overview of what the structure from soup project aimed to achieve on behalf of the *School of Computer Science* and the student involved. This section concludes the project by examining the practicality of the tool, by looking at possible extensions to the editor, by providing a critique on the development process, and by questioning the relative success of the project.

## 7.1 Practicality of tool

The warehouse source data provided at the start of the project was optimized for real-time rendering. A scene could be rendered from the data that a user could navigate around; but this data had no discernible structure to speak of. The task of identifying groups of polygons, made visible by a rendering a scene, is a non-trivial one. If this task was performed by hand, without the help of a software editor, it might have taken many months to break apart this specific set of data.

In contrary, the time taken to develop the structure from soup editor has made it almost trivial to break the same set of data down in a matter of hours. In a matter of minutes, useful structures such as furniture, walls, ceilings, and so on, can be identified and stored. In turn, time can be spent organising these structures to add additional semantics to help a user navigate through a scene helping them understand the data. As such, the tool is very practical at managing the sample data, producing structure from soup.

In terms of other types of data, it has been indicated that there are weaknesses with the tool regarding the underlying structure of model data. Without other data sources to test, it is not clear what problems could arise when using the tools provided. An example of which is the flood select tool which initially failed on the sample data, even though its implementation was fine. As such, the tools in the editor are optimized to work with one set of data, but it is unknown how effective the editor would be at handling other sets of data.

The knowledge gained from implementing the tools, and by working closely with the OpenSceneGraph framework suggest that extensions to the editor could be easily made to support new sets of data. The hard work of integrating the QT and OpenSceneGraph frameworks, providing multiple views of the scene, and writing easy to use camera tools, suggest that more time could have been spent addressing these issues of data dependency. This could be achieved by proving parsing tools for cleaning and preparing different sources of polygon data.

## 7.2 Extensions to the editor

The output of the structure from soup editor is a structured scene graph stored in the OSG file format. Created structures contain hierarchies of identified polygons, and these can be used as useful data for other applications.

As described in *Chapter 5.4 Exporting knowledge*, identified structures could be used in a searchable database of models. Such a database could be built on the named groups of polygons, or using spatial indexes of the model data. Spatial indexes would allow for shape comparison against supplied models, where as a textual search would rely solely on the naming of identified structures.

Picking on the idea of “semantics”, the structure of the scene hierarchy could also be searched in a more logical way. From the sample data a *Car group* was identified that contained a *Car chassis*, and four *wheels*; a semantic query of this scene could look for “Vehicles with at least four wheels” to return models of cars within the scene. This would require additional mark-up not available in the editor. Semantic and ontological standards such as *RDF Schema*[41] and *OWL*[42] could be used to enhance the hierarchical structure from the editor for use with *Semantic Web*[43] type systems. An option to export the structure as an XML document would be an easy way to facilitate this functionality.

Additionally, the editor lacks support for the X3D format. X3D is the successor VRML, and was identified at the start of the project as a suitable format for distributing structured data to the web and to other sources. During the time of the project, only the OSG file format was implemented; alternative output formats would allow structured data from the editor to be used elsewhere.

Texture support was not included in the prototype tool. This forces the assumption that polygon soup contains no texture information. In order to bridge this handicap, tools such as *Icarus*[44][45] that allow photo-texturing of models could be used in conjunction with structures identified using the editor. The process of laser scanning a scene to produce unstructured data could form the start of a work-flow where polygon soup is generated, structure is created using the SFS Editor, and photos are used to texture individual structures using Icarus.

## 7.3 Development process

The agile development process selected for this project was aimed at producing regular prototypes of the software for demonstration. This provided two major benefits; achievable deadlines were set on a regular basis, forcing progress to be made, and the process of development was thoroughly tested and understood, meaning the software and practices were well documented and easily repeatable.

As voiced by lecturer and guests, at the School of Computer Science, agile development methods are becoming widely accepted as cost effective and practical ways of developing software. Development methods that are slow to produce software create long delays when bugs are found and fail to highlight problems with requirements or bugs with the tool, until long after time and effort is invested into a project. By choosing an agile development method risks were mitigated and features suitable for the next milestone could be selected.

The major risk of the project was in the integration of the QT and OpenSceneGraph frameworks. The analysis done at the start of the project, in background and research, failed to identify that integrating the two libraries was a non-trivial activity, and this delayed work on implementing the high-level tools. Additionally, the complexity of the OpenSceneGraph data structures, which are optimized for display and rendering of 3D data, caused trouble in the implementation of algorithms to deal with these data structures. Poor documentation about these data structures created delays towards the end of the project before the useful polygon-selection tools could be implemented. The OpenSceneGraph user base will need to grow to provide more information about these low-level implementation details.

In retrospect, the Trolltech QT Framework was the strongest part of the software development process; the framework provides QT Designer for user interface design, suggested design patterns, low-level memory management, and well documented guidelines and examples for developing software. The classes and features of the QT Framework helped reduce and solve problems caused by OpenSceneGraph, and in the end, the two systems seem to work well together.

## 7.4 Success of the project

Developing a 3D editor to display and manipulate polygon soup has been a challenging one. Through the course of the project careful research was performed on the data structures and processes involved with modern software development.

In terms of breaking an unstructured data file down into identifiable structures, the tool successfully demonstrates the ease in which this can be achieved. Much of the time and effort lost when losing the source files for the *warehouse model* have been recovered by the simplicity of the tools available through the editor. As such, useful work has been performed as a result of producing this practical software. In turn, further research and development opportunities have been suggested based on extensions identified in this report.

By following practices taught at *The University of Manchester*, the problem of creating software was broken into manageable, repeatable steps, producing software that solves the initial problem. This resulted in a process that can be followed by others interested in developing similar software.

Finally, the task of producing working software suitable for demonstration has been achieved within the deadlines set by *The University of Manchester*, and knowledge of the processes involved have been recorded. Additionally, the student involved has learnt valuable skills with C++ programming, Trolltech's QT framework, OpenGL, and the OpenSceneGraph framework. It is the hope of the student that the tool will be used for educational purposes, both to explain the concept of polygon soup and as a demonstration to future students for what is achievable during the final year of a student's course.

