THE UNIVERSITY OF TEESSIDE

SCHOOL OF COMPUTING

MIDDLESBROUGH

CLEVELAND

TS1 3BA

CREATING A VIRTUAL WORLD DESCRIBED BY CUSTOM MAP DATA

BSc Computer Games Programming

April 2006

T R D Hill

Supervisor:	T P Davison
Second Reader:	T Nelson

Abstract

This project details the investigation of using map data, to automatically create assets, with the purpose of re-creating the world the map represents, carried out by Tom Hill as part of his Final Year Personal Practical Project.

The project was separated into three main stages. Firstly, the process of how a framework to support map data could be created is followed, resulting in the production of a design to follow to allow for provision of default functionality, while also letting it be replaced by the developer.

Next the process of how to create a three dimensional model from the two dimensional shape of the feature defined by the map is investigated. A feature may not just be the geometric shape defined by the map, and so the idea how extra information called Meta data which defines unique properties of the feature can affect the production of a mesh is followed.

This leads to the production of a system whereupon the mesh is separated into common zones of volume, produced by differing methods. To give a developer an amount of customisation to the production of these zones, extra objects called Zone Decorators are used to apply detail areas to the faces within the zone. These components when mixed provide the potential to automatically create a wide range of mesh types.

The final stage of the project seeks to test the designs set out before, by creating a test application aimed at seeking the perspective of a developer using the framework for inclusion within a visualisation application.

With the implementation of the test application complete, the project seeks to evaluate the design, and predict what testing requirements a real world framework would need if it were to serve its external developers correctly. Finally a collection of further work is recommended to future complement the work of this project.

Acknowledgments

Primarily, thanks go out to Tyrone Davison, the supervising tutor of this project for his guidance and support from beginning through to the end.

Secondly a 'bucket-load' of thanks goes out to a number of people who have put up with my ramblings, offered advice, opinions or even just some motivation at various stages throughout the project, these are in no particular order; Edward Smith, Alistair Parr, Peter Knee, and Tom Salter.

Thirdly much thanks goes to the University of Teesside's Learning Resource Centre which put up with me for far too many hours throughout the project duration.

Finally, but most importantly a massive thank you to Doreen Hill and Laura Stockhill for proof-reading this document which most likely covered material far outside their interest bounds, but still stuck with it anyway.

Thank you.

Preface

This document is centred on the technical side of the creation of a product combining two main subject areas; geography and virtual asset creation. As such technical language will feature within the report. It is intended that the average reader have the basic knowledge of how virtual assets are created, and what maps are, as well as most importantly a basic grasp of programming.

For further reading beyond this document, please consult the Bibliography in *Appendix B*.

Contents

1. Introduction

The world's information is becoming ever connected; this has come to mean that data is fast becoming not only a very important but an extremely accessible commodity. With the various technological improvements that have lead to this current scenario, one driving factor has been the increase in computational power on offer for devices. The power now being offered means that for the first time there is enough functionality to start visualising data that previously would have been too expensive in terms of computational and hardware requirements. It is an exciting time, leading many to find both useful and unique ways to show and link the data, and so producing very powerful and unique analysis tools.

As this trend continues, more quantity of analysis is naturally demanded merely because it is now possible. A higher amount of data can enrich a product and make it seem more valuable. However more data means more analysis and interpretation must be undertaken, which if this process is manual, will increase cost. It is this stumbling block that must be overcome, if applications are to be created that use the data in interesting and useful ways.

Interpreting map data is an excellent example of this. Many computer systems of today look to create virtual worlds, and what better to base those worlds on, than the real world itself? A number of products are taking geographical data and using it in useful ways. Google Earth for example, is currently attempting to create a virtual globe where every street and building can be found anywhere in the world. Not content with just that, the application actually attempts to recreate some major cities into three-dimensional simplified versions of their real life counterparts, as shown within *Figure 1.1*. Whilst in contrast, route finder systems

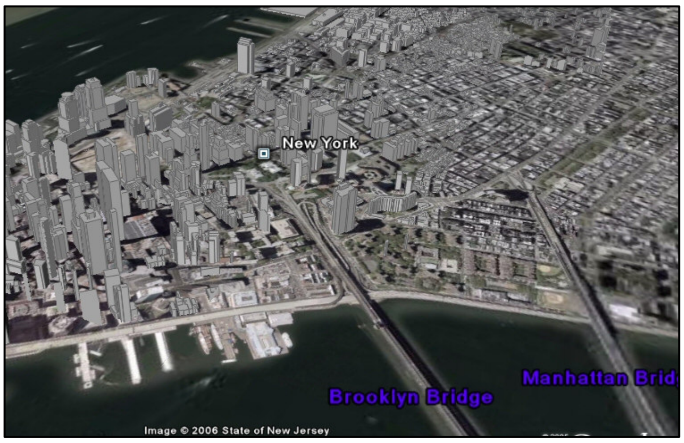


Figure 1.1 - Google Earth's Representation of New York

(such as *Tom Tom Navigator, Figure 1.2*) are using map data to calculate the user a path to follow to their destination.

What is common with these systems is that, as they become more technologically advanced, they are slowly incorporating 'extra detail' features that allow for much more complex representations of their geographical dataset to be created.

Most systems in this category share similar characteristics; they eschew as much manual input as possible, in favour of an automated system that can handle the high amounts of data for them. This keeps the cost of processing the map data to produce a visually pleasing output to a minimum, however many of these systems are quite underdeveloped and offer very simplified versions of the world their map represents.

Almost certainly, the trend for analysis of maps identified will continue, while it is highly likely much better quality representations of the data represented within maps will be demanded. As the technology that allows map analysis gains ground, it will be deployed within a larger range of applications to cross reference data, for a variety of different intents.

Figure 1.2 - Tom Tom Navigator Product

It is the intent of this project to first investigate a solution that could be used by software developers to easily ‘plug’ in map data into their application. Secondly it will investigate how to use this data effectively for the purpose of creating asset data for a three-dimensional world. More specifically the majority of the work in this area will be focused on the creation of three dimensional closed geometric models, known as meshes, from the elements and their properties contained within the map.

To investigate the first intent, the project will attempt to define what map data contains, what elements of this data are fundamentally important for the developer, and how the developer might want to successfully use them within a real application.

The mesh creation aspect of the project will be addressed by attempting to break down the process of creating a mesh into abstract and generic pieces, so that a generic solution can be found.

The fundamental areas of the project will be tested via an application created specifically for the project that will read in example map data and output a series of meshes representing the map's world. The success of this application, which will build upon the analysis and design described within the document, will be seen as a proof of concept for the overall ideas and techniques suggested within.

This document will begin by detailing the methodology applied to the work carried out during the project period. Next the findings of the analysis of the problem will be presented, and will move onto any required research in response to that chapter. From there, the design section will attempt to draw all of the previous information together to produce a template upon which to build the proof of concept application.

At this point, the application's test processes will be described, which will follow onto an evaluation of the system to access any successes or failures found within the solutions created. Finally, the document will present a collection of recommendations to the reader.

2. Methodology

2.1. Overview

The work for the project was centred on the production of a solution to the two problems defined within the Introduction Chapter. Work flow elements were split into four sections, described within *Figure 2.3*.

1) Produce a design for a framework for a developer to integrate map data into this application easily.
2) Produce a design for a system of asset creation, giving a number of parameters.
3) Produce the implementation of the systems described in (1) & (2).
4) Produce an application which mirrors the role of a developer, to test the systems in (3).

Figure 2.3 - Project Goals

2.2. Stage One & Two - Pre-implementation

These stages concerned themselves with the production of a framework to process map data. The framework would take into consideration developer requirements of a third party Application Program Interface (API).

1) Analyse the problem.
2) Research into background.
3) Generalise into core components.
4) Design each element.

Figure 2.4 - Stages of Work Process

Each sub-stage of the work process seen within *Figure 2.4* logically leads onto the next. This allowed for a linear approach to be taken for the pre-implementation work, and saw the production of both a design for a map framework, and a solution to asset creation.

2.3. Stage Three & Four - Implementation

The execution of these stages followed a more cyclic approach. As the implementation of the designs in Stages One & Two was constructed piece by piece, each element was tested via the test application required by Stage Four. This meant that the construction of the test application was directly reliant upon the construction of the design, but allowed for an approach which allowed immediate evaluation on each element of the implementation to see whether it worked, which encouraged the evolution of the system to incorporate feedback from a developer perspective.

1) Implement element of Stage Three.
    a. Implement Testing of element for Stage Four.
    b. Evaluate, if changes need to be made to element, do so.
2) Move onto next element.

Figure 2.5 - Work Process

3. Analysis

3.1. What does a Map contain?

Before representing maps within a software product, it must be understood just what information is contained within them and what information is essential to be held within a collection of data (or dataset) for it to be considered a map.

Looking to maps, it must be known what they can contain, so that an image of what data to expect can be created. The process to achieve this, can begin by observing what types of data is represented on a map. Looking at the range of represented data types within *Figure 3.6*, it can be seen that there may be some form of similarity between them all, so that an abstracted definition of a map can be formed. Every piece of data could be thought of as a property of the map, or as a feature of the dataset it represents.

- Buildings
- Relief
- Rivers
- Lakes
- Roads & Road features
- Stations
- Forests
- Churches
- Houses
- Bus routes
- 'Average traffic' data for roads (used by route planning software)
- Height of buildings
- Addresses
- Type of vegetation in area

Figure 3.6 - Typical examples of Map data

A map shows very little data that could not be contained within these features. In fact the map itself is really just showing how all of these pieces of data, representing objects in the world, are positioned in relation to each other. This positional data of the objects could even be thought of as a property of each feature and not a property of the map itself, therefore it can be deduced that the term 'map' means 'a collection of different types of features'. The map itself has very little data that it owns.

In *Figure 3.6* there are some data types that have no real visual representation on the map. For example, the height of buildings could not really be accurately conveyed on a plan drawn representation of a town. Data types such as this would need to be considered as properties of the features they are linked to. In this way the features could still contain information such as building height, but just not show it in an immediate way. Of course on some maps, the solution to this problem would be to simply draw a numerical figure on top of the feature that detailed its height. There are very few types of data listed within *Figure 3.6* that could not be considered either a feature, or considered a property of another feature on the map.

3.2. What Applications Might it be useful to use Maps within?

Before creating a solution to use map data, it would be useful to ask what this map data might be used for. The answer to this might contain clues or constraints as to how a solution would be created.

As map data is, by definition a representation of a world, the simple answer of what they are useful for, could be any application that requires data from a world. However this is not a clear answer. Maps are the alternative to creating a perfect copy of the world to be represented, they simplify the features within the world and store the key properties needed to identify the feature's key characteristics.

Maps are not always used to describe the real world. They might be used to define a world that does not exist. To use a map might be especially useful for applications creating the virtual worlds existing in Games, or Computer Graphics' work.

These disciplines typically have considerable artwork requirements, if a map could be used to initially define the environment, which could then be processed and the environment could be automatically visualised by software, this would save a huge amount of time within the asset creation timeframe.

If the map itself was kept simple enough to be able to be easily edited, then changes to the environment's make-up would be simple, and quick to make, further helping the creativity process.

It is also important to remember that the map itself does not just define geometry. In the case of games, a huge variety of features could be defined. Properties of the game such as Non Player Characters' movement paths or level goals could all be defined within the map. Not to mention the geometry of the levels within the game, being able to be easily edited and changed, promoting a faster throughput of asset modification.

These examples of uses mean that expandability for differing functionality must be strived for, while also generic definitions of features within the map will certainly be needed, as a feature may actually represent something that cannot be previously anticipated. The map must be thought of as a blueprint, able to define anything of the world it represents.

3.3. If the system was for a software developer, what exactly would they want to do with the data anyway?

All uses of map data by software developers would rely on first providing easy access to the map data itself. This translates to being able to let developers easily import and query the map dataset. A logical, easy to understand interface for any tools provided is a must, but this is also a requirement of any software system, as a developer cannot successfully use a tool if they do not understand it.

Taking these points further, while combining them with the nature of our dataset, a developer would probably like an easy way to be able to differentiate between map features; therefore these map features seem likely candidates to be their own specified objects within the framework. The developer would likely want to query each of these features for any properties associated with them, as well as extract their values.

