SIMPLIFIED COMPUTER-AIDED DESIGN SOFTWARE FOR MASS CUSTOMISATION

A thesis presented in partial fulfilment of the requirements for the degree of

Master of Engineering
in
Mechatronics

at
Massey University, Albany,
New Zealand

Travers Biddle
2014

1.0 Abstract

The advent of 3D printers has created a demand for products that are individually customized to satisfy a user's needs. Currently, there are no software tools for intuitively customising a part or object, suitable for printing with a 3D printer. Most Computer-Aided Design packages require substantial user training, and are prohibitively expensive.

There is a market gap for a product that allows a user to customise a product to their own tastes, with no knowledge required for how the product works or how its parts fit together. Instead of forcing a user to wholly create a product model, they should be given a working model of their desired product and the tools to customise it. They should be able to do so without needing any of the skills or expertise of a professional designer or engineer.

For this project, Existing CAD software was investigated to find their markets, their features and their advantages and drawbacks. No software was found that would entirely address the needs identified above.

A proof-of-concept piece of software was written that demonstrates simple modifications to single parts through the clicking and dragging of faces. The resulting customised part can then be saved in a format suitable for 3D printing.

The proof-of-concept software succeeded in being simple enough for an untrained user to be able to intuitively customize a part, and several features were identified that would need to be implemented for a release version.

2.0 Table of Contents

3.0 Table of Figures

4.0 Introduction

4.1 Inspiration for Project

Although 3D printing has been around for over 30 years, the recent advent of low-cost desktop 3D printers has created a surge of interest in the field, as well as a demand for users to be able to easily customize products to better suit their needs. However, there is a lack of suitable software tools for mass customisation to suit this market. Ideally, an individual with little or no technical ability should be able to easily design a part or product and 3D print it. Current Computer Aided Design (CAD) applications are prohibitively expensive and complicated for users inexperienced with CAD to realize a design. Most of these software packages require several months or years of training to attain the level of competency required to be able to create new, or modify existing parts.

Some types of CAD packages used in various industries are:

- Professional Packages (SolidWorks, AutoCAD, Solid Edge and many others)
- Open Source (FreeCAD, LibreCAD etc)
- Animation tools (Blender, Houdini)
- SketchUp, Rhino3D, AutoCAD 123D, TinkerCad

All of these force a user to design a complete model. To better suit the non-technically oriented user market, a user could benefit from being given a generic “template” item and customising it. They would not have to be aware of, or implement, the required features and dimensions for the item to function properly, and would not need the engineering knowledge required to design such an item entirely.

4.1.1 3D Printing Creating a Need for Customisation

A recent increase in the use and development of 3D printing has increased the demand for a simple method of creating or customising objects. Modern 3D printers build up a part by slicing up the virtual CAD model into thin slices, and then applying thin layers of material, for each slice of the model, over each other, until the part is complete. In the case of the now many available desktop 3D printers, the material is often a kind of polymer filament, which is heated to a semi-liquid glue-like state and able to solidify quickly once extruded from the nozzle that traces out each slice of the model. “Scaffolding” is also printed to support parts of an object that don’t otherwise have a base for material to be applied. This is essentially “Additive Manufacturing”, as opposed to machining methods like drilling/cutting/grinding etc

which remove matter from an original block of material. Because of this process 3D printing can produce objects not feasible with traditional machining, such as moving parts and extremely complex geometries.

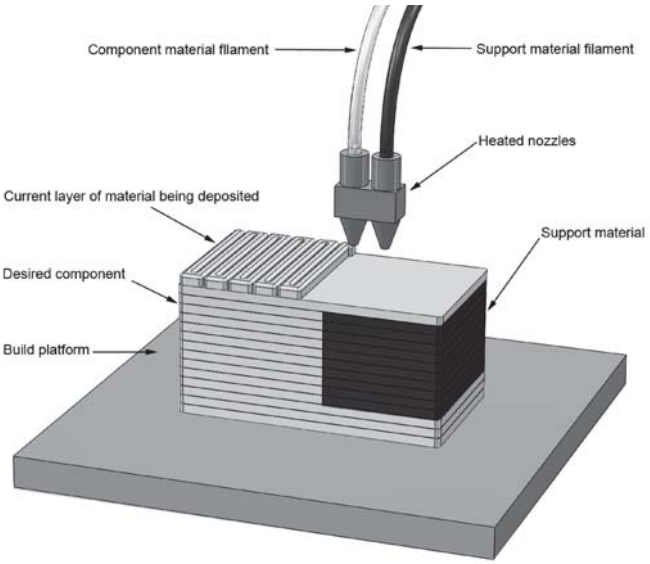


Figure 1. Fused Deposition Modelling 3D printing technology that desktop 3D printers are based on

Where 3D printers were once difficult and prohibitively expensive for all but commercial use (rapid prototyping etc), there are now 3D printers that are small and affordable enough for personal and educational use. These desktop 3D printers could conceivably be used for objects as small as a fingernail to objects as big as a basketball. For personal use, they could be used to print items such as phone covers, toys, and sunglasses frames etc. However, there is a general lack of tools to customise or personalise the virtual 3D models that are required before one can 3D print a part. A useful software application could provide a user with a template design and allow them to modify it in ways that would still result in a useful creation.

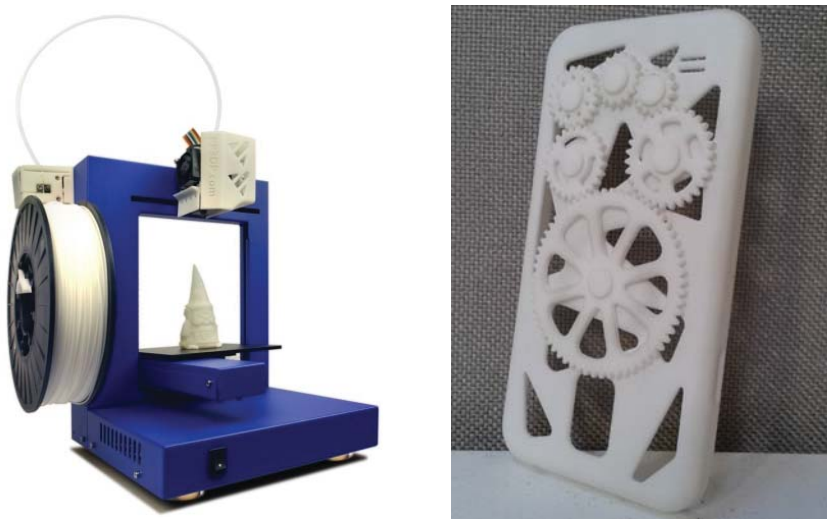


Figure 2. An Up! 3D printer for home use, and a 3D printed phone cover with freely rotating gears

4.1.2 Software Inspiration for Customisation Interface

In terms of intuitive user friendliness the software used in the gaming industry seems to have advanced significantly faster than the CAD software of the product design industry. Significant elements of user-friendly customisation can, for example, be found in the video game "SPORE", which allows a player to design and create a creatures, buildings and vehicles. The player is provided with an initial template form. They can select from a range of different features to add (eyes/ears/legs etc for creatures), and position, size, rotate and colour them to design unique creations, unforeseen by the game's creators. The interface is simple enough that children can use it.

A popular application, called Instagram, allows a user to modify their photographs with pre-built filters to give a result comparable to those of professional photographers. These photographers would use Photoshop (or other similar software) and apply a broad level of knowledge and expertise to achieve their result. Thus, users of the Instagram application can produce a result comparable to that of a professional, but without any of the knowledge or skills, which take time to acquire. Even though all users of the Instagram application are given the same conditions to work with (photo filters, border styles), they can achieve vastly different results, making it a good example of mass customisation.

Another piece of software, SketchUp, has a useful "push/pull" feature. If a face is clicked on it can be dragged to a different position, with the other faces still attached. For example, a cube could be made a rectangular prism by clicking and dragging one of its faces.

If the principles of SPORE, and other similar software, could be applied to product creation it would result in a software package in which an uninitiated user, or even a child, should be able to design and produce any item such as a toy, mobile phone covers, mugs, or sunglasses.

Figure 3. SPORE vehicle creation interface.

4.2 Objectives of this Thesis

This thesis aims to develop a proof-of-concept software package that demonstrates the customising of items in an easy-to-use manner. The software will facilitate printing the resulting creation with a 3D printer.

Rather than forcing a user to create an object entirely, the user should be presented with a fit-for-purpose design, which they can then modify and still achieve a working result.

The scope of this project will be limited to single solid parts.

The demonstration software created as part of this research project is provided on a CD-Rom appended to this thesis.

Summary

Desktop 3D printers have created a demand for the ability to easily customize parts for 3D printing. There are currently no software packages that allow a non-technical user to easily and intuitively customise a virtual object which can then be printed via a 3D printer.

The objective of this thesis is to provide a proof-of-concept demonstration of such a software package.

5.0 Literature Review

Computer-Aided Design is used to solve problems in many fields. It is commonly used to assist with product design, architecture, animation, photo-realistic rendering, game creation, personal-use 3D printing and other areas.

Most current CAD applications can be designated into one of several categories:

- Design / Surface Modelling
- Professional Engineering
- Architectural
- Animation

Each type of package has its own methodology for creation.

Many CAD applications already exist. Most of these require a user to wholly create a part or object, rather than modify or customise an existing one.

5.1 Design Modellers

The best example of a modeller for designers is Rhinoceros 3D (Rhino), a surface modeller. Objects are created by constructing surfaces, then connecting them together to give the illusion of a solid. All lines are assumed to be curves. A straight line is simply a straight curve.

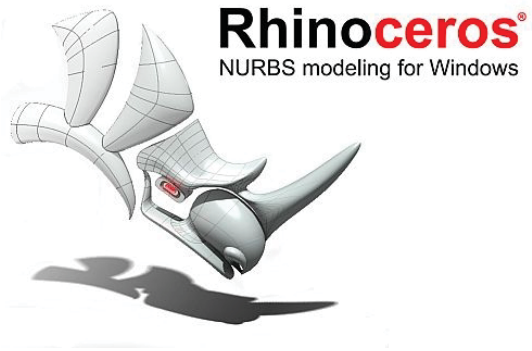


Figure 4. Rhinoceros emblem

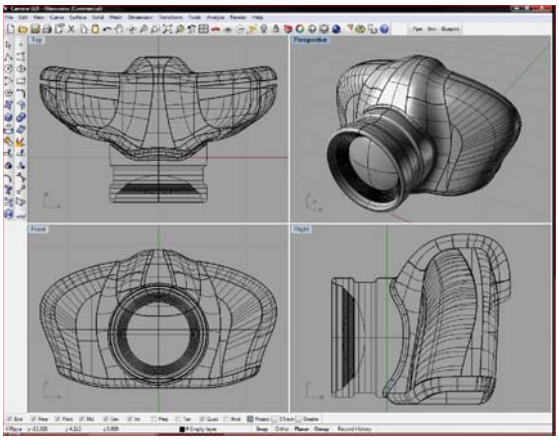


Figure 5. Sample object designed using Rhino

To create, for example, a house using the Rhino philosophy, you would draw a series of curves on screen, in the outline of a floor plan. The curves would then be connected together to form outlines and guide curves. Surfaces can then be generated from the curves and

connected together to form walls. A roof or ceiling would be constructed similarly, but in a different direction to the walls. The walls would then be extruded upwards to it.

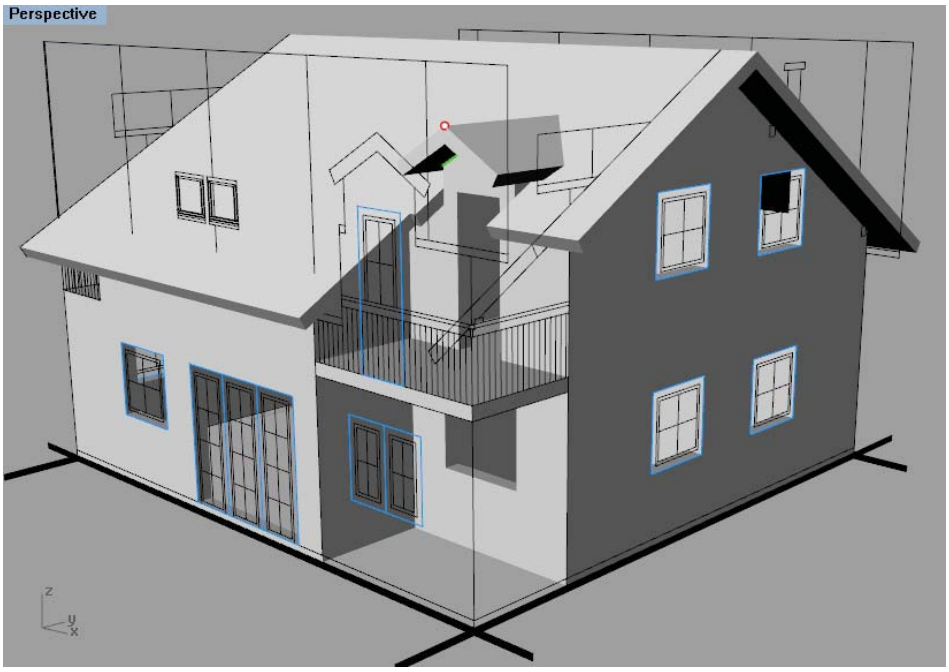


Figure 6. A House built using Rhino

Rhino has many shortcuts to make designing easier, such as an extrude feature. However, because Rhino does not have a history tree, when a series of surfaces are created with an extrude, they can only be modified as any other surface.