# Chapter 8 References

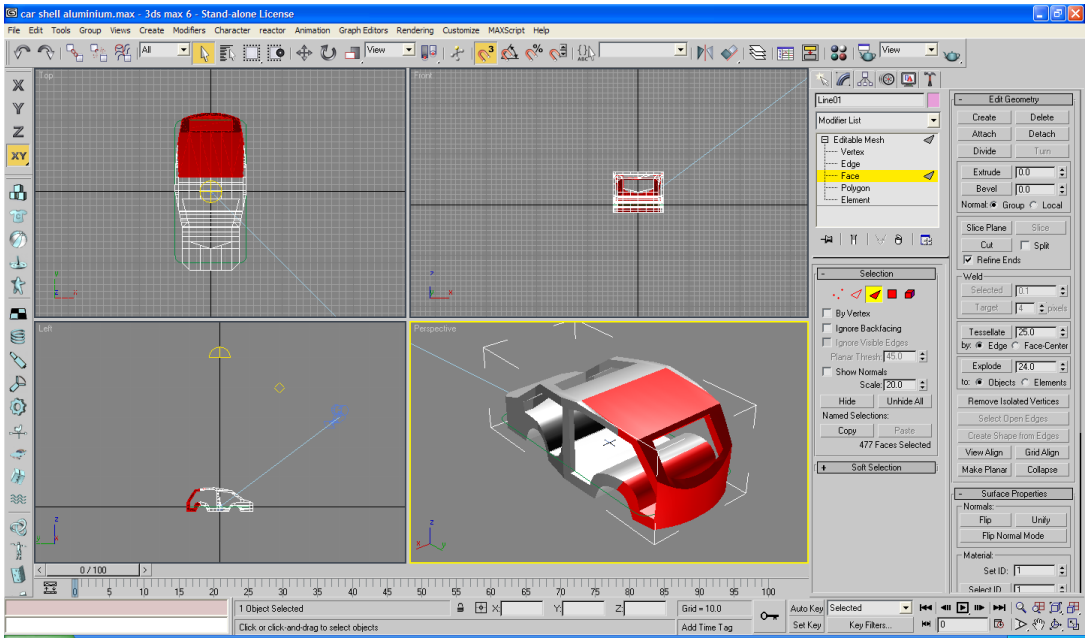

1. Christian Fruer, Siddharth Jain, Avideh Zakhor, “Data Processing Algorithms for Generating Textured 3D Building Facade Meshes from Laser Scans and Camera Images”, International Journal of Computer Vision, 2005, Vol 61, p159-184
2. “Autodesk 3DSMax”, 2008-04-26 <http://www.autodesk.com/3dsmax>
3. “Newtek Lightwave 3D”, 2008-04-26 <http://www.newtek.com/lightwave>
4. “Autodesk Maya”, 2008-04-26 <http://www.autodesk.com/maya>
5. “Blender”, 2008-04-26 <http://www.blender.org/>
6. “Google SketchUp”. 2008-04-26 <http://sketchup.google.com/>
7. “Moray”, 2008-04-26 <http://www.stmuc.com/moray/>
8. “Metasequoia”, 2008-04-26 <http://www.metaseq.net/english/index.html>
9. “Emperor Battle for Dune”, 2008-04-26 <http://en.wikipedia.org/wiki/Emperor:_Battle_for_Dune>
10. “Warzone 2100 Resurrection project”, 2008-04-26 <http://www.wz2100.net/>
11. David Arendash, “The Unreal Editor as a Web 3D Authoring Environment”, 3D technologies for the World Wide Web, 2004, 9, p119-126
12. Jesse C. Himmelstein et al, “Swept Volume approximation of polygon soups”, International Conference on Robotics and Automation, 2007, p4854-4860
13. Hamid Laga, Hiroki Takahashi, Masayuki Nakajima, “Geometry Image Matching for Similarity Estimation of 3D shapes”, Computer Graphics International, 2004, p490-496
14. Ryutarou Ohbuchi, Tsuyoshi Takei, “Shape similarity comparison of 3D models using alpha shapes”, Pacific Conference on Computer Graphics and Applications, 2003, p293-303
15. Kai Hormann, Bruno Lévy, Alla Sheffer, “Mesh Parameterization: Theory and practice”, ACM SIGGRAPH courses, 2007
16. “OpenGL”, 2008-04-26 <http://www.opengl.org/>
17. “Trolltech QT framework”, 2008-04-26 <http://www.trolltech.com/>
18. “OpenSceneGraph”, 2008-04-26 <http://www.openscenegraph.org/>
19. “HaiQ – Hand coders IDE for QT”, 2008-04-26 <http://groups.google.com/group/haiq>
20. Dave Shreiner, Mason Woo, Jackie Neider, Tom Davis, "OpenGL® Programming Guide: The Official Guide to Learning OpenGL", 6th Edition, Addison Wesley Professional, 2007
21. Dave Shreiner, “OpenGL® Reference Manual: The Official Reference Document to OpenGL”, 4th Edition, Addison Wesley Professional, 2004
22. “MinGW - Minimalistic GNU for Windows”, 2008-04-26 <http://www.mingw.org/>
23. “Microsoft Visual Studio”, 2008-04-26 <http://msdn.microsoft.com/vstudio/>
24. “Eclipse - an open development platform”, 2008-04-26 <http://www.eclipse.org/>

25. "ISO - International Organization for Standardization", 2008-04-26 <http://www.iso.org/>
26. “Valve Software - Half-life 2”, 2008-04-26 <http://www.valvesoftware.com/>
27. “Adobe Photoshop”, 2008-04-26 <http://www.adobe.com/photoshop>
28. “ossimPlanet”, 2008-04-26 <http://www.ossim.org/OSSIM/ossimPlanet.html>
29. “FlightGear Flight Simulator”, 2008-04-26 <http://www.flightgear.org/>
30. David Thomas, David Heinemeier Hansson, “Agile Web Development with Rails”, 2nd Edition, Pragmatic Bookshelf, 2006
31. “Big O notation”, Dictionary of Algorithms and Data Structures, 2008-04-27 <http://www.nist.gov/dads/HTML/bigOnotation.html>
32. “Using Viewing and Camera Transforms, and gluLookAt()”, 2008-04-27 <http://www.opengl.org/resources/faq/technical/viewing.htm>
33. Martin Fowler, “Refactoring - Improving the Design of Existing Code”, Addison-Wesley, 2006
34. Jasmin Blanchette, Mark Summerfield, “C++ GUI Programming with Qt 4”, Prentice Hall PT, 2008
35. “X3D at the Web3D Consortium”, 2008-04-27 <http://www.web3d.org/>
36. CppUnit - Test Driven Development framework, 2008-04-14 <http://cppunit.sourceforge.net/cppunit-wiki>
37. Craig Andera, “TTD Is Great... Except When It Isn't”, 2008-04-14 <http://www.pluralsight.com/blogs/craig/archive/2004/10/17/2852.aspx>
38. “Concurrent Versions System (CVS)”, 2008-04-27 <http://www.nongnu.org/cvs/>
39. “Subversion (SVN)”, 2008-04-27 <http://subversion.tigris.org/>
40. “School of Computer Science – The University of Manchester”, 2008-04-27 <http://www.cs.manchester.ac.uk/>
41. “RDF Vocabulary Description Language 1.0: RDF Schema”, 2008-04-27 <http://www.w3.org/TR/rdf-schema/>
42. “OWL Web Ontology Language”, 2008-04-27 <http://www.w3.org/TR/owl-features/>
43. “W3C Semantic Web Activity”, 2008-04-27 <http://www.w3.org/2001/sw/>
44. S. Gibson, J. Cook, T.L.J. Howard, and R. J. Hubbold, “ICARUS: Interactive reconstruction from uncalibrated image sequences”, SIGGRAPH 2002 Sketches and Applications Programme, 2002, p197
45. S. Gibson, T.L.J. Howard, “Interactive reconstruction of virtual environments from photographs, with application to scene-of-crime analysis”, Proceedings of ACM Symposium in Virtual Reality Software and Technology 2000, 2000, p41-48
46. UC Berkeley City Modelling Project, Fast 3D City Model Generation <http://www-video.eecs.berkeley.edu/~avz/3d/3d.html>, 2008-04-27

