Table of Contents

Introduction

This chapter discusses potential uses for 3D model reconstruction, and then goes on to cover various methods that can be used to create 3D models of real-life objects.

Uses of 3D Model Reconstruction

In the past 20 years, vast increases in the processing power of computers has enabled creation of interactive 3D graphics. While computers are now able to create very lifelike and spectacular images, they are still only as good as the information they are given to work from. Every object that is rendered must be described for the computer in terms of primitive objects that can be drawn. In most cases this information will be in the form of a list of polygons.

Objects may now be described and edited relatively easily within the confines of the computer using CAD and other software. Almost all modern products are designed inside a computer before ever being produced in the real world. From the point of view of advertising this is very useful, as very lifelike images of the product to be marketed may be produced.

However, there are still many occasions when having artists describe the object by computer is not the best method. Perhaps an accurate likeness of an already existing object is required, or the object could be produced far more quickly by a sculptor using clay. Very good examples of this are seen in the film industry, where indistinguishable virtual scenery or stunt doubles of actors must be created. In addition, a lot of entirely computer generated characters are produced by sculptors beforehand.

As technology has progressed, computers have been able to produce three-dimensional graphics that are real-time and interactive. The same need for real-life 3D models has been found in the interactive sides of computing, such as computer games and the Internet. These new areas have introduced slightly different demands of the technology. They may require hundreds of 3D models, but not put the same high demands on quality and accuracy. They also probably require a lower cost solution.

That solution is the aim of this project. Most professional solutions to the problem of model reconstruction have so far focused on high-cost, low volume markets. The process of creating a 3D model from a real-life object is usually divided into two steps - acquisition and reconstruction.

Methods of Model Acquisition

There are many ways to capture a three-dimensional model of a real-world object. Perhaps one of the simplest of these is a pointer on a jointed arm containing position sensors. The operator then places the end of the pointer on many different parts of the object, and would gradually build up a set of points on the model in the computer. These can then be joined up to create a mesh of polygons which represent the outside of the object.

This can also be implemented automatically in a very crude manner – the object is placed on a turntable, and a motorized pointer is driven outwards until it hits the object, where its displacement is measured.

Faster and less intrusive methods of reconstruction tend to use lasers. By shining a laser beam, and then looking for the dot with a linear sensor, measurements can be taken quickly and accurately without disturbing the object. A similar idea is used by some scanners, in which a pattern of light is projected onto the object, and information read back via a digital camera (changes in displacement or focus of the pattern are used to determine depth).

A variation on this technique involves no patterns, but instead uses a camera with a very narrow depth of field (that can be refocused to different depths). Image processing techniques can determine how far away each pixel of the image is by finding the depth at which that area of image appears most sharp.

Another method of model acquisition is used in a professional product called the Z-Cam. This uses a grey scale CCD with an extremely fast shutter and a defocused infra-red laser. The shutter is closed, and the laser fires a burst of light. At the time the reflected light is expected back the shutter opens, and then closes again after most light has returned. The brightness value in the CCD will then be proportional to the distance away of the object (the proportion of the laser pulse that returned). Many other methods of model acquisition exist, such as laser range finding, ultrasonics, and radar.

Methods of Model Reconstruction

Using the information gained from the above methods to create a 3D model is quite easy in most cases. Most of the methods produce a series of three-dimensional points, whose exact connection is known so they may be connected by triangles relatively easily.

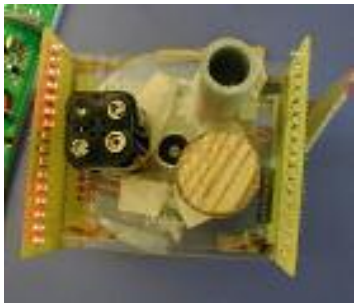
It is however possible to obtain three-dimensional models using just a series of two-dimensional images from a normal camera. In this case reconstruction becomes more complex. One (quite popular and simple) method worth mentioning involves vast amounts of manual intervention (PhotoModeler – [1]). Several images are taken, and key points are marked and identified by a human.

The software then takes this information and calculates the coordinates of these points in 3D. Ideally the whole process could be automated, but the brain is very good at pattern recognition. In fact, using a human to pick out key points often results in a more efficient and accurate mesh, purely because the points chosen will be at rapid changes of surface shape rather than just where the colour changes suddenly (as most recognition algorithms would tend to do).

Another problem to be considered is that of very small features on an object. Obviously an object can be scanned at extremely high resolutions, but this is impractical for most uses (apart from high budget, low volume industries such as films and engineering). A common method of adding detail to a low resolution mesh is to use a texture. This is an image that is effectively painted on to the

polygons that make up the model, and for features that do not protrude far from the model it works very well.

To gain this information, several images of the object from different viewpoints are required (or an extremely high-resolution colour scan, which is probably not practical). Therefore from the point of view of cost it makes sense to find a solution to the problem that could use the same images for both model reconstruction and texturing.

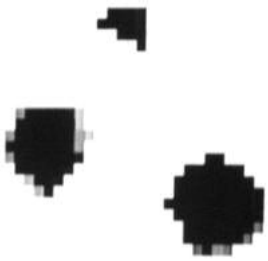


MIT's 2D Tomography Unit

There has been a sizeable amount of research into techniques for reconstructing models from cameras; however the work that first caught my eye and gave me the idea for the project was that done by two masters students at MIT (Jim Paris and Mariano Alvira) for use in a hardware practical class. This didn't actually use a camera at all, but involved 16 light sensors arranged in a line with opposing lights and a rotating turntable.

The apparatus would detect if a parallel beam of light could pass through the test objects in each one of 16 positions, and for different angles of the turntable. Using this 2D tomography technique, a very low resolution 2 dimensional slice of objects on the turntable was produced using only a 1 dimensional sensor and a stepper motor on the turntable.

The algorithm used for this is based on the Radon [2] transform. This is the same algorithm as used for X-Ray tomography, which is often used to reconstruct 3D models of structures inside objects that cannot be opened or seen through (such as luggage, or items of archaeological importance).



Result produced by MIT's tomography unit

A simple explanation of the Radon transform is that it sums all the input data in such a way that every point in the 2D result is the sum of every point in each 1D projection that can 'see' that point. This involves summing the 2D matrices obtained by extruding each 1D projection into a 2D matrix, and rotating it by the angle from which it was taken. The Radon transform will be covered in more detail later.

MIT's approach differs slightly from the Radon transform in that light and opaque objects are used. Light either passes through the gap between objects or doesn't, and this makes the algorithm far more intuitive (instead of summing, a boolean AND is performed). It does introduce some interesting problems, such as

the way that a hollow object (such as that shown in the image above) can never be known to be hollow, and so will appear solid (as shown by the result of the scan). This is not the only problem - in fact all concave surfaces suffer in a similar way.

MIT's 2D Tomography apparatus could easily be extended to produce a three-dimensional model, either by using a two-dimensional sensor, or by moving the object in an extra dimension. The obvious way to create a two-dimensional sensor is to use a digital camera, which is not only two dimensional, but very high resolution.

The algorithm used in the 2D tomography apparatus relies on the sensors detecting parallel beams of light, which causes a bit of a problem. It would require a series of lenses in addition to the Camera to make it detect only parallel beams of light. However, by changing the algorithm slightly, the sensor can simply be a camera, viewing the object with perspective. There is also no need to detect light passing through the object – colour matching could detect the difference between the colour of the object and a uniform coloured background – effectively detecting the object's silhouette. This allows for a very low cost solution, with just a turntable needed. In the lowest cost of cases, a minute-long video of an object rotated by second hand of a clock would suffice.

The shape of the object can be reconstructed from the silhouettes of the object from different angles. As has already been mentioned, in order to make the model more realistic, a texture needs to be added to it. The texture itself can be created from the same series of images of the object.

This is by no means the only method of creating a model from a series of two-dimensional images. It is the method that occurred to me after seeing MIT's tomography apparatus. In the Preparation section I will describe other methods that have been used, and their good and bad points.

Preparation

Background Reading and Research

To start with, it seemed a good idea to research other methods of model reconstruction to see if any of these would be more useful than the method described previously. Computer vision is an extremely wide area, so I focused my research on retrieving information from calibrated images. Retrieving information from uncalibrated images (where the camera field of view, orientation and position is not known) is often a far more difficult problem, and is probably not advisable to undertake for a dissertation project.

Model retrieval from calibrated images has been explored quite deeply, mostly as research, and at least once as a commercial product. One of these products was a system that was produced and sold by Olympus, called ScanTop. It retailed for $7000, and contained just a piece of software and a computer-controlled rotating turntable. The user had to provide their own camera, and this would be used to take several pictures of the object from different angles against a coloured background. From the advertising literature that I could find [3], the ScanTop used an almost identical method to the one I described in the introduction.

On the whole though, algorithms have relied on matching features found in images and triangulating their position (e.g. [4]). At first glance, this looks like the best solution. It does however introduce the problem of matching features between all images, and then producing triangles that join these together in the correct way.

These approaches have both some nice features and drawbacks. They are able to reproduce concave surfaces well, and automatically produce vertices that might be a good starting point for triangulation. The main problem is that they rely on the matching of features in images. A large amount of modern man-made objects are extremely smooth, and lack features that could be easily found. Consider also repeating patterns such as the buttons on a computer keyboard or telephone keypad. The edges of the keys look exactly the same, and could very easily confuse a simple algorithm that worked without good knowledge of the rest of the scene.

Another approach taken [5] is a more brute-force approach to model reconstruction. It considers all points in a volume where the object in question would be. For each of these it projects out to where it would be seen in the images taken from each angle. If the colour of the point is similar in around half of the images, then it can be deduced that what was being viewed was a surface (viewed from all around the 180 degrees it is visible). This sounds like it would be very robust, but some further scrutiny shows some serious shortcomings. This is an algorithm that produces Voxels [6] not polygons, which are hard to produce triangular models from. It also suffers badly from large areas of similar colour on an object.

Considering the explanation of the last approach, the method based on MIT's idea can be described in a much more intuitive manner: For each point in a volume, project onto each available image where it would appear if it existed. If in

any image you can see the background where the point should be, then it obviously doesn't exist. When performed over an entire volume, each point is then known to be either inside or outside the model. This is a perfect set of data for other algorithms to work on.