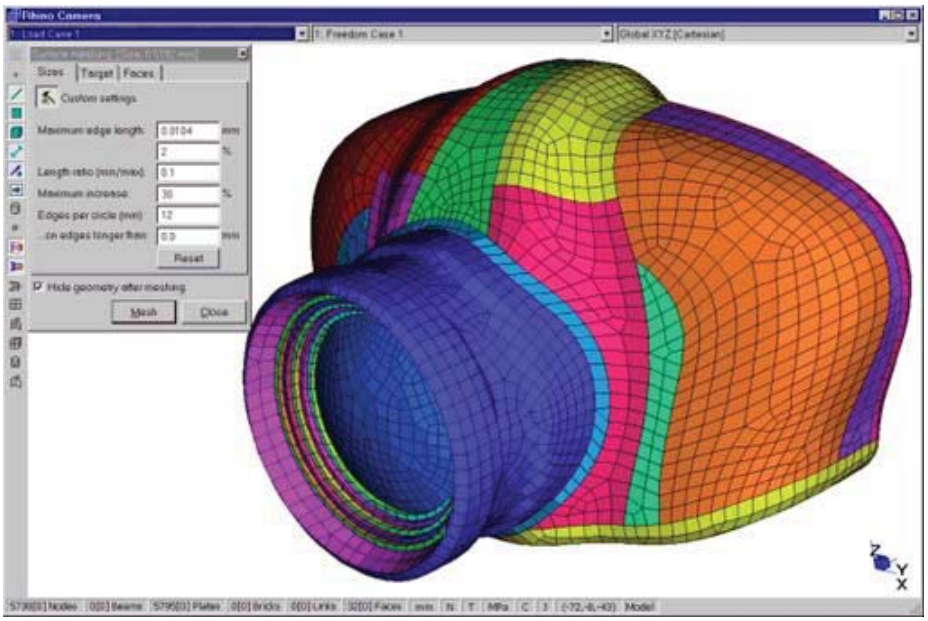


Figure 7. Different surfaces used to form a camera

Rhino includes a command bar, in which text commands can be entered. The commands must be memorised and the command bar used frequently to make the use of Rhino truly effective.

5.2 Engineering CAD Packages

Examples: SolidWorks, AutoCAD, Pro/ENGINEER, SolidEdge

Sometimes referred to as "feature-based CAD", these packages are commercial applications, geared towards large businesses. They often include advanced features such as photo rendering, force/motion study ability etc, making them unaffordable for personal users. An exception is Cubify Invent, which is very similar but without the extra features at a much lower cost.

Unlike Rhino, which is typically used just to create mock-ups or give an impression of how something will look, items designed with these packages are often used for manufacturing reference. The designs must therefore be perfect representations of the items that will be reproduced.

Also unlike Rhino, features are presented as solids, rather than connected surfaces.

Figure 8. SolidWorks emblem

To create an item, a user must know every detail of what it is and how it works. The user must know what the item's features are, the dimensions of those features and how its different parts fit together. Editing another users design is possible, but difficult when the history tree must be studied to determine where changes must be made to get the desired outcome. Making a change to a feature in the history tree can easily invalidate features further down the history as they were built upon the changed feature, called an "external reference". The larger or more complex a model becomes, the more interdependent features there will be. A tedious fixing process must be done to resolve the object.

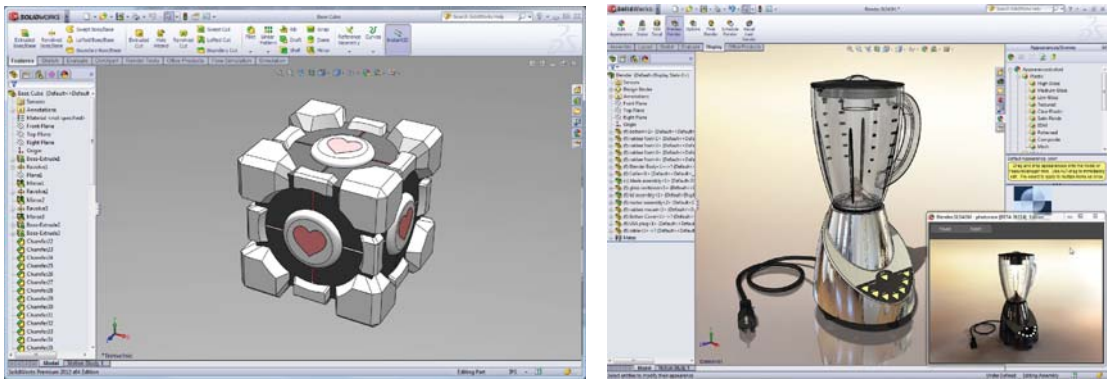


Figure 9. Sample usage of SolidWorks

The approach to designing objects with SolidWorks is to identify "features" such as forms, shapes, planes, patterns etc. To build a mock-up house with SolidWorks, the main feature would first be identified, probably an extruded box-like shape. Doors, windows, roofing etc would then be added on to this extrude using other features; revolve, loft, linear/circular pattern, extruded cut etc.

Small adjustments to lengths, widths and heights can be made easily, allowing an iterative process to get something to look right.

5.3 Architectural Packages

Examples: ArchiCAD, Revit

These are strictly architectural tools designed specifically for building/construction. They have many shortcuts for implementing windows, doors, trees, etc.

The ArchiCAD methodology is to create a floor plan from a bird's-eye view. A floor plan can be created for each floor each floor of the building. Three dimensional walls are automatically generated from the line drawings.

A "roof tool" is included to easily apply a roof to the boundary of the house. There are other tools for simply adding doors, windows and steps.

ArchiCAD allows a user to quickly and easily switch between a floor plan view and a 3D view. A user can easily see how changes in the floor plan will be reflected in the final construction.

SketchUp

SketchUp has found popularity as an architectural tool. As it is free to use it is also popular for home users.

This software makes it difficult to adjust dimensions once they've already been defined. Its most useful feature is its push/pull tool, which allows a user to turn a 2D drawing into a 3D form by dragging it.

The typical SketchUp process is to draw 2D drawings that can then be "pulled" into a three-dimensional solid using the Push/Pull tool. Drawings can be made on the walls of these solids, which can be Pushed/Pulled to create an indent or protrusion in the solid.

Figure 10. A House designed using SketchUp

To create a house a floor plan would be drawn and pulled upwards to create walls. A roof outline can be drawn on a different plane and dragged across the length of the house to create the roof.

5.4 Animation Modelling

Examples: Houdini, Blender, Maya

Models built with these modellers are represented by a mesh, rather than geometric relations. This makes it difficult to make logical geometric changes to a design, especially where curves or circles are concerned.

Animation modellers do not ensure validity of the model for realisation.

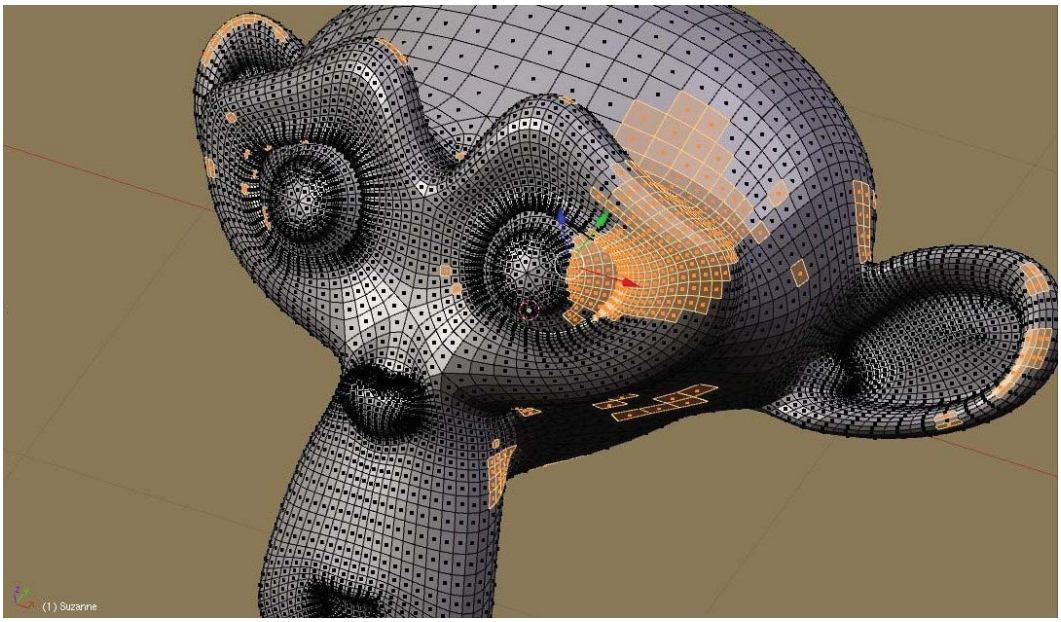


Figure 11. Monkey face created with Blender

Different packages have different methods for creation, some acting like surface modellers, others requiring a more parametric approach. The end result is represented by some form of mesh. Blender is the only one capable of saving models in a *.stl format.

5.5 Mass Customisation Software

A new product called Odo, by the company Digital Forming, aims to give users the ability to customise pre-made parts. It does so by providing different options for features such as handles, surfaces, text, legs, lids etc. These options have to be created by the original designer of the part. This allows only a finite number of possible combinations and prevents a user from creating their own design for a feature.

Odo is a web-based editor. Customised designs can be saved to the Digital Forming servers. They can then be printed by Digital Forming and mailed to the customer at a price. The option for a customer to print their design themselves is not available.

A web-based application, Sculpteo, offers a range of customisation services for iPhone cases, keyrings, digital pottery, toys and other items. The Sculpteo business model is centred on 3D printing user-uploaded designs and mailing the physical product to the user. The items that can be customised are limited to what Sculpteo develops interfaces for (iPhone case etc).

Another web app, Cubify, has a digital sculpting tool. This lets a user 'sculpt' a digital object as though it were clay by using the mouse to push or pull on the surface of a digital blob.

5.6 Recent Developments in Direct Modelling

Siemens' Synchronous Technology combines both history-based modelling with direct editing so a user/designer does not have to choose which approach to take. A history tree is retained and immediately re-calculated when a feature is adjusted through click-and-drag. This technology was combined with Siemens' Solid Edge in 2008.

Autodesk has a similar project called Fusion that lets a user choose whether to design using a feature-based method or a direct modelling method. This has been incorporated into Autodesk Inventor, an application for mechanical design.

As touch screen technology becomes more important there have been developments in CAD to make use of it. Another Cubify product, Cubify Draw, is an iPad/iPhone app where a user can draw a shape outline with their finger; the app then extrudes the shape into a three-dimensional solid.

Figure 12. Cubify Draw allow a user to easily create an extruded design using touch input

Summary

Most CAD editors in the market fall into one of three groups: surface/design modelling, professional engineering and animation.

Surface modelling involves forming surfaces using guide curves and control points. The surfaces are connected together to give the illusion of a solid.

Professional packages like SolidWorks take a parametric feature-based approach to creation.

Animation modellers have various methods of creation that result in a mesh.

There are several web-based applications that offer some kind of customisation. Their capabilities are rather limited. They focus on mailing a 3D-printed version of the customised object to the customer.

Siemens and Autodesk have made attempts at integrating direct modelling into their products. Since their release, developments have been infrequent.

6.0 Approaches to Solid Modelling

6.1 Modelling Methods

To represent a solid object, it needs to be stored somehow in computer memory. There are many ways to do this, but most can be grouped into one of the three core approaches identified here.

6.1.1 Cell Decomposition

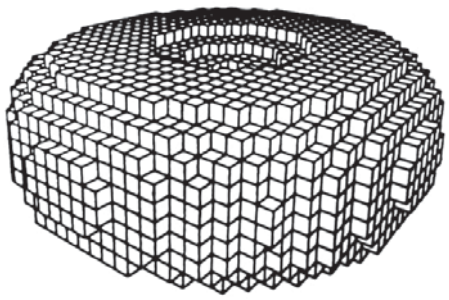


Figure 13. A rounded object divided into cells [9]

A solid object can be represented by dividing its volume into smaller volumes or 'cells'. Cellular decomposition produces an approximate representation of the object because some cells will be partly in the object grid and others will be discarded. Cells need not be cubic or identical in shape [1].

From Boundary Representation Modelling Techniques [14]:
The straightforward cell decomposition method is to divide object space into unit-sized elements (cubes or spheres) and to represent shapes as collections of these. Because of practical limitations in computers, it is not possible to divide space into infinitely small elements, so cell decomposition methods have to be approximate representations. A refinement of the technique using variable sized elements is called the octree technique. Using this method, objects are represented as trees of elements. Each element is classed as being full of material, partially filled or empty. Partially filled elements are subdivided, and the smaller elements are again classified as filled, partially filled, or empty. The elements are always subdivided into eight smaller elements, hence the name octree.

This approach can make it impossible to accurately represent some shapes, particularly where curves are concerned. It is useful in the area of Finite Element Analysis to measure effects of stresses.

6.1.2 Constructive Solid Geometry

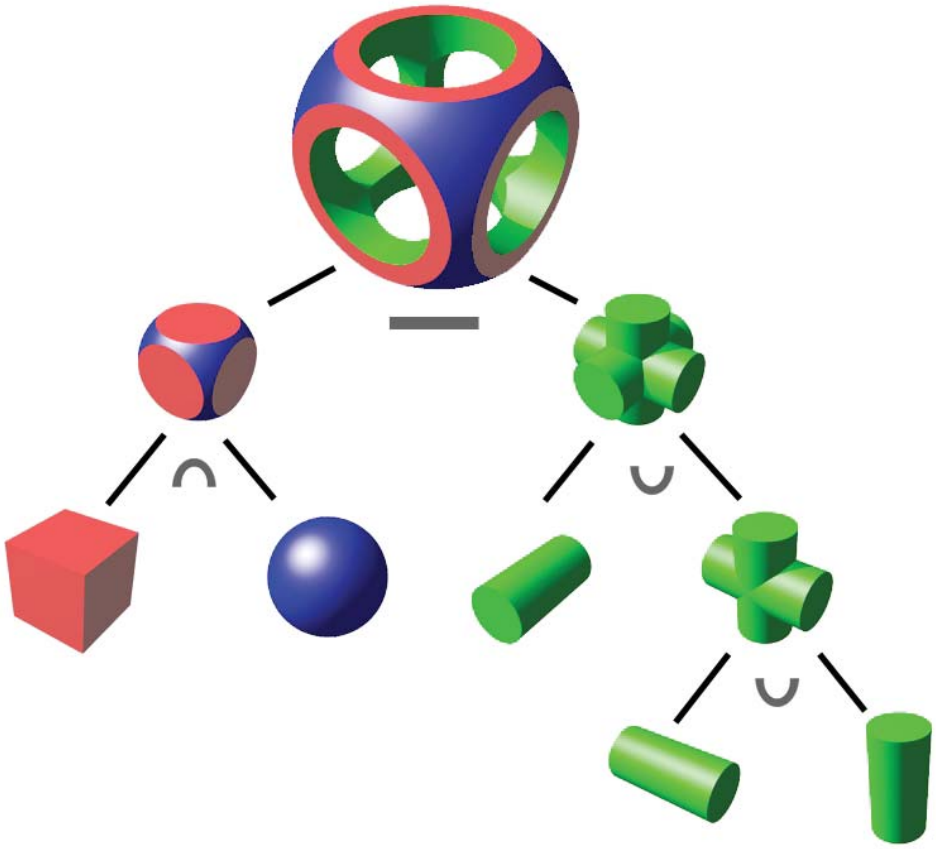


Figure 14. Demo of constructive solid geometry tree. Created in POV-Ray. [16]

Constructive Solid Geometry is a scheme for modelling rigid solid objects as set-theoretical compositions of primitive solid "building blocks" (Voelcker & Requicha, 1977). The constructive representation scheme relying on closed regular primitives, rigid body motions, and the regularized set operations is called Constructive Solid Geometry (Voelcker & Requicha, 1977).