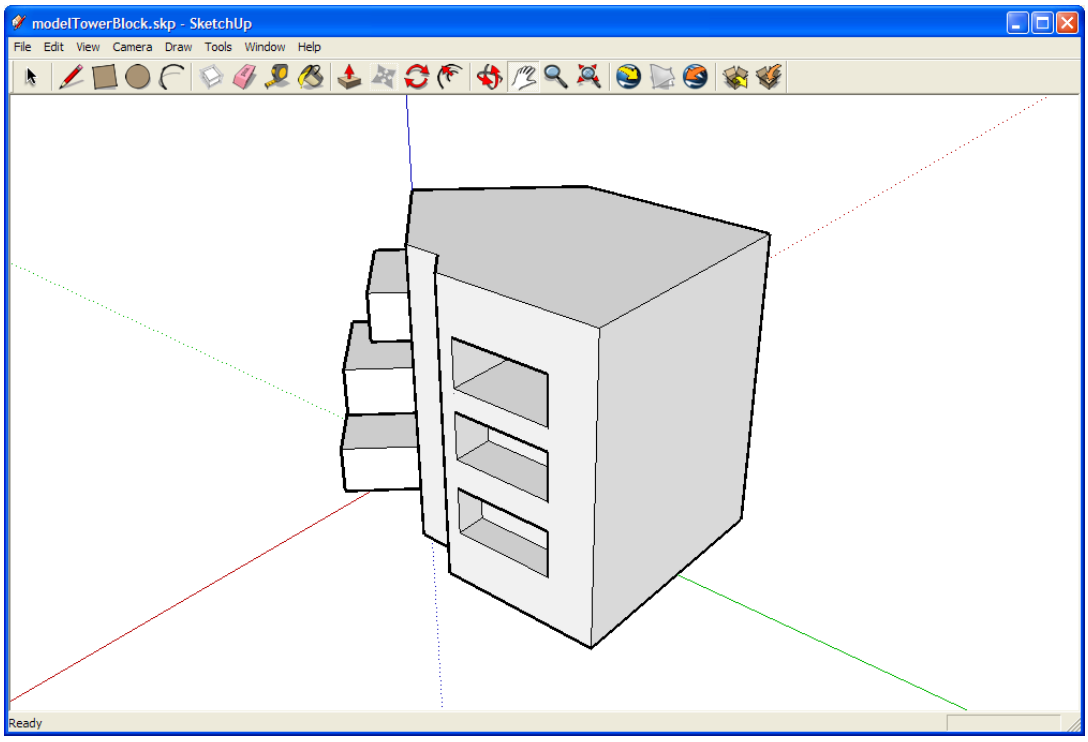
# Appendix A: Screenshots

## Examples of 3D Editors

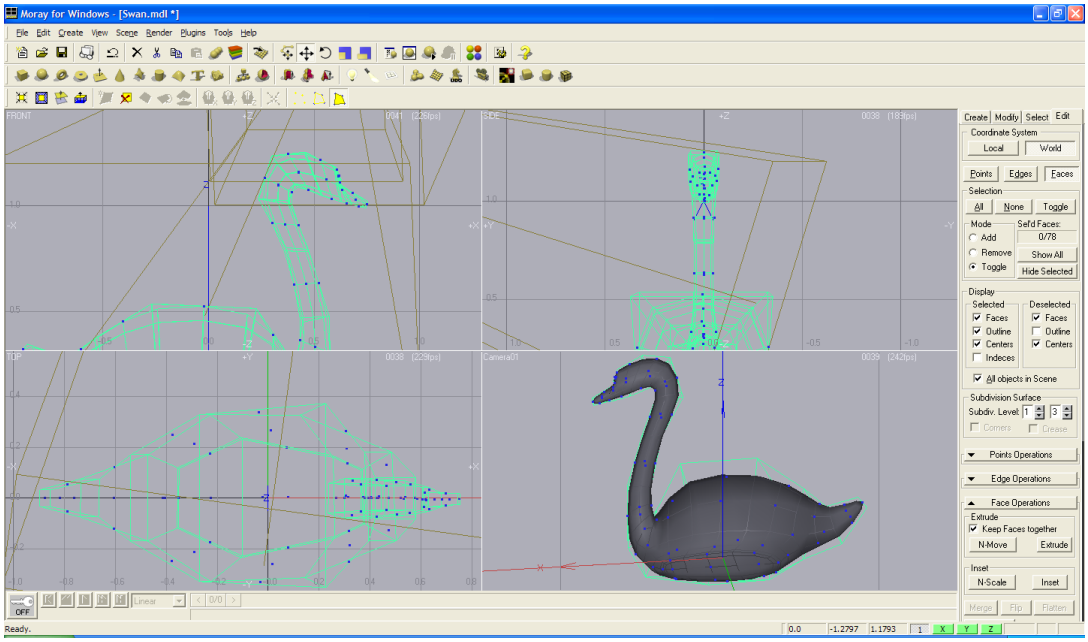
### Autodesk 3DSMax



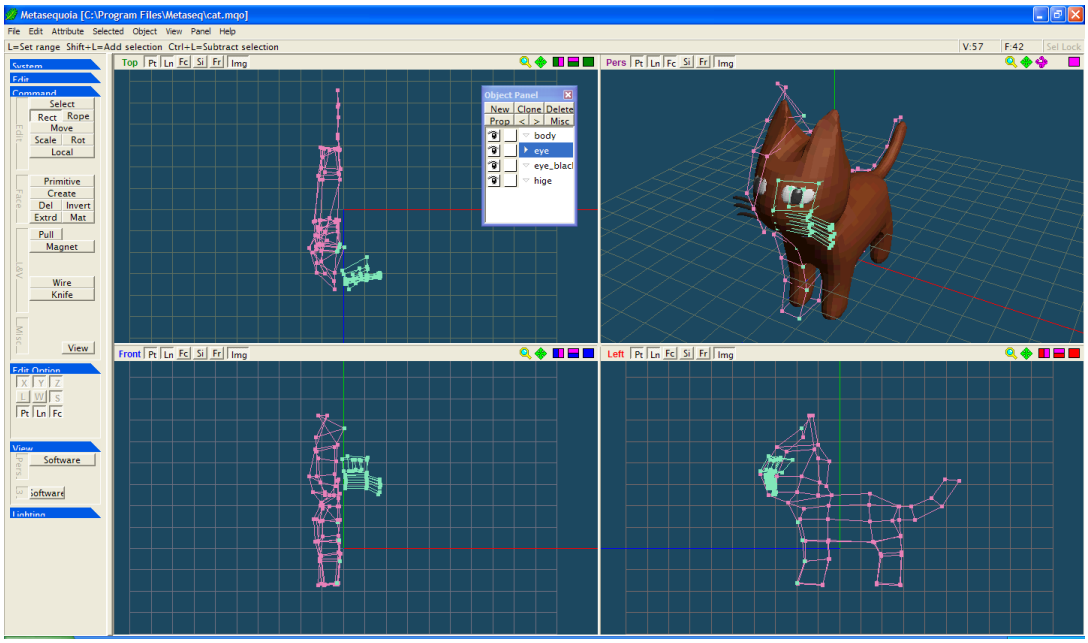

### Google SketchUp



## Moray

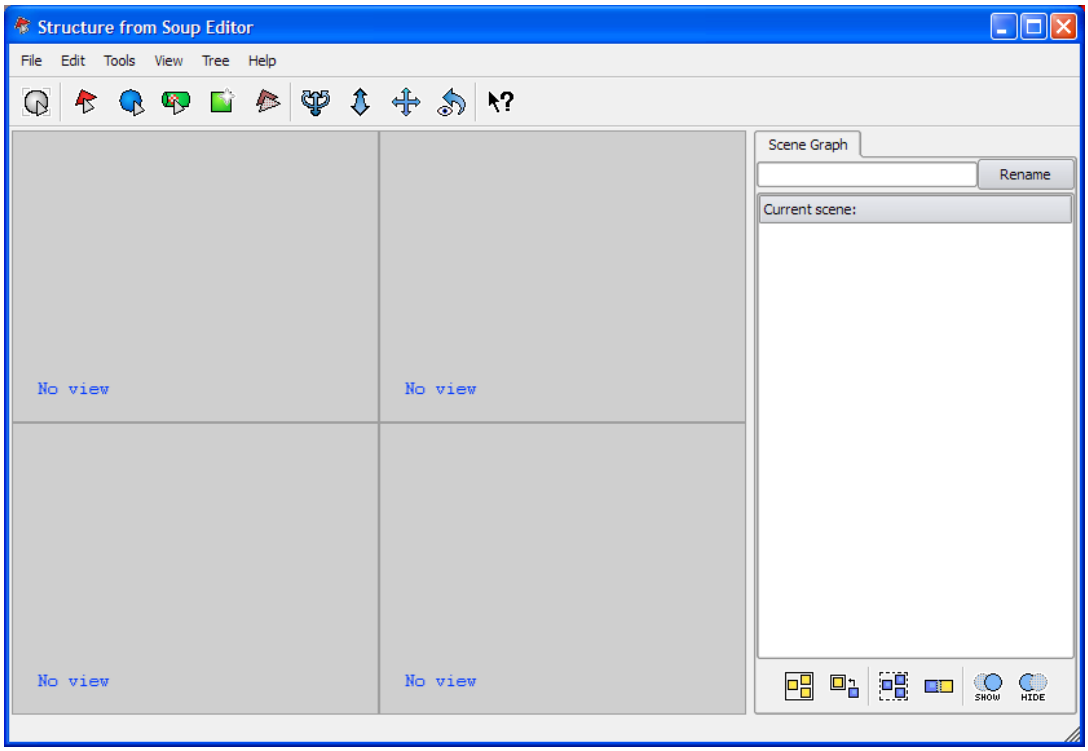



## Metasequoia

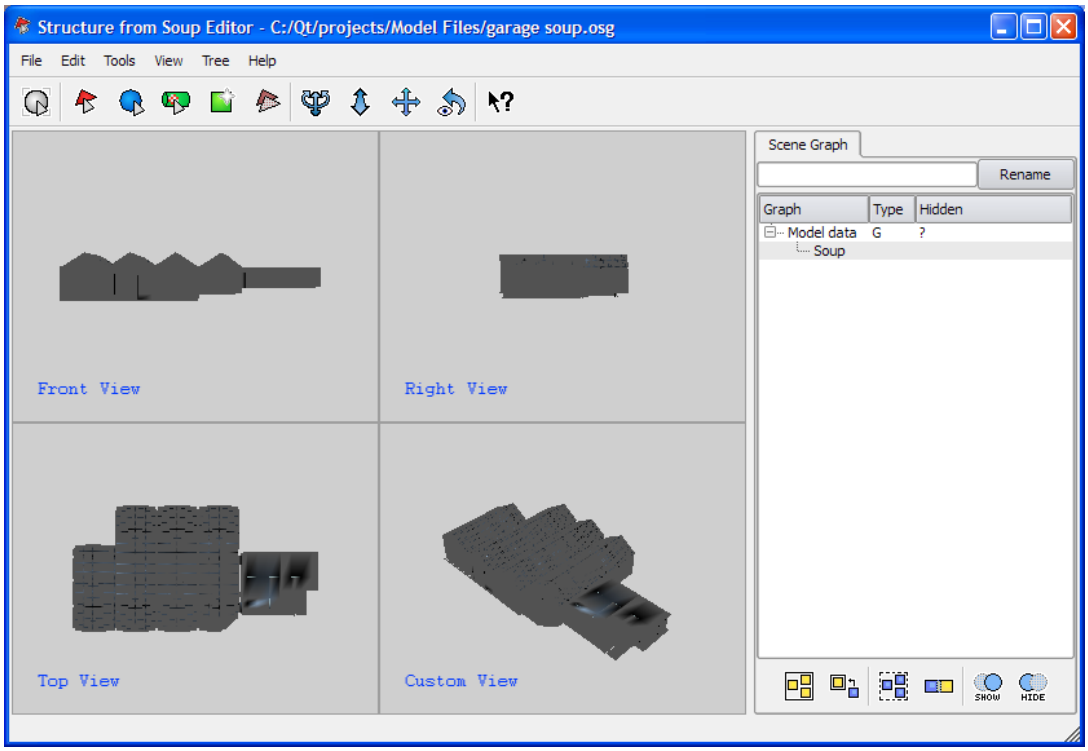