This one property of the framework may prove to be rather difficult to abstract as the data itself could be anything, from a name or address, to a quantitative piece of data like a height value. From this it seems that the only mutual property all of these have in common is that they are 'pieces of differently typed data', which add detail in some unspecified way, and are associated with the feature.

The last point to be made for the developer’s requirements is that the developer will likely want to produce some form of output from the map data after they have processed it in the way they envisaged. This output will very likely need to be in a form that can be customised according to the application. This suggests that, if needed, the developer should be able to introduce their own functionality into the framework’s processes to allow them to get the results they want.

3.4. Maps need to be simple and abstract in concept, then they can be pragmatically represented

Features on a map need a way to be abstracted, and simplified, so as to not only make the implementation of a framework easier to produce, but to increase the usability aspect of the software from a development perspective. Therefore, the more generically constructed the representations within the framework, the more situations they can be applied successfully to. This will result in a higher usability factor for the framework.

3.4.1. The Basics of a Feature

To begin this process, it would be useful to know what properties a feature of a map may have, and which of these are common amongst all different possible feature types. If this bare basic representation of a map feature can be abstracted successfully, then the pragmatic representation of the map as a whole will be far simpler, as we have already defined a map as a collection of features.

All features would seem to have a shape to define themselves within the world, and then a set of data elements of unknown size which define their unique properties (which are not shared between features, for instance a tree may have a 'leaf size' property, but a building almost certainly does not).

Only one potential conflict with defining the map as a set of features arises, and that is when considering the actual relief area itself. Is this a 'feature' or is it part of a 'map'? Whereas other features would normally have a shape that is mapped onto the underlying relief below it, the whole area of the map defines the relief. Does this make it a special case, or could it still be considered a separate feature? Considering that the only thing that differentiates a feature and relief is the height data, it is a feasible jump in logic that a feature should also be able to contain height data if it is available. Terrain could then be treated just like any other feature within the map.

1) Features have a position within the world.
2) Features have some description of their shape / dimensions
3) Features have some sort of type to describe themselves (house, tree, grass etc.)
4) Every feature can have a number of unique properties describing data about themselves
5) Features could have height data attached.

Figure 3.7 - Per-feature Data

3.4.2. Simplifying the properties of a Feature

While looking at the list of common properties in *Figure 3.7*, both *Points (1)* and *(2)* describe location based data, *(1)* could be made redundant by simply incorporating it within the shape definition of *(2)*.

Points *(3)* and *(4)* again are open to debate as to whether they can be combined. Is it correct to force every feature to have a type? Perhaps the map only describes one type of feature, in which case type data would be made redundant from a feature definition. From a developer perspective too, is it correct to force a type of the framework's devising onto them, to then have questions arise as to how they will fit their own type into the range of pre-defined ones? This suggests that type data should be an optional property, and so would be part of a piece of custom information. Therefore *(3)* could be considered part of *(4)*.

With point *(5)*, it too could be argued that it is a custom piece of data to be held within *(4)*. However, values will always be numerical representing the height at a specific location within the map. This situation is different than that of *(3)*'s merge with *(4)*. Type data could be an integer ID, or it might be a type name stored within a string. As *point (5)* is actually describing the geometry of the feature, it would make sense to make *(5)* an optional property of the shape definition *(2)*.

In this case, it is recommended that height values across the shape of the feature be considered a standard part of every feature. The height values are taken from point samples within the map. As the type of data of these height values can be predicted (it would be a single numerical value), the framework can support them as standard. This is also an opportunity to take a value adding opportunity in relation to the framework's functionality. If functionality is added that can allow the developer to easily process these values, perhaps being able to access a model of the surface they describe would significantly simplify the process of using the data. This analysis transforms the definition of a feature to that described in *Figure 3.8*.

1) Features have some description of their shape / dimensions.
    a. This shape data can have height data attached.
2) All features have a number of unique properties describing data about themselves.

Figure 3.8 - Revised Per-feature Data

3.5. How can a system support 'any' form of data?

To let the developer change the type and elements of data contained on each feature of the map to comply with *Point (2)* of *Figure 3.8* is a useful capability of the proposed framework. To be able to support this data within the framework, the size, type, or even the number of elements of data within it must be known. If these Meta properties of the data are left unknown, it will be impossible for the framework to allow a straightforward query of its values.

These values cannot be accurately worked out by the system alone, as it is impossible to predict what data a user will have entered for each feature. This translates to become a new requirement; any data for a feature will need to be considered Meta data, by having a description of what data types it contains accessible.

3.6. How would the map data itself be stored?

Traditionally maps are thought of as paper based, and this has directly translated to them being represented as images on computer systems. In this representation, as far as the computer is concerned, it has an image file, and not a piece of map data. For the computer to be able to start recognising information from the map, it must follow the same process as humans do - recognise each feature as a separate entity. This suggests that the system needs image recognition abilities, i.e. to develop map reading skills comparable to a human's, or the data is marked by some other way to allow the computer to recognise the map data.

Obviously from a system perspective, to directly translate a map in image format into each of its separate features would be an intensive process, presumably requiring much shape recognition processes to be carried out on the image by the system. The alternative would be to manually mark-up the information, effectively making the human do the complex tasks of feature recognition.

As the point of the project is to try to decrease asset creation timings, this is an unacceptable scenario when handling maps stored as images. However, for the scope of the project, creating a complete shape recognition solution for marking up features on maps stored as images would be outside of its scope. Therefore a compromise must be created for the project, where it will support map images, but not to the extent that it will be trying to recognise complex feature types. If the images are simplified to solid colour shapes, with no text labels, this should decrease the complexity of the overall image, and make marking out the relative features far simpler for an algorithm to accomplish.

In relation to input files for the framework, to use map data previously marked up, or to mark it up manually while processing, is a key issue of map creation decisions for the framework. If the raw sample data of the mapping process was exposed, then the marked up version, which from here on in will be referred to as the 'vector' style, will be more feasible to use. However, if this raw data is not available, then the alternative will be to use image data (herein referred to as 'raster' style), as this type of map will demand less time to create initially, but require a potentially complex process to mark it up.

3.7. Raster or Vector Map data

Without considering whether they can be supported due to the issues described within the previous section, raster and vector data style maps must be more precisely defined. When a 'vector' type map is talked about, it is thought that the map, as a dataset, is marked up into its various features with their properties correctly. For raster map types, this is not the case, the connections between properties and features must be made by the reader of the map, as the data is stored as just a visual representation of the map area.

Using these definitions, it is preferable to use vector type data in regards to the framework, as this is the type of data that precisely defines each feature within the map. Little pre-processing is required as opposed to the raster type data, which is data that defines each feature within the map visually and may not even do this as precisely as the vector option. This is due to the encoding process of storing the data within the image file which has limited resolution with which to store the range of data. Whereas the vector format can provide support to precise positions, the raster will only allow positions at the set pixel points within the map area.

So not only does vector type data provide the computer more knowledge of what the map contains, when its Meta data carrying properties are taken into account, it will also be far more precise for feature shape definitions. For the developer, this offers increased accuracy and a far richer dataset to play with, a much preferable choice.

With these issues in mind, some points should be kept in mind for the project, defined within *Figure 3.9*.

- There can be more than one type of input to the map system.
- Vector map data should probably take a prominent role within the framework, as it allows better usage by the developer.
- Perhaps a 'raster' to 'vector' data convector would be useful for situations where map data is encoded in raster format.

Figure 3.9

3.8. What sort of issues could automatic asset creation come up against?

Once a developer has raw data from the map, there are quite a number of different issues that they will immediately be faced with when trying to produce a form of output that mimics that of what the map represents. The immediate concern is how to transform a feature, currently defined as a shape on the surface of the map, into a representation of what it is in the real world.

In the case of the creation of a mesh for a basic building, while the base shape of the mesh is fully accessible as it is part of the standard feature definition as defined in *Figure 3.8*, how is this two dimensional shape built upon to become a three dimensional mesh?

A basic solution to this problem could assume that the building’s height information would be part of the feature’s Meta dataset. Using this, the mesh could be formed by extruding this base polygon upwards to the height defined within the Meta data thus a basic building shape could be defined.

From this basic shape, it should be possible to expand upon what the feature should look like in three dimensions by adding extra detail. Using some extra Meta data values that define the properties of these details would be an excellent use of the custom data properties of features. Perhaps even if these were not defined, they may take randomised values that conform to a range of values allowing even the custom data to be optionally defined by the user.

While it might be very simple to just extrude a feature's shape upward by a defined height, this does not take into account scenarios where a feature’s true shape is variable throughout its height. The feature's map shape might be its visibility from a plan view above, hiding its true shape at the base. The reverse situation could also be true.

The framework must not consider that features have constant dimensions over the range of its height. In the same vein, a feature may not even have matching appearance when viewed from different angles around its circumference. Mesh creation must therefore make differentiation between the faces of the base shape and areas of height of these faces.

These observations lead to the requirement that custom functionality must be supported within the mesh creation process at a number of levels.

- The base shape of a feature could be used at any point of the creation process.
- Meta data of a feature must be available to be able to influence the creation process.
- Faces and areas of height must be differentiated between within the creation process.
- All areas of the system (from properties of data, to functionality) should be able to be customised.

Figure 3.10

3.9. Requirements of a produced mesh

Taking a step back from the mesh output, it should be questioned as whether there are any requirements on how to form the output. It goes almost without saying that the mesh would be much more use if its surfaces are tessellated into triangles.

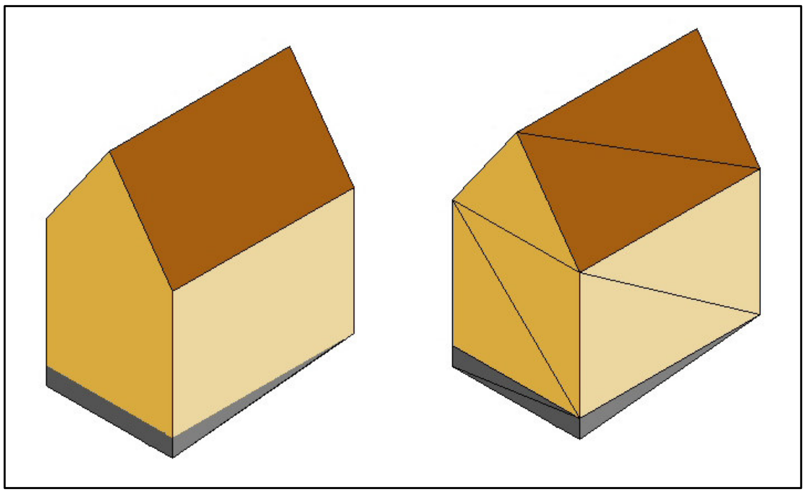


Figure 3.11

Tessellation of polygons is a required step within the creation of the mesh (*Figure 3.11*). Simple convex polygons might well be simple to tessellate, but seeing as any shape of feature is to be supported, this suggests that non-convex polygons could become an issue, and certainly an issue that developers would likely want a solution for.

3.10. The Process of Conversion from a Feature to an Asset

Aside from the exact process of how the volume of the asset is derived from the feature, if this process is to be defined as expandable, it will be useful to define, from a developer perspective, the overlying process.

Many different types of features might want different types of meshes to be created. As mentioned above planar dimensions cannot be guaranteed to be the same throughout the range of height, and so this suggests perhaps the mesh creation will need to define the creation of these differing areas as separate creation processes. If they are separate, then they are viable for overriding from a developer.

This could hold true for any areas of detail on meshes, as in the case of a simple building. First its framework could be created, which is merely a mesh defining its volume, then the surfaces of the mesh that depict the walls of the building could be filled with details such as windows, signs or guttering, to help define what the mesh is supposed to represent.

The ability to add detail areas is a very important requirement of the system. Any meshes created could be aimed at anywhere between very simplified to very accurate representations of the world the map represents. The extreme case would be to reproduce the real world to an extent where the mesh and feature are no longer distinguishable. Allowing developers to be potentially able to add an infinite amount of detail therefore is a requirement of the mesh creation system.