The primitives can such three-dimensional shapes as prisms, spheres, cylinders, pyramids etc.

The boolean operators can be union, intersection or difference.

The libraries OpenCSG [24] and GTS [18] can be used to facilitate a CSG design program.

The attractive properties of CSG include conciseness, guaranteed validity (by definition), computationally convenient boolean algebraic properties, and natural control of the solid's shape in terms of high-level parameters defining the solid primitives and their positions and orientations, the relatively simple data structures and the elegant recursive algorithms [13].

All constructive approaches are limited by their ability to compute and manage topological neighbourhood information of the points in the represented sets. A related significant drawback of CSG and all other implicit representations is the lack of explicit representation and parameterization of the solid's interior and particularly its boundary [13].

6.1.3 Boundary Representation

Boundary representation is a method of representing an object in terms of the surfaces separating the outside of the object from the inside [3]. The surfaces are defined as Faces. Faces are bounded by sets of edges, which are portions of curves lying on the surface of the faces on either side of the edge. The points where several faces meet are called vertices [14].

Boundary representation allows much more arbitrarily-shaped models than CSG or other methods. It is also easier to implement useful operations, extrusion, chamfer, shelling etc. However, there is no guarantee of validity, so checks must be performed on the data to ensure a valid physical model is represented.

The elements for a boundary representation modeller were summarised by Stroud [14]:

• VERTEX: A vertex is a node of the data structure, lying at a point in space.

• EDGE: An edge is a segment of a curve, running between two vertices. In an Eulerian model edges lie in two loops, or possibly occur twice in the same loop.

• FACE: Faces are portions of surfaces. Faces are bounded by loops, which are ordered sets of edges.

• LOOP: In its simplest form, the model data structure consists only of faces, edges, and vertices. However, this does not allow multiply connected faces, where there is an outer boundary and one or more inner boundaries. To allow for this kind of model, the edges bounding a face are divided into closed circuits of edges, called Loops.

…

• SHELL: Each closed set of faces in the object forms a Shell. It is useful to represent these shells explicitly in some way in a model, rather than having to retrieve the information by traversing a face set to see if it is closed.

…

• POINT: A zero-dimensional entity, a position in 3D Euclidean space defining the position of a vertex.

• CURVE: A one-dimensional entity defining the shape of an edge.

• SURFACE: A two-dimensional entity defining the shape of a face.

…

A practical application of this can take the form:
A list of coincident vertices.
A list of Edges
A list of adjacent faces

Each Edge stored as:
The start and end vertices.
A list of connecting faces
A list of adjacent Edges

Each vertex stored as:
A list of adjacent vertices.
A list of connected Edges
A list of adjacent faces

As shown, the model has plenty of redundancy, making it easier to navigate the data structures. This comes at the cost of ensuring that when a topological structure (vertex, edge, face, shell) is added or removed, the rest of the model is updated accordingly.

Holes

To represent a hole in a face, the face can be represented as:
A list of Vertex lists (V1, V2, V3, V4), (V5, V6 V7), (V8, V9, V10, V11)
A list of Edge lists (E1, E2, E3, E4), (E5, E6, E7), (E8, E9, E10, E11)

The first vertex/edge list represents outermost boundary and all subsequent vertex/edge lists represent the holes or pockets inside.
For convenience and compatibility with rendering agents such as OpenGL, the outer loop can be listed in clockwise order, while inner loops (the holes) can be listed in counter-clockwise order.

From here, a triangulation/tessellation algorithm is required to divide the face into triangles or other simple shapes that can be rendered with a graphics API.

Triangulation is a very complex topic with limited choices of libraries and functions to perform it. One is the now deprecated OpenGL function gluTesselator.

Using gluTesselator in OpenGL, the following figure is divided like so:

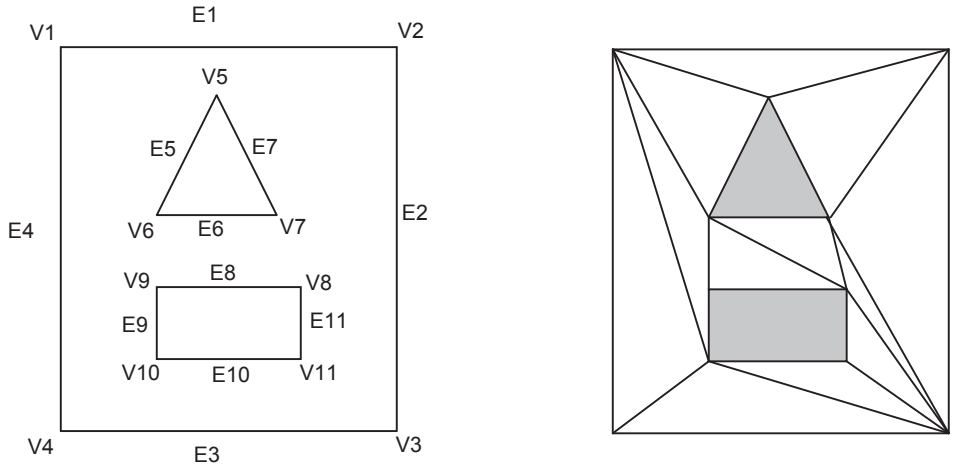


Figure 15. Example triangulation. An arbitrary shape triangulated with gluTesselator

6.1.3.1 Validity

For a boundary represented object to be topologically and geometrically valid it must:

- Satisfy the Euler-Poincaré formula
- The set of faces totally encloses, leaving no missing parts
- Faces do not intersect each other except at common vertices or edges
- The face boundary loops do not intersect themselves
- Be two manifold (a surface that does not touch or intersect itself)

Euler-Poincaré Formula :

$$V - E + F - (L - F) - 2(S - G) = 0 \qquad (6.1)$$

V: The number of vertices

E: The number of edges

F: The number of faces

G: The number of holes that penetrate the solid, usually referred to as genus in topology

S: The number of shells. A shell is an internal void of a solid. A shell is bounded by a 2-manifold surface, which can have its own genus value. Note that the solid itself is counted as a shell. Therefore, the value for S is at least 1.
L: The number of loops. All outer and inner loops of faces are counted.

An Euler operation is a combination of changes to v, e, f, h, b, and g, such that the Euler-Poincaré formula is still satisfied.

e.g. If V is increased by 1, another term in the Euler-Poincare formula must be increased or decreased accordingly, such as E increasing by 1. The Euler operation would be: make a vertex and edge. This ensures the expression resolves and the model is topologically valid at every step.

The Euler operations do not ensure geometric validity.

Geometric Validity

According to Stroud [14]:
Faces may have one or more outer loops of bounding edges and any number of inner loops of edges. The edges bounding a face may not intersect or touch each other except at the vertices. The implicit face orientation must agree with the surface orientation. A single model may have a single outer shell and any number of inner shells, or cavities. The object may not be self-intersecting, although parts of it may be coincident or 'non-manifold' parts. The curves and surfaces of the model must be curvature continuous in the portion used in an edge or face.

Winged-Edge Data Structure

The winged-edge data structure stores the two points that define it, the adjacent faces, and the preceding and succeeding edges for both left and right traversal (each of the two faces) [2].

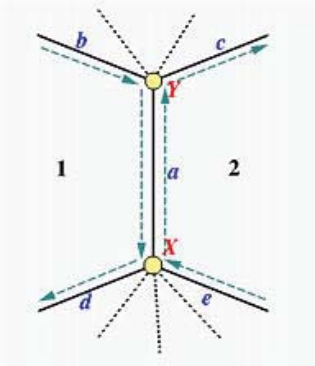


Figure 16. Winged-Edge data structure. [20]

Half-Edge Data Structure

A half-edge data structure stores only one "half" of an edge. The structure contains a pointer to the other "half". If one half has start vertex A and end vertex B, its pair, the other half, will have start vertex B and end vertex A. This makes traversal easier for defining edge loops needed for creating faces. In theory, it also makes the structure a fixed size and operations can be performed in constant time.

6.2 Curves

The dominant mathematical technique of representing curves in surface and solid modellers is the use of NURBS (Non-Uniform Rational Basis Spline) curves. NURBS are a type of B-Spline. B-Splines are in turn based on Bezier curves.

NURBS equations are used to define curved surfaces.

6.2.1 Bézier curves

Bézier curves are a form of "parametric" function.
The formula can be expressed as:

$$B(t) = \sum_{i=0}^{n} \binom{n}{i} (1-t)^{n-i} t^{i} P_i \qquad (6.2)$$

$P_0 \ldots P_n$: control points, usually two or three-dimensional.
n: The degree of curve (linear, quadratic, cubic). n is equal to the number of control points minus one.

6.2.2 B-Splines (Basis Spline)

A B-Spline is a generalised version of a Bezier curve where the order of the curve does not depend on the number of control points. The order can be chosen to be linear, quadratic, cubic etc.

The formula for a B-spline is:

$$C(u) = \sum_{i=0}^{n} P_i N_{i,p}(u) \quad (6.3)$$

where:

$$N_{i,0}(u) = \begin{cases} 1 & \text{if } t_i \le u < t_{i+1} \text{ and } t_i < t_{i+1} \\ 0 & \text{otherwise} \end{cases} \quad (6.4)$$

$$N_{i,j}(u) = \frac{u - t_i}{t_{i+j} - t_i} N_{i,j-1}(t) + \frac{t_{i+j+1} - u}{t_{i+j+1} - t_{i+1}} N_{i+1,j-1}(u) \quad (6.5)$$

$$T = \{t_0, t_1 \ldots t_m\} \quad (6.6)$$

$$p \equiv m - (n-1) \quad (6.7)$$

$$P = \{P_0 \ldots P_n\} \quad (6.8)$$

B-splines have the concept of "knots" which exist to join B-splines together. Therefore, a B-spline can be made up from a series of B-splines. When the knots values (T) are equidistant the B-spline is said to be uniform. Otherwise, non-uniform. If two knots t_j are identical, any resulting indeterminate forms 0/0 are deemed to be 0.

6.2.3 NURBS (Non-Uniform Rational B-Splines)

NURBS are a generalisation of B-splines. "Non-Uniform" means the knot values can be arbitrarily spaced. "Rational" means the control points can be assigned different weight factors.

$$C(u) = \frac{\sum_{i=0}^{n} N_{i,p}(u) w_i P_i}{\sum_{i=0}^{n} N_{i,p}(u) w_i} \quad (6.9)$$

p: order (degree +1)

Here, n is the number of control points $P_i w_i$ are the corresponding weights.

The weighting components can all be set to 1 to give each point an equal weight.

NURBS surface equation:

$$S(u,v) = \frac{\sum_{i=0}^{m}\sum_{j=0}^{n} N_{i,p}(u)N_{j,q}(v)w_{i,j}P_{i,j}}{\sum_{i=0}^{m}\sum_{j=0}^{n} N_{i,p}(u)N_{j,q}(v)w_{i,j}} \qquad (6.10)$$

6.3 Datums

Datums are theoretically perfect points, lines and planes. They establish the origin from which the location or geometric characteristics of features of a part are established [4].

Datums are used to define dimensions of features (length, width, angle, distance).

6.4 History-Based modelling and Direct Editing

There are two foremost ways of presenting model editing, history-based modelling and direct editing.

Typical professional CAD packages use a history-based method where features must be designed in order.

Siemens has a recent (2013) version of Direct Modelling they refer to as “Synchronous Technology” [27].

6.5 User Interface (UI), Usability and Aiding Inexperienced Users

Unfortunately, no research directly relating to ease-of-use of Computer-Aided Design systems could be found. Together, Hamade, Artail & Jaber have done some interesting research on the area [6][7][8] but have not addressed which interface features lead to ease-of-use.

The unmeasured general consensus is that direct editing/modelling is easier and more intuitive, especially for inexperienced users. This is the type of design that video games, such as SPORE, inevitably use.

Editing a History-Based model requires the user to become familiar with the history of the feature tree. Depending on the feature configuration, a desired change may require re-designing a number of features 'downstream' to keep the model valid.

R. B. Miller [10] identified an ideal response time to a control activation as immediate, no more than 0.1 seconds.

6.6 Existing Modelling Kernels

6.6.1 OpenCASCADE

Open CASCADE [23] Technology is a software development platform freely available in open source. It includes C++ components for 3D surface and solid modelling, visualization, data exchange and rapid application development.

Open CASCADE Technology can be best applied in development of specialized CAD/CAM/CAE applications. The typical applications built with help of OCCT are pre- and post-processors for finite element analysis software, CNC/CMM path generators, numerical simulation programs, etc.

Open CASCADE Technology (former CAS.CADE) exists from the mid 1990-s and has already been used by numerous commercial clients belonging to different domains − from software edition to heavy industry.

Open CASCADE is free to use.

A package of e-learning courses consisting of:

- Fundamentals (includes Geometric modelling + Advanced modelling + Visualization)
- OCAF (Open CASCADE Application Framework)
- Shape Healing

cost €1,920.00 at the time of application (2012).

Experience with and documentation of Open CASCADE indicated that it expected a very history-driven approach to modelling. To make a solid model editing program using it, a model would have to be totally rebuilt for any change made to any of its geometry.

The Open CASCADE documentation includes very little about visualization [25] (Version 6.5.3 of Open CASCADE was used for this research).

Open CASCADE uses OpenGL to do its graphics rendering.

6.6.2 Parasolid

Parasolid is the Siemens-owned kernel used by SolidWorks and Solid Edge [28].

The Parasolid software can not be obtained without a license. Siemens does not offer evaluation versions of Parasolid. They do offer campus-wide licenses throughout a university for academic purposes. A grant is possible for this purpose. A license for commercial use is prohibitively expensive. Attempts to obtain this grant and software were unsuccessful.