# Using the Structure from Soup Editor

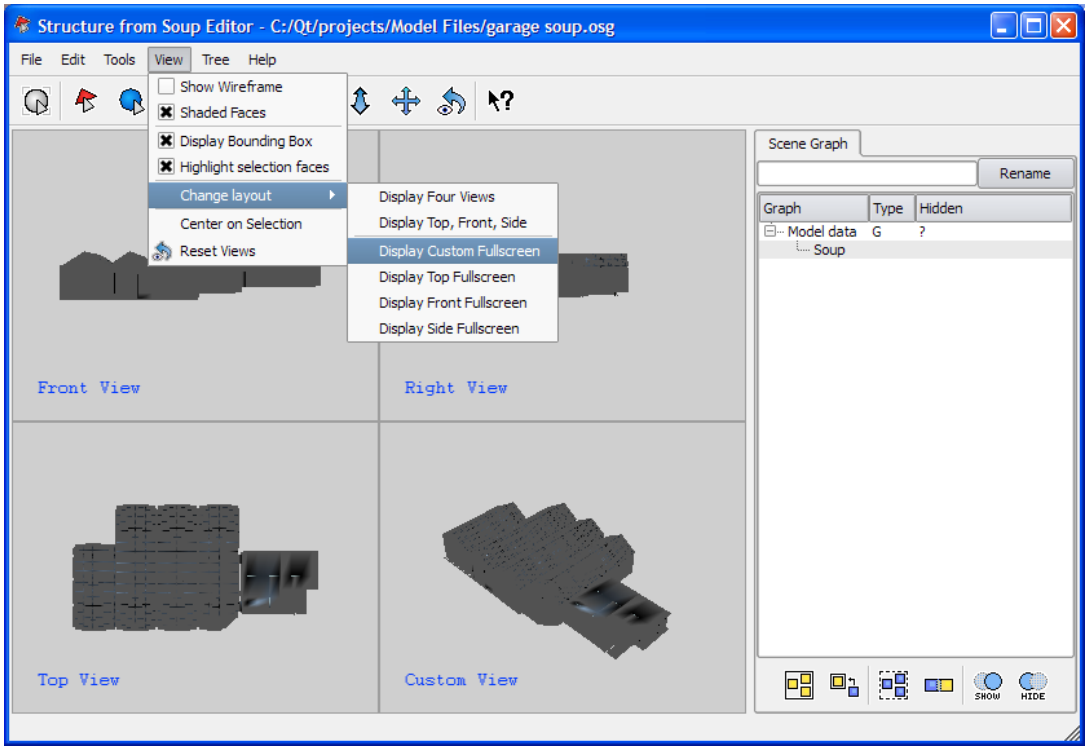
### Starting the editor



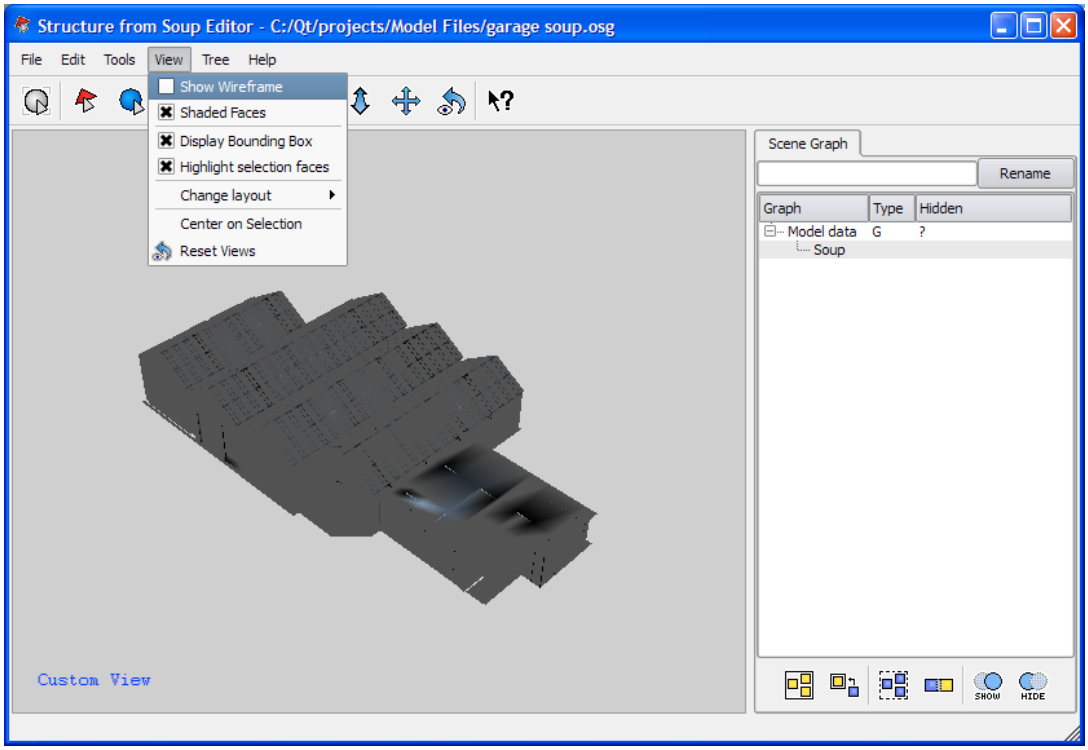

### Loaded model soup - warehouse model – Front, Side, Top and Custom views



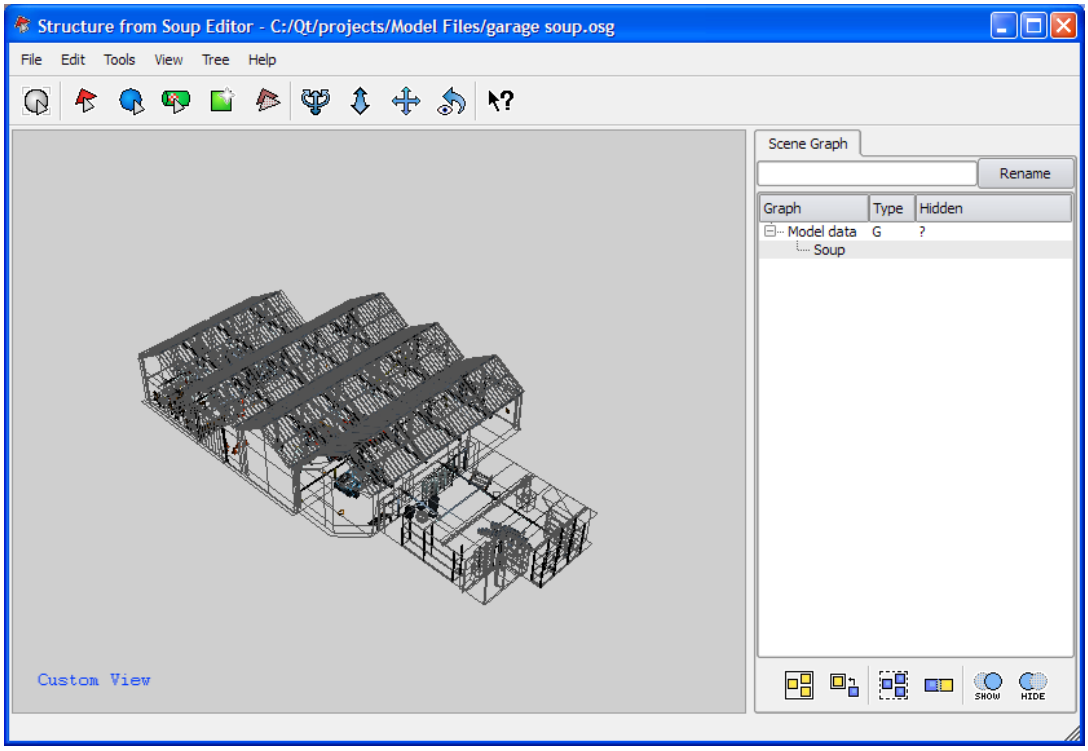
## Changing the view mode



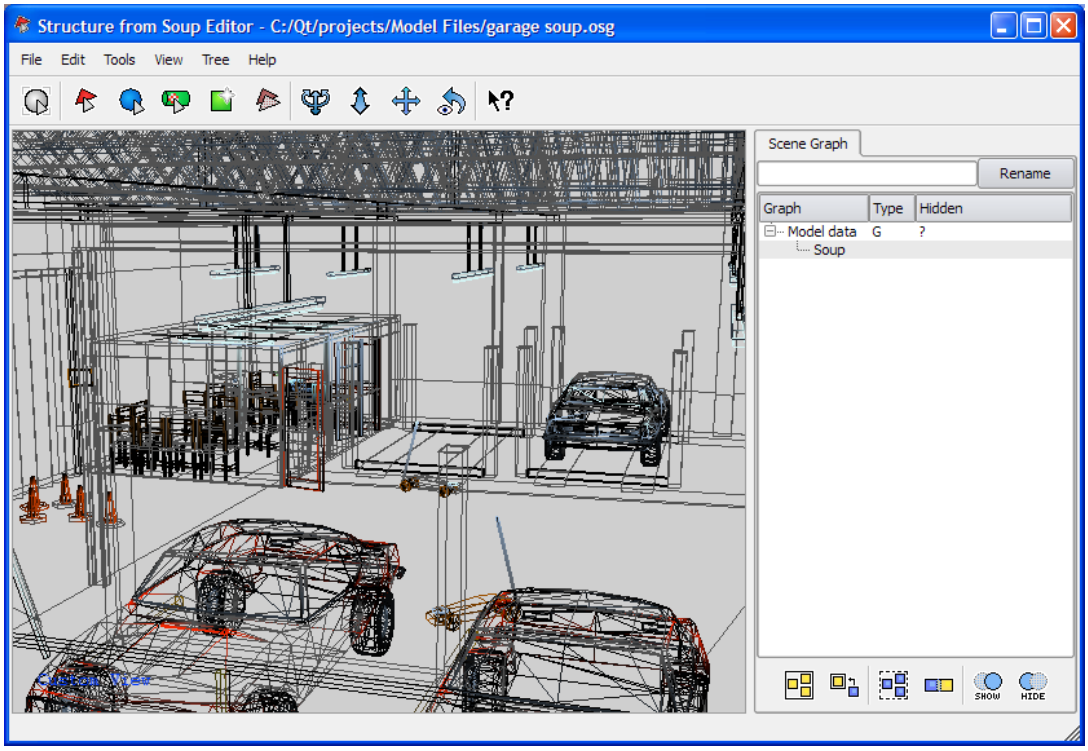