Of course reproducing an automatically created photo-realistic mesh will be beyond the scope of this project's investigation, but the underlying feature set will need to be produced to at least allow the possibility of this outcome in future. The ability to create a base mesh which could then be manually edited further would also be seen as a success, as this would help hit the project's aim of decreasing asset creation time.

Whatever the implementation eventually produces, it must consider a mesh to be broken down into separate construction elements, which are then able to be replaced and reused to the developer's prerogative.

4. Research

4.1. Storage File formats

There have been two different data types that have been defined as able to store map data.

Raster data would obviously be stored within an image format, but which? Well it is questionable as to whether this choice matters as long as the map information's data, i.e. the shape of features, is not lost within the encoding process.

Vector type data, due to its highly expandable properties would need to be contained within a file format that also contained a description of the data described within it. As described in the Analysis section above, this is a requirement of the Meta data capabilities.

Before discussing what file formats would be supported by the project's implementation of the framework, it would first be useful to look at other professionally produced mapping software.

4.1.1. Ordnance Survey Master Map

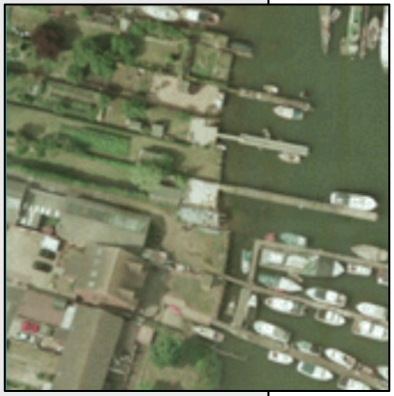
At the forefront of examples of a professional application using map data storage is the Ordnance Survey (OS) Master Map solution. The maps for this solution are defined as a marriage of all of the Ordnance Survey's datasets. These different types of data are defined as layers, with the Master Maps themselves split into four distinct layer groupings.

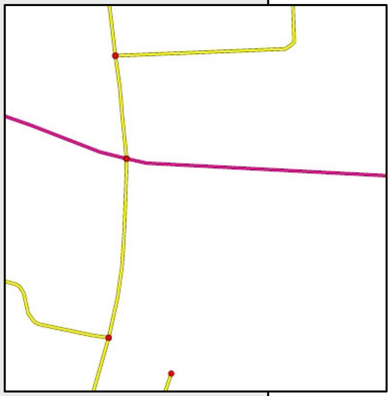
- **Address Layer**
  Essentially this is a database of every address in Great Britain (originally taken from Royal Mail's Postcode Address File, PAF®), but with one useful inclusion - a precise map location of where that address is on the map. This then allows for other layers to link up the address information to specific features contained within the map.

- **Imagery Layer**
  A collection of colour aerial to-scale images of the area contained within the map.

- **Integrated Transport Network (ITN) Layer**
  The ITN layer contains data on the road network within the map. Each junction is a node at which multiple roads can join together. The network is able to be queried to generate route planning information, and as each road contains properties such as traffic calming measures, bridge height limits or whether they are one way only these can be used within the calculations too.

- **Topography Layer**
  The topography layer is what should be considered as the 'traditional' map layer. It contains all of the data normally expected of a map, with features such as buildings, height data, roads, land and administrative boundaries. All of these features are accessible in this layer, allowing to be cross referenced with the other layers provided. Each feature is considered its own entity, allowing them to be filtered and selected with ease.





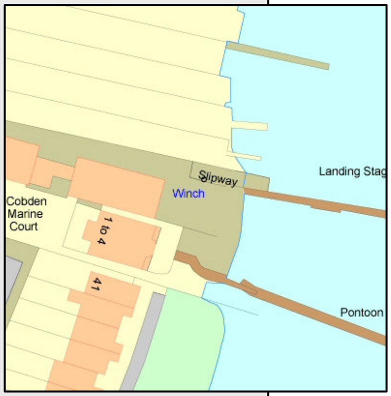


Figure 4.12 - OS Master Map Summary

Considered separately, each layer of the OS Master Map described within *Figure 4.12* presents a detailed set of data. However its real use is in how it allows connections between these different datasets to be formed. For every piece of data, there is an accompanying locator tag, which holds a unique identifier. By using this identifier, the information for features between datasets can be cross referenced. The system on a whole is given a massive amount of usage value onto its feature set by allowing this. To use the OS Master Map as the starting point for this project would seem at first highly beneficial, as the map data conforms to the analysis of the vector type format described previously.

However, there are a number of factors which prevent this. Firstly, although the data is raw (and potentially extremely useful), it is questionable as to how appropriate it is to use it for the nature of automatic asset creation for the project. The data would contain a high amount of detail that is unlikely to be used within the asset creation aspects of the project to actually affect the outcome, as to do so would require introducing the same amount of functionality. This would likely be too much of a labour intensive task for the scope of the project.

Ultimately however, the most important factor in the decision of whether to use the OS Master Map system is that in order to gain access to the digital map data, the project would require either a license holder membership to the OS, an extremely costly one off payment for limited use of a specific area's map data, or finally through a subscription to an Athens package, to which the University does not currently subscribe.

These two options mean that usage of the OS Master Map system is really limited to just observing how specific areas in functionality are handled by OS.

4.1.2. Geography Mark-up Language (GML)

The most important and poignant issues from the project's point of view on the map data initially is that of how map data is stored, as this will dictate the initial functionality needed to be implemented to use this data. The OS Master Map software has obviously encountered some of the issues outlined previously in the Analysis section of this document as regards to Meta data. The major problem of how to support multiple types of features, all with different definitions of how their data is stored, seems to have been solved by the OS Master Map solution by using XML to store all of the map data.

However, upon investigation this implementation is not of the OS' own doing. They are using an open standard of geospatial description called the Geography Mark-up Language (GML). GML is a set of XML Schemas which describe basic types of features able to be represented within map data. The important point of GML is that it is an international standard which currently seems to be only supported in a

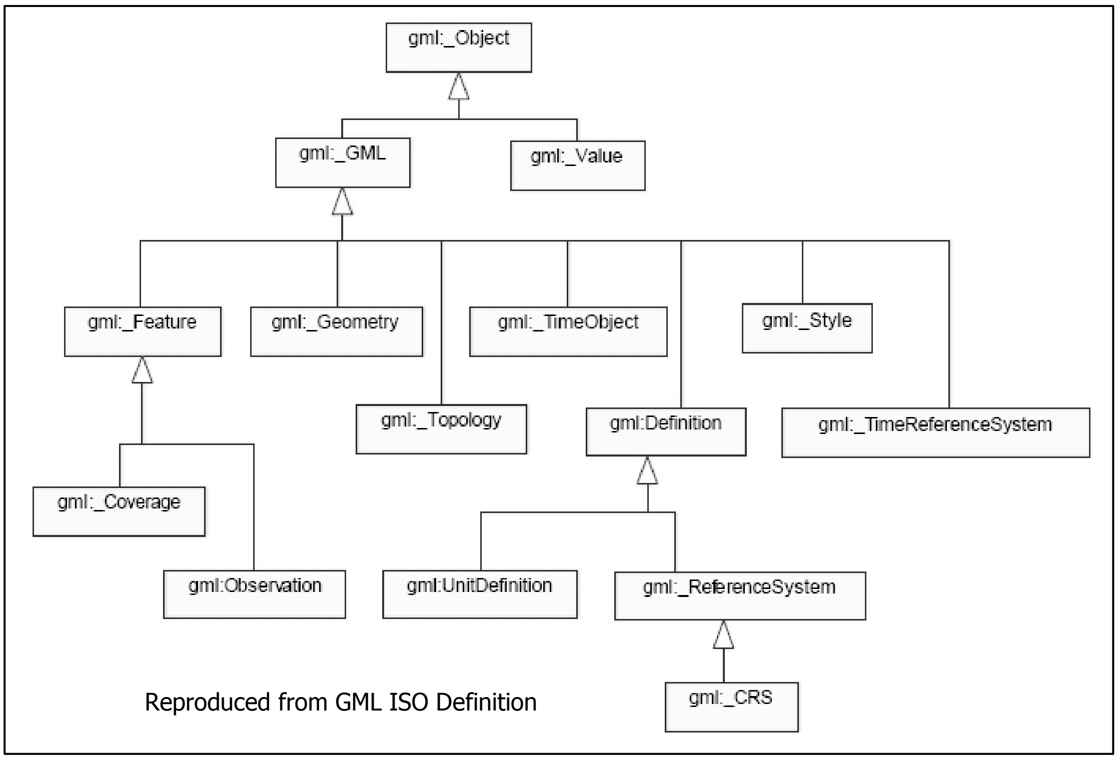


Figure 4.13 - GML Object Hierarchy

significant way by the Ordnance Survey and some scientific research outlets. Its existence is warranted by the aim of being able to have a consistent format to map data, so that it may be easily imported and exported by a wide variety of applications.

The GML standard defines a set of object hierarchies to which any implementation must adhere. Essentially it is a design for when abstracting map features into common properties. As visible within *Figure 4.13*, GML separates map data at the top level into a number of different types, but most remark-worthy are Features, Geometry, Topology, and Time Objects. These are discussed in more detail within *Figure 4.14*.

This wide range of definable features suggests that GML has been created in mind of encompassing all known possibilities when it comes to representing map data. While this immediately seems to fit within the scope of the project, questions must be raised over whether the definitions themselves are too much for the scope of the project.

- **Features**

  Features within GML are defined as the information of any specific feature on the map. Inside them is stored all sorts of Meta Data that the feature contains, as well as unique locator and identification tags.

- **Geometry**

  Geometry is the physical geometric representation of any feature on the map, whether that be a collection of straight lines used to define a border, a circle object to define a roundabout, or any number of vector representations such as curved lines used to create faces of a shape.

- **Topology**

  Topology is the representation of how the geometry contained within the map is collected together. In this way, edges between adjacent features can be shared, decreasing the overall size and cost of the database, and allow easy editing of the bounds between features.

- **Time Objects**

  Time Objects are information sets that are representative of data of a specific time period. Differing sea levels or seasonal data for example could be included via a Time Object.

Figure 4.14 - Analysis of GML Base Objects

For the purpose of the project, will examples such as Time Objects really be needed? Will all the other various data types etc. actually be used within the project? Let alone the underlying point that GML is just a file format standard, it does not provide a way to actually read the files in. If the decision was made to work to the GML specification in regards to map data storage, it might very well mean an increase in work to get the system to conform, for very little overall gain seeing as map data held in GML form is currently unavailable for use. Whether it is used or not, it seems the project will possibly need to make its own maps, and so it would save many implementation headaches converting to and from the GML format as opposed to a custom format.

4.1.3. Raster data

Considering vector map data will be difficult to access for the project purposes, it may actually be a more profitable exercise to try to mark-up common image based maps into a form more akin to the developer friendly vector format. Raster images are image files that contain an indiscriminate number of pixels. Each pixel holds a value reflecting the colour the pixel is meant to take; these pixels are rendered by the computer in order to produce an image on screen.

As their design dictates that they only hold colour values, they naturally have a maximum limit on the amount of data they can carry, which is most typically a Red, Green and Blue channel per pixel (RGB), with an optional Alpha channel making an appearance within some image formats (RGBA). With this in mind, they will be highly unlikely to natively support the Meta data requirement previously laid out. With some arrangement of data, they may be able to hold a basic dataset of information, and then the application could rely on an accompanying solution to the Meta data.

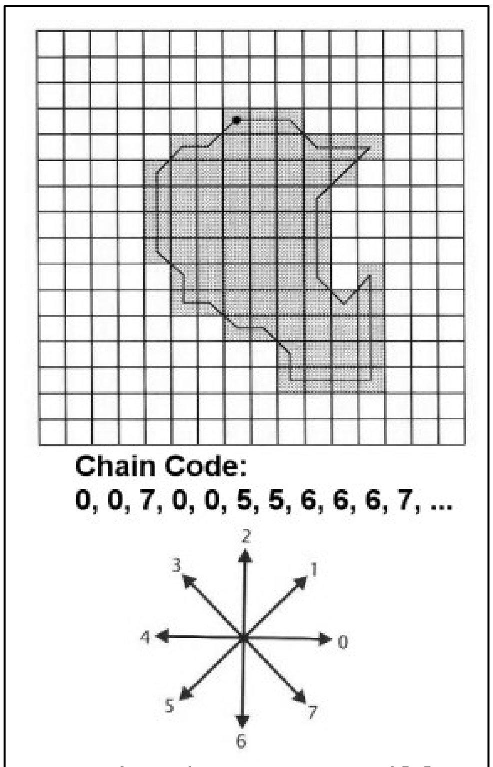
There are a number of options for deciding what data the images will hold. Firstly, maps contain shapes; these shapes are usually defined by lines that signify borders of a feature shape. Sometimes the feature shape will be shaded; sometimes it will just be an outlined shape. Traditional maps such as the OS maps would sometimes have markings for names of areas overlapping this essential data. Therefore, even though the shapes are there, they may have overlapping shapes that too define a feature.