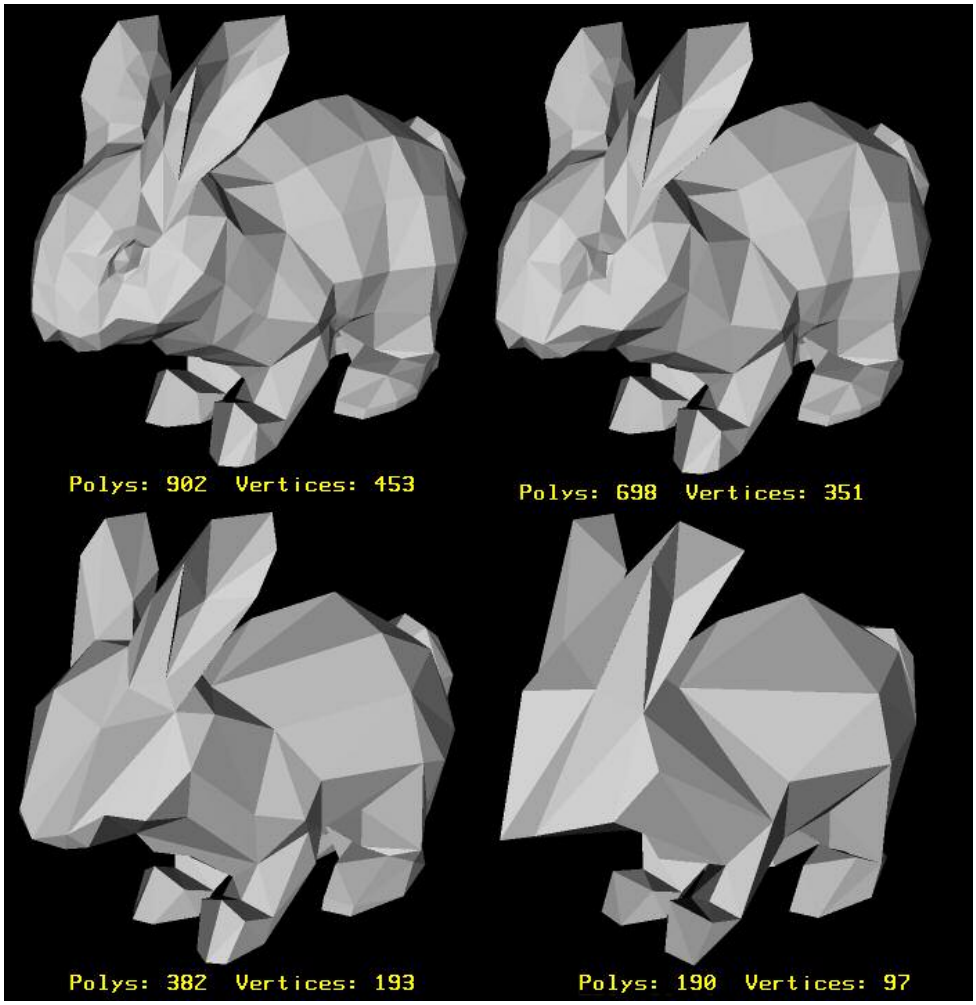
Requirements Analysis

The aim of this project is to produce a low-cost method of model reconstruction that can be used for interactive applications. For the majority of these applications, high accuracy will not be required. Since this is a low cost solution it should be capable of running on a standard PC, and should use very little external hardware.

Because of this, I decided to set the following objectives for this project:

- Should run on a normal PC
- Should not require the PC to be modified in any way to interface hardware
- Will automatically take pictures of an object from known angles
- Will complete calculations in a useful amount of time. e.g. less than an hour
- The model produced should be an obvious likeness to the real-life model

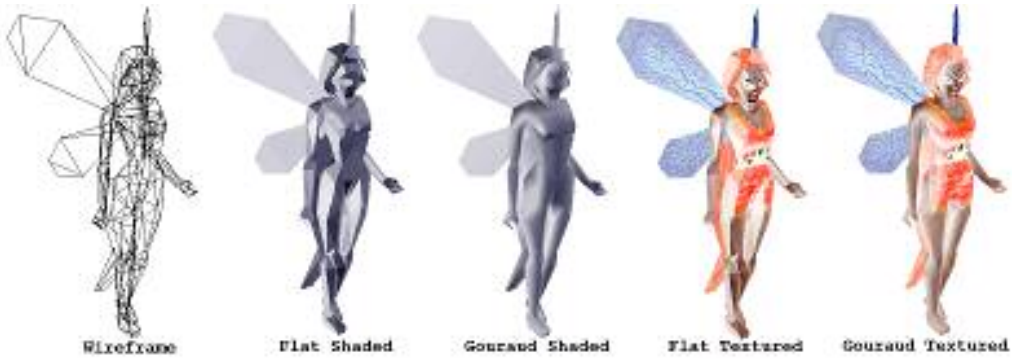
Another question that needs answering is how high do we expect the resolution of the model to be? Since this is for interactive applications, the accuracy does not need to be high. Using the method of model construction given, very little fine surface detail will be produced anyway, so a bare minimum of triangles will be needed.



Different Resolution Models from Stan Melax' Polygon Reduction Demo

The diagram above shows the same object, but made from different amounts of triangles. While the low resolution model obviously contains very little detail, there are a few tricks that can be used to make it look almost as good as the high resolution one. The first one of these is not to calculate light for triangles, but instead to calculate light for each vertex, and use the triangles to blend between the different colour values (Gouraud Shading - [7]). The second is to apply a texture onto the triangles to introduce more detail.

When these two approaches are combined, the result is much better, and it is obvious that a large amount of triangles should not be needed. See the diagram below for an example. The amount of triangles needed varies drastically with the complexity of the object. However I only intend to scan moderately complex objects, and I believe 2000 triangles is more than enough to adequately describe an object when textures are applied. Any more than this and it would be impractical to download the object (and its texture) quickly over a slow Internet connection.



Different Ways of Rendering the Same Model (654 triangles)

Choosing Algorithms

Reconstruction

After considering the other methods of model reconstruction mentioned, I decided that the original idea (using a modified Radon transform) was still the easiest and most robust to implement. In its normal form, the Radon transform converts representations of data from matrices representing pixels in an image, to matrices representing lines that make up the image.

Performing the transform on an image will yield another image where the brightness of each point specifies how much of a certain type (angle/offset) of line there is in the image. The axes usually represent the angle of the line (theta) and the smallest distance between it at the origin of the coordinate system.

The Radon transform itself is very intuitive. For every point in the transform, the value may be calculated by adding all the points in the source image along the line it represents. The formula below shows this :

$$g(\Phi, s) = \iint f(x, y).\delta(x\sin\Phi - y\cos\Phi - s)\,dx\,dy$$

δ *is 1 when its argument is 0, and 0 elsewhere*

f(x,y) *if the original image*

x = x coordinate

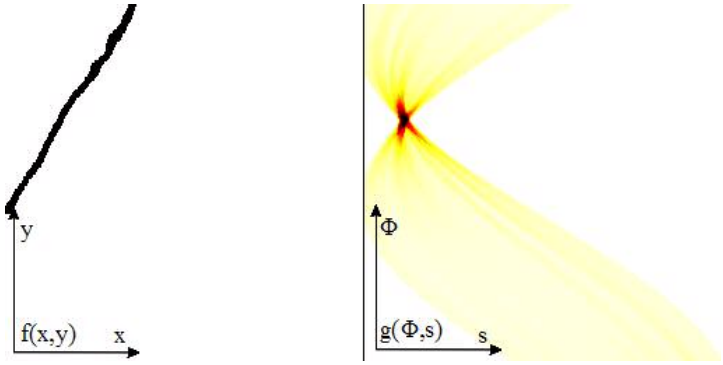
y = y coordinate

g(Φ,s) *is the transformed image.*

Φ = angle (in radians)

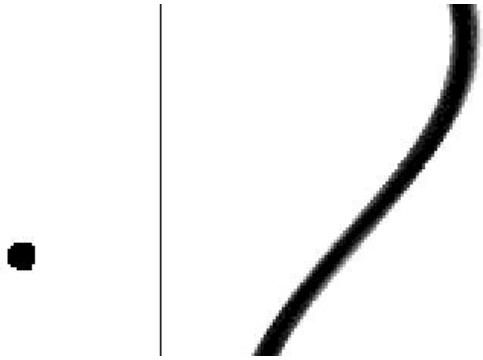
s = offset from the origin

The line integrated over is usually to infinity, as the equation suggests. However by defining the image as being 0 everywhere except where data is defined, we have to sum only pixels in the image.



The original image (left), and its Radon Transform (right)

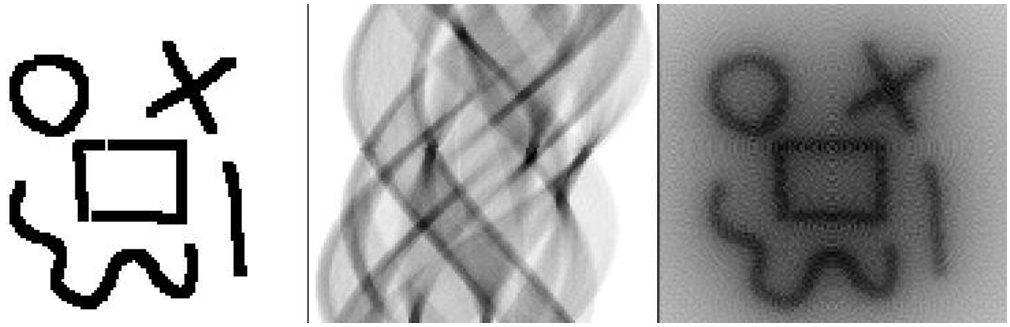
Above is a radon transform of a hand-drawn line. It is obvious that even though the line is not perfect, at one point in particular the radon transform produces a high value. This represents the line's angle and distance from the origin.



A Circle, and its Radon Transform

Above is the Radon Transform of a Circle. Imagine the circle placed off-centre on a turntable, and then looking at its projection as it rotates. The result of the Radon transform looks very similar to this (1D projections along the X axis, rotation along Y).

This turns out to be very useful, because in these circumstances the inverse Radon transform allows the original object to be more-or-less reconstructed. This is the same idea used in X-Ray tomography.



An Image, its Radon transform, and finally the result of an Inverse Radon transform

To perform the inverse transform, you must integrate along a sine wave instead of a line. A far more intuitive way of thinking about the Radon transform is to imagine taking the 1D projections of the 2D object, and extrude them over a 2D array. Then, this array is rotated by the amount the object was rotated, and added into an accumulation array (which is originally zero). The end result is the same.

This transform can easily be extended to 3D, by extruding 2D projections. It turns out that if the extrusion is replaced with a perspective calculation, a simple parallel projection is not needed, and an image as taken from a camera can be used instead.