6.6.3 ACIS

Like Parasolid, the ACIS libraries are not obtainable without a license. Owned by the Spatial Corporation, ACIS is the kernel used by AutoCAD. An evaluation version of ACIS for universities requires professorial approval. Attempts to obtain an evaluation version were unsuccessful.

6.7 Graphics APIs (Application Program Interface)

There are two major graphics APIs that are commonly used: OpenGL [26] and Direct3D (Part of the Microsoft DirectX package). OpenGL has better compatibility among systems, whereas Direct3D is only supported on Microsoft Windows.

Figure 17. OpenGL and DirectX logos

It is feasible that the project software will be applied to mobile devices (usually smartphones and tablets), which run operating systems such as Android, iOS, Windows mobile etc. At 16th May 2013, market research indicates [21] that Android devices make up 75% of the worldwide market for smartphones, iOS devices make up 17.3% and Windows at 3.2%.

OpenGL supports Android and iOS mobile devices through OpenGL ES [22]. OpenGL ES is also used in WebGL for browser-based rendering. Windows Phone 8 supports Direct3D natively with some limitations [19]. Newer versions of the Microsoft Surface Pro (a tablet) run Windows 8, which supports Direct3D.

6.8 Output

The project requires an output format that is compatible with 3D printers. Two standard file formats have been identified for this use: the *.stl and *.amf formats.

6.8.1 Stereo Lithography Format (*.stl)

An *.stl file is stored as a list of triangles that make up the outside 'skin' of an object. It does not support curves, but they can be approximated by using small enough triangles (Roscoe, 1988).

The *.stl format can either be in a plain-text format, or binary format.

An *.stl file does not contain any scale information. Most programs that use *.stl files assume them to be in millimetres or inches.

A plain-text *.stl file must start with the text "solid" optionally followed by the name of the object.

Each triangle must then be represented like so:

```
facet normal nx ny nz
      outer loop
            vertex v1x v1y v1z
            vertex v2x 21y v2z
            vertex v3x v3y v3z
      endloop
endfacet
```

where each *n* or *v* is a floating point number in sign-mantissa 'e'-sign-exponent format. Each vertex component must be non-negative and non-zero.

The file must end with “endsolid” optionally followed by the name of the object.

6.8.2 Additive Manufacturing File Format (*.amf)

AMF is an XML based format. It is designed as an improvement upon the STL format.

The AMF format lists all the vertices of the object. The order they are specified gives them an index number from zero to the number of vertices. The triangles making up the surface are then defined by their index numbers. This helps prevent holes or tearing of the surface where vertices don’t match up. The format also supports material types and colours. It also supports curves by specifying the normals of the vertices.

Summary

Solid Representation Schemes:
Cell Division
Constructive Solid Geometry
Boundary Representation

Validity of boundary-represented models can be checked using the Euler-Poincare formula, and making sure no faces intersect each other.

To display an arbitrary polygon, the polygon must be divided into triangles, a process called triangulation.

Many CAD packages represent curves through NURBS surfaces. These are based on NURBS curves, a subset of B-splines, which are a generalization of Bézier curves.
NURBS surface equation:

$$S(u,v) = \frac{\sum_{i=0}^{m}\sum_{j=0}^{n} N_{i,p}(u) N_{j,q}(v) w_{i,j} P_{i,j}}{\sum_{i=0}^{m}\sum_{j=0}^{n} N_{i,p}(u) N_{j,q}(v) w_{i,j}} \qquad (6.10)$$

History-based modelling has been the more widely used approach to the user interface until recently. Direct modelling has been increasing in popularity and is more intuitive to work with. Insufficient literature was found regarding user interaction with CAD systems.

There are three major modelling kernels available for building solid modellers: Open CASCADE (open source), Parasolid (owned by Siemens PLM) and ACIS (owned by Spatial corporation).

Two graphics APIs are available: OpenGL and Direct3D. On a typical computer they are very similar. However, OpenGL has better support for mobile devices.

The two output formats suitable for 3D printing are *.stl and *.amf. The *.stl format is older and simply comprises a list of triangles. The *.amf format is newer, having support for colours, materials, textures etc, but does not yet have the same adoption as *.stl.

7.0 Methodology

The project objective was to create a modelling program that could demonstrate mass customisation ability, for non-technical users. The building of any program necessitates some choices to be made regarding the approach. A modelling application has additional decisions to be made around the solid representation etc.

7.1 Solid Representation Scheme

A boundary representation scheme was chosen as the solid representation scheme. Boundary representation has the most flexibility for storing objects. CSG is limited in the models that can be created without making oversized levels of structures. Cell decomposition is not suitable for any solid modelling given its finite approach to storing models.

7.2 Modelling Kernel

Neither Parasolid nor ACIS could be easily obtained. They are products designed for business consumption, their acquisition requiring negotiation with sales staff. They are prohibitively priced for individual/academic use. In the interest of getting a demo running in a timely manner, the free Open CASCADE was chosen as the modelling kernel. An existing library was desirable for the project, rather than custom code.

Using a library means you avoid writing custom code, work which has already been done. The libraries have support available, for future efforts on the project code. Open CASCADE has been written over several years and thoroughly tested by its users.

Open CASCADE eventually proved too complex and overly-featured for the project purpose. Although simple models could be built, there was little to no documentation or guidance on interacting with the model e.g. selecting a face with the mouse and having that face change colour. Searching through the source code did not help – there are over 14,000 files in the OCC include directory. For changes to be made in a model's geometry, the part would have to be totally rebuilt each time. For an approach where a click and drag system is used this would mean rebuilding the model every time the screen refreshes while a face is being dragged. CAD systems are notoriously slow at building models, so a complex object represented in this project's application could cause it to slow down to the point of non intuitiveness.

Another approach to Open CASCADE was attempted with PythonOCC. This was abandoned when the current version of Python would not work with PythonOCC on Windows, and Open CASCADE could not be made to work on Linux.

Although other software projects have managed to successfully use the Open CASCADE libraries, this project did not have the time or resources to continue effort using it. If there was more time to work on the project, it might have benefitted from the exhaustive features of Open CASCADE or Parasolid/ACIS. However, it did not.

A custom boundary representation scheme was instead used, using data structures like so:

```
Co-Ordinate{
    x,y,z;
}

Vertex{
    Pointer to a Co-ordinate
    List of connected edges
    List of adjacent vertices
    List of adjacent faces
}

HalfEdge{
    List of Connected Vertices
    Pointer to Half-Edge's pair
    Pointer to the next edge in a loop
}

Face{
    List of lists of edges (first list is the face's outer edge loop, subsequent lists are loops for
        holes in the face)
    List of lists of vertices
    List of adjacent faces
    A, B, C and D where Ax + By + Cz = D, the equation of the plane the face lies on
}
```

```
Shell{
    List of Faces defining the shell
}

Object{
    List of Shells defining the object
    List of Faces in the object
    List of Edges in the object
    List of Vertices in the object
}
```

The half-edge representation scheme was chosen as it was deemed easier to work with, and is the typical way of storing edges in solid modellers.

7.3 Language

C++ was chosen as the programming language. C++ has the benefit of being an object-oriented language and allowing pointers. C++ is the native language of OpenGL and many other libraries that may need to be used – glm, Eigen, Freetype…

The author is most experienced in C++, resulting in quicker development times without the need of learning a new language and compiler. C++ does lack abstraction. This is not seen to be an issue for this project.

Additionally, CAD programs can be notoriously slow when dealing with complex objects. A lower level language helps to keep the software respond quickly to user interaction. AutoCAD and SolidWorks are written in C++.

7.4 Application Framework

MFC (Microsoft Foundation Classes) was used as the application Framework. MFC has exhaustive documentation on its classes and functions at the MSDN (Microsoft Developer Network) website. Other options, including Qt and WTL, have much less documentation, which is invaluable in learning how to use parts of an API.

An MFC application looks like a native Microsoft Windows Application. This is not as aesthetically attractive as a GUI for non-technical users would call for but is what Windows

users are very familiar with. In theory the application framework can be changed without difficulty.

7.5 Graphics

OpenGL was used for the graphics API. The decision was not overly important. The only real alternative is Direct3D, which would be fine. However, OpenGL is less platform-dependant, so the project can be more easily transferred to other systems such as Mac and Linux. Through OpenGL ES, it is also more applicable to mobile devices. OpenGL ES is supported on Android and iOS which make up a much larger part of the market than mobile Windows devices using Direct3D.

OpenGL version 3.1 was chosen, being the most recent version of OpenGL that would work on the test machine. The newest versions of OpenGL require a different programming methodology than the older versions that ship with Windows (v1.3). They involve using shaders and Vertex Array Objects (VAOs) for rendering.

7.6 Interaction Method

Direct Editing is heavily favoured for the user interface. Direct modelling is how games such as SPORE approach creating. The alternative would be a parametric approach, where dimensions must be specified, probably using a keyboard. This would be undesirable for the following reasons:

1. It forces the user to switch between the mouse and the keyboard
2. There is a lag between wanting to make a change, and seeing that change on screen
3. The user may not know how big/long/thick they want a dimension to be without seeing it.

A history-based approach would be detrimental to the user experience as it would force the users to learn a part's build history before being able to make changes. Although Siemens and others have had success in combining direct and history modes, it has been decided to use a direct modelling approach exclusively. The user interacts directly with the face/edge/point they want to edit.

7.7 Triangulation

As the program must accept all arbitrary solids, (providing they match the rules of validity) the program author is unable to imagine all objects that will be created/edited using the program. It must therefore support arbitrarily-shaped faces with arbitrary numbers of holes. The faces must be divided into triangles, in a process called triangulation, so they can be rendered to the screen. Some of the available triangulation libraries are:

- "Triangle" by Richard Shewchuck
- poly2tri
- General Polygon Clipper
- GluTesselator

"Triangle" only works in two dimensions, so points would have to be converted to a 2-dimensional plane, fed into Triangle and the result converted back into 3-D space. Holes are only supported by supplying Triangle with a point that lies inside the hole, a non-trivial problem.

poly2tri has a lack of documentation on usage, and reports indicate that it crashes regularly for degenerate cases.

The General Polygon Clipper reportedly uses a type of decomposition that introduces many t-junctions and has a prohibitive license.

GLUtessellator was determined to be the most suitable for the job. It is part of GLU, the OpenGL Utility library, making it technically deprecated for OpenGL 3.0+. Although considered deprecated, it has not been officially replaced, making it the most suitable triangulation algorithm available for this application.

7.8 Output Format

All three observed output formats, ASCII STL, binary STL and AMF, were implemented. The ease of implementation for the formats was simple enough to do this. ASCII STL and AMF are both text-based formats. Binary STL is essentially the same format as ASCII STL but in a binary format.

7.9 Other Libraries Used

- glm –A math library is needed with modern OpenGL to work with view, projection and model matrices, specifically applying transforms to them. glm is the defacto standard for use with OpenGL.

- Eigen – A library was needed for solving systems of linear equations. Eigen appears to be a stable library, while still being quite fast.

- Freetype – Freetype is used to read fonts from a font file and convert them to bitmaps. This was needed for displaying dimensions on screen with numbers. The characters, e.g. '5', were converted to a bitmap, which could then be mapped to a quad polygon, thereby displaying a number to the user.

Summary

After attempting to use Open CASCADE, a custom boundary representation scheme was developed in C++.

MFC was chosen as an application framework because of the immense resources available for it.

OpenGL was used as the graphics API. The deprecated GLUtesselator was chosen for triangulation after failing to find a better alternative.

Both types of STL format and the AMF format were implemented for output.

8.0 Implementation

The first part constructed was the basic topology data structures, the vertex, edge, face structures etc. These are present in topology.h.

It is important to visualize a model to check its accuracy, so the graphical aspect of the program was developed next. A basic OpenGL 1.3 window was set up (A more modern version of OpenGL would be used later, when other parts of the program were working).

Creating a basic OpenGL window involves defining a window class structure and registering it. Then, a device context structure is needed to render to a window. Next, a pixelformat is found and set for the device context. Finally, a rendering context is found and activated, connecting OpenGL to Windows.

8.1 Triangulation

Before solid models can be rendered the program needs to be able to triangulate faces into triangles, which can then be rendered. GLUtesselator does this, and was therefore needed in place before more work could be tested.

GLUtesselator is used like so:

```
for each face in object {
      gluTessBeginPolygon(GLUtessellator *tess, void *userData)
      for each edge loop in face {
            gluTessBeginContour(GLUtessellator *tess)
            gluTessVertex(GLUtessellator *tess, GLdouble
            cords[3], void *vertexData)
            gluTessEndContour(GLUtessellator *tess)
      }
      gluTessEndPolygon(GLUtessellator *tess)
}
```

When gluTessEndPolygon() is called a set of callback functions are called including vertexCallback(). vertexCallback() gets called with a vertex as an argument in an order that

defines triangles that can be stored and rendered so as to represent the total polygon. These functions are a part of draw.cpp

8.2 Building the Object

Before displaying objects can begin, there must be functions to create vertices, edges, faces and shells, to connect edges into edge loops, and associate each data structure with its adjacent elements.

Build.cpp includes the following functions:

```
void MakeVertex(Object *Obj, double x, double y, double z,
char *name);
void MakeEdge(Object *Obj, char *V1,char *V2, char *name);
void ConnectEdges(Object *Obj, char *E1, char *E2);
void MakeFace(Object *Obj, char *E1, char *name);
void MakeHole(Object *Obj, char *F1, char *E1);
void MakeShell(Object *Obj, node<Face> *faceptr, char *name);
```

MakeVertex() takes x, y and z positions and creates a coordinate from the data. A Vertex is then made, referencing the coordinate. The Vertex' name is set to the name supplied as an argument. The Vertex is added to the Object referenced in the first argument.

MakeEdge() Searches Object *Obj for two vertices, V1 and V2. If both vertices can be found, a new HalfEdge is created with VertexListFirst = V1 and VertexListLast = V2. When a HalfEdge is created, another 'pair' HalfEdge must also be created with VertexListFirst = V2 and VertexListLast = V1.

ConnectEdges() sets the nextEdgeInLoop pointer on two HalfEdges such that one HalfEdge A points to B. Which one depends on the vertex the HalfEdges have in Common.