There are two important points to remember about raster information; firstly a raster image has no connotations of where or what a shape is, it is just a collection of colour data. Secondly the image has a limited resolution as to what its accuracy can be.

4.1.4. Techniques for raster processing

The basic problem of converting raster data to the vector data needed is how to find the geometry of features represented in the map. There are many image recognition techniques for shape recognition, but our requirement states we must be able to know what the overall border shape is. Therefore we need edge information for all of the shapes in the map, so this could suggest that we need to process each edge one by one for each shape.

Research for this problem went into many areas of image processing, and pattern recognition, until inspiration for a solution was found. One way of storing two dimensional shapes as parts of images is to store using a boundary based shape representation (Davies, 1986 [1]). Essentially this uses what is referred to as a chain code which details the movement direction from one pixel to another. To store the chain code of a shape, all that is needed is the starting pixel, and the chain code that details the path of its boundary.



From 'Digital Image Processing' [2].

Figure 4.15 - Chain Code Shape Storage

What is not described is how to produce a chain code from an image. If the chain code could be generated from a raster image, it would be a very simple step up then to generate a vector type shape for the feature. Implementation for mark-up of a raster map should investigate this possibility.

4.1.5. Defining multiple features

Defining one feature within a raster map is simple. A background colour could be set, and all different regions of this colour within the image could be treated as a feature. Multiple features that were neighbours could then be differentiated against by using differing colours. The image itself would be segmented into regions, with each region representing a feature *(Nevatia, 1986, [1])*.

With the available channels Red, Green, Blue and Alpha (RGBA), one or possibly two at minimum would need to be taken for feature definition purposes. The 'alpha' channel is made available for extra data to be stored per pixel that may not actually be drawn with the RGB values to screen. Traditionally it is used as a transparency mask when rendering the RGB values and as such within image manipulation applications such as *Adobe Photoshop®* is only visible within a specific viewing method.

Image manipulation applications will traditionally not render the image with a visible alpha channel, merely use that channel to control how it renders the RGB channels. In this vein, it is traditional for height map data to be stored within this 'greyscale' channel, which can describe the height value at specific points on the map. From an editor standpoint this is quite acceptable, as the output image is almost treated as two separate entities, one as a set of RGB channels and one as the Alpha channel.

This idea of treating the alpha value of pixels as height sample values of the terrain, ties in with how GML handles height data. It too offers a collection of height set points that contain the height level at a specific location. The raster image in this case would be describing the height at every point possible within the image; at a per pixel level.

4.1.6. Vector Data

Vector type data has already been defined as the easiest data format for a developer to process. Vector data, when talked about in relation to geometric shapes, would revolve around definitions such as solutions for lines, curves, circles, rectangles and any other geometric shapes used. This again is mirrored within the GML format. Vector data itself fits with the 'any' kind of data previously laid out within the analysis section, and so perhaps if stored as raw data, it too would benefit from being stored in a similar way to Meta data.

4.1.7. Meta Data

As described within the Analysis, Meta data for the map would require a storage format that also contained a description of the data within. The vast majority of file formats are fixed in their specification, and so do not allow for the type of expansion needed. Requiring a description of the data held within the file, sounds very much like a property of the Extensible Mark-up Language (XML).

XML allows any data to be held within it, but requires that each element of data be described. This would allow the system to immediately support any number of developer customisations.

While programming languages such as C# and Java have freely provided interfaces to read in XML data, some languages such as C++ have comparatively less successful methods. In the case of C++, there is no language standard implementation, and while there are a number of third party solutions many have requirements to use Dynamic Link Libraries (DLL), or other such devices which would make distribution of the framework to developers (and potentially their clients) more difficult, as checking of the existence of these components would need to be carried out before using them.

XML parsers fall into two categories; The Document Object Model (DOM) approach, which keeps all of the hierarchical information of the data described (one data element can be the child of another and/or a parent of another), and the Simple API for XML (SAX) approach, which simply reads off data as it is found within the file [6].

For the benefit of the developer using the framework, a DOM orientated approach would be more useful, as it would allow the extra option that Meta data could be considered part of a hierarchy, with elements being children of other elements - this relationship data would not be lost, but would also be easily accessible.

With a C++ environment though, the large question of how this XML would be read into the application still remains. After searching around, it was found that *Tyrone Davison's TXF XML* file reading API built for students of the *University of Teesside* was most suited to the needs of the project. It retains the hierarchical information of any XML file, provides a simple set of clearly named functions, while also requiring just one DLL that can easily be packaged together with any framework based application.

4.2. Shape Tessellation

Once the feature shapes are within the framework’s data, developers will need to be able to easily access and process them. Whilst processing them for the cause of outputting a mesh, it will be a common request to provide a triangulated representation of the shape held by features. While this is a relatively simple process for convex polygons, it is not the case for non-convex polygons.

4.2.1. Convex Polygon Tessellation

A simple convex polygon can have a simple algorithm applied to it to split it down to a number of triangles (*Berg, 2000 [3]*). A vertex is chosen within the polygon, and then diagonals are drawn to every other vertex within the shape, bar its neighbours. This is shown within *Figure 4.16*.

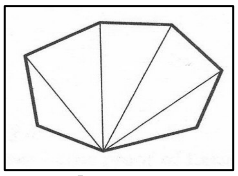


Figure 4.16 - Convex Polygon Tessellated

4.2.2. Non-convex Polygon Tessellation

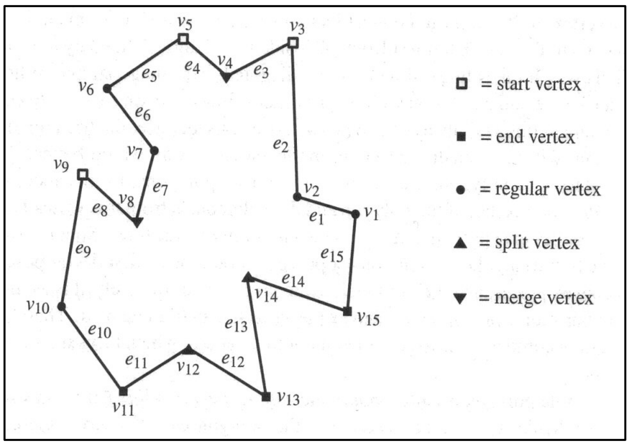


Figure 4.18 - Vertex Mark-up of a Non-convex Polygon

Non-convex polygon tessellation is considerably more complex than that of its convex polygon counterpart. Many solutions strive to convert the non-convex polygon to a number of convex polygons first *(Berg, 2000,[4])*, then use the simple tessellation approach already defined. However by far the simplest method found, is to break the non-convex polygon up into Monotone pieces (*Lien & Amato, 2004, [3])*. The technique to break the polygon up into monotone pieces is far simpler than trying to produce a number of convex polygons to then tessellate. It begins by marking up the vertices of the polygon into different types (*Figure 4.18*) then using this type to split the polygon into y-monotone pieces. A y-monotone piece is defined as a polygon of which, if the edges are traced from the top most vertex downwards, the edge does not travel back

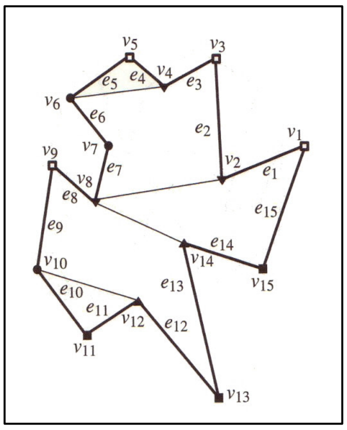


Figure 4.17 - Non-convex Polygon Broken into y-Monotone Pieces

in an upwards direction at any point. This approach even allows for complex non-convex polygons (polygons with holes within them) to be tessellated.

Once the split into y-monotone pieces is complete (*Figure 4.17*), tessellation of the polygon can occur. The tessellation process of the individual y-monotone pieces themselves is comparable in complexity to that of a convex polygon, so overall this allows for the simplest solution found for a generic polygon tessellation algorithm.

4.3. Testing bounding with Feature shapes

Developers may need to carry out intersection tests of one shape with another. While shapes are a series of edges, at first it might seem applicable to carry out plane intersection tests with each individually. This technique will only work with convex polygons however. To support non-convex polygons, it is recommended that the shape be tessellated using the technique described within *Section 4.2.2*, and then any intersection calculations be carried out on the resulting set of triangles.

4.4. Tools choice

The framework will need to support inclusion within a wide range of applications, which may be across a multiple number of platforms, and in a variety of languages. Supporting this range of requirements is a huge task, and so for the scope of the project, the primary platform of *Windows XP* has been chosen. This should allow the maximum number of applications to be able to use the framework. While the framework could no doubt be implemented within any development environment, for the project, C++ will be used due to the higher experience level of the implementer in this language. Programming will take place within *Microsoft Visual Studio .NET 2003*.

The final deliverable to developers could be developed as a DLL, which would allow for smaller sized executables from the developer and would abstract the developer and framework source code completely. This option means whichever computer system the developer's application is run on, will also need the framework's DLL available to the system to be able to run correctly. It is deemed that this is again too large a drawback, and so the project will aim to produce a library file, that can be linked at compile time with the developer's application. This option is far simpler for the scope of the project, at the minor expense of some potential redundant code within the application.

5. Design

5.1. The Underlying Framework

So far, a number of points have been collected about the structure of the software application framework for the map reading application. Expandability is at the core of all design considerations, closely followed by the ease of use of the product from a developer perspective.

To meet the first requirement of expandability, the design must be modular. However to meet the second consideration, modularity must be carefully produced, it must not modularise the framework for the sake of modularising it. It must also not make modules too large. To provide functionality to the developer, but for them to take advantage of that and be rewarded with having to re-write functionality that was provided in the default place but is now not available to them, is just as large a design failure, as to not provide expansion opportunities at all.

Therefore default functionality must be provided that would be of use to a developer, but this functionality must be cut into small scale modules, of which a developer could potentially override any property of the system's actions. This core design requirement must be adhered to if the framework is to be a success.

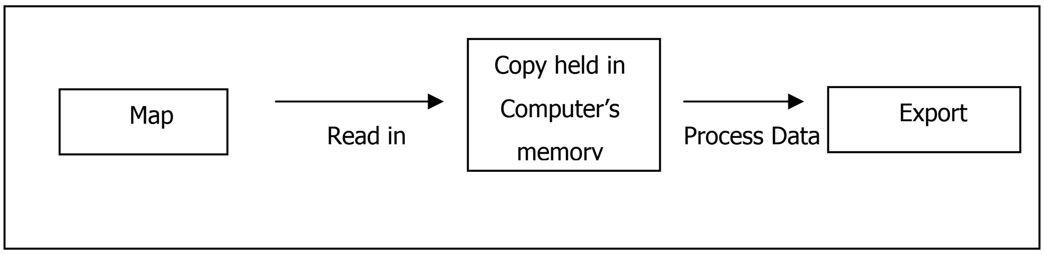


Figure 5.19 - Map Reading Process

The two core actions of the requested framework are to be; reading in a map, and to process this data into a developer chosen output as shown in *Figure 5.19*. This immediately suggests two modules for the framework, one that reads in map data, and one that uses this map data to export whatever the developer requires. To be able to export what they want, the developer must be able to easily query this data to a point where they can filter it to the appropriate conversion process required.