In the series of images above, the original image has been blurred slightly. If this was created from silhouettes, the Image would be either black or white, however none of the concave structures would be reproduced.

The Images shown in this section were produced from a interactive Java applet that I made to test the Radon transform. It is available on my website, the address of which is at the end of this document.

Background Detection

For the previously mentioned algorithm, the background needs to be reliably detected from the foreground of each image. I decided that the best way to do this was to place a solid coloured background behind the object when images were taken of it.

The output of the background detection needs to be a single-channel image in which the value of each pixel represents whether that pixel was foreground (object) or background. I decided not to limit this to a boolean value and to instead use a graduation of values to represent pixels that the algorithm wasn't certain about.

I didn't research into algorithms for background detection, because it seemed that it could be done reliably simply by comparing the colour within some tolerance. This matching does however need to be done in HSB colour space because the colour should be matched far more tightly than the brightness (lighting will almost certainly be different in different areas of the background).

Polygonisation

In order to convert the data in the volume into polygons (which would be easier to render and to put textures on) there are several different methods available. However, the algorithm would be working from a three-dimensional volume full of points spaced at regular intervals. The two main algorithms I could find that were of use were one involving Tetrahedrons [8] (which doesn't appear to have a name), and Marching Cubes [9][10]. Marching cubes is widely used in many areas, and I decided to use that because of its relative simplicity to implement.

Marching Cubes works by dividing the volume into a series of small cubes, and draws a polygon inside each cube (if it is needed) to approximate where the surface should lie (where the value in the volume is some constant). The algorithm itself was covered in the Advanced Graphics course of CST Part II [11].

Polygon Reduction

Performing the Marching Cubes algorithm on the volume could produce a very large amount of polygons, and because of the intended use for the models (interactive rendering) it makes sense to reduce the number of polygons used in areas where they are not needed (e.g. flat parts of object). This is not vital though, so I decided this would be left as an exercise if I got time.

Quick research into this area turned up many methods of polygon reduction, however the one that appeared most simple, and visually to give the best results, was from an article in GDMag written by Stan Melax [12].

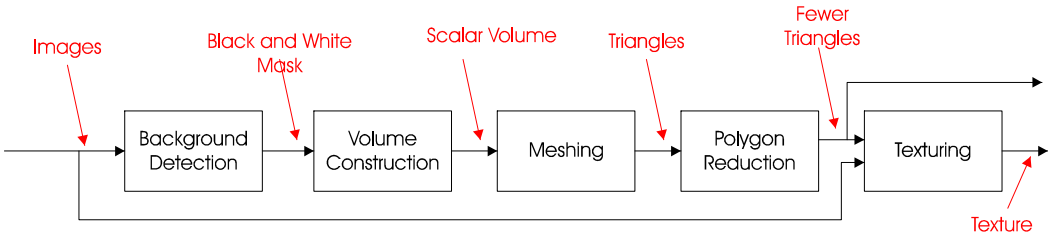
Texturing

Finally, a texture needs to be added to the polygons. I decided that this could be done by projecting each image from the camera onto the model after the shape had already been calculated. This would use texture mapped polygon drawing with Z-Buffers for the bulk of calculation.

Texture Mapping and Z-Buffers are discussed in the CST Part IB course titled 'Computer Graphics and Image Processing' [13] and the other courses it refers to.

System Design

The whole system now consists of a chain of algorithms:



Background Detection is an algorithm that produces grey-scale images as output. Black means background, and white means the object. Anything inbetween will be defined as not being known exactly. The algorithm works by converting the images into the HSB colour space, and finding the vector distance between each pixel and the colour of the background.

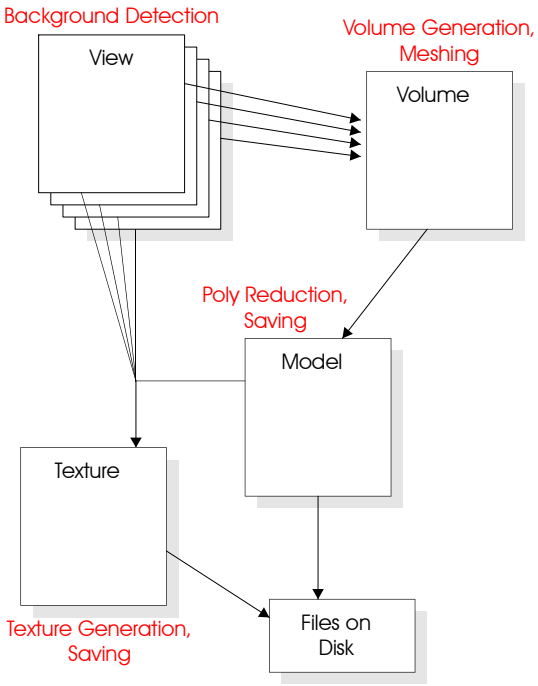
Volume Construction takes the images from background detection and projects them into a 3D volume. Any element in the volume is deemed to be present if any one of the images was background at that point (modified Radon transform).

Meshing uses the Marching Cubes algorithm to create a set of triangles that make up the object. These are placed in a Mesh object that contains the triangles and vertices.

Polygon Reduction uses Stan Melax's polygon reduction algorithm [12] to reduce the amount of polygons in the model.

Texturing first assigns each triangle an area of a large texture, and then makes an average of the colour of every pixel in the texture according to all the input images where that pixel is visible (e.g. Not occluded by the other side of the object).

Model Saving is not shown in the diagram, but saves the texture and model information to a file that will be read by the model viewer.



In order to make development easier, and to keep code tidy, I decided to implement the code in an object-oriented language. Data structures such as images, volumes, meshes and textures would all be represented by objects, with the algorithms that operate principally on them being implemented in methods in those objects. The diagram to the left shows how I intend the objects to link together.

Provision for Testing

I decided to start by implementing dummy objects containing blank methods, and then implementing the functions in the order in which data flows from the start (with the exception of Polygon Reduction, which is an extension). This increases the functionality of the project over time, while allowing me to test each part using as input the information I tested from the last finished block.

This is very much like the 'Bottom Up' software design philosophy, in that the parts of software needed for the rest to function are implemented first.

The program itself would contain a simple 3D preview window to allow objects to be debugged without introducing extra places for error (file load/save). Below is a table of the type of test output expected from each stage.

Background Detection	Grey-scale images showing what is considered background and what isn't.
Volume Construction	A 3D model rendered to the screen volumetrically – placing points everywhere the volume is deemed to be solid.
Meshing	A 3D wireframe model, followed by solid, and finally smooth-shaded (when normals have been calculated)
Polygon Reduction	A smooth-shaded 3D model
Texturing	A textured 3D model, as well as the texture output to a file.
Model Saving	The texture saved to a file, as well as the file produced itself. This should be viewable by the viewer – no other test information is feasible.

I planned to use POV-Ray to create a series of test images. These would have a known, solid background colour, and a known field of view. It is then much easier to check that my algorithms are working correctly.

Proof of Feasibility

As a brief proof that the idea would be successful, I decided to make a simple version of the Volume Construction algorithm. I did this by creating a program in Pascal - mainly because it was very quick to prototype in, and had good file I/O.

The program rendered a model consisting of 4 cubes of different heights from different angles. It then used these images in the way that has been mentioned to create a volume of points, of which slices were rendered. This allowed me to test if the method would work satisfactorily, and also to gauge how much processing power would be required for the algorithm.

Choice of Development Tools and Hardware

As well as creating the 3D model, the project should also provide a way of viewing it on-line. The obvious choice for this is Java, which will allow the object to be viewed from a web browser on any Java-enabled platform.

ActiveX controls are a possibility for this, and could be written in a variety of languages (C/C++/VB/Delphi), however it is Windows-only and so not particularly suitable.

For the project to work, very few extra resources are required. I needed a method of retrieving images from different angles, and a camera to capture the images.

Changing Angles

In the original Project Proposal, I suggested the use of a 3-jointed robot arm to capture images and keep lighting constant. I built it, but during this stage I realised that keeping a constant colour background would prove very difficult, and the original reason for it wasn't a problem after all.

The Robot Arm created for this project

The original reason was that lighting might be best kept constant if feature matching were to be done (but I have since decided to use the silhouette method). In texturing, the texture will be averaged over all views, and lighting will not be a problem in that phase either.

I decided the added complexity of positioning the arm wouldn't gain me any advantage, and instead opted for the fall-back idea of using a turntable.

The turntable (shown below) was made from perspex and contained an old hard disk stepper motor that required 200 steps per revolution. I still decided to go with the original idea of using the PIC microcontroller controlled stepper driver, which I would have to program.

The Turntable to be used

There are C compilers available for PIC microcontrollers, but due to the simple nature of the program I opted to write it in assembler. PIC assembler consists of only 32 instructions, and so is relatively easy to write and debug.

The PIC microcontroller would have to accept commands via an RS232 port and output the correct series of signals to turn the stepper motor. In case of the failure of this, I found a relay control box that worked from the PC printer port's data lines. Controlling the printer port data lines directly from Windows is non-trivial, and so this would be a fall-back measure using an old PC running DOS.

Camera

Before starting the project I made sure I had a camera capable of taking pictures on demand. The one I obtained was an Olympus C2020Z, which contained a serial port interface. Some free third-party software (PhotoPC [14]) can be used to control the camera. In case of failure, I also had access to another Olympus camera with the same interface.

Programming Language

The programming language that the project should be written in and any extra libraries that might be needed have to be considered. Java would be the obvious option here, but my experience has shown that it could be far too tedious and slow to be used for image processing.

Java makes access to raw image data quite hard, to the extent that its Graphics class doesn't contain a special getPixel or drawPixel method, and requires quite a large amount of code to retrieve image data. This would make code far more obscure than it needed to be, and the majority of my time would be

spent wrestling with the various APIs.

For this reason, I have opted to use C++. There are a vast array of libraries available to use. For this project, I require both to be able to read images that the digital camera has output (in JPEG format), and to be able to output the the result quickly and easily in 3D.

To load images, there are several options. The three of these I found most appealing were:

- The JPEG Image loading code
- ImageMagick Image Library
- DevIL Image Library

The JPEG code will load and save JPEG files quite happily, however it doesn't provide any other utility functions that I might need (such as flipping images, resizing, blurring, adjusting contrast). ImageMagick and DevIL are very similar in this - both allow images to be loaded and saved in many formats, and provide many manipulation functions. However DevIL appears to be alone in the fact that it supports integration with DirectX and OpenGL, which would be very useful from the point of view of displaying debug output. In addition DevIL allows the binary pixel data to be accessed extremely easily, so it seemed the obvious choice for this application.