MakeFace() takes the first edge of an edge loop and creates a Face. For the edge loop to be valid, the first edge must point to another edge, which must point to another, and so on until the first edge is involved again.

MakeHole() adds an edge loop to a face. The first edge loop of a face is its outer edge, while subsequent edge loops are inner loops defining holes. Adding an edge loop to a face effectively creates a hole.

MakeShell() creates a Shell from a list of Faces.

From here, an object can now be represented.

A cube would be defined like so:

```
MakeVertex(Object, 1.0, 1.0, 1.0, "V1");
MakeVertex(Object, 1.0,-1.0, 1.0, "V2");
MakeVertex(Object,-1.0,-1.0, 1.0, "V3");
MakeVertex(Object,-1.0, 1.0, 1.0, "V4");
MakeVertex(Object, 1.0, 1.0,-1.0, "V5");
MakeVertex(Object, 1.0,-1.0,-1.0, "V6");
MakeVertex(Object,-1.0,-1.0,-1.0, "V7");
MakeVertex(Object,-1.0, 1.0,-1.0, "V8");

MakeEdge(Object,"V1","V2","E1");
MakeEdge(Object,"V2","V3","E2");
MakeEdge(Object,"V3","V4","E3");
MakeEdge(Object,"V4","V1","E4");

MakeEdge(Object,"V5","V6","E5");
MakeEdge(Object,"V6","V7","E6");
MakeEdge(Object,"V7","V8","E7");
MakeEdge(Object,"V8","V5","E8");

MakeEdge(Object,"V1","V5","E9");
MakeEdge(Object,"V2","V6","E10");
MakeEdge(Object,"V3","V7","E11");
MakeEdge(Object,"V4","V8","E12");

ConnectEdges(Object,"E1","E4");
ConnectEdges(Object,"E4","E3");
ConnectEdges(Object,"E3","E2");
ConnectEdges(Object,"E2","E1");
MakeFace(Object,"E1","Front");

ConnectEdges(Object,"E5","E6");
ConnectEdges(Object,"E6","E7");
```

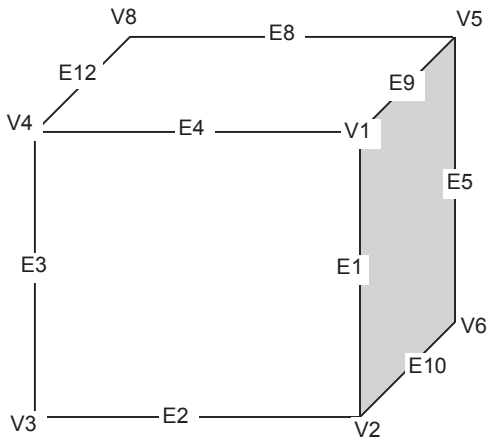


Figure 18. Constructed cube

```
ConnectEdges(Object,"E7","E8");
ConnectEdges(Object,"E8","E5");
MakeFace(Object,"E5","Back");

ConnectEdges(Object,"E4", "E9");
ConnectEdges(Object,"E9", "E8");
ConnectEdges(Object,"E8","E12");
ConnectEdges(Object,"E12","E4");
MakeFace(Object,"E4","Top");

ConnectEdges(Object,"E1","E10");
ConnectEdges(Object,"E10","E5");
ConnectEdges(Object,"E5", "E9");
ConnectEdges(Object,"E9", "E1");
MakeFace(Object,"E1","Right");

ConnectEdges(Object,"E2","E11");
ConnectEdges(Object,"E11","E6");
ConnectEdges(Object,"E6","E10");
ConnectEdges(Object,"E10","E2");
MakeFace(Object,"E2","Bottom");

ConnectEdges(Object,"E3","E12");
ConnectEdges(Object,"E12","E7");
ConnectEdges(Object,"E7","E11");
ConnectEdges(Object,"E11","E3");
MakeFace(Object,"E3","Left");

MakeShell(Object,Object->FaceListFirst,"Shell1");
```

8.3 Graphics

Each Face data structure has a pointer to a structure containing rendering information (VertexArray). When the vertexCallback function (from GLUtesselator) runs, that structure is filled with vertex data for each of the triangles that make up the face.

A Vertex Array Object (VAO) and Vertex Buffer Object (VBO) must be prepared for sending to the shader. This can be done like so:

```
glGenVertexArrays(1, &vaFaces->vaoID[0]); // Create Vertex
Array Object
glBindVertexArray(vaFaces->vaoID[0]); // Bind Vertex Array
Object so it can be used

glGenBuffers(3, vaFaces->vboID); // Generate Vertex Buffer
Object

glBindBuffer(GL_ARRAY_BUFFER, vaFaces->vboID[0]); // Bind
Vertex Buffer Object
glBufferData(GL_ARRAY_BUFFER, vaFaces->numVertices *3*
sizeof(GLdouble), vaFaces->vertices.data(), GL_STATIC_DRAW);
// Set the size and data of the VBO
glVertexAttribPointer((GLuint)0, 3, GL_DOUBLE, GL_FALSE, 0,
0); // Set up vertex attributes pointer
glEnableVertexAttribArray(0);
```

Rendering is done using shaders. In this program the GLSL (OpenGL Shading Language) was used. The main vertex shader for this application calculates the position of a vertex by multiplying the projection, view and model matrices passed to it. It also calculates the intensity of light reflected on a face.

8.4 Interaction

8.4.1 Face Selection

The first part of interacting with the model on screen is to left-click on a face and have it change colour.

First, the position of the screen that was clicked is taken in two-dimensional screen coordinates. The coordinates are passed through the glm library function unProject() for z coordinates of 0.0 and 1.0, creating a ‘ray’ that lies directly underneath the mouse from z = zero to the end of the projective plane.

Each face lies on a plane and each of those planes are tested to find the intersection with the ray, and the intersection's distance from the screen. The faces are ordered from nearest to farthest as the intersections are found. The face nearest to the screen that is intersected by the ray will be the one selected.

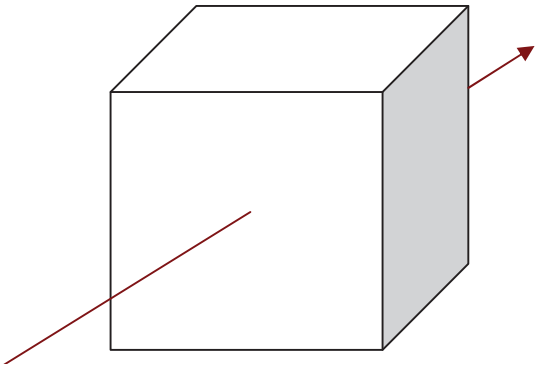


Figure 19. The ray intersects both the front and back faces, but only the front face should be selected

Each face of the object is then tested for intersection against the ray. This is done by using the triangles created for rendering each face. If the ray passes through any of triangles, then it must pass through the face. To test if a ray passes through a triangle, the ray was tested to see if it passed 'above' or 'below' each of the three sides. If it passes above all three, then it passes through the triangle. If it appears to pass below all three, then the ray is coming from the opposite direction to the triangle normal, and it passes through the triangle.

The test for whether a ray passes above or below a side is as follows:

$V1 = T1 - P0$ (8.1)

$V2 = T2 - P0$ (8.2)

$N = V2 \times V1$ (8.3)

if$(Px - P_0)\bullet N < 0$ {

Px is below line (T1,T2)

}

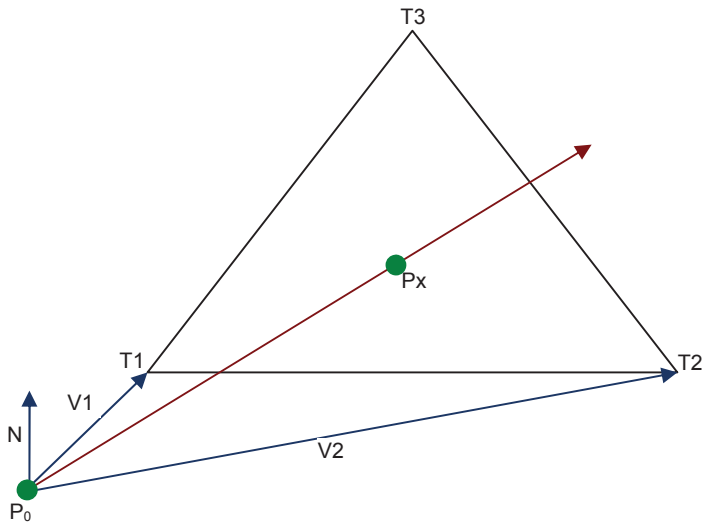


Figure 20. Vectors used to determine intersection of triangle

This works because $(Px - P_0) \bullet N$ (where $(Px - P_0)$ and N are normalised) is equal to cos θ, where θ is equal to the angle between N and $Px - P_0$.

if $\theta > 90°$ then Px is below the line.

if $\theta < 90°$ then Px is above the line.

When a face is found to be selected it is a simple matter to render it in a different colour, making it clear to the user that the face has been successfully selected.

8.4.2 Face Dragging

When a face is dragged in this program it moves in the direction of the face's plane's normal (N), and retains its shape. This involves moving each vertex position of the face by a specified distance in the direction of N.

Firstly, the distance to move must be determined by the mouse's position.

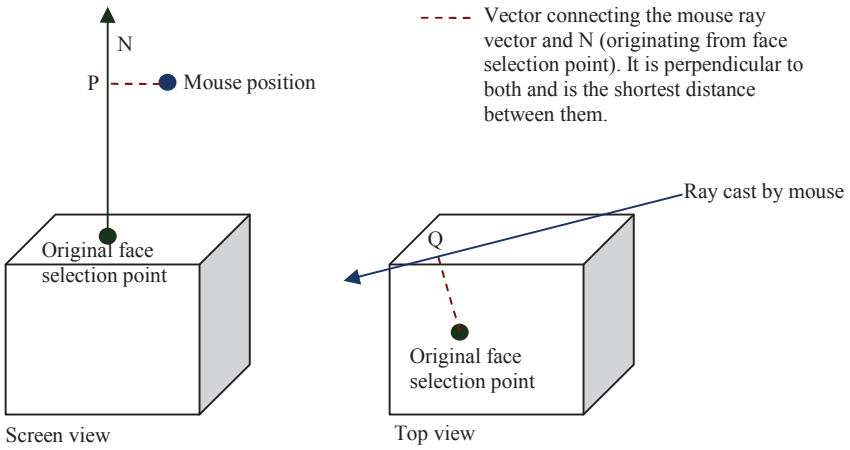


Figure 21. Front and top views of the mouse ray related to the face normal vector originating from the original selection point

O: the original face selection point

N: face normal

R_0: mouse ray origin

R: mouse ray direction vector

V: vector connecting the face normal and mouse ray.

It was decided to move the face up to point P, where:

$$O + tN = P \quad (8.4)$$

$$P + sV = Q \quad (8.5)$$

$$R_0 + uR = Q \quad (8.6)$$

$$O + tN + sV = R_0 + uR \quad (8.7)$$

$$O_x = tN_x + sV_x = R_{0x} + uR_x \quad (8.8)$$

$$O_y = tN_y + sV_y = R_{0y} + uR_y \quad (8.9)$$

$$O_z = tN_z + sV_z = R_{0z} + uR_z \quad (8.10)$$

V is perpendicular to both N and R. As such it can be found by taking the cross product of N and R. This leaves a system of linear equations with three unknowns which could be solved using a linear solver like Eigen. However, a different approach was used for this program.

The distance d between N and R is

$$\mathrm{d} = \mathrm{O} + \mathrm{tN} - \mathrm{R}_0 - \mathrm{R} \qquad (8.11)$$

Because we are looking for the smallest positive distance between the vectors we can simplify by finding d^2.

Using partial differentiation, $\frac{\partial f}{\partial t}$ and $\frac{\partial f}{\partial u}$ can be found. When these are both zero the local minima has been found, hence the points on N and R where the distance between them is minimal.

(See appendix for calculations)

$$t = \frac{R2(NR_0 - NO) + (NR(OR - RR_0)}{(N2 * R2) - NR^2} \qquad (8.12)$$

where:

$$R2 = (Rx + Ry + Rz)^2 \qquad (8.13)$$

$$N2 = (Nx + Ny + Nz)^2 \qquad (8.14)$$

$$NR_0 = NxR_0x + NyR_0y + NzR_0z \qquad (8.15)$$

$$NR = NxRx + NyRy + NzRz \qquad (8.16)$$

$$NO = NxOx + NyOy + NzOz \qquad (8.17)$$

$$OR = OxRx + OyRy + OzRz \qquad (8.18)$$

$$RR_0 = RxR_0x + RyR_0y + RzR_0z \qquad (8.19)$$

Once t is found finding the distance the vertices must be moved is straightforward. The face's list of vertices is traversed, adding t*N (N is normalised) to its original position. To facilitate this, a list of the face's vertices and their original positions is created when the face is selected. The list is cleaned up when the left mouse button is released.

There is another matter to consider when face dragging. Consider the following case:

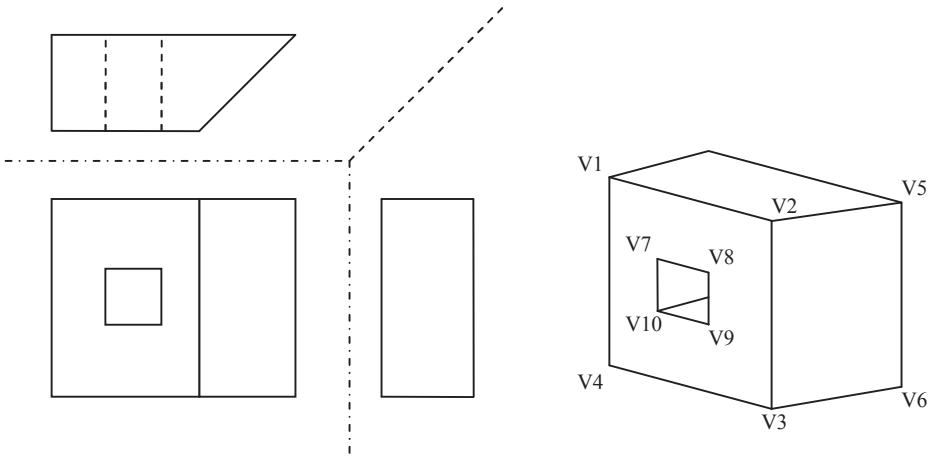


Figure 22. Orthographic and isometric views of a demonstration object

The right face is at a 45° angle and the front face has a hole that extends to the back face. If the right face were to be dragged and only the right face vertices were moved (V2,V5,V6,V3) then the vertices that make up the front face (V1,V2,V3,V4,V7,V8,V9,V10) would not all lie on the same plane. This problem is not only true for faces with holes but any face that has more than three vertices where at least two vertices (that are not part of the edge being moved) are different distances from the edge that is being moved (the edge moved in the above case is V2-V3, where the front face is concerned).