## Setting wireframe view



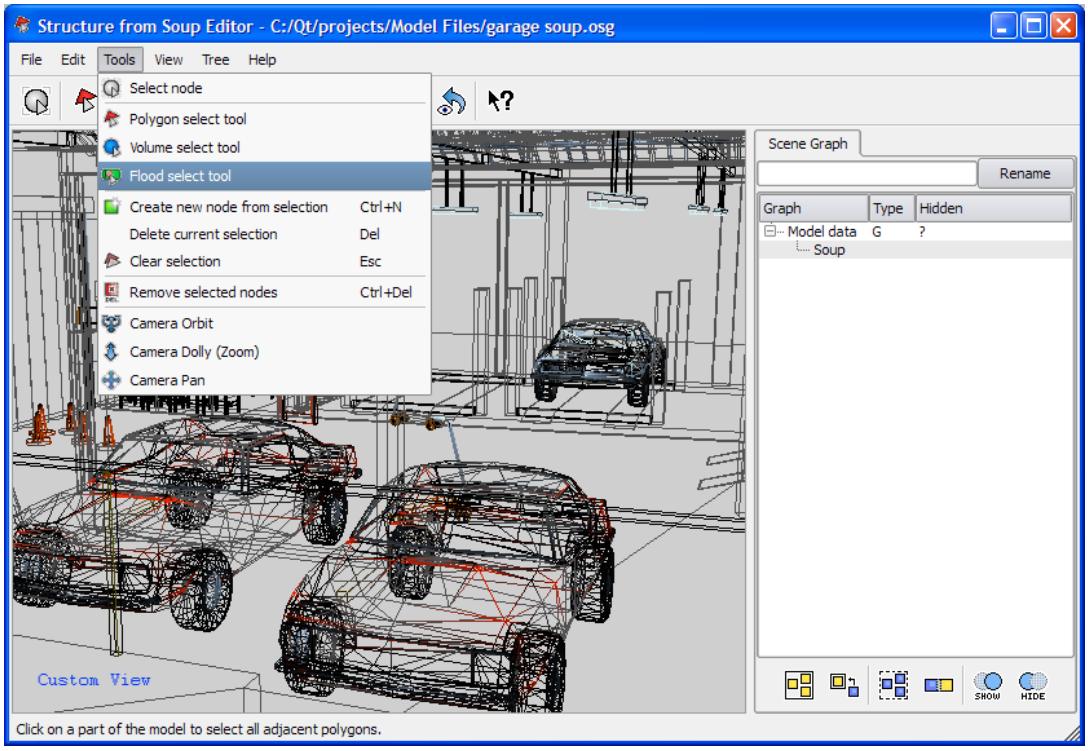
**Wireframe view of warehouse model**



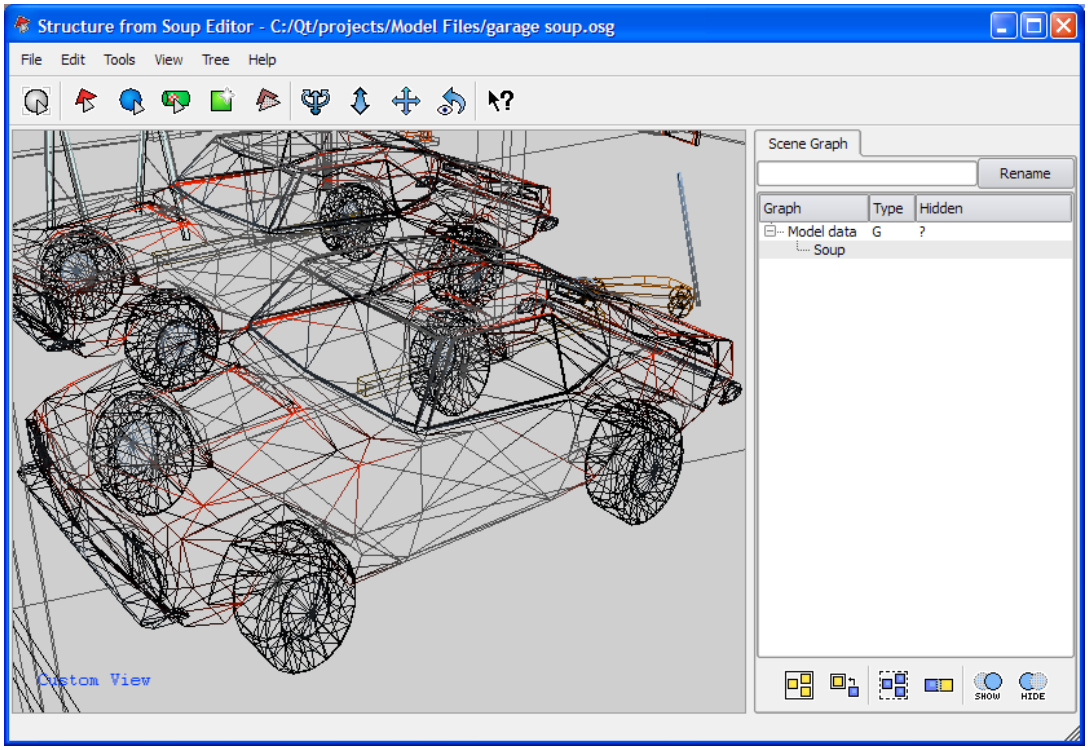

**View of warehouse from inside**



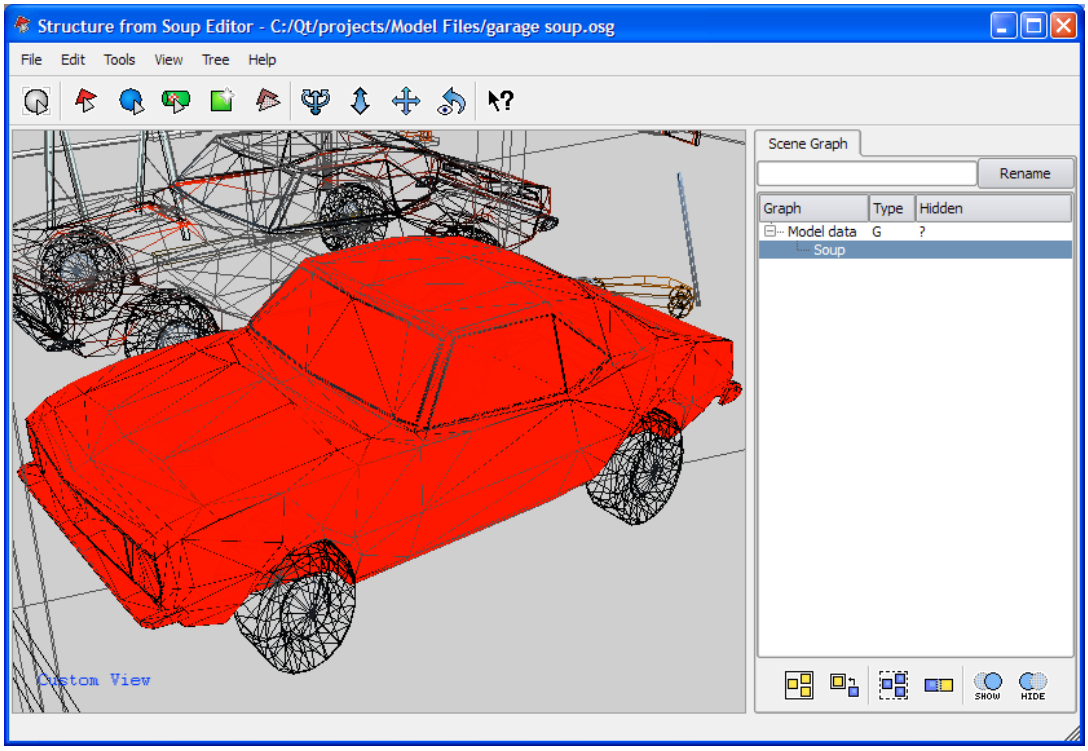
**Tools available from the tools menu**



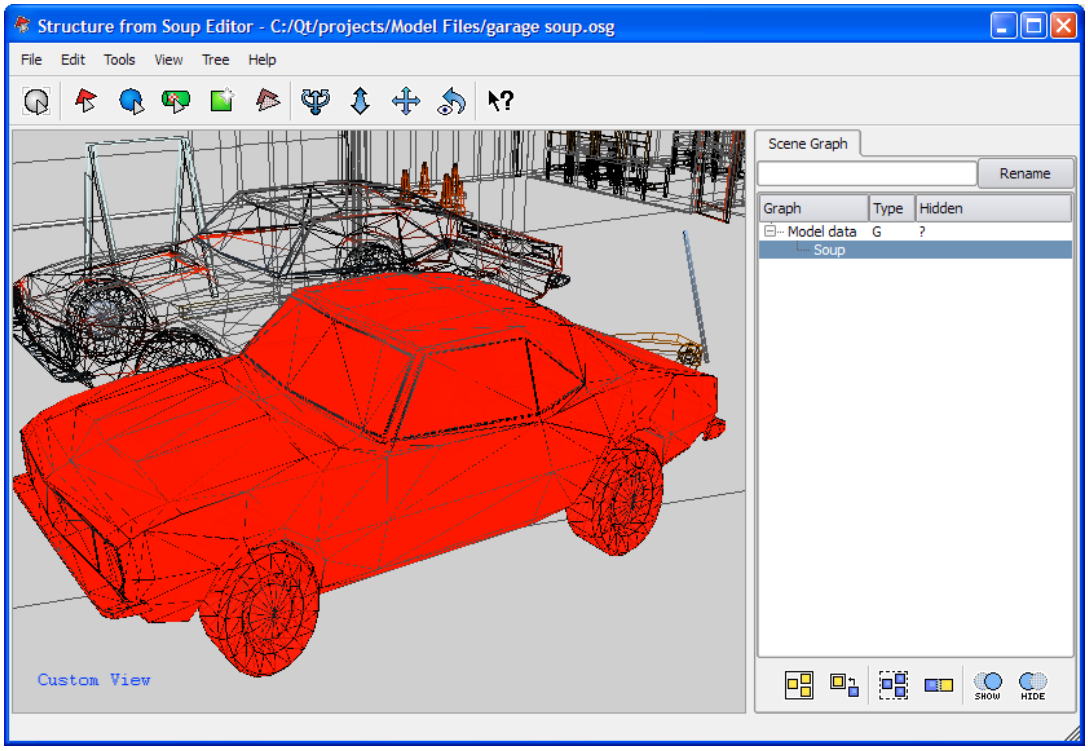