For simple debugging I wanted an easy way of displaying 3D graphics. The only 2 viable options here were DirectX and OpenGL, and I quickly discounted DirectX because it wasn't multi-platform and took an excessive amount of work to initialise. The OpenGL GLUT library allows OpenGL to be initialised on any platform in just a few lines of code, and rendering in OpenGL is almost as simple as specifying the coordinates of each triangle.

Using these two libraries which are both very low level should reduce the amount of effort required to interface to the bare minimum, and allow more time to work on the algorithms.

The viewer will be written in Java. Most 3D toolkits for Java (Java3D, JavaGL) use native enhancements to make them interface with display hardware, and so need a separate installer.

These aren't really feasible for a website since a user is unlikely to want to download anything extra to view objects. While a few Java software renderers exist (Jazz3D), writing a simple Z-Buffered texture mapper is relatively easy and so I opted to write the entire applet myself.

Backups and Source Control

In order to ensure that I lost the minimum of information in the case of a hardware failure I decided to copy all files to another computer at the end of each day I worked on the project. Every week I would then write the files from each day off to CD-R to ensure I had a proper backup of everything that I had done.

Given the way I decided to design this project, it only consists of 5 or 6 source files (plus headers). I came to the conclusion that source control systems such as CVS would impose too much overhead because:

- There were very few source files
- From the backup I had a full copy of each day's work anyway
- Due to the way the project was being implemented (Bottom-Up), only one source file would require major changes each day.

Analysis of Project

The parts of the project I foresee having problems with are the retrieval of images, polygon reduction and texturing. I have left extra time for these in the schedule. Background detection, Volume Construction and the Java viewer should not pose any serious problems.

Implementation

I have chosen to describe the aspects of the project's implementation in the order in which data flows between them, to cut down on references between sections. Then order they were implemented in was exactly as described in the original Project Proposal.

Test Images

The first step of development was to produce a series of test images. This was done with POV-Ray [15], using the chessboard and pawns example.

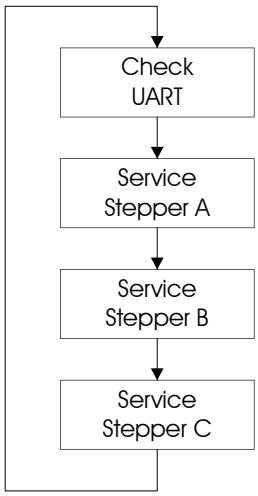
Pawns.pov - an example POV-Ray file by Douglas Otwell

The chess board was removed, and the pawns moved closer together. The camera position was set to a combination of sine and cosine, which rotated it around the pawns to produce a series of test images. I chose a white background, because this seemed easiest to recreate in real life (using spotlights on white paper). After looking at the images produced it was however obvious that specular reflections on the objects were appearing white too.

With careful lighting in a real-world situation (with non-reflective objects), it is unlikely this would ever happen, but due to the nature of the intended volume construction algorithm (where data is effectively boolean 'AND'ed together) any part of object that is mistaken for background will be removed from the computer's model, making a hole in it.

I re-rendered the test model on a red background, with the intention of using a bright red background for real-life models too. The choice of colour could in theory be anything that did not occur on the object being scanned. However, I had some red velvet (velvet is very good at giving a constant colour) so it seemed sensible to use a similar colour for the background o the test images.

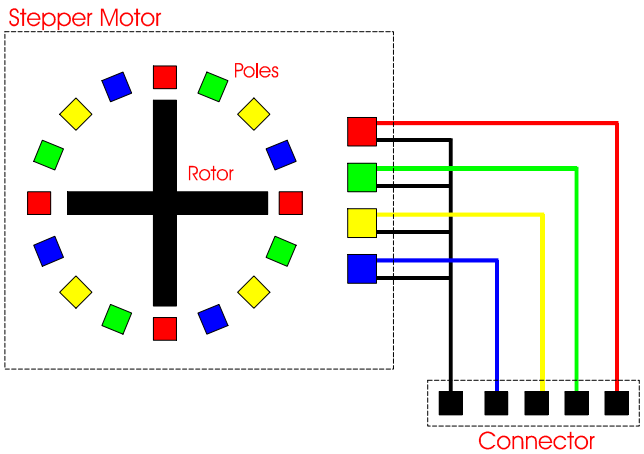
Stepper Motor Controller



Round-Robin in PIC

To program the Microchip PIC microcontroller on the stepper motor control board I used the Microchip In-circuit debugger. This allows the PIC's flash memory to be programmed while it is still in the controller board, and also provides good debugging functions such as breakpoints and watching of registers.

The controller itself needs only to perform very simple functions based on input it receives from an RS232 port (This the only realistic communications method provided by the controller board). It must move the stepper motor a certain number of steps in a certain direction by changing the state of 4 output pins over time. Since timing is not critical, I decided not to use interrupts and the PIC's built-in timer, and to instead opted for a round-robin. Each time around the loop the UART was checked for data, and then the state of the output pins was changed if enough time had passed.



Wiring of Stepper Motor

As the diagram above shows, there are 5 pins on the stepper motor. One of these must be connected to either ground or VCC. I chose VCC because

the stepper motor driver used is more efficient at pulling its outputs to ground. In order to rotate the stepper motor, the lines must be pulled low in order:

Clockwise	**RED, GREEN, YELLOW, BLUE, RED, GREEN, ...**
Anti-Clockwise	**RED, BLUE, YELLOW, GREEN, RED, BLUE, ...**

The protocol was designed to be as simple as possible for the PIC. The data rate chosen was 19200 baud (very little data needed to be transmitted), with no parity and one stop bit. The last 4 characters are stored in a buffer, and each new character received is checked to see if it is a command.

The length of the round robin loop was arranged so that the time taken between checks of the UART was far less than the time taken for a character to be received (at 19200 baud, less than 0.5ms per character), so that overflows could be avoided.

If the last character is A,B or C (referring to the 3 stepper motor drivers) the PIC looks back in the buffer for a direction (either + or – character) and a 2 digit hexadecimal number specifying the amount to move. Characters in hexadecimal numbers are lower-case to avoid confusion with commands.

I chose this reversed method of sending commands (you would expect the command character to come first) to avoid having to implement a state machine. Using this method, only a simple check needs to be done each time around the round robin.

There is also no notification that a stepper has finished moving. I intend the controller to only move the turntable by a few steps (4 or 8) and movement will be finished by the time the digital camera has received the request to take a picture. This limits the use of the controller, but for this project there is no point in adding any extra complexity that will not be used.

Examples of commands are :

`01+A`	Move stepper A forward by 1 step
`ff-B`	Move stepper B backwards by 255 steps
`7f+A7f+B7f+C`	Move all steppers forward by 127 steps
`D`	Disable all steppers (save on power)
`E`	Enable all steppers

Image Capture

No problems were encountered while programming the PIC, and the turntable was tested using TeraTerm. The digital camera was also controllable via an RS232 port, and the free command-line software (PhotoPC [14]) was used for this. It takes roughly a second to initialise the digital camera – plenty of time for the turntable to stop moving. It seemed sensible to do the capture of images with a simple batch file. Copying data to 'COM1:' saves on the unnecessary complexity of the Java Serial port APIs, and the command-line nature of the PhotoPC software was perfectly suited to this. Below is a small section of the batch file

used:

```
mode COM2 BAUD=19200 DATA=8 STOP=1 PARITY=n
copy rleft.txt COM2:
photopc -l COM1: snapshot
copy rleft.txt COM2:
photopc -l COM1: snapshot
copy rleft.txt COM2:
photopc -l COM1: snapshot
...
...
```

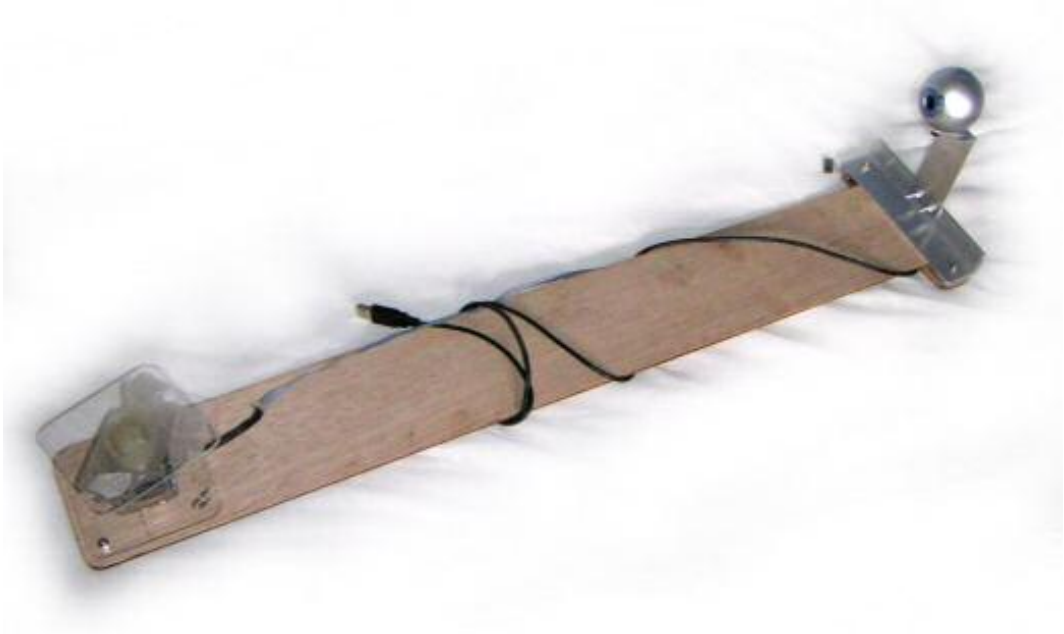
The file **rleft.txt** contained the command needed to rotate the stepper motor. In the case of this file, 25 rotations were made, and the contents of **rleft.txt** was:

```
08+A
```

After some investigation into minor inaccuracies in the sequence of images obtained I noticed that the base of the turntable was moving very slightly relative to the floor over the rotation of the model. This meant that over the course of what should have been a 360° rotation, the top of the table had rotated by only 350°.

The solution to this was simply to mount the turntable and camera on the same piece of wood. However, at the same time I decided it would be worthwhile to find a more universal method of image capture. PhotoPC supports around 10 cameras, but all of these are out of production now.