Most vertices have three adjacent faces. Zero or one adjacent faces are not possible when only flat surfaces are involved. Two adjacent faces are possible, though unlikely. They may exist in a case like so:

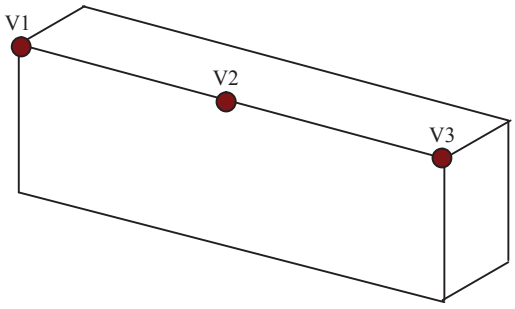


Figure 23. Example case where a vertex (V2) has only two adjacent faces

Four or more adjacent faces are very possible with pyramid-like structures, but are as yet unaddressed.

Because most vertices have three adjacent faces, it was decided to take the following approach:

When the movement of a face would cause the vertices of an adjacent face to not lie on the same plane, we find the vertex (Vf) on the adjacent face that is farthest from the edge (E) shared between the moved face (Fm) and the adjacent face (Fa), before movement, in a line perpendicular to the edge. The edge is comprised of a start and end vertex (V_E1,V_E2). Now Fm can be moved, which also moves E. Vf, V_E1 and V_E2 form a triangle which defines a plane (P). Where the vertices of Fa have three adjacent faces, we move the vertex such that it lies on the intersection of two of those adjacent faces and P (instead of Fa).

A plane has an equation of the form:

$$Ax + By + Cz = D \qquad (8.20)$$

The intersection of three planes requires solving the system of equations:

$$A_1x + B_1y + C_1z = D_1 \qquad (8.21)$$

$$A_2x + B_2y + C_2z = D_2 \qquad (8.22)$$

$$A_3x + B_3y + C_3z = D_3 \qquad (8.23)$$

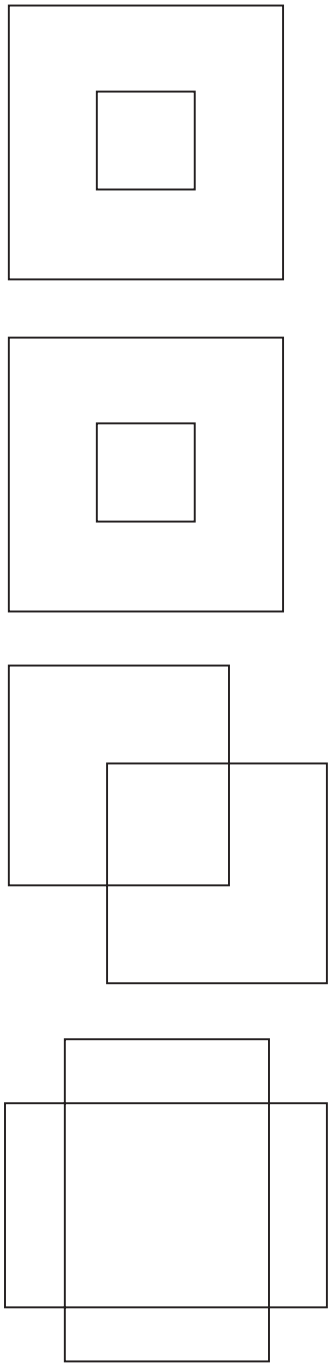
where x, y and z are unknowns. In the program this is solved by using the Eigen library function `colPivHouseholderQr().solve()`. From the Eigen tutorials [17]: “Here, `colPivHouseholderQr` is a QR decomposition with column pivoting. It's a good compromise … as it works for all matrices while being quite fast”

In theory, there are several types of combinations of three planes that do not intersect at a single point. It has not been proven that these cases will not apply in the program and no effort has been made to compensate should they occur. This may cause errors and/or crashes for some models.

8.4.3 Face Dragging Limits

When a face is dragged, it needs to stop if it is going to intersect another face. Intersection with another face is invalid geometry for a solid object.

The solution to this is to find the limits that a face can be dragged when that face is selected. Currently this is done in interaction.cpp with the function FindLimits(). Four general cases have been identified where a limit must exist on a faces movement.



Case 1: All vertices of face 1 lie 'inside' of face 2, and all vertices of face 2 lie 'outside' face 1, when looked at from a perspective perpendicular to face 1.

Case 2: All vertices of face 1 lie 'outside' of face 2, and all vertices of face 2 lie 'inside' face 1, when looked at from a perspective perpendicular to face 1.

Case 3: One or more vertices of face 1 lie 'inside' face 2 and one or more vertices of face 2 lie 'inside' face 1, when looked at from a perspective perpendicular to face 1.

Case 4: All vertices of face 1 lie 'outside' face 2 and all vertices of face 2 lie 'outside' face 1, but the faces would collide if face 1 was moved, when looked at from a perspective perpendicular to face 1.

Figure 24. Sample intersection cases

To address the first case, we create rays from each of the vertices of face 1 that are perpendicular to face 1. These rays can then be tested for intersection with face 2 using the method described in the face selection section. If a ray intersects the face its distance from vertex-to-face is recorded. The smallest distance between a vertex and face 2 is taken as the limit of face movement.

To address the second case, the same method is used, but with face 1 and face 2 reversed.

If both of the first two cases are addressed, then there should not be a need for a separate test for the third.

The fourth case can not be addressed by ray intersection testing like the others as no vertices would hit the other face if face 1 was to be moved. Instead, the edges must be tested for intersections.

If a straight edge, E, is made of two vertices, V1 and V2, it equation of line is:

$$V1 + t(V2 - V1) \qquad (8.24)$$

When a face is moved, all its vertices move in the direction of the face's normal, N. The line equation of an edge on a moved face is therefore:

$$V1 + sN + t(V2 - V1) \qquad (8.25)$$

The line equation for an edge on face 2 is:

$$W1 + u(W2 - W1) \qquad (8.26)$$

The lines will intersect when:

$$V1 + sN + t(V2 - V1) = W1 + u(W2 - W1) \qquad (8.27)$$

If both t and u are between 0 and 1, the line intersection occurs between the vertices of each edge. The edges will therefore intersect if face 1 is moved a sufficient distance.

This is implemented like so:

```
for each edge in face 1{
        for each edge in face 2{
                solve (Equation 8.17)
                if s < s_max { // if the distance to intersection is less than the smallest distance
                already found
                        if 0 < t < 1 and 0 < u < 1 { // if edges will intersect
                                s_max = s
                        }
                }
        }
}
```

A face can be moved both forward and backward, so there will be two limits – one for the positive direction and one for the negative direction. This is not addressed in the pseudocode above, but is a simple matter of seeing if s is positive or negative, and testing against the positive or negative limit accordingly.

8.4.4 Face Intersections

It was discovered in certain cases that dragging a face would cause one of its adjacent faces to move in such a way that it would intersect another face.

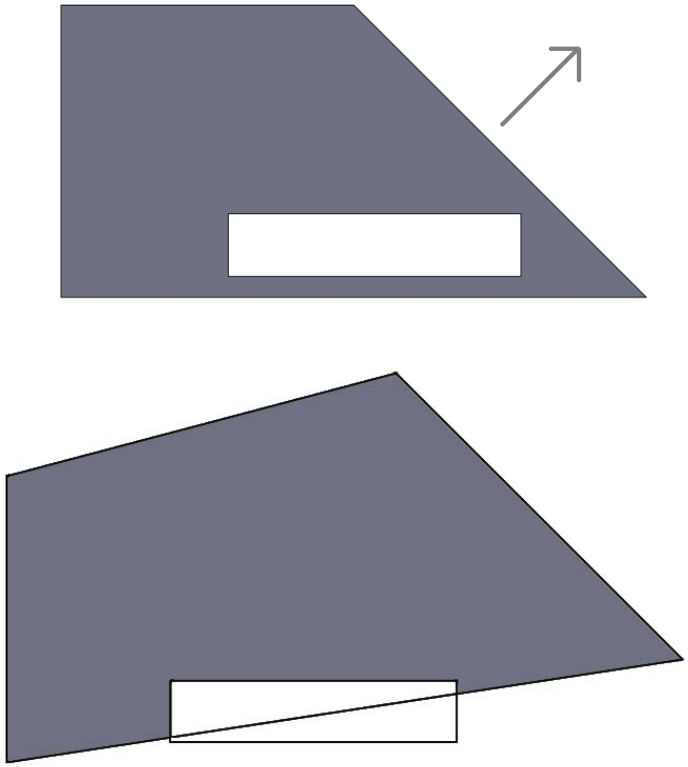


Figure 25. Front view. When the right-side face is dragged it also moves the bottom face, causing it to intersect with several of the hole faces.

A test for intersecting faces was created to address this issue. If faces are found to be intersecting, the user can be alerted that the geometry is invalid. This test can also be performed after the model is first built to ensure validity. No literature could be found on how to solve this problem so an original solution was developed.

Two non-parallel planes will intersect in a straight line (L). Flat faces will each lie on a plane. If L intersects the edges of either of the faces there is a possibility that the faces are intersecting. This can be investigated further by ordering all the intersections by distance from a point on L. The distance can be negative. If a face causes an intersection, the next intersection must be by the same face if the faces are not intersecting at that area.

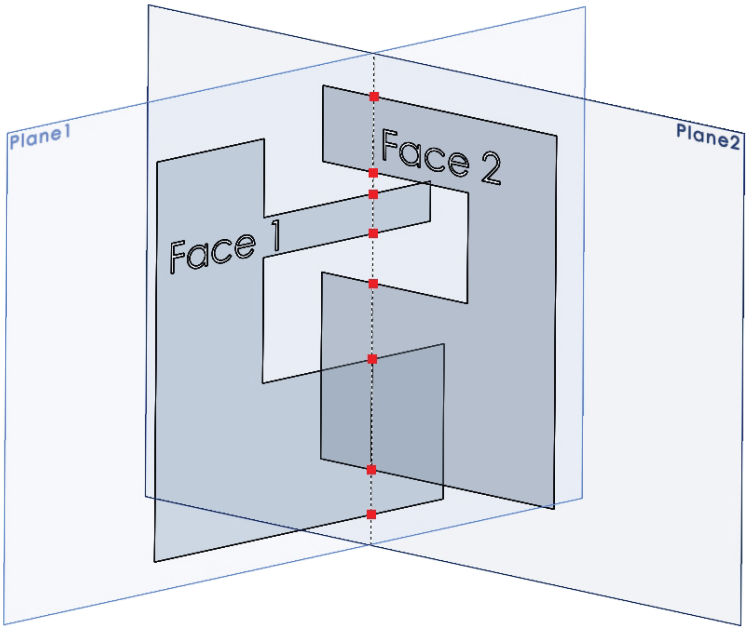


Figure 26. Intersection of two face planes. The red dots show where the edges of each face intersect the plane of the other face.

Using Figure 17 as an example, Plane 1 and Plane 2 intersect on the dotted line (L). From top to bottom, face 2 is the first to intersect L. Face 2 intersects L again before face 1 does, so the geometry is valid up to that point. Still moving from top to bottom, face 1 does the same thing and the geometry is still valid. Face 2 is the next to intersect L. However, before it intersects again, L is intersected by Face 1, which means the faces are intersecting at that point.

The total number of intersections that occur for each face must be an even number.

The pseudocode is:

```
Calculate L // L = (Lx0,Ly0,Lz0) + t(Lx,Ly,Lz)
for each edge in face 1{
        if edge is intersecting L {
                InsertIntoList ( t, face 1)
        }
}
for each edge in face 2{
        if edge is intersecting L {
                InsertIntoList ( t, face 2)
        }
}
loop{
```

```
        if List (i).face != List (i+1).face{
                faces are intersecting
                return true
        }
        i = i + 2
}
return false
```

Because this is in three dimensional space, two lines may not intersect each other when they should, due to truncation error, rounding or various other factors. Therefore, when finding the intersection of two lines, we 'flatten' the equation into two dimensions. The easiest way to do this is to simply omit the x, y or z equation, whichever will have the least effect. The equation to omit is chosen by finding the largest absolute term of the normal of the face we are working with. E.g. If the face equation is $3x - 5y + z = 10$, its normal is $\begin{pmatrix} 3 \\ -5 \\ 1 \end{pmatrix}$. The largest absolute term is the y term (-5). We therefore omit the y term from the line equations.

$$\begin{pmatrix} Lx0 \\ Ly0 \\ Lz0 \end{pmatrix} + s\begin{pmatrix} Lx \\ Ly \\ Lz \end{pmatrix} = \begin{pmatrix} Ex0 \\ Ey0 \\ Ez0 \end{pmatrix} + t\begin{pmatrix} Ex \\ Ey \\ Ez \end{pmatrix} \qquad (8.28)$$

becomes:

$$\begin{pmatrix} Lx0 \\ Lz0 \end{pmatrix} + s\begin{pmatrix} Lx \\ Lz \end{pmatrix} = \begin{pmatrix} Ex0 \\ Ez0 \end{pmatrix} + t\begin{pmatrix} Ex \\ Ez \end{pmatrix}. \qquad (8.29)$$

The first equation has two unknowns (s and t) but three equations, which means that a solution might not be possible. The second equation is guaranteed to have a solution, provided the lines are not parallel.

There is a problem when a vertex lies on the plane intersection line. We are not immediately able to tell if the edge loop passes through the intersection line or if the vertex lies on the line with both edges connected to it on the same side of the line.