**Wireframe view of a car within polygon soup**



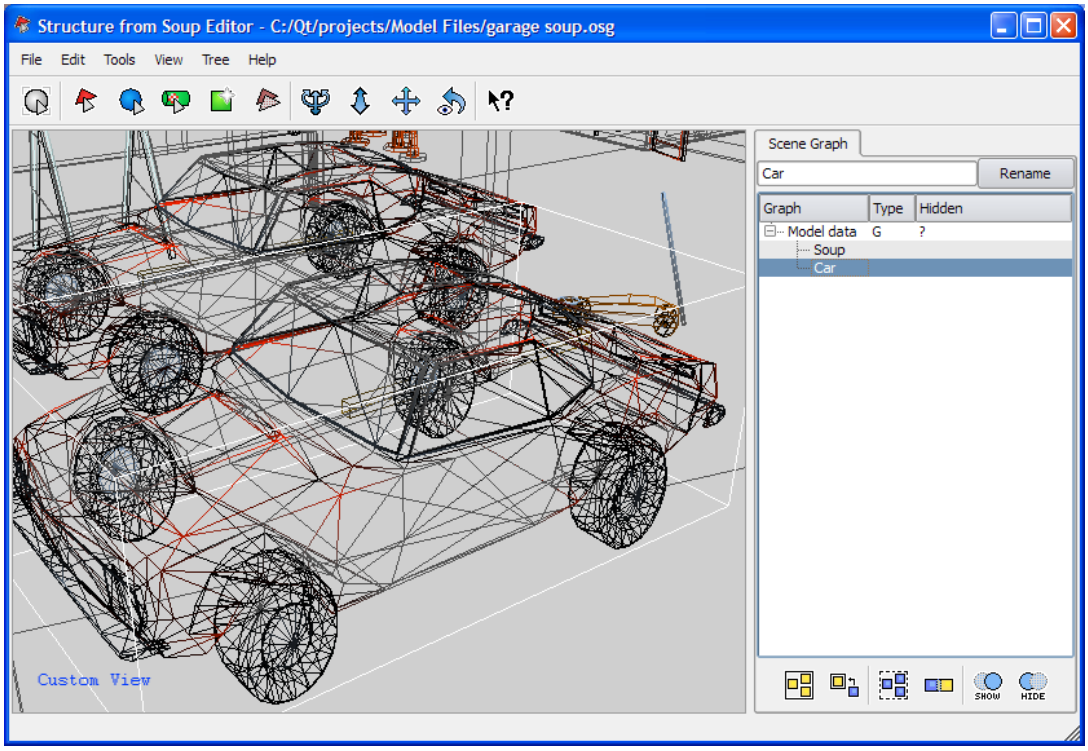
**Car chassis selected**



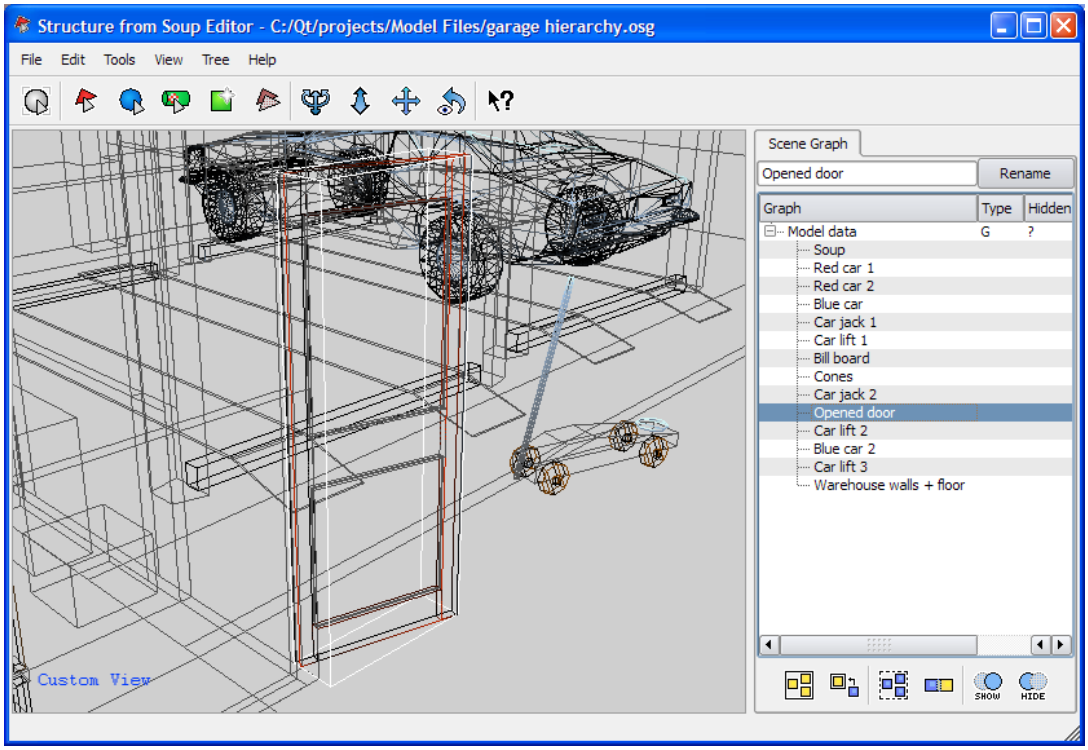

**Full car selection of chassis and wheels**



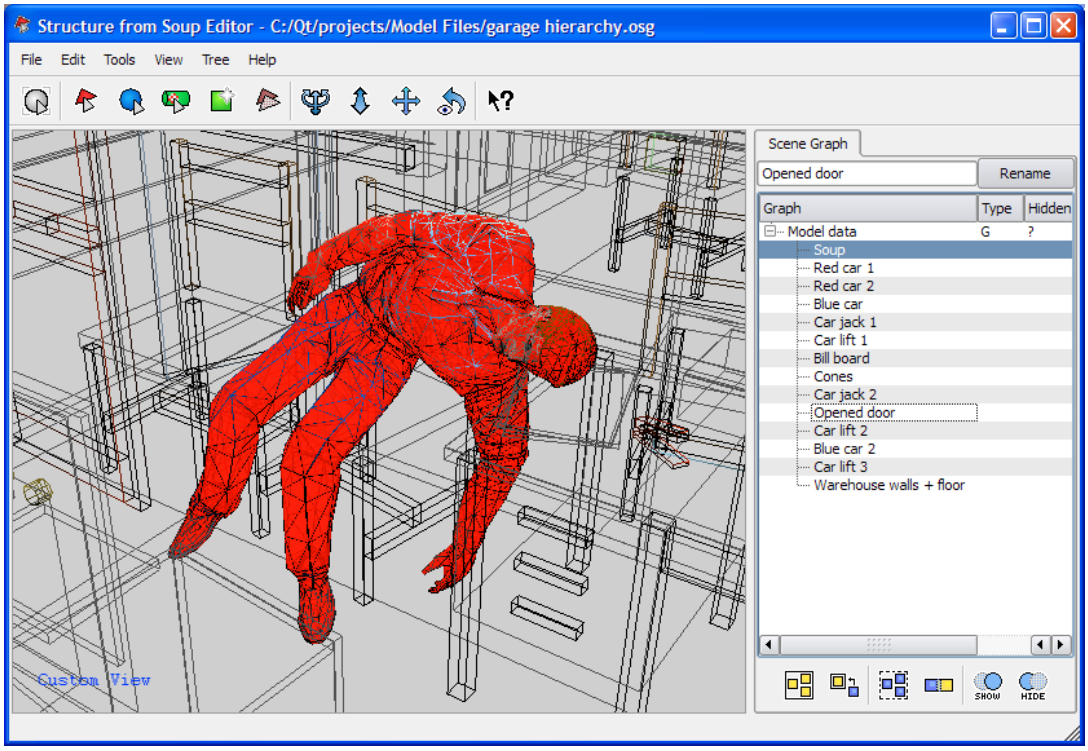
**Car selection added as name structure to scene graph**