The solution I found was to use the Java Media Framework to capture images from any video input source (specifically a web cam). This has the added benefit that since the camera now used USB, the whole system was easily usable on a laptop, which usually only contains a single RS232 port. This made setting the apparatus up much easier.



The web cam and turntable mounted on a wooden platform

I made a small Java program to capture Images, and also to control the turntable by opening and writing to the file 'COM1:'. Again, the Java

Communications API was overkill for the very simple serial control that was needed.

For the coloured background, I was able to use dark red velvet. This has the nice property of having very little visible texture, and also very low reflectance, so the camera sees a very uniform colour. The images gained from this look very easy for software to distinguish background and foreground from.

Background Detection

Using the red velvet, the background of the image appears to be an almost constant colour (apart from changes in lighting). Detecting the background in this kind of image should prove a simple case of colour matching, assuming there is no trace of a similar colour in the model. The algorithm should produce an image as output where a value between 0 and 1 signifies how likely the pixel is to be background (0) or object (1).

The first part of the colour matching process is to convert the entire image from the RGB colour space to HSB colour space. This allows the matching algorithm to pay more attention to the hue of the background than the brightness, since changes in lighting throughout the picture are bound to occur.

The conversion to HSB colour space is relatively simple. Brightness and saturation are calculated from the formulae below.

$$HSB_B = MAX(RGB_R, RGB_G, RGB_B)$$

$$HSB_S = \frac{(HSB_B - MIN(RGB_R, RGB_G, RGB_B))}{HSB_B}$$

HSB$_{H,S,B}$ *are the Hue, Saturation and Brightness*

RGB$_{R,G,B}$ *are the Red, Green and Blue values*

The hue depends entirely on the colour. The RGB value is normalized, and then the correct value for hue is chosen depending on the relative amounts of the two non-zero colours left :

After the change to HSB colour space, a scaled vector distance is

calculated for each pixel between the pixel's HSB value and the intended HSB value. Scaling occurs in each dimension to allow for different sensitivity for hue and brightness.

The formula for this looks like:

$$D = \sqrt{\begin{matrix}(H_{scale} * (H_{match} - H_{img}))^2 + \\ (S_{scale} * (S_{match} - S_{img}))^2 + \\ (B_{scale} * (B_{match} - B_{img}))^2\end{matrix}} \quad - \quad offs$$

$\mathbf{H_{scale}}$, $\mathbf{S_{scale}}$ and $\mathbf{B_{scale}}$ *are values that determine how accurately H,S and B are to be matched*

$\mathbf{H_{match}}$, $\mathbf{S_{match}}$ and $\mathbf{B_{match}}$ *are the ideal values that the background should be*

$\mathbf{H_{img}}$, $\mathbf{S_{img}}$ and $\mathbf{B_{img}}$ *are the HSB values of the image at that pixel*

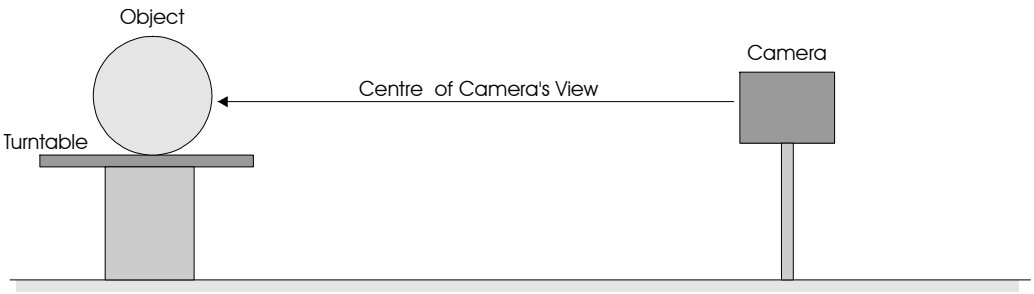
offs *is the value that determines how much we expect the background to vary in colour*

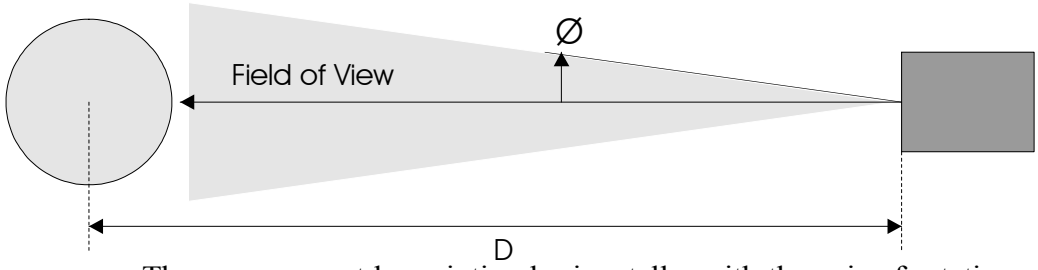
This is then clipped to the range [0,1], and saved into another image which is used for the volume construction stage.

Volume Construction

The information retrieved from background detection must then be used to produce the Volume of Scalars which will eventually produce the Polygon-based model. This section of the project depends very heavily on how the apparatus is set up.

To keep the calculations simple, the apparatus must be set up as follows:





The camera must be pointing horizontally, with the axis of rotation of the turntable positioned in the middle of its field of view, pointing parallel to the y axis of the sensor. The relative height of the camera to the turntable will not make any difference except to the position of the scanned object.

From a theoretical point of view, first a volume must be filled with the background information, where the X and Y from the source image map directly onto the volume, and every point on the Z axis is the same:

$$A[x, y, z] = B[x, y]$$

A *is the extruded Volume,* **B** *is the Background Image generated in the last section*

Then, the volume must have a perspective transform applied:

$$C[x, y, z] = A[\frac{x}{z+d}, \frac{y}{z+d}, z]$$

C *is the Volume with perspective*

And finally this must be rotated around the y axis by the angle the image was taken from. Note that the y axis can be left alone – this is a consequence of the camera being horizontal.

$$D[x, y, z] = C[(x\cos t)+(z\sin t), y, (z\cos t)-(x\sin t)]$$

D *is the fully transformed Volume*

An Accumulation Volume **Acc** must be created. This is originally initialised to all '1'. We know the volume **D** now contains information on whether each entry is deemed to be part of the background or foreground. We could simply perform a boolean AND between **Acc** and **D**.

However the Volume **D** doesn't contain boolean values, but instead values between 0 and 1. To merge the two volumes it makes sense to use the MIN operator, to find the minimum value out of the two volumes:

$$Acc[x, y, z] = min(Acc[x, y, z], D[x, y, z])$$

In the actual C++ code, the above transforms were done in one step without the intermediate volumes, by just transforming coordinates. This not only increases readability and speed, but also accuracy (performing the steps separately introduces more rounding errors than are needed).

The calculated volume rendered as points

Meshing

Meshing used the Marching Cubes algorithm. In order to implement this, I used the description in a paper called 'Polygonising a Scalar Field' [9]. The algorithm would add triangles by attempting to find the vertices that made them up, otherwise it would add new vertices to the model and use those instead. This meant that some connection information could be gained as the model was created and a lower computational cost.

The C++ object that contained the model was implemented using doubly-linked lists for triangles and vertices. This is of very little importance at the moment, but had very little extra overhead, and allows fast removal and addition of data when Polygon removal is to be implemented.

There were several reasons for this. The first was that connection information allows normals of vertices to be calculated, which helps to produce a far better looking, lit model (Gouraud Shading). This information also allows the Polygon Reduction algorithm to detect adjacent triangles more easily, and in addition reduces the file size of most saved models by removing redundant vertices. Fewer vertices also means less geometry calculations and so faster rendering.

The winged-edge data structure could have been used, but this allowed more information to be stored than was needed, and would have caused more trouble than necessary when it came to keeping all the information consistent while performing operations on the mesh.

On test runs with models, meshing took a very large amount of time to complete because of the searching of the vertices that was needed – a potentially $O(n^3)$ problem where **n** is the volume size.

However, by adding the most recent vertices at the front of the list, the spatial locality of the triangles can be exploited. Because marching cubes will

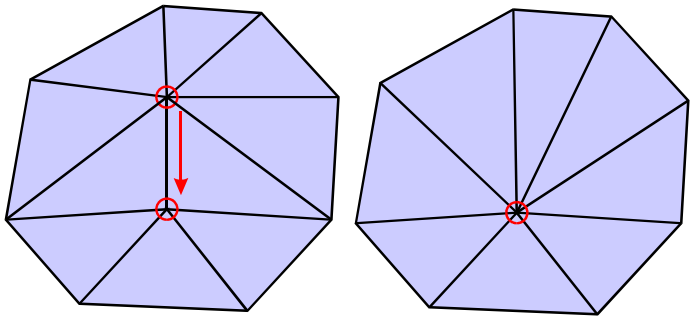
work its way in sequence through a volume, triangles in a 2D slice of the volume may only be joined to triangles in the previous slice. This then reduces the complexity of the algorithm to $O(n^2)$. Of course in order for this to happen, the search must stop when all the vertices created in the previous slice have been tested (and I did not implement this).

The Triangles produced by Marching Cubes rendered as a wireframe

To speed this up even more, a 3 entry vertex cache was created to take advantage of the fact that the marching cubes algorithm often produces adjacent polygons one after the other (especially in the case where two triangles are created in one 'cube').

Polygon Reduction

Polygon reduction uses the algorithm described in a GDMag article by Stan Melax [12]. Actually removing triangles from a mesh is not ideal. Instead, an edge of a triangle is chosen to be collapsed down to a point. This will result in one or more triangles being reduced to simple lines, which can then just be removed, leaving no gap in the model.



Removing triangles by collapsing an edge

Choosing an edge involves keeping information on which triangles are connected to each vertex, and also which vertices are adjacent. This information must not be calculated each time a triangle is removed, because it would make the edge-finding algorithm of $O(n^2)$ complexity, and therefore unusable for the amount of polygons that I would require it to work on. Instead it must be cached, and only the affected entries updated.

The amount of triangles adjacent to a vertex does not on the whole depend on the total number of triangles in the mesh. Therefore storing connection

information and updating is as it changes is only O(1) apart from the initial O(n) construction. This reduces the whole edge-finding algorithm to O(n) complexity.