This situation presents a slight practical problem however, if there are just two single modules such as these, then how would the system enforce a standard 'map data' that once read in, could be passed to the export module. This is an important point as the export module will need to rely on the map data being defined in a preset way, to be able to process it. With developers able to override the import and export modules as they please, there will be potentially an infinite number of possible combinations between the two. Therefore there is potential for an infinite number of problems with compatibility between the two types of module if the way in which communication occurs between the two is left for the developer to decide.

There are two realistic solutions to this scenario, the first is to have no extra module and leave the requirement as is. This would require the reading module to pass the data direct to the export module in a clearly defined way. This could either mean that the developer must store the data somehow while reading it in, and then pass it all at once to the export module, or they would pass each element's data to the export module as it is read by the import module.

The first solution would mean more responsibility to the developer to store the data, translating into more work for their implementation. The second would dilute the import, and export processes, with the two happening at the same time. This could make the entire map import and export process more confusing than it needs to be, and suggests that a cleaner solution could be used. Keeping the module count at two, seems to either require the developer to create too much functionality themselves, or means that the framework design is too complex.

The second solution would be to introduce a framework storage solution to the map data. This would mean that the read module would run, and would fill up the 'framework storage' module, which once complete would then signal for the export module to run, and process the data within the framework storage. Developers would now not need to worry about any issues with storage. This is essentially a solution to the problem laid out within the first option of the first solution detailed above.

Overall, there are now three main modules within the framework design as seen in *Figure 5.20*. At this point their inner workings need to be fleshed out. Seeing as the framework storage is the middling ground of the two others, and the only one which would not require expansive properties it seems a sensible place to start. Before work began on the framework storage design, work would need to begin on the design of what the storage module would need to hold.

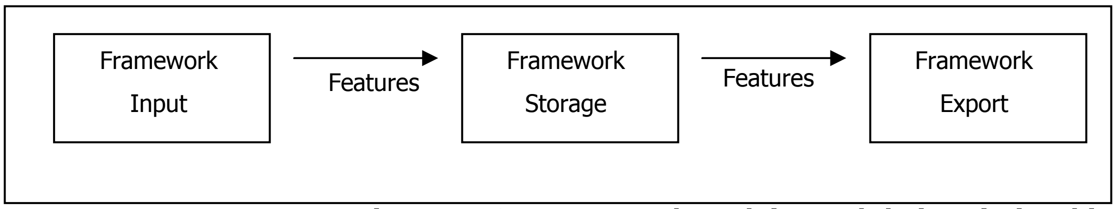


Figure 5.20 - Framework Modules and their Relationship

5.2. Definition of a Feature

As described within the Analysis, maps are collections of features within a specific area defined by the map, therefore it can be concluded that the only data the map contains are these features. Our framework storage is essentially our in-memory vector format collection of the features within the map. The framework storage, stores these features that contain properties as defined in *Figure 3.8*.

The Feature object within the framework will therefore have a make-up as described in *Figure 5.21*.

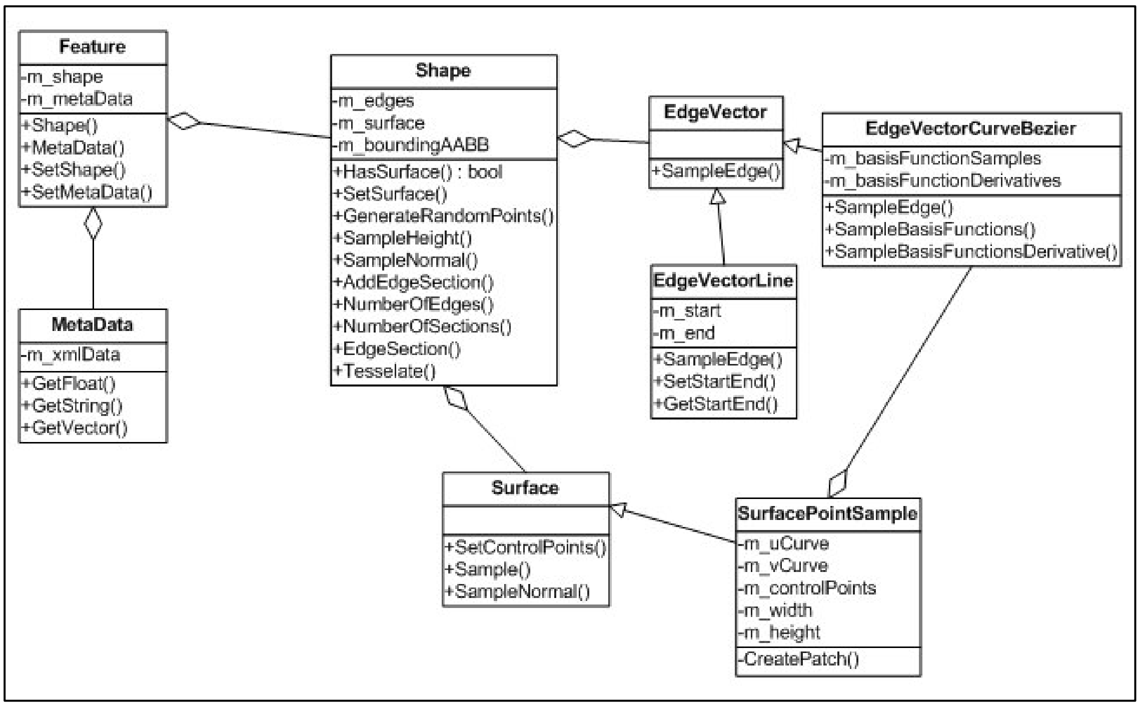


Figure 5.21 - Design of a Feature

5.3. Framework Storage Module

The framework storage module has been defined as a collection of features gained from the map reading process. It is merely a collection object, relatively simple in nature, but there to force any developers using the framework to conform to the framework's definition of what a feature is, and how import and export modules should pass their data to one another. Considering it must be used by the two different modules, the methods required of this object can be deduced.

- **Add Feature**
  This method will be needed by the input module.
- **Get Number Of Features**
  This method is needed by the export module to be able to know how many features it must process.
- **Get Feature**
  This method will be used by the export module so that it can access a feature in the feature list to process them.

Figure 5.22 - Feature List Methods

The object diagram of the Framework storage can be deduced from *Figure 5.22* as shown in *Figure 5.23*. It has also been decided that as the framework storage is essentially a list of features, the module will be renamed to Feature List.

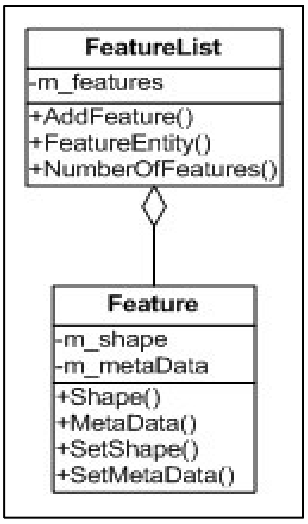
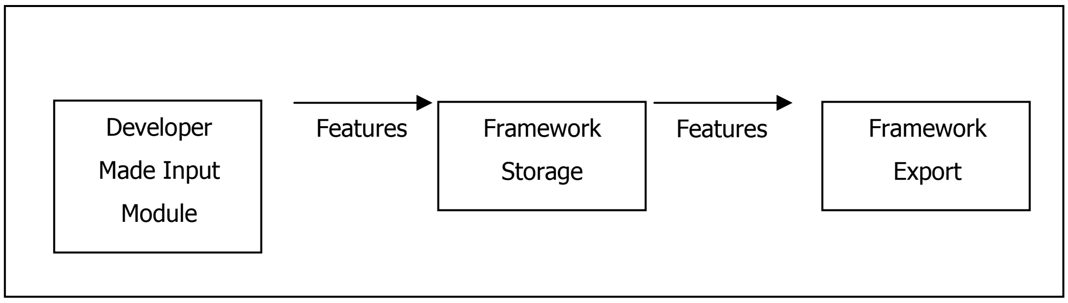


Figure 5.23 – Feature List Design

5.4. Framework Input

The only interface that the input module will require of the Feature List functionality described in *Figure 5.22* is the 'Add Feature' method. This suggests that, despite previous assertions, it really does not matter what form of implementation the input module takes from the framework's perspective. The framework does not need to force the developer to make their input module take a specific form.

The only requirement will be for the developer to provide the Feature List with complete Features using the 'Add Feature' method. This means that a developer has as much freedom as possible when implementing an input module for reading in a map. By following this design decision there are now no requirements for the developer to produce an input module conforming to what the framework has pre-defined.



This does not mean that the project will not provide a default implementation of an input module of its own, as a bare basic requirement to provide default functionality to the developer is to be able to read a map in. It does however mean that the design shown in *Figure 5.20* has changed, as technically there is no requirement for a specific input module for the framework as now seen in *Figure 5.24*.

5.5. Framework Output

The starting point of the export module will be the Features contained within the Feature List module. A question that must be asked is what will a developer want to do with the Feature List? A developer will want to use the data contained within a feature to influence how they handle this feature in the export process. Depending on the feature's data, it will be processed in a number of different ways, to the preference of the developer.

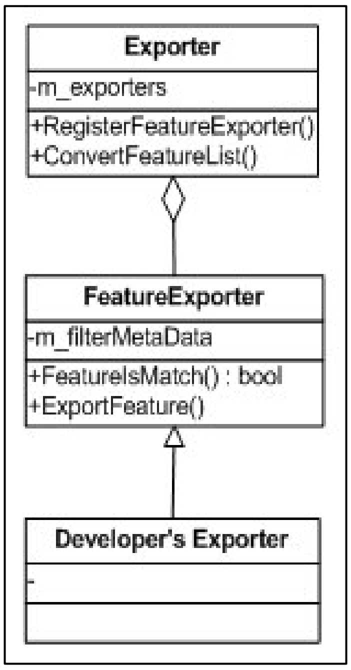


Figure 5.25 - Exporter Design

This suggests that an export module will need to be able to filter the Feature List, by comparing each feature's data values with certain values that the developer decides are appropriate. Once the feature search is complete, the developer would like to be able to use a specific overriding method to process and convert them into what is required. The export module whilst conforming to the design laid out in *Figure 5.25* has overall behaviour keeping to the points within *Figure 5.26*.

- Process the Feature List, finding features that hold matching properties to the developer's request.
- Match these features up with the developer's proposed exportation method.

Figure 5.26 - Export Process Behaviour

5.6. The Proposed design

With all modules now considered, the framework design overall will take a form as described within *Figure 5.27*.