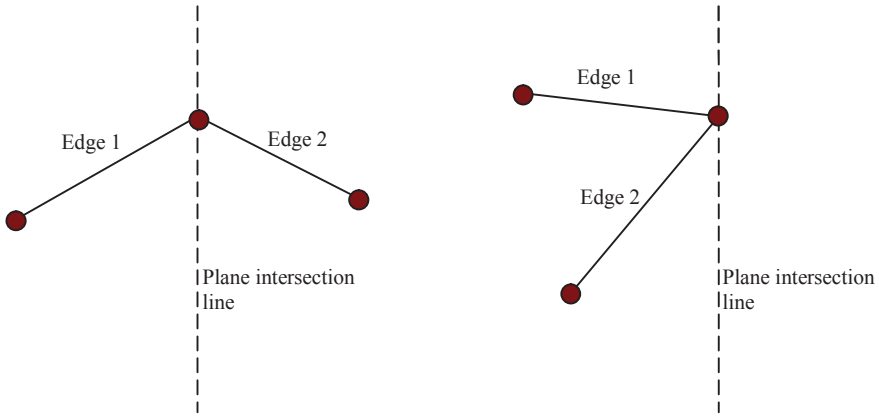


Figure 27. Left: the Edge loop passes through the intersection line. Right: the edge loop does not pass through the intersection line but touches on it.

Using the example in Figure 18, the left example should be perceived as one intersection for successful face intersection detection. The right example should be perceived as no intersections or two intersections in the same place (both edges intersect the line an infinitely small distance from the vertex). Ideally, we would guarantee it to be two intersections, otherwise there could be a situation where the vertex of one face lies 'in' another face, which would be non-manifold and hence, invalid. However, problems that will be detailed later lead to the approach being left undeveloped.

A standard solution to this problem is to hypothetically displace the plane intersection line an infinitely small distance in a perpendicular direction, and see if it still intersects the edge we are looking at.

In the program, the equation of the edge is:

$$\begin{pmatrix} Ex0 \\ Ey0 \end{pmatrix} + t\begin{pmatrix} Ex \\ Ey \end{pmatrix} = \begin{pmatrix} V1x \\ V1y \end{pmatrix} + t\begin{pmatrix} V2x - V1x \\ V2y - V1y \end{pmatrix} \qquad (8.30)$$

The equation

$$\begin{pmatrix} Lx0 \\ Ly0 \end{pmatrix} + s\begin{pmatrix} Lx \\ Ly \end{pmatrix} = \begin{pmatrix} Ex0 \\ Ey0 \end{pmatrix} + t\begin{pmatrix} Ex \\ Ey \end{pmatrix} \qquad (8.31)$$

is solved for t with the equation

$$t = \frac{Ly(Lx0 - Ex0) + Lx(Ey0 - Ly0)}{LyEx - LxEy} \qquad (8.32)$$

If t is equal to zero or one, the beginning or end of the edge, then the intersection is on a vertex. To move the plane intersection line, the starting point $\begin{pmatrix} Lx0 \\ Ly0 \end{pmatrix}$ would be moved by some distance (u) in the direction of $\begin{pmatrix} Ly \\ -Lx \end{pmatrix}$, a perpendicular vector to the line equation.

$$t_{new} = \frac{Ly\left(\left(Lx0 + uLy\right) - Ex0\right) + Lx\left(Ey0 - \left(Ly0 - uLx\right)\right)}{LyEx - LxEy} \tag{8.33}$$

$$t_{new} = \frac{Ly\left(Lx0 + uLy - Ex0\right) + Lx\left(Ey0 + uLx - Ly0\right)}{LyEx - LxEy} \tag{8.34}$$

$$t_{new} = \frac{Ly\left(Lx0 - Ex0\right) + Lx\left(Ey0 - Ly0\right) + u\left(Lx^2 + Ly^2\right)}{LyEx - LxEy} \tag{8.35}$$

$$t_{new} = t + \frac{u\left(Lx^2 + Ly^2\right)}{LyEx - LxEy} \tag{8.36}$$

u is said to positive. $Lx^2 + Ly^2$ has two terms that are both squared, so it is guaranteed to be positive (It cannot be zero because of earlier algorithms). This leaves the denominator. If $LyEx - LxEy$ is positive, then t_{new} will be greater than the original t. If negative, then t_{new} will be less than the original t. This result can be used to determine whether the edge still intersects the plane intersection line.

if ((t = 0 and $t_{new} > t$) or (t = 1 and $t_{new} < t$)) {

The line and edge intersect when the line is moved

}

It was found that round-off error of the floating-point types affected the result of t. When t should be exactly zero or one the result was sometimes a very small amount different. This resulted in an uneven number of edges detected as intersecting the line. A solution to this might be to add a tolerance to the 'if' statement testing for zero or one. However, floating-point types have greater precision when working with smaller numbers and less precision when working with bigger numbers, making the choice of a tolerance difficult.

There is, however still a problem.

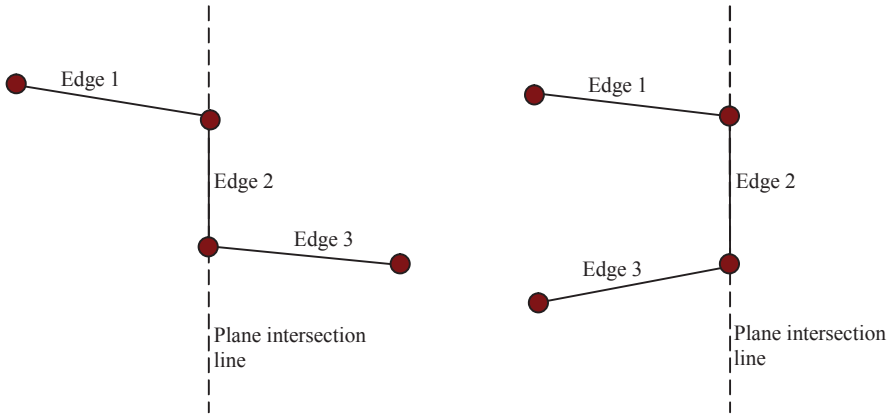


Figure 28. Edges are collinear with the plane intersection line

Edges may be collinear with the plane intersection line. The right example in Figure 28 is non-critical as it can be said to have no intersections or two intersections on Edge 1 and Edge 3. The left example needs special handling. This problem was not handled in the program.

8.4.5 Object Stretching

Another feature implemented is the ability to stretch the object by dragging a face.

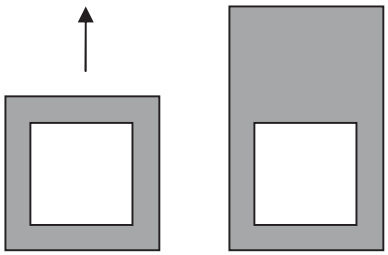


Figure 29. Without stretch feature enabled

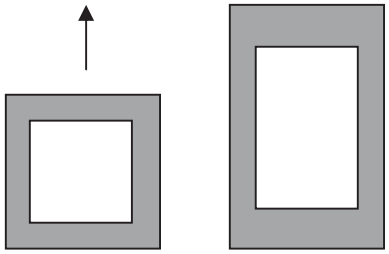


Figure 30. With stretch feature enabled

This feature is implemented by finding the farthest vertex from the selected face, when the face is selected. That vertex acts as a limit and other vertex positions are adjusted according to it. It must be 'below' the face. The distance is measured perpendicular to the face.

Where the plane equation is: $Ax + By + Cz = D$ (8.37)

and the line from plane to vertex is: $\begin{pmatrix} Vx \\ Vy \\ Vz \end{pmatrix} + t \begin{pmatrix} A \\ B \\ C \end{pmatrix}$, (8.38)

where $\begin{pmatrix} A \\ B \\ C \end{pmatrix}$ is the normal to a plane.

$$A(Vx + tA) + B(Vy + tB) + C(Vz + tC) = D \qquad (8.39)$$

$$t = \frac{D - (AVx + BVy + CVz)}{A^2 + B^2 + C^2} \qquad (8.40)$$

The vertex that gives the largest value of t is 'farthest' from the face. Call this largest value t_{max}.

When a face is dragged, all vertices in the model, except the one farthest from the plane, move in the direction of the face normal. Their magnitude of movement is determined by the ratio of the vertex-to-face distance to t_{max}.

8.4.6 Model Rotation

Model rotation using the mouse has been implemented using an arcball. An arcball works by matching up mouse coordinates with a sphere centred on the object. When the mouse is moved, the sphere is also moved so that the coordinates still match up. This creates an intuitive method of model rotation.

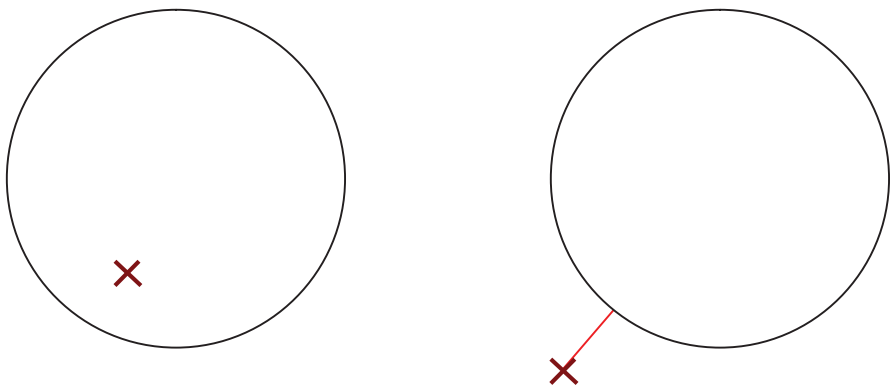


Figure 31. A mouse click on the sphere (left) puts a control point in the same place. A click outside the sphere (right) maps the control point to the closest point on the sphere.

A mouse click creates a ray from zero to infinity. The equation of this line is

$$E + tV\,, \qquad (8.41)$$

where E is the 'eye' location, V is the vector of the mouse ray. The points on the sphere equal r^2 (r: radius of sphere), assuming the sphere is centred at (0,0,0). Equating the equations gives:

$$(E+tV)^2 = r^2 . \qquad (8.42)$$

The solution is:

$$t = \frac{-2EV \pm \sqrt{4(EV)^2 - 4V^2(E^2 - r^2)}}{2V^2} . \qquad (8.43)$$

t should be taken as the minimum of its possible values for $E+tV$ to lie on the closest side of the sphere to the viewer. If there is no solution (the equation inside the square root is negative) r is adjusted so that the equation within the square root is zero.

$$r_{new} = \sqrt{E^2 - \frac{EV^2}{V^2}} \qquad (8.44)$$

The ratio of r_{new} to $r_{original}$ is taken. Now

$$t = \frac{-EV}{V^2} \qquad (8.45)$$

and the point mapped to the sphere is

$$(E+tV) \times \frac{r_{new}}{r_{original}} . \qquad (8.46)$$

Otherwise, the point is simply $E+tV$. (8.41)

The rotation is based on the current point compared to the point selected when the mouse was clicked. A normalised cross product of the current and original points (as vectors from the centre of the sphere) is taken as an axis of rotation. The angle of rotation is the dot product between the vectors.

To apply this rotation to the OpenGL context the rotation matrix is determined. When the rotation vector and angle are known the matrix can have the form:

$$\begin{bmatrix} x^2(1-\cos\theta)+\cos\theta & xy(1-\cos\theta)-z\sin\theta & xz(1-\cos\theta)+y\sin\theta & 0 \\ yx(1-\cos\theta)+z\sin\theta & y^2(1-\cos\theta)+\cos\theta & yz(1-\cos\theta)-x\sin\theta & 0 \\ zx(1-\cos\theta)-y\sin\theta & zy(1-\cos\theta)+x\sin\theta & z^2(1-\cos\theta)+\cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad (8.47)$$

where x, y and z are the elements of the rotation vector and θ is the angle.

The model matrix of the OpenGL context is multiplied with this rotation matrix to get an update view of the scene.

8.5 Dimensioning

8.5.1 Scaling

An option has been given to provide a scaling factor to the model by implementing a number box to the side. The number in the box can be easily changed to provide an overall scaling factor to the model. The scale is then reflected in exported *.stl files and the dimensioning graphic (detailed below).

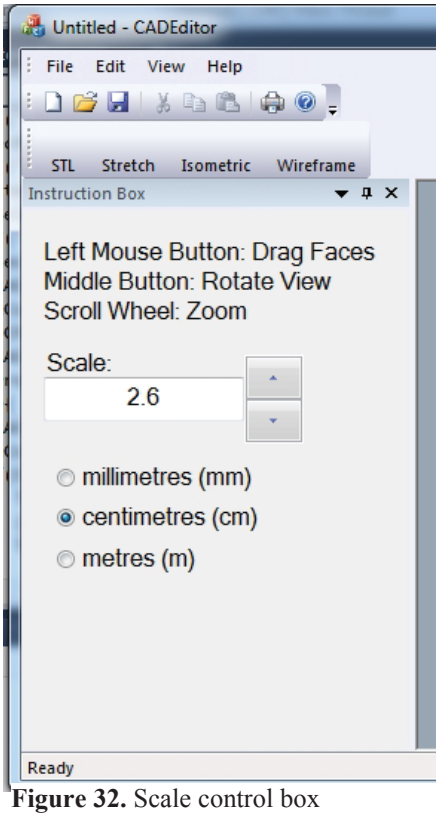


Figure 32. Scale control box

There are also radio buttons to use units of millimetres, centimetres or metres. Switching from millimetres to centimetres for example, is equivalent to applying a scaling factor of 10, but the units will also be applied to the dimensioning graphic. This is to make it easier to visualise the model in a real world setting, by using convenient units.

8.5.2 Dimensioning Graphic

A graphic has been applied to give the user an idea of how large their object is. The graphic appears whenever a face is selected or dragged. It displays the distance from that face to the farthest point on the model from that face.

The bar appears as though it exists in three-dimensional space, but it is really a 2D overlay. The desired direction of the bar has to be found to position it correctly. This direction is a vector that is the cross product of the face normal and the vector going directly into the screen.