Using the stored information, the edge which could be collapsed with the least amount of distortion to the model is found. Determining this properly would require a large amount of processing, so instead a simple heuristic is used, based on the length of the edge, and the angles between their normals (shown below). After the collapse, all adjacent triangles and vertices have their stored data recalculated.

$$cost(u,v) = \|u - v\| \times \max_{f \in Tu} \left\{ \min_{n \in Tuv} \left\{ (1 - f.normal \cdot n.normal) \div 2 \right\} \right\}$$

u,v *are the two edges*

Tu *is the set of triangles that contain* **u**

Tuv *is the set of triangles that contain both* **u** *and* **v**

x.normal *is the normal vector associated with the triangle* ***x***

I implemented the algorithm quite quickly, but had some difficulty getting it to work reliably (triangles would be removed, leaving just a hole in the mesh). In order to debug it I used the 3D preview display, highlighting polygons that were to be removed to see if they were the correct ones, and whether they had been reduced correctly. Text debug output was also produced containing the connection information of each vertex, and I noticed that vertices had an average connectivity of 2 triangles.

This meant that the model had not been joined correctly, and I traced the problem back to the add triangle method of the Model object. This was performing a normal floating point comparison of vectors, which was failing in most cases due to floating point inaccuracy in the Marching Cubes Algorithm.

The marching cubes algorithm calculates coordinates for each triangle by interpolating along the edges of cubes. In almost all cases the same calculation is performed more than once since each edge is shared between 4 cubes. The calculation is also performed in a different order, and this is almost certainly what caused the inaccuracies.

This was quickly resolved by altering the Vector equals function to check whether Vectors components were within a defined range, instead of the previous binary comparison.

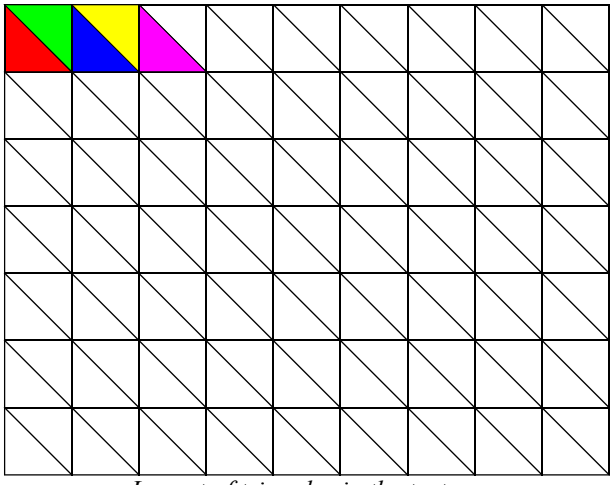
Texturing

Texturing was one of the most difficult aspects of the project, as I was able to find very few exact details from research papers. Papers such as [16] and [5] seem to use some form of texturing, but do not mention this at all. However they use millions of triangles to describe objects, and could possibly just colour these instead.

Texturing first requires every triangle in the model to be given unique,

non-overlapping coordinates in a texture, and then for the pixels in this texture to be calculated from all the different views, using the position of each triangle.

Giving unique coordinates to each triangle is not a hard problem – I divided the texture into a grid, and then divided each grid square into two right-angled triangles. Since the triangles are all roughly the same size, it is reasonable to do this as it is very compact.



Layout of triangles in the texture

A decision had to be made about whether each point on the texture was made from just the view that was deemed to represent it best, or from an average of every view that represented it. I came to the conclusion that the latter option would produce a far more robust algorithm, and would also require less calculation (finding the most representative would require calculating normals for every pixel).

Parts of an object occluding others

The algorithm also has to avoid mixing textures from parts of the object that overlap. In the picture on the left, the tail of the cat overlaps itself as well as the cat's head.

The solution to this problem that I settled on was to project the computer's model onto the original view. Then, when the texture for each triangle was to be drawn, the computer-generated image could be consulted to find out whether the triangle had been visible in that view or not.

This approach required a polygon-drawing algorithm that could render the 3D model from the source triangles, making sure nearer triangles were drawn over ones further away. I decided to use a Z-Buffer for this, because it was simple to implement, and information on how far away which parts of the object were was also very useful. To draw these triangles, the scan-line approach was taken, as explained in the CST IB Computer Graphics course [13].

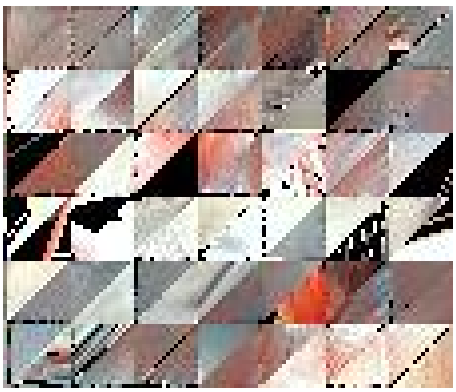
The code for this was also reused for the final part of the algorithm

with a few modifications. This calculated the texture that should be applied to each triangle for each view. The routine calculated texture coordinates exactly like a normal texture-mapped triangle draw routine, except that it copied pixels from the image to the texture.

The texture consisted of 32-bit accumulator planes for R,G and B, as well as an extra component that contained the number of pixels drawn. This allowed an average to be kept of all pixels drawn. After the algorithm had completed, the R,G and B components were divided by the fourth plane to give the final texture.

Since this part of the project relies heavily on image processing and rendering, it made sense to test and debug it by being able to view the various data structures. This was done by saving them to disk as image files. For example to save the rendered model, the R,G and B colour planes were used to encode the number of the triangle at that position. This allowed me to inspect the image with a paint program, and even extract the triangle number via the colour of each pixel.

Computer's model rendered with triangle number as colour



A section of texture from the scan of the cat model

Model Saving

One of the aims of this project was to produce a model that could be used for applications on the Internet – in order for this to be viable, the file that was produced had to be small. To save the texture, there was really only one obvious choice – JPEG. There is already good support for this in Java, and it has very good compression for natural images (because the texture was produced from a camera in the first place). The lossiness of the format is not much of a problem as a user is unlikely to notice when the texture is wrapped around an object.

For the model file itself, I decided to save the vertex coordinates, texture coordinates, and vertices separately. I could have used triangle strips, but it would have been more difficult.

For an example of a model format I decided to look at Quake 2's MD2 file [17]. The MD2 file had to store many frames of animation in a single small file. It used no compression, but stored all data in a binary format. Vertex coordinates were stored using 8 bits per dimension, but with a scale and offset factor for the entire object.

This seemed perfect for the model format, especially since the marching cubes grid size was unlikely to be greater than 256^3 (mainly because of processing time), and the model would always be the same size, eliminating the need for scale and offset information. I used similar ideas from the MD2 file format by using 16 bit values for texture coordinates, as well as 16 bit values for the vertex indices in textures. While I could have applied compression to the object file to make it smaller (vertex indices would never use all 16 bits) I decided that a file written in this form would probably be small enough.

The exact format of the model file is given in the appendix.

The implementation of this part of the project went fine, however there is no simple way to test it without the Java viewer. In reality I developed the code to save models and the Java code to read them at the same time, and this allowed me to test both parts of the software.

Java Viewer

The Java viewer consists of 4 main classes: The Applet, the Model, the Graphics class and a Texture Loader:

Applet Class (JModel)

This class acts as glue. On initialisation it reads a parameter from the web page to determine which model and texture to load, and attempts to read them. It initialises the Graphics class, and responds to mouse and paint events to allow the user to interact with the object.

Model Class (Model)

The Model class contains a list of triangles and vertices, as well as the code to load the model file. It also contains a list of transformed vertex coordinates – this reduces the calculation required for drawing at the expense of extra memory usage.

Graphics Class

The Graphics class uses an array of integers, and the ***MemoryImageSource*** class to allow raw data to be written to an Image and subsequently onto the screen. It used a Z-Buffer based texture mapper that used exactly the same principle as that used in the Texture Generation above.

Texturing didn't need to be perspective correct because the polygons are on average only around 16 pixels wide – and perspective problems in a triangle so small are almost imperceptible. Software renderers often only correct for perspective every 16 pixels, so there would be no difference in graphics quality between the two.

To increase speed the texture mapping polygon routine assumes the texture is 512 pixels high and wide. By performing a boolean AND on the texture coordinates, range check errors can be avoided without costly conditional jumps.

Texture Loader Class

The Texture loader is a class that implements the ***ImageConsumer*** interface – this is the only, rather long-winded way of retrieving image information from Java that I was able to find. It contains an integer array which is updated as the image is loaded – this array is passed to the Graphics class for texture mapping.

Implementation and Testing of the Java Viewer

I implemented the Applet and Model classes first, drawing the model first as points to ensure the model file was loaded correctly (which is when the sign bit problem was found), and then adding a solid polygon renderer, followed by a modification to display polygons with textures.

I decided not to implement real-time lighting, as the texture already contained enough lighting information (the average of all lighting over the course of the rotation) to make the model look realistic.

Evaluation

Since this project was very dependent on real-world data, there were only a certain number of things that could be done to evaluate its performance. It was tested on several real-world objects, and the computer-generated test object of pawns.

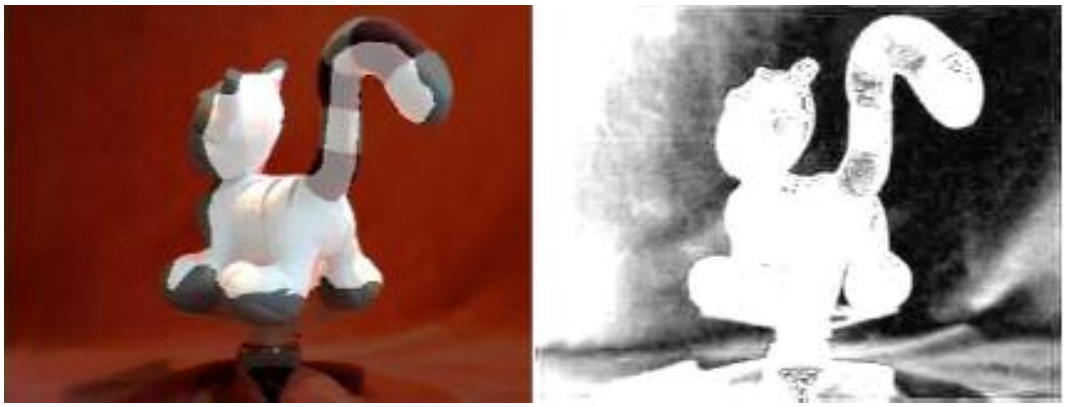
Testing

The obvious evaluation of the system is to attempt to reconstruct a variety of objects and look at the system's performance with each of them. If was in possession of an object and an accurate computer model of it then I would be able to compare the two mathematically and produce hard figures.