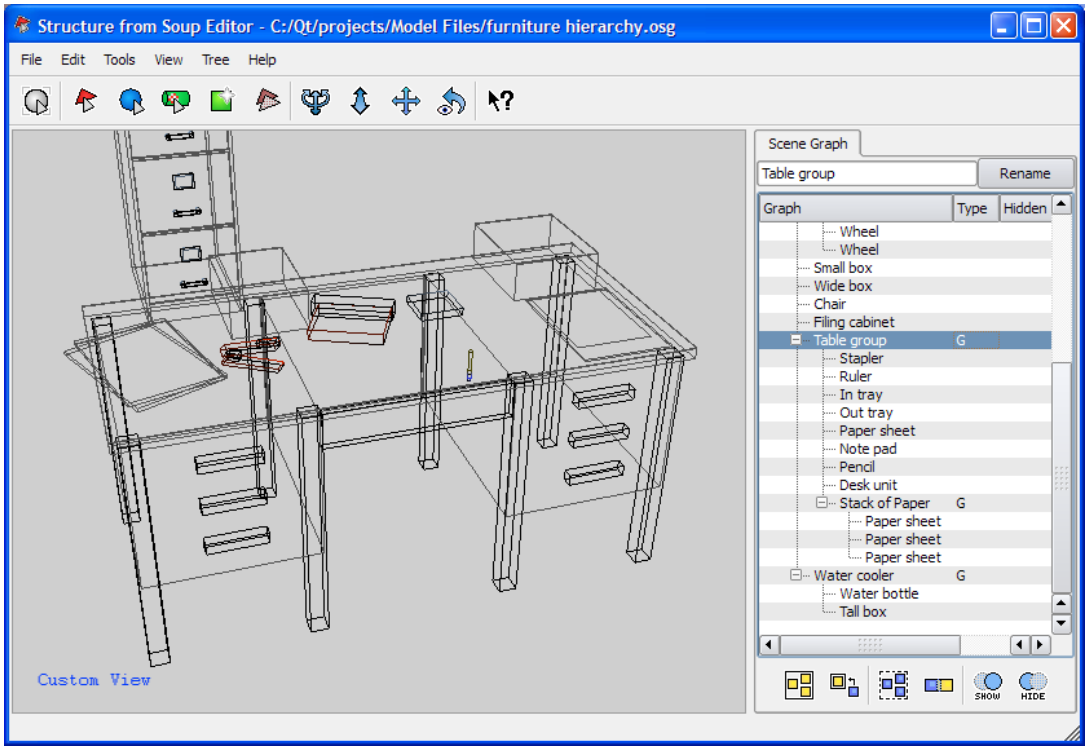

**View of "Opened Door" within scene**



**View of complex selection made using the *Flood Select* tool**

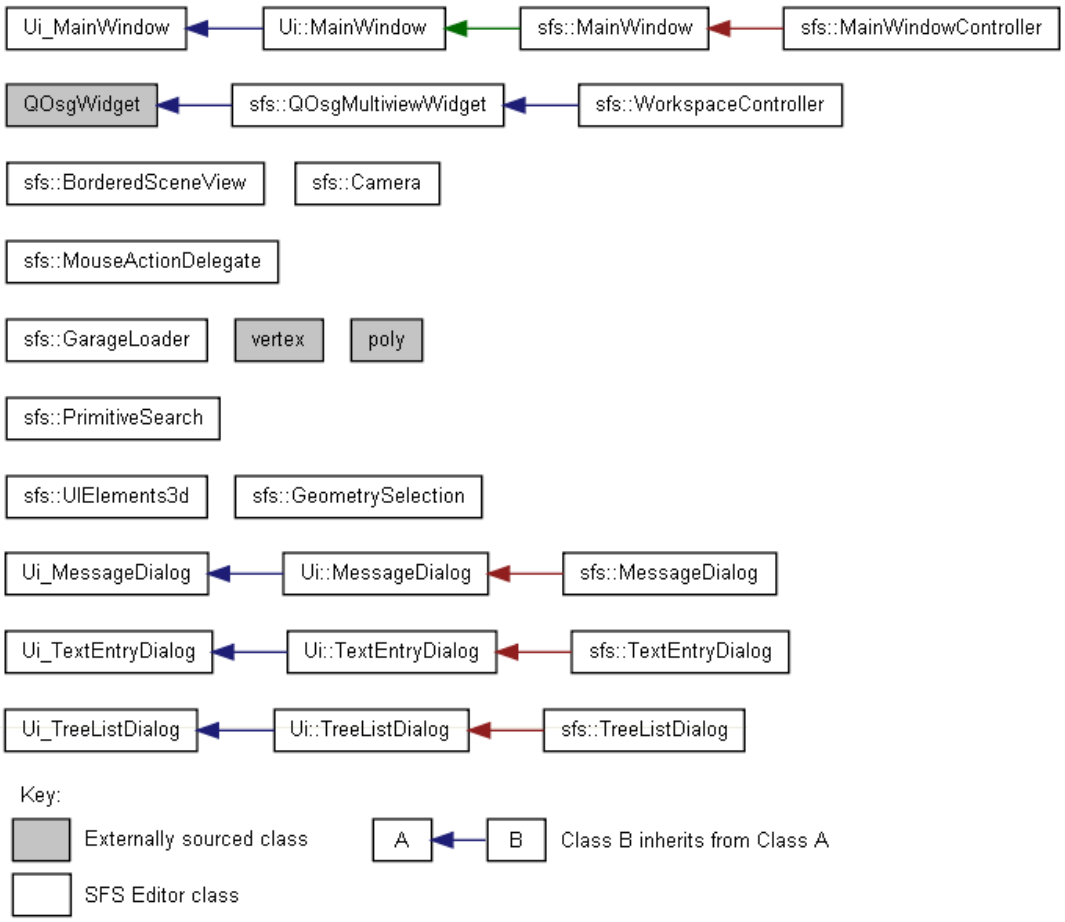



**Hierarchical structure of desk unit made from soup**



# Appendix B: Program structure of final project

**SFS Class inheritance hierarchy**

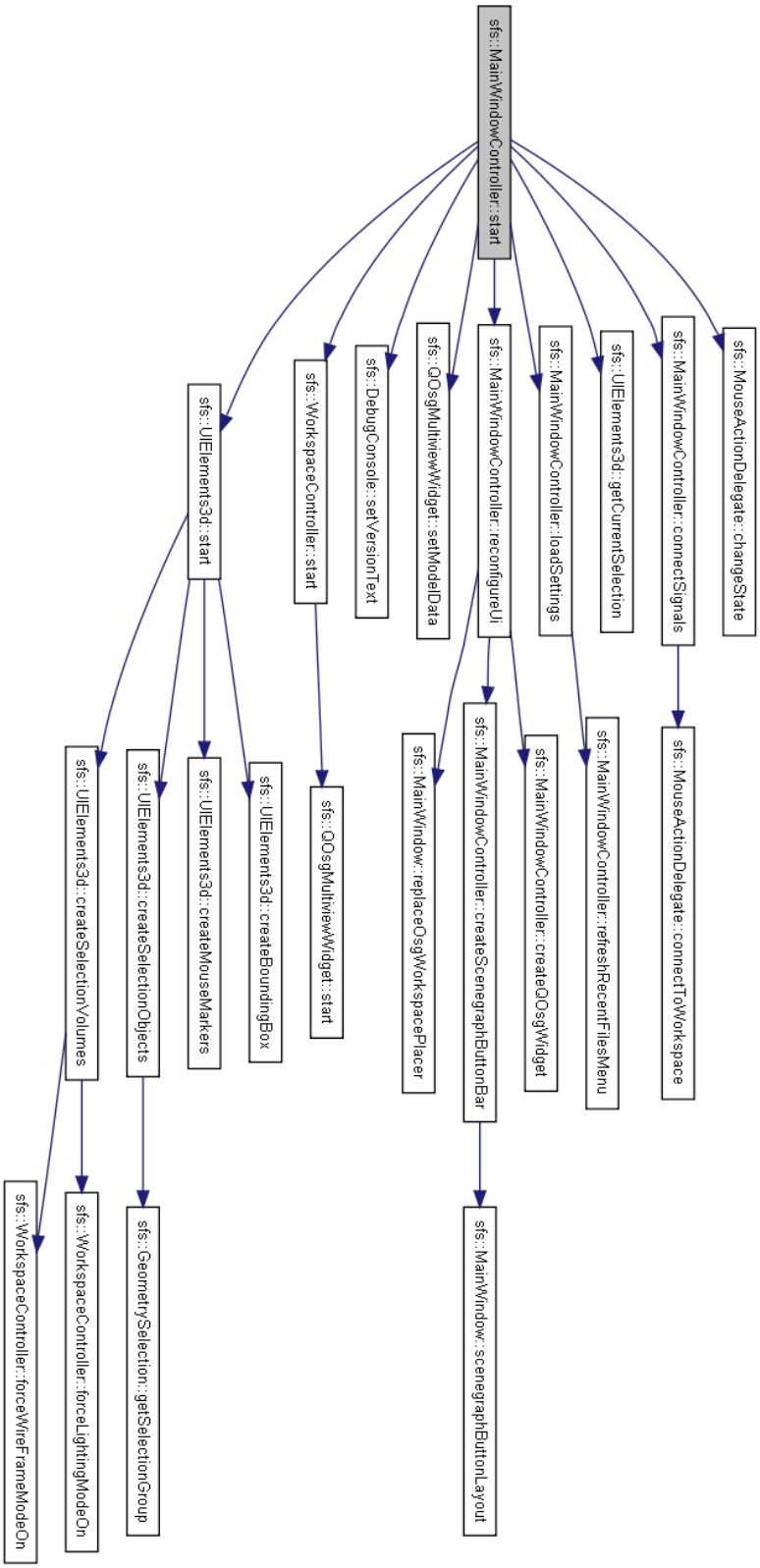



The *QOsgWidget* class was created by Antoine Hue on August 2006, based on some early work by Gazihan Alankus.

The *vertex* and *poly* classes are part of the *GarageLoader* code provided by Toby Howard at the start of the project. The *sfs::GarageLoader* class is a heavily modified version of that supplied code.

All other classes were created by John Beech for the SFS Editor project.

**Call diagram for start() method of MainWindowController**



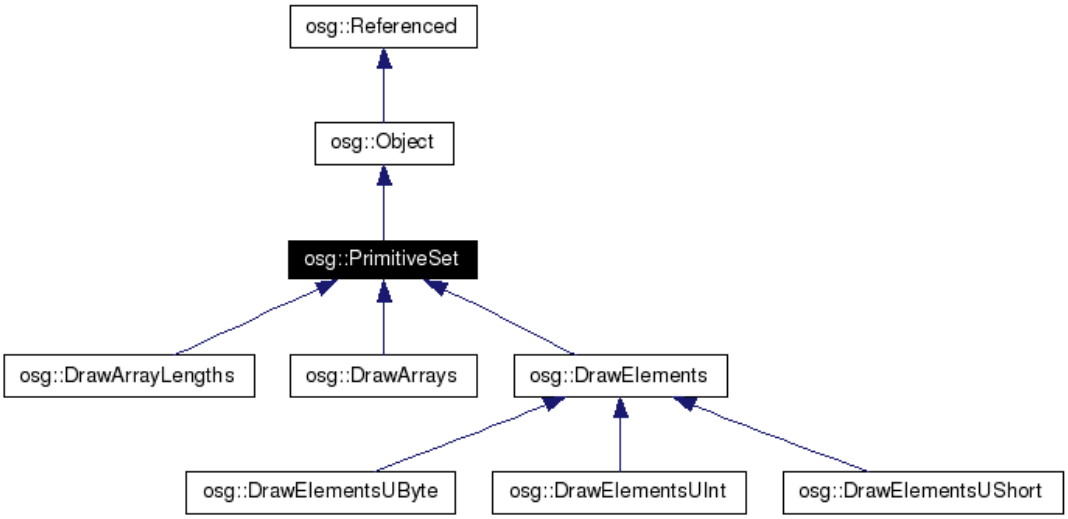