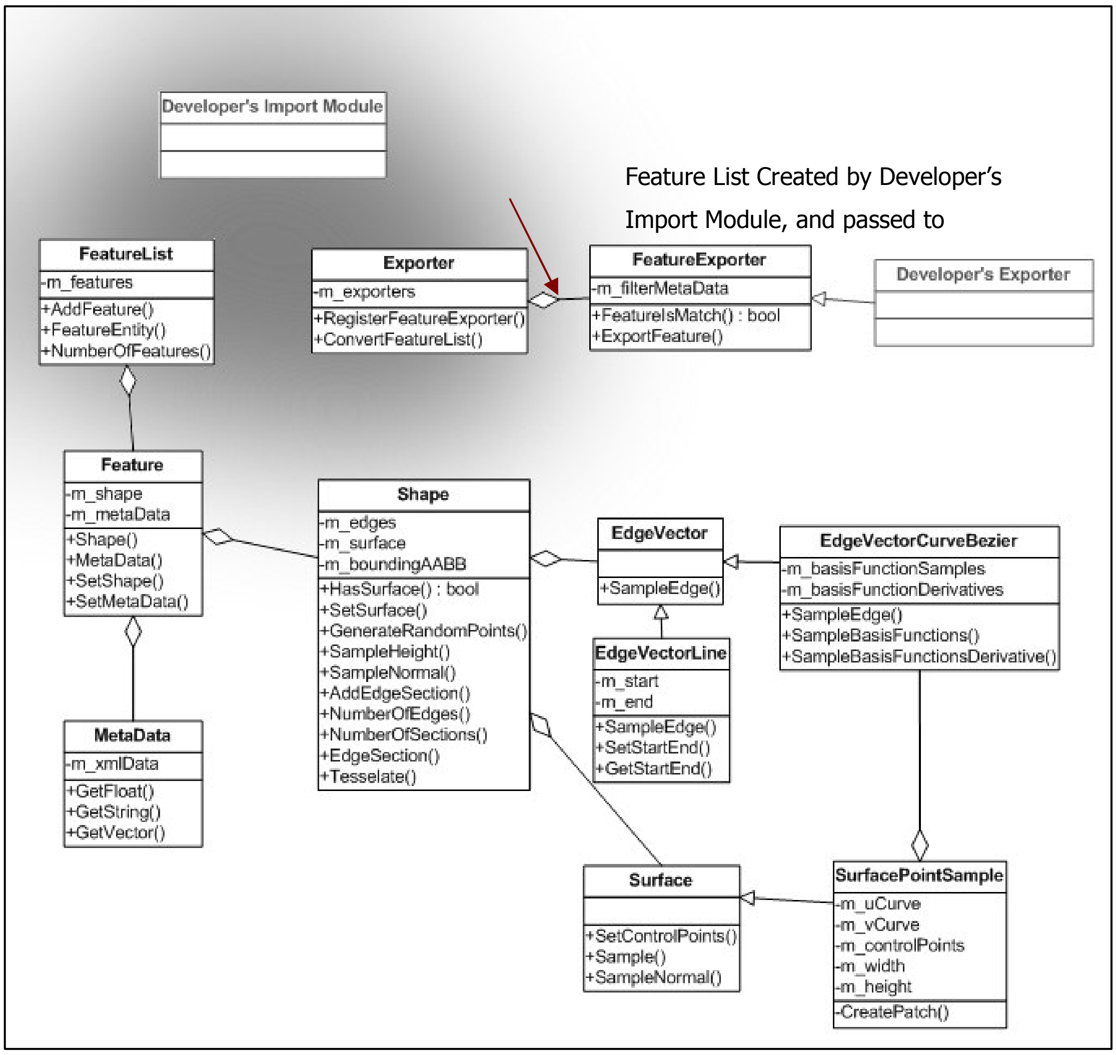


Figure 5.27 - Complete Design

5.6.1. Observations

From a merely curious glance, it is would seem that the proposed design described within *Figure 5.27* and any implementation of just this design, would not actually provide a significant quantity of work for the entire scope of the project, or of developer aid to the core problem identified by the project.

At this point, it is obvious, that while the framework is useful as a general guide to helping developers processing map data, there must be more problems a developer would face while implementing. If this is the case, then the real work will involve providing solutions to these individual problems within the framework that the developer can realistically use.

This theory is further followed through, when observing just where the most amount of functionality seems to be located within *Figure 5.27*. The majority is located towards the lower objects in the hierarchy, focused around making the storage, and access of feature data easier for the developer to carry out. This supports the notion that developers using the framework would appreciate smaller solutions to help solve individual problems with the process.

Side note: It must be noted that due to these observations the project changed direction from that originally laid out in the specification of the project within *Appendix A*, to focus not only on trying to solve individual problems in the overall process, but to that of automatic asset creation.

5.7. Exploration of Helper functions

The primary focus for the project is for the processing of map data to become easier for a developer. It has been found that a solution for developers would also need to include solutions to problems when importing and exporting the map data. This 'helper' functionality will need to become part of the framework itself. As defined in the analysis, the most important area for added functionality would be to help convert a feature's geometric data into forms a developer could easily use. Tessellating the shape or even just allowing easy access to the shape itself or any Meta data of the feature would be high priority in helping the developer.

All of the discussed functionality above could be held in per-feature methods, and as long as the developer could access each feature they could use these methods. However, helper functions to help process and convert the largely two dimensional features, into three dimensional models, represented as triangular meshes would employ more functionality.

By breaking down the core process of creating a mesh, and providing a pragmatic solution to the process would help inject a quantity of work into the project to match its required scope.

Focus should also be given to the input side of the framework; a generic solution to be able to read in a map stored in raster data would be extremely useful in initially integrating the framework into an external application.

5.8. Finding features on a Raster Map

5.8.1. Recognising shapes

To recognise the shapes on a raster map, to recognise the features contained within it, regions of equal colour will be assumed to define each feature. By moving upwards through the image one scan line at a time, searching for a pixel of a not-yet-encountered region, it is safe to use the shape tracing algorithm described in *Section 4.1.4* as soon as a pixel matching is found. Once the trace has reached its start point, the feature shape has been recognised. At this point all pixels within the shape are considered part of this shape, and can be removed from the future search to find a start-of-edge pixel.

The output chain code will be easy to process to find the corner points; a corner is simply each part of the chain code which changes from the previously defined movement directions. This chain code, in conjunction with the corner points can be used to create a shape for the feature constructed entirely from straight line edges. It is feasible that the shape could also be constructed from other line types, such as curves; however this would need a more detailed chain code analysis by the software.

Once a shape is recognised, the rest of the image would still need to be analysed. To do this, the algorithym will pick up from the start pixel, and continue its search along the scan lines of the image.

5.8.2. Pairing Up Meta data

While the features are defined within the raster data contained within the image file, the Meta data will need to be supplied in a partnering XML data file. As seen within the OS Master Map, locator information will be needed to match up any Meta data defined, to the features within the image file. The locator decided to be used here, is simply a pixel coordinate within the feature the Meta data belongs to. The framework should then be able to match the two up relatively easily. This also means that from a user perspective of defining the locator information on a Meta data element the process is much simpler, as it is relatively easy to find a pixel coordinate from an image.

5.9. Creating a mesh

With the analysis of what a feature consists of, it cannot be concluded that there is an immediate way of generalising the creation of a mesh to represent it. There is one shape that represents it in two dimensions in the map plan, and there is an undefined amount of Meta data used to describe unknown attributes.

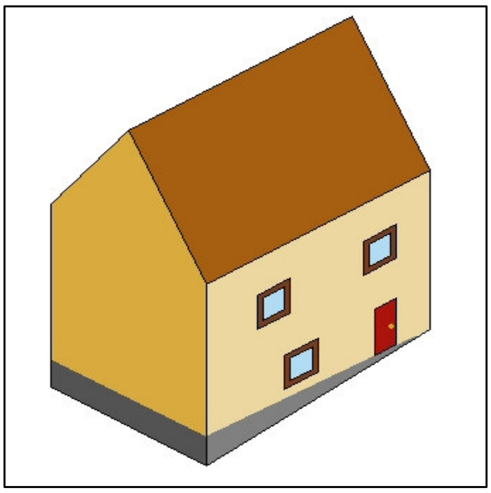


Figure 5.28 - Perceived Future Output

As shown within the Analysis chapter, it seems that the mesh creation must treat the feature as a set of different zones. These zones are then to be 'decorated' with whatever detail is needed to be put onto that particular zone of the feature. Both Zone Makers and Zone Decorators that encapsulate this functionality should be thought of being able to be overridden by the developer.

5.10. Mesh Maker

The Mesh Maker is intended to be used within the per-feature export functionality of the framework.

- **'Add Zone Maker'**
  Needed to register a Zone Maker with the Mesh Maker
- **'Build'**
  Executes all of the Zone Makers in order, orchestrating the Mesh making process.
- **'Get Mesh'**
  Some form of method to allow the developer to extract the final created mesh from the Mesh Maker.

Figure 5.29 - Mesh Maker Functionality

To begin with, all that the Mesh Maker has is a feature to build from. This means it has both shape and Meta data information. Within the export component of the framework for this feature type, the developer must register different Zone Makers to build the mesh to meet the specification they require for the output mesh. Once picked, the Zone Makers are passed to the Mesh

Maker in the order that they are to be used to create the feature's mesh, from its base through to its top.

The Mesh Maker must control the creation process, by processing the Zone Makers in the correct order, and then collating the output of all Zone Makers into the final output mesh. From a developer perspective the Mesh Maker will be the object giving them the mesh they want. The Mesh Maker will therefore have the functionality as described in *Figure 5.29*.

5.11. Zone Makers

Each Zone Maker will be given the feature data for it to create its section of mesh. A Zone Maker is expected to trace around the vector format edges of the shape of the feature one by one (sampling them into straight faces in the case of curved based edges). This will allow each Zone Maker to create a set of faces for the mesh which match the shape of the feature and conform to any Meta data attached to it. The output of this will be to create a basic frame which defines the mesh volume for this zone (*Figure 5.30*).

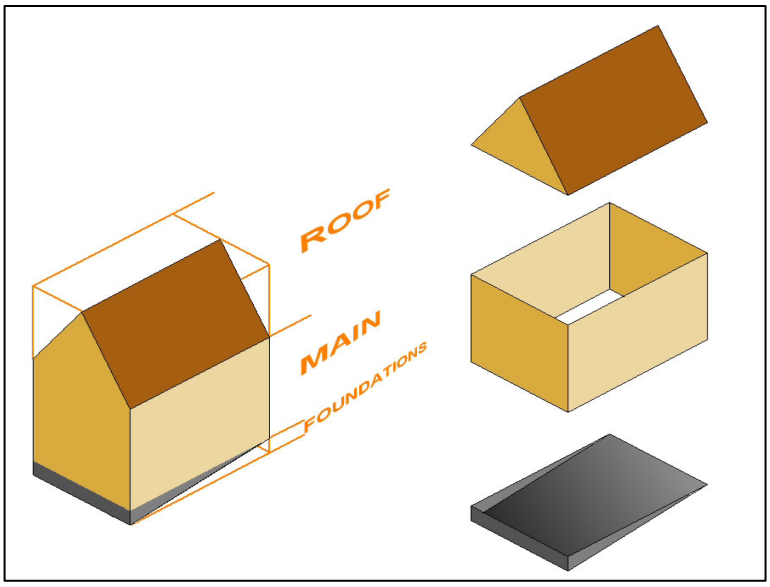


Figure 5.30 - How zones fit Together

These basic frames may be shared between different types of feature, for example a wall zone could be shared between a house and a shop. To differentiate between features, further areas of detail would need to be added. While this detail could form part of the Zone Maker, it makes no sense to add further complexity to this object. Instead it would make sense to be able to add custom areas of detail to this one zone, promoting code reuse, and meeting the expandability requirement of the framework.

A Zone Maker would be in charge of directing the process of adding this detail to the zone. Zone Decorators would be held per Zone Maker, which add specific parts of detail to the zone. Each Zone Maker would need to mark-up the faces which could have detail added to them as an Adorn Surfaces.

By taking this route, the design would allow developers to use one Zone Maker for the house/shop scenario described above, and then multiple smaller areas of Zone Decorators, which would mean less code duplication, and less overall work for the developer.

5.12. Stripping for the Decorator

As discussed within the analysis chapter, a zone's Adorn Surface needs to be split into generic sections that can be treated the same by the Zone Decorators before the decoration process can be begin. It must be the responsibility of the Zone Makers to split their Adorn Surfaces into these sections, and then give them to the Zone Decorators to act upon. The form that these sections take must be found, so that a specification can be made for the Decorator functionality.