Unfortunately I wasn't able to obtain any complex-shaped objects with models associated with them, so the comparisons I am able to perform are more limited.

Testing Accuracy

As has been mentioned previously, in order to debug the texture generation code I saved the rendered model to disk as an image file. By overlaying this model onto the original image I am able to see easily how close a match it is to the original.



Left: Original Image overlaid with computer's model

Right: Computer's Foreground/Background differentiation

Above left is this image overlaid onto the original. It is far smaller than the original, and is missing one ear. One thing responsible for this is the background detection. You can see on the right the image that was produced by the background detection algorithm.

While parts of the background in this are white, this is of no consequence. What causes the most trouble is the way that parts of the test object are dark. If any part of the object in any image is darker than 50% white, then it will be removed from the computer's representation of the object. This has

happened in several places on this object, including the aforementioned missing ear.

Projection is also to blame for this problem. As the object rotates, if the perspective assumed is too great or too small then the background 'eats' into the representation of the object, because the size of features is either under or over-estimated.

Currently background detection and projection use constant values that need to be changed by hand to produce the best results – obviously some form of automatic calibration from images would be very useful and would make a good addition to the software. One would hope that once set up, none of these settings would need to be changed. However, due to changes in ambient lighting throughout the day, and the camera's auto white-balance compensation slight changes are needed to produce the correct results.

In some cases the object may have a colour very similar to that of the background, requiring more sensitive (and so more conservative) background detection to be used. The best solution to this problem is to use a different colour background though.

A much better version of the scan can be seen below. This was created mostly by changing values to move the camera closer to the object and increase its field of view. In places the computer's model is too large, but this could be attributed to the background detection which in order to avoid classifying object as background classifies large areas of background as object. Upon close inspection it can be seen that the top of the turntable appears to be considered part of the object too – this backs up the background detection argument.

Computer model overlaid onto real image after modifications to background detection and perspective

The fact that part of the tail and ear are missing is probably down to the Projection settings not being perfectly right again. It may also be that the camera is not perfectly horizontal. Again, automatic calibration may be able to detect these problems and remedy them.

Testing on Different Models

I decided to use 4 Models to test out the reconstruction. One of these was a toy cat, the other was a toy of a cartoon cow, with some very complex shapes. There was also a shiny metal water spray can, and some china with a reasonably complex pattern on it. The results are shown below, rendered by the Java Viewer rather than OpenGL:

These unfortunately all have a red tint, which is due to the red background, reflected light, and the fact that with polygon reduction the reduced model may be slightly bigger than the original (and will pick up background as texture). The fact that the computer generated pawns don't appear to suffer as badly hints that this is probably just the camera. This could easily be remedied by shifting the hue of the texture slightly.

However it should be noted that for each one of these models, a few minutes of fiddling with values had to be done to get the best result. While the water spray has a small section of handle missing it shows a remarkable amount of accuracy given the size of feature it was asked to extract (around 1.5mm by 4mm). Shiny objects are usually considered the most difficult to reconstruct, but due to the nature of the algorithm it has worked almost perfectly. Texturing has been of the average of the reflections, which happen to be red.

As can be seen very clearly in the model of the jug, polygons at its top are incorrectly textured. This is because the software was able to infer the presence of a surface, but was not able to find an image from the camera in which the colour of it could be viewed.

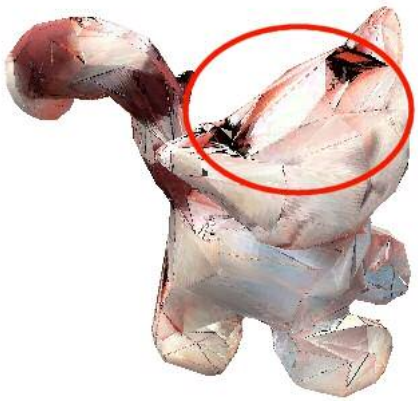
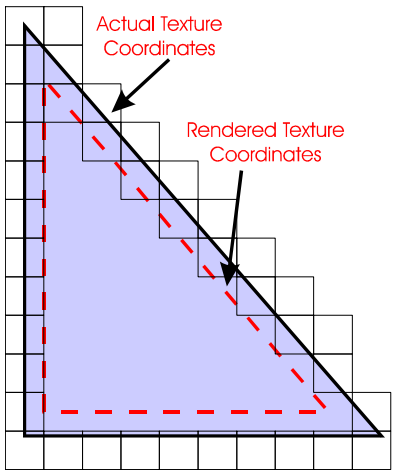


Image showing texture holes in the top of the model

The same artefacts occur in all objects. They are especially prominent in models with big flat tops. One solution to this problem is to just make these polygons transparent. Another solution is to make an algorithm that searches for the nearest pixel in the texture that has a defined colour, and just use that.

Small coloured lines can also be seen in the models that are being displayed. As the diagram below shows, if the actual texture coordinates are rendered, pixels from outside the triangle's texture will be rendered in error.

The artefacts have been reduced by moving the texture coordinates for the renderer inside the coordinates for the textured triangle. Unfortunately I moved the coordinates in an acceptable amount for the 1024x1024 textures produced in OpenGL. When these were resized to 512x512 for the Java viewer, the problem became worse, and the coloured lines re-appeared. This could be fixed either by moving the rendered coordinates in, or by using 1024x1024 textures for the Java viewer.



Texture coordinate misalignment intended to reduce texture artefacts

Comparing Performance

As has been mentioned, it is hard to compare quantitatively the performance and accuracy of my system with other systems available, because I have very little information on other systems. It is obvious visually that the system lacks the quality of the professional scanning systems.

However the level of detail obtained is still quite acceptable for some uses, and considering the low cost of hardware compared with other systems it may still be a viable proposition. With proper calibration and good background detection the system may produce models of good enough quality to be used for on-line shopping. As it is, there may still be uses for gamers and the growing community amateur computer-generated film makers.

As for speed, I have only heard performance figures mentioned for the Olympus ScanTop – a very similar product. This has been said to take around 2 hours per object depending on the detail settings used but does not mention detail settings or computer specification.

However, compared to that, the average of 2 minutes computation time on a 1Ghz laptop for this system would seem to be very encouraging. Using a web cam and turntable the time for image acquisition could easily be reduced to below 30 seconds.

Below is a table showing the time taken (in seconds) for each stage of the reconstruction of the cat model for different volume sizes. This was produced on a 1ghz Laptop.

Volume Size	*Loading, Background and Volume*	*Meshing*	*Reduction*	*Texturing*	*Total*
48	6	<0.5	<0.5	5	14
64	8	4	1	5	21
80	11	12	6	5	37
128	23	109	68	5	209

It can be seen that meshing scales very badly with volume size, and a better data structure should probably be used (or the previously mentioned search enhancement made). The slowdown is also partially due to excess noise in the test set of images. With less noise, less useless polygons would be reduced and both meshing and reduction would be far faster.

Results

The full results obtained from the software can obviously not be shown here, however they are available to be viewed in full 3D on my website:

Goals Achieved

I believe I have achieved my goals of creating a model reconstruction system. I have managed to fulfil the tasks I set for myself in the requirements analysis:

- Should run on a normal PC
- Should not require the PC to be modified in any way to interface hardware
- Will automatically take pictures of an object from known angles
- Will complete calculations in a useful amount of time. e.g. less than an hour
- The model produced should be an obvious likeness to the real-life model

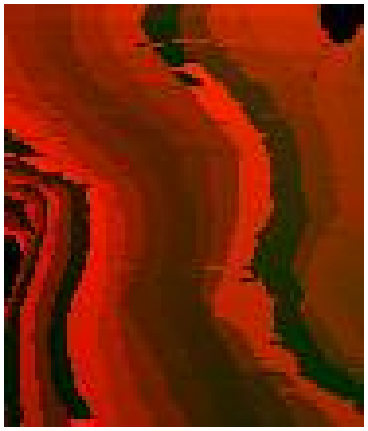
Residual Bugs

There still appear to be two minor faults in the code:

Occasionally in polygon reduction one or two polygons will end up completely removed, leaving a gap in the model. This only happens sometimes with huge polygon reductions such as 20:1, and has proved very difficult to track down due to the huge amounts of data around when the problem occurs.

It is possible that small errors creep into the data structures that maintain the state of connection of the model. These may cause the polygon to look unconnected and so to be wrongly removed. Inserting more checks into the code to report any unconnected triangles may help to isolate this problem.

The second fault does not appear to effect operation, but it may cause minor texturing artefacts. There appears to be a small error in one of the triangle draw routines that occasionally makes it draw a small (slightly offset) line at the end of a triangle. As can be seen on the picture on the left, small dashes are wrongly added to the rendered image.



Texturing artefacts (the short horizontal lines)

This proves a slight mystery, because all 3 triangle draw routines (Model Draw, Texture Generation, and Java Model Draw) are basically the same, and yet only the Model Draw texture mapping routine appears to contain the bug.

Attempts to fix this have included restricting the range of lines that are scanned to only those that are within the polygon's bounds. This appears to have reduced the problem, as well as increasing rendering speed. However it has not eliminated it and scrutiny of more pictures shows that problems don't always occur at the edges of polygons.

Conclusions

Summary of Findings

I believe that this project produced very good results given the quality of the apparatus that was used, and the very low processing time needed to create the 3D models. A few potential problems with this style of reconstruction were encountered - such as the lack of texture on the top and bottom of objects.

Overall though, I believe this method has a lot of promise for applications where accuracy is not needed. While the lack of concave surfaces would appear to be a major problem, visual analysis of the test objects that have been scanned shows that when a texture is added, artefacts become much harder to detect. For simple objects, the lack of concave surfaces is quite acceptable (e.g. the cat).

Hindsight

Taking into account the problems that have been encountered with texturing (specifically the lack of texture in areas that the camera cannot see), I think restricting the camera to a horizontal view of the turntable was a bad idea. Facing 30 degrees downwards would have been far better, however this could be potentially hard to set up.

Ideally the object could be placed on a test pattern, and the field of view, distance and rotation of the camera could be automatically calculated from images taken from any angle by hand. This not only allows the texture to be reconstructed without gaps, but means that there is no longer any need for a turntable, further reducing the cost of the system down to just that of the software and a sheet of paper with the pattern on it.

Further improvements could place the object on a sheet of glass or a very thin stand so images of the bottom could be taken too.

Further Work

After the creation of the software, a lot of time was spent changing the various values I had available for background detection and perspective in order to gain the most accurate result. Being able to automatically calibrate the background using an image with no object on the turntable (or an area of image known to always be background) could save a lot of time, and potentially match the background much better, giving more accurate results. In addition taking an image of a test object (perhaps a wireframe cube) on the turntable would allow the software to calculate perspective itself.