This diagram shows how the *MainWindowController* class creates and initialises all of the major objects used by the SFS Editor. The call path was generated from program code using the *Doxygen* documentation tool <http://www.doxygen.org> and *GraphViz* graph visualisation software <http://www.graphviz.org/>.

# Appendix C: Key OpenSceneGraph class hierarchies

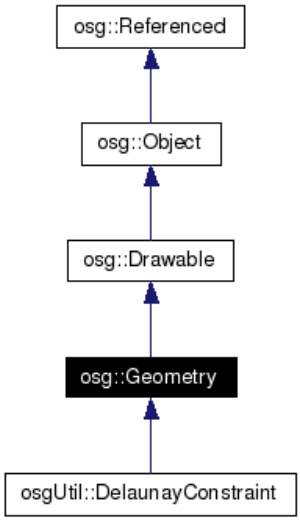
The following class hierarchies are taken from material available from OpenSceneGraph documentation. Full hierarchies are available online at:

http://www.openscenegraph.org/documentation/OpenSceneGraphReferenceDocs/

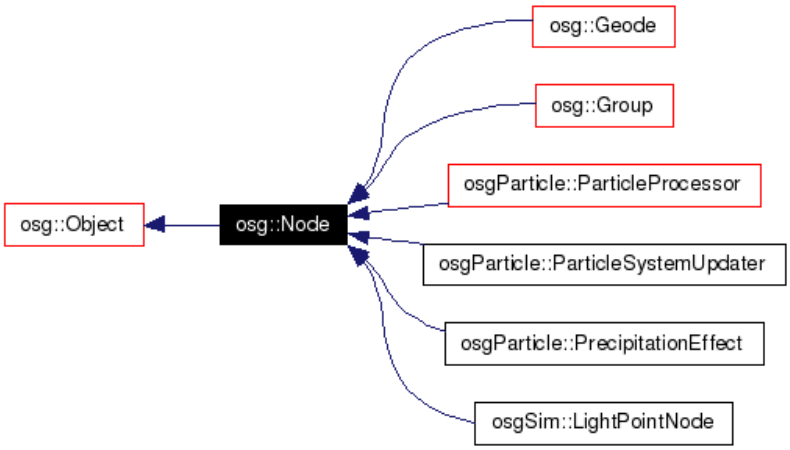
**Class hierarchy for osg::PrimitiveSet**



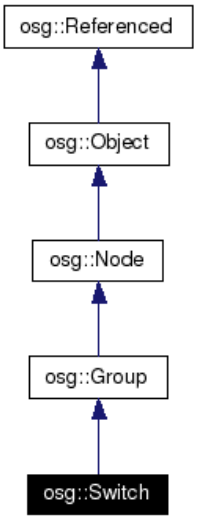

**Class hierarchy for osg::Geometry**

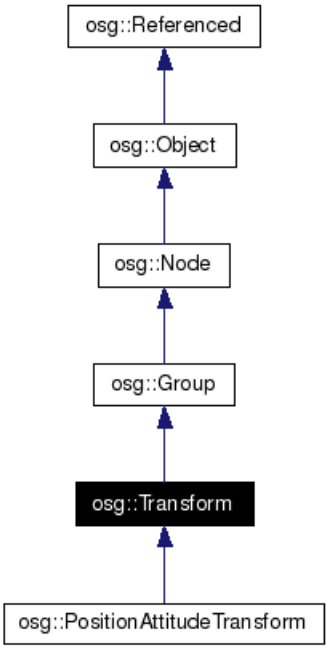


**Class hierarchy for osg::Node**




**Class hierarchy for osg::Switch**




**Class hierarchy for osg::Transform**