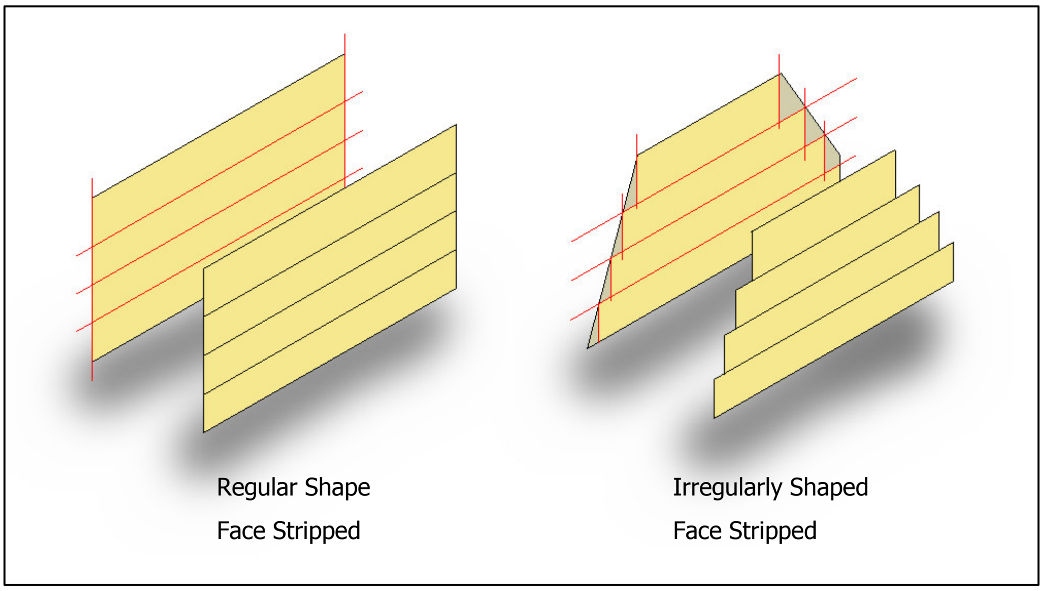


Figure 5.31 - Marking up Regular & Irregular Faces

The first attempt at a process would be to merely give the single Adorn Surfaces of each zone entirely to the Zone Decorator to do as it pleases. While this allows freedom, certain situations do not hold up to this approach. For instance, a zone is not defined as a single floor of a building, as such the Zone Decorator would be charged in figuring out how it would deal with multiple floors if given a Adorn Surface of the mesh. This pushes more functionality onto the decorator, increases code duplication and decreases expandability of the framework.

On a second run, this problem could be recognised, and rectified by splitting the Adorn Surface into floors (in the building's instance), or 'Strips' in a more generic instance. Thereby the Zone Decorator would treat a Strip as its own; it would be allowed to do anything it wanted to, over the whole vertical distance of the Strip. This removes a requirement for Zone Decorators and leads to an easier set of expansion possibilities.

However, what happens if the face that the Zone Maker wants to pass to the Zone Decorator is not square, but irregular (perhaps the mesh is to make a pyramid). Well by putting in a requirement that the Strip given to the Zone Decorator is rectangular and has a vertical edge perpendicular to the straight base edge of the Adorn Surface, this problem is eliminated. As shown in *Figure 5.31* there is a slight problem in this approach of potential space in zone faces that cannot be decorated, but it is an adequate solution for the purposes of the framework at the current time.

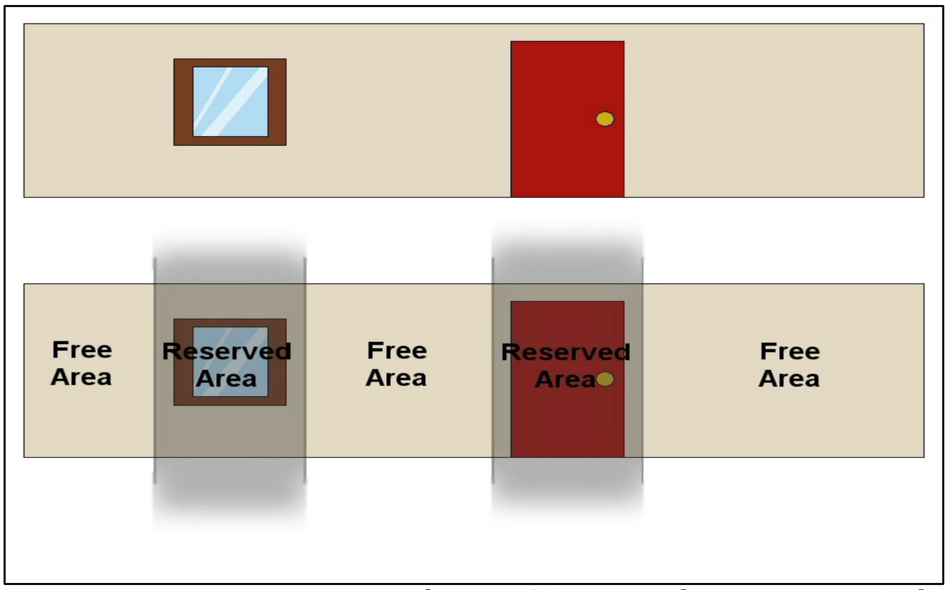


Figure 5.32 - Reserving Areas on a Strip

The final situation that may break this approach will occur when there are multiple Zone Decorators acting on the same Strip. What decides that they won't place a detail feature overlapping another already there? Could there be situations where doors and windows overlap? The Strip must be able to be queried to see if details have already been placed within areas of it (*Figure 5.32*), so that future decoration will not unwittingly collide with the decoration already applied.

5.13. Decorating the Strips

As multiple decorators will work one at a time on a zone's Strips, there will be a priority list determining their order of execution. To simplify proceedings, the order in which a developer registers the Zone Decorator with the Zone Maker will be the order in which they are executed.

As the decorators are adding detail to an automatically created mesh, they too should have a level of automation in regards to their placement on the mesh. While some detail features such as doors, might have a specific location on the mesh that they must take, others such as windows might be too costly to accurately define each location.

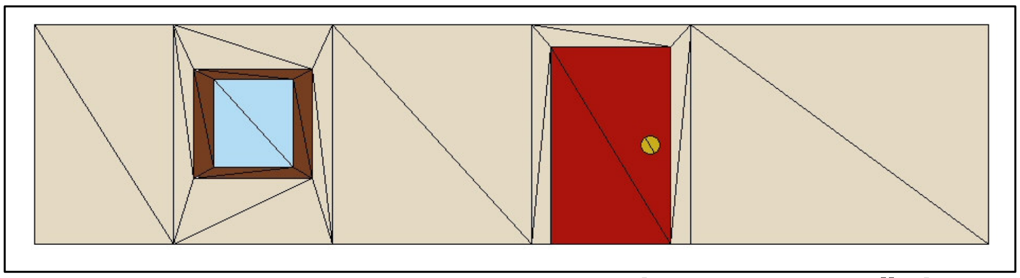


Figure 5.33 - Tessellating a Strip

Therefore Zone Decorators would need to fall into one of two categories; Fixed or Fill. Fixed decorators would position their detail at a specific location within the Strip of mesh face, while fill decorators would take any available space in a Strip and fit as many of them in as possible.

Naturally fixed decorators would have a higher importance than fill decorators, as their defined location denotes that the detail they give to the mesh must go there. Therefore Zone Makers would need to execute Fixed type Zone Decorators first, and then Fill type Zone Decorators on the remaining spaces within the Strip.

To deal with any tessellation issues, the decorator should be responsible to tessellate the mesh for its own area of the Zone Strip that it modifies (*Figure 5.33*). While a rather crude approach to a solution of the problem, it is flexible enough for the purposes of allowing the decorators to modify the mesh, and makes sure that the mesh is closed, with no holes.

5.14. Keeping Surface Information

To allow for detailed visuals to be supported, the final produced Mesh must be able to be constructed of a number of different surface types. Supporting texturing for example would allow for a vast range of detail to be supported. Therefore when all Zone Makers and Zone Decorators are executing, they must have access and calculate any texture coordinates available for the vertices of the mesh. They must also be able to add new surface types if required.

6. Testing

6.1. Proof of Concept

The first step when testing a framework aimed at being used by developers is to see just how suitable it is for an external developer to use. Therefore, a test bed application that was built to emulate the process an external developer would go through was created.

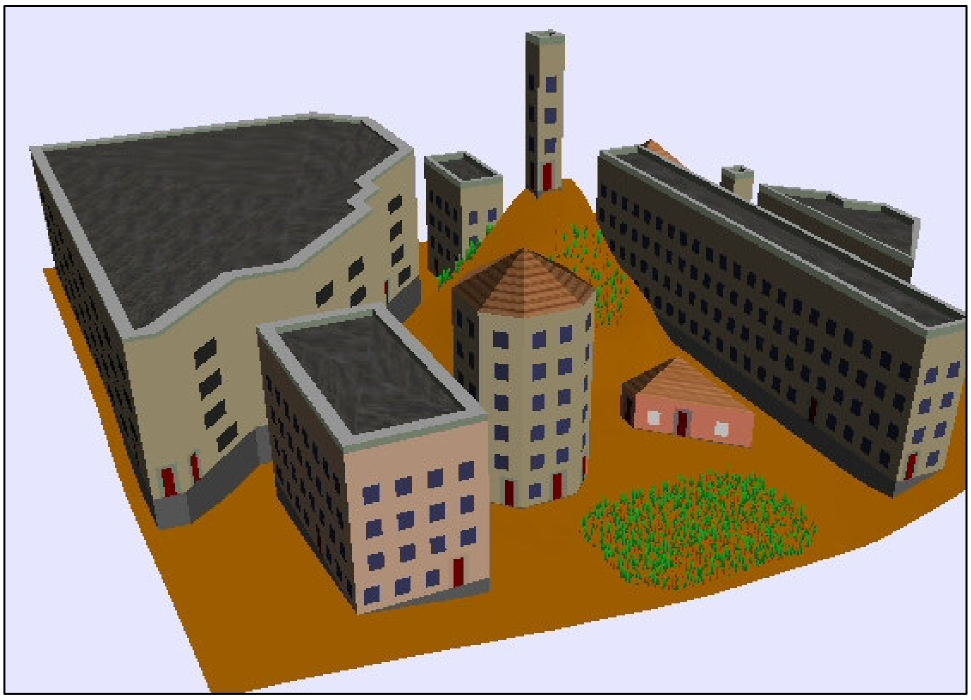


Figure 6.34 - An Example Scene

This application uses the implemented framework to read in a map that has its positional feature data stored in raster form and attempts to then recreate this as a set of meshes to be viewed in real time. This image file is paired with an additional XML Meta data file which was used to store all of the additional information for the map features. From here, the application uses the image reading component of the framework, to extract the individual features, and then uses this feature data to use a number of the different asset creation aspects of the framework.

The application itself can be found on the CD-ROM accompanying this document, while further information on how to use it is provided within *Appendix C*. Overall during the testing process three major aspects were attempted to be tested, these are described within the following sections.

6.1.1. Raster & Meta data Reading of the Framework

Primarily testing within this area focused on whether the shapes displayed within the map would have equal representations when processed and converted into meshes for the purposes of the application. All manner of polygon shapes were tested, including a variety of non-convex polygons (of which due to time restraints tessellation could not be implemented). Meta data which defined the height of each feature was used to create a wide variety of outputs.

Limitations here were polygons that had one pixel wide dimensions in the raster map. These depending on the situation would render out as free standing faces within the world, or were not rendered at all. All features should have dimensions of 2x2 pixels or above.

6.1.2. Ease of use of the Framework Architecture

Framework wise, the process of inputting and processing a map was relatively simple. The requirements from a developer in the test application, was to create an instance of the Image reader, pass it the image and Meta data file names. This then analysed the data and organised it into per feature vector type data. From here, the developer was required to provide a number of export objects to process this data. It was here that the majority of the developer’s effort went, but if using the provided Zone Makers with the Mesh Maker functionality, even this process was relatively small. Overall the architecture was useful, and did not prove too large a burden to integrate within the application.

6.1.3. Mesh Creation functionality

Mesh creation proved very modular, and simple to set up. Unfortunately it was the creation process of the Zone Makers or Decorators that proved rather cumbersome. While creating, and introducing custom Makers into the system proved simple to accomplish, actually creating new ones was rather lengthy.

Despite this, a variety of different looking meshes were able to be produced, a selection of which can be seen in *Figure 6.34*. Many use the same mix of Zone Makers and Decorators, but just change their Meta data values, even this approach can be used to create a variety of meshes.

6.2. Testing for the future

The framework has been designed from the beginning to be a third party solution to developers creating a wide range of applications. If this were to be a commercially available product, there is obviously a very wide range of testing opportunities that would need to be taken up to guarantee the framework correctly functioned at all times.

Part of the answer to this requirement would be to implement an automated testing framework, which would be run when any new functionally was added and would check that it would not be of detriment to the framework as a whole.

The subjects that automated testing solutions for software engineering projects normally concentrate upon are; Build and Testing. For this reason, the overall system is called a Build Verification Test solution.

6.2.1. Build Tests

The build process of the testing is designed to test any code changes made to the framework. It checks for things such as compile or link time errors. A successful output from this stage would be a fully built product. If the test is not successful, the basic product will not exist, and further tests cannot begin.