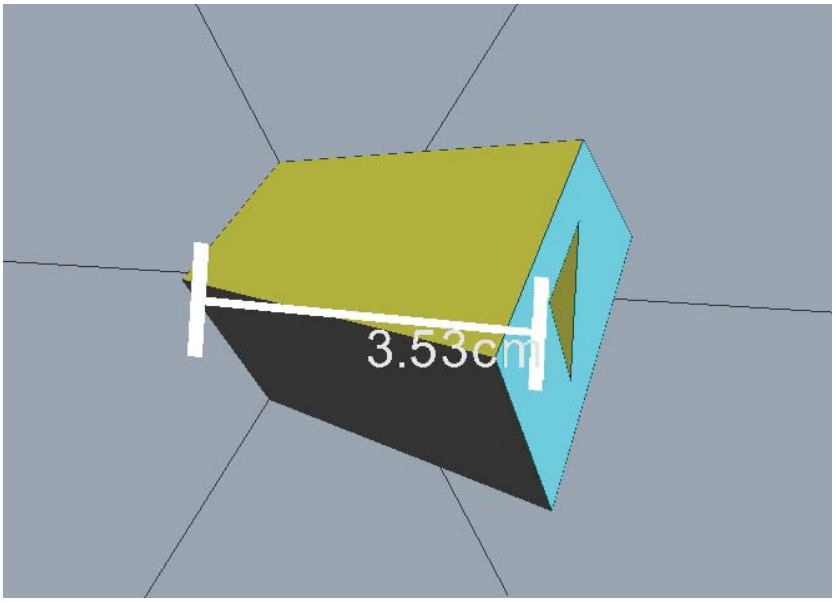


Figure 33. When a face is selected, a graphic appears to give the user an idea of the scale

The first step is to find the furthest vertex from the selected face, which has been done for many other purposes in this project.

Using the farthest vertex data and the point used to drag the face, the end points of the graphic can be determined. These points are then projected onto the 2D plane. From there, the graphic can be created by generating triangles which will be sent to the renderer.

By working in the 2D plane, it is easier to create a graphic of consistent size, no matter how far zoomed in or out the object is.

The text that shows the distance must also be added. For the digits 0 to 9, and the characters '.','c', and 'm', bitmap images are generated using the Freetype library. The Freetype library exists to convert fonts in *.ttf files to bitmap images. These images can then be mapped to flat quad shapes. In the program, the quads are generated and positioned just to the lower-right of the dimensioning graphic midpoint (in 2D space).

An integer to string conversion is required, where the number is displayed to three significant figures. The character '.' is appended, followed by "mm", "cm" or "m", depending on what units have been selected by the user.

8.6 STL File Export

The triangle data that makes up an *.stl file was taken from the graphical information stored in the face data structures. Happily, the graphical representation is also stored as triangles so little conversion is needed.

As an *.stl file must contain only positive definite values for vertices, we first go through all the vertices to find the minimum values in the x, y and z directions. The value ((0 – {min. value}) + 1) can then be added to each point just before the value is written to file. This will ensure all points have values of at least 1 in all three dimensions.

The next step is to apply the scaling information selected by the user. As most programs will assume the file to be in millimetres, the file should use those dimensions. Each number must be multiplied by the scale selected by the user. The number must then be multiplied by 10 if ‘centimetres’ has been selected, or by 1000 if ‘metres’ has been selected. This takes place after the minimum values have been found, but before the value ((0 – {min. value}) + 1) has been added to each vertex. The minimal values must also be multiplied by the correct scale to assure a correct *.stl file.

9.0 Results

The work resulted in a workable program that allows a user to change the shape of an object by dragging its faces, thus 'customising' it.

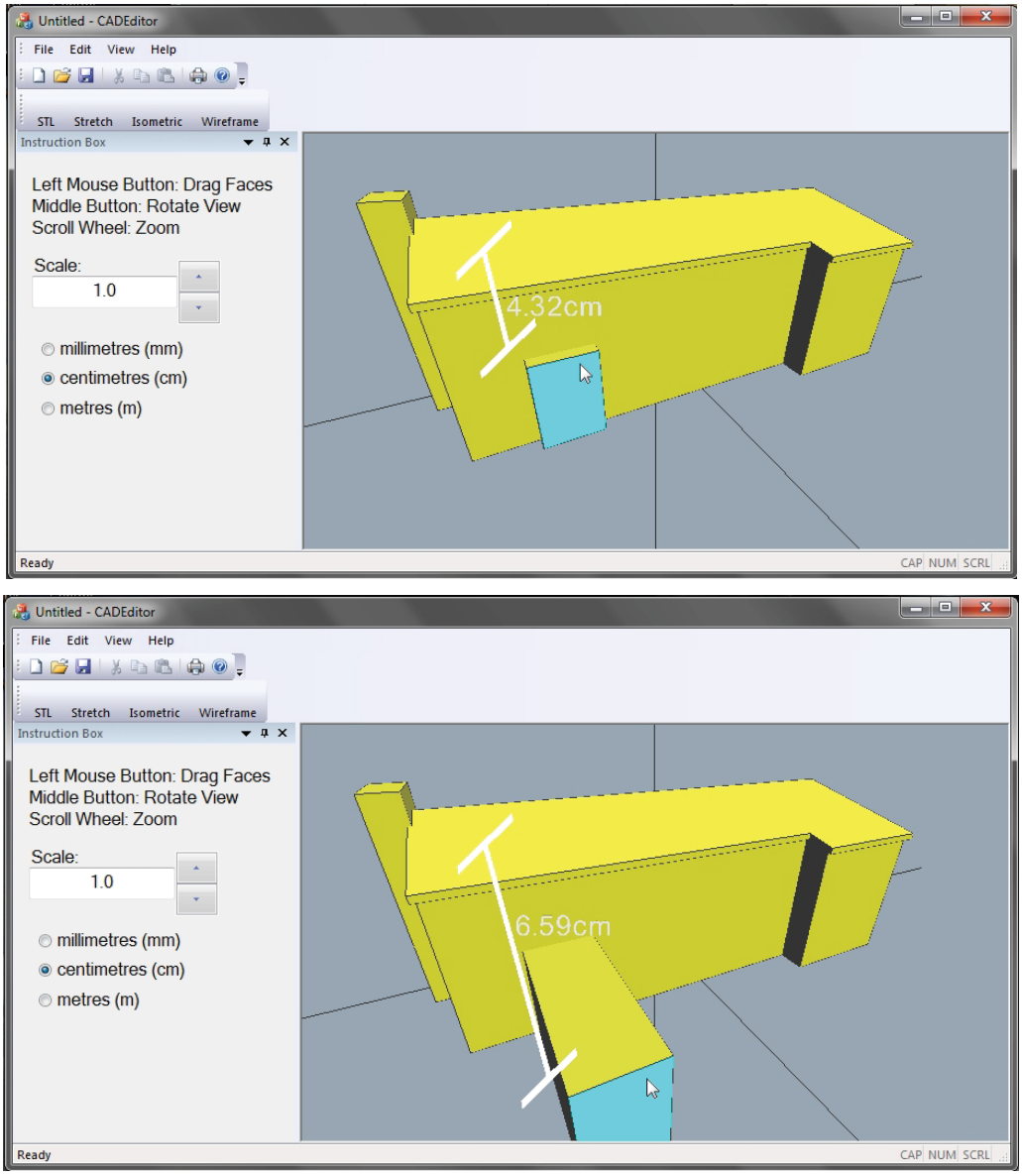


Figure 34. Elongation of feature by clicking and dragging face

The program periodically crashes when dragging faces. Debugging efforts reveal that when this happens, the program tries to access unallocated memory. It was not determined what caused this.

Occasionally, dragging a face would cause the object to form itself in an unexpected manner with totally invalid geometry. It was eventually determined that this was caused by small errors in floating point numbers exponentially accumulating. For instance, if a face was

dragged by moving the cursor 15 pixels, the new position of the face would be calculated 15 times (using the data of the previous position). The errors introduced by calculations in floating point numbers would consequently increase as each new position was calculated. This was somewhat rectified by giving each Face both rendering co-ordinates and actual co-ordinates. The calculations for rendering co-ordinates were always calculated from the starting position (the actual co-ordinates). The rendering co-ordinates replaced the actual co-ordinates when the user let go of the mouse button. The aforementioned problem was not experienced after this adjustment was made.

With this implementation it is not possible to modify models with vertices that do not have exactly three connecting faces. A square-based pyramid for example has a vertex at the top with four adjacent faces.

10.0 Conclusions

The direct editing method used is intuitive and easy-to-use. It has another benefit of being more compatible with touch screens, a technology increasing in popularity.

To be truly useful the application needs to handle curves and circular shapes. The most common approach to general curves in CAD is to use NURBS curves. However, the tests for intersecting faces and limit finding are much more difficult. The functions to determine these need to work for a general case. Dragging a face to change a dimension would have to work in a different way.

It needs some kind of restraints implemented so that certain dimensions can't be changed. This would result in an object retaining a certain shape when something is dragged. If a part needs to conform to certain dimensions to be useful (e.g. a mobile phone cover needs to fit the phone it's designed for), then those dimensions need to be locked somehow. An approach might be to try 'locking' vertices to a certain position relative to another vertex/edge/face.

A "mix n' match" type component with textures and features would be useful. SPORE and UCODO implement this so users can easily customise designs. Part of the original goal was to facilitate any shape that could be imagined and selecting components would result in a finite number of possible designs. However, it might more useful to a user to just select a feature they like and apply it to their design. This could be partly implemented by having 'texture files' that could be dragged on to a face, giving the whole face a texture.

Although the invalid geometry problem (mentioned in the previous section) was not experienced after the adjustment was made, it could still theoretically occur when many modifications are made to the model. This represents a weakness in the use of floating-point numbers.

The use of floating-point type variables is very limiting. Tests comparing two float types that should be equal frequently return false. A proposed solution [5] is to use a class that stores a numerator and a denominator as integers, representing a fraction.

```
class rational{
      integer numerator;
      integer denominator;
```

```
}
```

This has a distinct limitation of not dealing with square roots, which are used in finding the normalised normal of planes and faces. The normals themselves are used extensively in calculations regarding limit finding, intersection testing, dragging and *.stl file generation, among other uses. The class above also does not handle π, which would be needed if true circles were to be implemented in the program. Arbitrary precision math libraries exist, but would not be useful where irrational numbers are concerned. Some kind of math library that can represent √2 would be needed.

If the purpose of the program remains providing an interface for customisation to non-technical users, then accuracy of models might not be important. However, stronger tests for invalid geometry would be needed.

To address cases where four or more faces are connected to a vertex, an intended behaviour must be defined. One option is to 'lock' the faces with these vertices so they cannot be moved.

11.0 Appendices

Appendix A

Project files – see attached CD

12.0 References

[1] Amirouche, F. (2004). Principles of computer-aided design and manufacturing / Farid Amirouche. Upper Saddle River, NJ : Pearson Prentice Hall, 2004.

[2] Baumgart, B. G. (1974). Geometric Modelling for Computer Vision, Rep. STN− CS, 74-463.

[3] Braid, I. C. (1973). Designing with volumes / I.C. Braid. 1974.

[4] Cogorno, G. R. (2006). Geometric dimensioning and tolerancing for mechanical design. McGraw-Hill.

[5] Ghali, S. (2008). *Introduction to geometric computing*. Springer.

[6] Hamade, R. F., Artail, H. A., & Jaber, M. Y. (2005). Learning theory as applied to mechanical CAD training of novices. International Journal of Human-Computer Interaction, 19(3), 305-322.

[7] Hamade, R. F., Artail, H. A., & Jaber, M. Y. (2007). Evaluating the learning process of mechanical CAD students. Computers & Education, 49(3), 640-661.

[8] Hamade, R. F., & Artail, H. A. (2008). A study of the influence of technical attributes of beginner CAD users on their performance. Computer-Aided Design, 40(2), 262-272.

[9] Mäntylä, M. (1988). An introduction to solid modeling.

[10] Miller, R. B. (1968, December). Response time in man-computer conversational transactions. In Proceedings of the December 9-11, 1968, fall joint computer conference, part I (pp. 267-277). ACM.

[11] Requicha, A. G., & Voelker, H. B. (1977). Constructive solid geometry, TM-25 / by A.A.G. Requicha, H.B. Voelker. Rochester, N.Y. : Production Automation Project, University of Rochester, 1977.

[12] Roscoe, L. E. (1988). Stereolithography interface specification. America-3 D Systems Inc

[13] Shapiro, V., Farin, G., Hoschek, J., & Kim, M. S. (2001). Solid modeling.

[14] Stroud, I. (2006). Boundary representation modelling techniques [electronic resource] / Ian Stroud. London : Springer, c2006.

[15] H.B. Voelcker and A.AG. Requicha, “Geometric modelling of mechanical parts and processes”. COMPUTER, col. 10, no. 12, December 1977; also published as Tech. Memo. Mo. 23, Production Automation Project, University of Rochester, October 1977.

[16] commons.wikimedia.org/wiki/File:Csg_tree.png?uselang=en-gb
Accessible as of 14/01/2014.

[17] eigen.tuxfamily.org/dox/group__TutorialLinearAlgebra.html
Accessible as of 14/01/2014.

[18] gts.sourceforge.net
Accessible as of 14/01/2014.

[19] msdn.microsoft.com/en-us/library/windowsphone/develop/jj207062(v=vs.105).aspx
Accessed July 2013.

[20] www.cs.mtu.edu/~shene/COURSES/cs3621/NOTES/model/winged-e.html
Accessible as of 14/01/2014.

[21] www.idc.com/getdoc.jsp?containerId=prUS24108913
IDC, Android and iOS Combine for 92.3% of All Smartphone Operating System Shipments in the First Quarter While Windows Phone Leapfrogs BlackBerry, According to IDC (May 2013)
Accessible as of 14/01/2014.

[22] www.khronos.org/opengles
Accessible as of 14/01/2014.

[23] www.opencascade.org
Accessible as of 14/01/2014.

[24] www.opencsg.org
Accessible as of 14/01/2014.

[25] www.opencascade.org/getocc/download/
Open CASCADE documentation – "Visualisation: User's Guide" – Version 6.5.2 / April 2012. Accessed July 2012.

[26] www.opengl.org
Accessible as of 14/01/2014.

[27] www.plm.automation.siemens.com/en_us/plm/synchronous-technology.shtml
Accessible as of 14/01/2014.

[28] www.plm.automation.siemens.com/en_us/products/open/parasolid
Accessible as of 14/01/2014.

13.0 List of Acronyms / Abbreviations

API	Application Programming Interface
B-rep	Boundary Representation
CAD	Computer-Aided Design
CSG	Constructive Solid Geometry
GLSL	OpenGL Shading Language
MFC	Microsoft Foundation Classes
NURBS	Non-Uniform Rational Basis Spline
OpenGL	Open Graphics Library
Rhino	Rhinoceros 3D
UI	User Interface

File Extensions:

***.amf** Additive Manufacturing Format

***.stl** STereoLithography