As the framework is a third party solution, this stage must also test whether there would be any problems with an external developer linking the product with their application. For this stage it would be useful to have a number of example test products to use, to see whether they build with the changes made to the framework.

6.2.2. Automated Testing

This stage concerns itself with the actual functionality of the framework. Test programs will be run which will be designed to use a certain system within the framework. Each test would be different, but an acceptable range of subjects to focus on, would be those tested for the test product of this project described in *Section 6.1*. An excellent idea here would be to take some of the developer's projects and use them as test beds. This would allow for testing to directly assess the impact of any changes upon the developers using the product itself.

6.2.3. Ease of use

While many subjects can be tested automatically for the framework, developer ease of use of the system is something that can only be answered by the developers themselves. To gain this input into the process, facilities would need to be in place for a developer to be able to communicate their thoughts back to the framework team. Tools such as online feedback forms, or even simply an e-mail address would work well.

7. Evaluation

7.1. The Product

The test application produced for the project was not as complex as originally envisioned. While not comparing to professional products such as OS Master Maps, in terms of data complexity, it did at least explore the area of automated mesh creation. It was in this area it was hoped that the product would be more in-depth, unfortunately only a limited number of Zone Decorators were implemented (Windows and Doors). However, completely closed meshes were able to be created as standard, which can be seen as a success.

However overall, it is pleasing how the final product reproduced map data in three dimensions, and a true indication of how a more advanced product would have a place in the asset creation process of the future.

7.2. The Project

When originally conceived, the project's potential scope was incredibly huge. At first this proved somewhat daunting, and research to find a path was unfocused within this broad area, and so moved slowly. This resulted in lower motivation, and fatigue in relation to the work flow of the project.

Eventually focus was achieved, and work began upon the framework's structure, this proved beneficial, as the project could be visualised easier, however it was less clear in what goals the project should take forward, and so this resulted in the project still moving at a sluggish pace.

A better overall approach would have been to try not mixing the research and design stages. By mixing them, focus was continually taken away from the project work, and seemed to dart off into different directions. To have set clear concise smaller scale goals after the research stage had completed, would have meant that the most important parts of the project were worked upon in the correct order. As it was, while working it seemed like some parts of the project were not completed in the most efficient, and inspiring (in the case of getting something simple to later build upon) manner, and this may have affected the project in some ways, in relation to the test implementation product, and the framework specification.

This should not detract from the considered success of the specification for abstracting subjects so large as mapping and asset creation however. While the implementation further proved how the analysis and design approach for the framework was successful. The work completed is definitely a firm basis for further work within this field, and could be especially built upon for the asset creation aspects of the project; this factor in particular can be seen as a success.

8. Recommendations

The total possible scope of the project is massive, obviously not all subject areas could be explored, and so a number of areas have been identified that could do with further exploration in the future.

8.1. Integrate 'True' Map data

Having data from the real world would mean comparisons between application output and the area the map represents would be easy. It would also mean that creating test data would no longer be a chore, as there would be an extensive selection to choose from already.

8.2. Increase number of Zone Maker & Decorator Types

Creating a large catalogue of Zone Makers and Decorators for the application would mean that more areas of algorithmic asset creation could be explored. For instance, there are many different types of roof left unexplored by the project, how to create all of these different types of roof is a huge subject area to explore by itself, but when considered with all the possible types of zones or decorators the scope is unlimited.

8.3. Different types of Test Applications for the Framework

Currently there is only one application which tests the framework for the purpose of displaying the map's contents in real-time. Perhaps an application which converts the mesh data into a modelling file format for a 3D modelling package would be a worthwhile test for the system. Other applications which perhaps do not even focus on the geometry of the map, but on the Meta data could be other avenues for continued investigation.

8.4. Map Features that do not Represent Physical Objects within the World

Perhaps of interest to games, map features could be used to define such things as AI routes, areas where enemies can spawn from or any other game specific feature. An investigation into how the concepts of automatic asset creation could be applied in this context may be a worthwhile cause.

8.5. Implement Non-convex Polygon Tessellate

By adding this functionality to the system, it would instantly make it able to process a much higher in complexity amount of possible map data, and produce a wider variety of output meshes.

8.6. Investigate how this technology affects the art pipeline

The most important further work in this project's priorities would be to investigate how using a two dimensional map contained within an image, and accompanying Meta data could be used to affect the art creation process when creating a virtual world. Could a successful implementation be created where the whole world was instantly modelled, or would a simplified version, which was then manually expanded upon by an artist have to be settled for? How much time would it really save the art pipeline in a production?

9. Conclusions

The investigation into a solution for software developers to easily integrate map data into their own applications proved to be a fruitful exercise, which produced a design for a framework that allowed expandability of its functionality to support as many different applications as possible.

Maps were defined to be a collection of features, with each feature consisting of a shape (containing the geometric shape, and any point sampled height data) and an undefined amount of Meta data. This Meta data was defined as any custom property unique to a single feature.

The framework produced was defined to have three separate processes; input, storage and export. The import module is entirely within the developer's responsibility in how they construct it, as it adds single features to the storage module. The export module will then process the features within the storage module, filtering them based on what types or values of Meta data they contain. Once a match is made, this feature is processed using the developer's defined conversion process, and the feature data can be used to create an asset.

It was found that for feature data to be useful for the developer, a number of query methods would need to be provided, to allow them to use the data contained easily. Without these, there would be little benefit to using the framework, over creating an in-house custom solution.

It was found that to generalise the mesh modelling process, it was essential to break the geometry up into common zones. This was because these common zones, would likely share common methods to create them. By separating these methods up in this way, and encapsulating them into a 'Zone Maker', it not only allowed for reusability within mesh creation, but considerably simplified the concept of creating the mesh itself.

Extra detail within the mesh was handled by 'Zone Decorators', which are used to fill each zone with decorative features (such as Windows). Again these could be completely overridden by developer functionality, allowing for an infinite amount of expansion and possible outputs.

A proof of concept application was created, which read in basic map data, consisting of an image depicting the geometry of the world, and an XML file containing extra Meta data for each of the features contained within. While not producing features of an extreme detail quality, the application provided opportunity to see the potential of different mixes of Zone Makers, Decorators or input values to produce a wide range of different output meshes.

The work here should be seen as a basis of for further work within the realms of asset creation. Recommendations for the subjects of this work are provided within *Chapter 8*.

Overall this project has concluded that the idea of using a map as a simple-to-define blueprint for a much more complex world, is a very powerful one in the fight to lower asset creation timings for virtual worlds, and is almost certainly worthy of future work dedicated to it.

10. References

[1]	Young & Fu	*The Handbook of Pattern Recognition and Image Processing* Chapter 9, "Image Segmentation", Pages 215-232 Chapter 10, "Two-Dimensional Shape Representation", Pages 234-235 Academic Press, Inc., 1986
[2]	Bernd Jähne	*Digital Image Processing, 6th Edition* URL Springer, 2005
[3]	M. de Berg	*Computational Geometry, 2nd Edition* Chapter 3, "Polygon Triangulation", Pages 45-61 Springer, 2000
[4]	J Lien & N Amato	*Approximate Convex Decomposition of Polygons* Texas A&M University, 2004
[5]	Open GIS Consortium	*Geography Mark-up Language (GML) Implementation Specification* www.geospatial.org/specs OGC, 2005
[6]	Dietel & Dietel	*XML: How to Program* Prentice Hall, 2001
[7]	Paul Bourke	*Piecewise Bezier Curves* astronomy.swin.edu.au/~pbourke/curves/bezier/cubicbezier.html Viewed Feburary 2006

Appendix A - Specification

Creating A Virtual World Described By Custom Map Data

Synopsis

The major objective of this project is to create a software framework to aid asset creation. The software will form a part of the asset creation pipeline, it will read in map data, which could come from the real world or could describe a fictional place, and convert and represent these in a form that will be useful for a number of applications to access. The most desirable of which will be to recreate the map into a three-dimensional virtual world. The project will allow the exploration of the problems and solutions involved in an asset pipeline that tries to support as much functionality as possible.

Together with this basic 'surface/area definition' in the map data, Meta Data will be used to add extra information to the features on the map and further define them. This Meta data will be entirely customisable, allowing a user to use the tool in any which way they please. Coupled with user-definable data, the map features themselves will be able to be accessed in custom ways, by designing the system to be as generic and expandable as possible, with the aim that a developer could take the system and modify it to suit their needs perfectly.

With the framework, it is then desirable to create an application using it to use the system from a developer's perspective. This application will focus on testing the asset creation stage, and not how it uses any data produced.

Design of the framework will take the process of fitting the technical requirements laid down, and then trying to see if they fit any theoretical situations that may occur. Once this is deemed successful, the same process will be applied in its development. Development will follow the design, but will see a process of adding in different map features to the system and tracking whether they work or not. If they do not work, the system will be re-evaluated and allowed to evolve.

Minimum Objectives

- Create a framework that is clear and easy to understand, that can handle as many forms and situations of Meta, and map data.
- Create a demonstration application that will show the process in which the framework can be used.
- Meta data from the framework must be able to be fetched, and filtered easily by the host application.
- Map data to contain workable examples of at least height data, buildings, surface types (grass, concrete, etc.), roads, and 'special features' (such as statues, libraries, and basically any sort of unique feature).

Schedule

9th November 2005 - January 2006
Research and development of 'map reading' features and heavy development of Meta data aspects. Begin testing of framework, by introducing one map feature at a time to a test application (built onto it), following it through the framework, making use of it by displaying the new data in the application, and then continuing this process for as many features as possible.

January 2006
Interim report, fully evaluate framework by beginning final phase of 'test application' development. Test application should now take as many features as the map data provides and reproduce the location described in 3D. Test application should be completed.

February 2006 - Hand in 3rd April 2006
Main project should be complete, development of 'extra' features to further test and demo the asset pipeline/framework (expansion on the number of features that maps can support for example). Complete final report.

Appendix B - Bibliography

[1]	A Brodersen	*Real-Time Visualization of Large Texture Terrains* University of Aarhus, 2005
[2]	M Bertram X Tricoche H Hagen	*Adaptive Smooth Scatter-data Approximation for Large-scale Terrain Visualization* University of Kaiserslautern, 2003
[3]	-	*Ordnance Survey* www.ordnancesurvey.co.uk Viewed November 2005
[4]	W Niblack	*An Introduction to Digital Image Processing* Chapter 9, "Image Segmentation" Chapter 10, "2D Shape Representation Prentice Hall, 1986
[5]	A Fournier	*Triangulating Simple Polygons and Equivalent Problems* University of Toronto, 1984
[6]	T Dey	*On Good Triangulations in Three Dimensions* University of Purdue, 1991
[7]	M Neal	*A Software Tool and Techniques for Converting Map Data into Object Orientated Representation* University of Wales, 1998
[8]	C Marshall	*Making Metadata: a study of metadata creation for a mixed physical-digital collection* Xerox Palo Alto Research Center, 1998

[9]	J Beaujardiére	*The NASA Digital Earth Testbed* NASA, 2000
[10]	L Barnett	*A "Roads" Data Model: A Necessary Component for Feature-Based Map Generalization* 3M Company & University of Minnesota, 1997
[11]	T Davison	*University of Teesside Intranet* outranet.scm.tees.ac.uk/users/u0018196 Viewed January 2006
[12]	Microsoft Corp.	*MSDN* *msdn.microsoft.com* Viewed March 2006

Appendix C - User Instructions

Initial Instructions

To run the supplied copy of the test application, insert the accompanying CD-ROM.

Wait for the autorun to begin. If this does not happen, please 'explore' the CD-ROM, and open the file named "readme.txt". Follow the instructions inside this file to begin running the application.

Background on Application

The application is designed to process a number of example maps. It has a number of basic commands to aid the viewing of the produced meshes. Exact control configurations are available on the CD-ROM, but options to render the scene in wireframe or solid are available, as too are back face culling methods.

There are a number of example maps to view. To view the map, please open the corresponding '.tga' file in an image editor of your choice. To view the Meta data file, please open the '.xml' file.

To get a taste of what output awaits, without running the application, please refer to Appendix D.

Appendix D - Screenshots

The following are a selection of Screenshots from the test application. Shown are the original raster maps, and the output in either solid or wireframe mode.