Another small oversight in my design was the lack of a settings file to save the settings. All settings were constants in a header file, because originally I had envisaged that they would not require modifying. Ideally, this information

would be stored alongside the images for a particular model. This is a reasonably simple addition, and should have been specified as part of the project in the first place.

As was mentioned in the last section, a pattern that allows the camera's position to be calculated from the source image would solve a lot of problems. These is nothing in the software that means the camera must work horizontally, so at the expense of some speed all existing algorithms could easily be adapted. There already appears to be some software that works on a similar principle.

As has been mentioned previously, there are sometimes faint lines at the edge of triangles. This could be reduced greatly by using a more intelligent algorithm to place triangles on a texture. This would place them according to their neighbours on the 3D object, so colours at their borders would match.

Bibliography

Virtually all research for this project was done on the Internet, due to the large amount of on-line publications in this area by companies, research groups and individuals.

[1] RSI GmbH, PhotoModeller
http://www.rsi.gmbh.de/photomodeler_e.htm

[2] Peter Toft, The Radon Transform
http://eivind.imm.dtu.dk/staff/ptoft/Radon/Radon.html

[3] Olympus, ScanTop Brochure http://www.flexist.de/pdf/h28z288.pdf

[4] Miguel Sainz, Renato Pajarola, Antonio Susin, Photorealistic Image Based Objects from Uncalibrated Images
http://www.ics.uci.edu/~msainz/publications/vis03.pdf

[5] Yasemin Kuzu, Photorealistic Object Reconstruction Using Colour Image Matching http://www.fpk.tu-berlin.de/~kuzu/Pub/isprs2002_voxel.pdf

[6] Alexander Stevenson, Voxels and Volumetric Representation
http://www3.telus.net/ah/voxels/voxels.htm

[7] H. Gouraud, Continuous Shading of Curved Surfaces

[8] Paul Bourke, Polygonising a Scalar Field Using Tetrahedrons
http://astronomy.swin.edu.au/~pbourke/modelling/polytetra/

[9] Paul Bourke, Polygonising a scalar field
http://astronomy.swin.edu.au/~pbourke/modelling/polygonise/

[10] W.E. Lorensen, H.E. Cline, Marching Cubes: a high resolution 3D surface reconstruction algorithm

[11] Dr Neil Dodgson, CST II - Advanced Graphics 2003-04
http://www.cl.cam.ac.uk/Teaching/2003/AdvGraph/

[12] Stan Melax, A Simple, Fast, and Effective Polygon Reduction Algorithm

[13] Dr Peter Robinson, Computer Graphics and Image Processing
http://www.cl.cam.ac.uk/Teaching/2003/Graphics/

[14] Eugene Crosser, PhotoPC

[15] PovRay.Org, POV-Ray Website http://www.povray.org

[16] Miguel Sainz, Renato Pajarola, Antonio Susin‡, Photorealistic Image Based Objects from Uncalibrated Images
http://www.ics.uci.edu/~msainz/publications/vis03.pdf

[17] John Carmack, MD2 File Format http://www.idsoftware.com

Appendices

JMOD Model File Format

Sections

The JMOD file consists of 3 blocks, which come one after the other in the file:

Header

Vertices

Triangles

The following data types are used:

uint32	unsigned 32-bit integer
uint16	unsigned 16-bit integer
int16	signed 16-bit integer

Header

Offset (Hex)	*Type*	*Description*
0x00	uint32	Magic number = 'JMOD' Ensures this is the correct file
0x04	uint16	Vertex Count (**VCOUNT**)
0x06	uint16	Triangle Count (**TCOUNT**)

Vertices

This block consists of VCOUNT copies of the structure below:

Offset (Hex)	*Type*	*Description*
0x00	int16	(X*32767) (-1<=val<1)
0x02	int16	(Y*32767) (-1<=val<1)
0x04	int16	(Z*32767) (-1<=val<1)

Triangles

This block consists of TCOUNT copies of the structure below:

Offset (Hex)	*Type*	*Description*
0x00	uint16	Index of 1st vertex in previous block

Offset (Hex)	***Type***	***Description***
0x02	uint16	Index of 2nd vertex in previous block
0x04	uint16	Index of 3rd vertex in previous block
0x06	uint16	1st Vertex X Texture Coordinate (tx*65536)
0x08	uint16	1st Vertex Y Texture Coordinate (ty*65536)
0x0A	uint16	2nd Vertex X Texture Coordinate (tx*65536)
0x0C	uint16	2nd Vertex Y Texture Coordinate (ty*65536)
0x0E	uint16	3rd Vertex X Texture Coordinate (tx*65536)
0x10	uint16	3rd Vertex Y Texture Coordinate (ty*65536)

POV-Ray Pawns

```
// Modified Persistence Of Vision raytracer version 3.5 sample file.
// ---------- Original file
// "Pawns", a study in wood... three pawns on a chessboard
// File by Douglas Otwell
// ----------------------------
// Modified by Gordon Williams to remove chessboard
// and put Pawns closer together

global_settings { assumed_gamma 1.8 }

#include "colors.inc"
#include "shapes.inc"
#include "textures.inc"

//
//   Yellow pine, close grained
//
#declare Yellow_Pine = texture {
  pigment {
    wood
    turbulence 0.02
    color_map {
      [0.000, 0.222  color red  0.808  green  0.671  blue  0.251  filter
0.000
               color red  0.808  green  0.671  blue  0.251  filter 0.000]
      [0.222, 0.342  color red  0.808  green  0.671  blue  0.251  filter
0.000
               color red  0.600  green  0.349  blue  0.043  filter 0.000]
      [0.342, 0.393  color red  0.600  green  0.349  blue  0.043  filter
0.000
               color red  0.808  green  0.671  blue  0.251  filter 0.000]
      [0.393, 0.709  color red  0.808  green  0.671  blue  0.251  filter
0.000
               color red  0.808  green  0.671  blue  0.251  filter 0.000]
      [0.709, 0.821  color red  0.808  green  0.671  blue  0.251  filter
0.000
               color red  0.533  green  0.298  blue  0.027  filter 0.000]
      [0.821, 1      color red  0.533  green  0.298  blue  0.027  filter
0.000
               color red  0.808  green  0.671  blue  0.251  filter 0.000]
    }
    scale 0.1
    translate 10*x
  }
}

// Yellow_Pine layer 2
texture {
  pigment {
    wood
    turbulence 0.01
    color_map {
      [0.000, 0.120   color red  1.000  green  1.000  blue  1.000  filter
1.000
               color red  0.702  green  0.412  blue  0.118  filter 0.608]
      [0.120, 0.231   color red  0.702  green  0.412  blue  0.118  filter
0.608
               color red  0.702  green  0.467  blue  0.118  filter 0.608]
      [0.231, 0.496   color red  0.702  green  0.467  blue  0.118  filter
0.608
               color red  1.000  green  1.000  blue  1.000  filter 1.000]
      [0.496, 0.701   color red  1.000  green  1.000  blue  1.000  filter
1.000
               color red  1.000  green  1.000  blue  1.000  filter 1.000]
      [0.701, 0.829   color red  1.000  green  1.000  blue  1.000  filter
1.000
               color red  0.702  green  0.467  blue  0.118  filter 0.608]
      [0.829, 1       color red  0.702  green  0.467  blue  0.118  filter
0.608
               color red  1.000  green  1.000  blue  1.000  filter 1.000]
    }
    scale 0.5
    translate 10*x
  }
}

//
//   Rosewood
//
#declare Rosewood = texture {
  pigment {
    bozo
    turbulence 0.04
    color_map {
      [0.000, 0.256   color red  0.204  green  0.110  blue  0.078  filter
0.000
               color red  0.231  green  0.125  blue  0.090  filter 0.000]
      [0.256, 0.393   color red  0.231  green  0.125  blue  0.090  filter
0.000
               color red  0.247  green  0.133  blue  0.090  filter 0.000]
      [0.393, 0.581   color red  0.247  green  0.133  blue  0.090  filter
0.000
               color red  0.204  green  0.110  blue  0.075  filter 0.000]
      [0.581, 0.726   color red  0.204  green  0.110  blue  0.075  filter
0.000
               color red  0.259  green  0.122  blue  0.102  filter 0.000]
      [0.726, 0.983   color red  0.259  green  0.122  blue  0.102  filter
0.000
               color red  0.231  green  0.125  blue  0.086  filter 0.000]
      [0.983, 1       color red  0.231  green  0.125  blue  0.086  filter
0.000
               color red  0.204  green  0.110  blue  0.078  filter 0.000]
    }
    scale <0.5, 0.5, 1>
    translate 10*x
  }
  finish {
    ambient 0.5
    diffuse 0.8
  }
}

// Rosewood layer 2
texture {
  pigment {
    wood
    turbulence 0.04
    color_map {
      [0.000, 0.139   color red  0.545  green  0.349  blue  0.247  filter
1.000
               color red  0.000  green  0.000  blue  0.000  filter 0.004]
      [0.139, 0.148   color red  0.000  green  0.000  blue  0.000  filter
0.004
               color red  0.000  green  0.000  blue  0.000  filter 0.004]
      [0.148, 0.287   color red  0.000  green  0.000  blue  0.000  filter
0.004
               color red  0.545  green  0.349  blue  0.247  filter 1.000]
      [0.287, 0.443   color red  0.545  green  0.349  blue  0.247  filter
1.000
               color red  0.545  green  0.349  blue  0.247  filter 1.000]
      [0.443, 0.626   color red  0.545  green  0.349  blue  0.247  filter
1.000
               color red  0.000  green  0.000  blue  0.000  filter 0.004]
      [0.626, 0.635   color red  0.000  green  0.000  blue  0.000  filter
0.004
               color red  0.000  green  0.000  blue  0.000  filter 0.004]
      [0.635, 0.843   color red  0.000  green  0.000  blue  0.000  filter
0.004
               color red  0.545  green  0.349  blue  0.247  filter 1.000]
      [0.843, 1       color red  0.545  green  0.349  blue  0.247  filter
1.000
               color red  0.545  green  0.349  blue  0.247  filter 1.000]
    }
    scale <0.5, 0.5, 1>
    translate <10, 0, 0>
  }
  finish {
    ambient 0.5
    diffuse 0.8
  }
}

//
//   Camera ...
//
camera {
  location <8*sin(clock*3.14159*2.0), 1, 8*cos(clock*3.14159*2.0)>
  direction <0, 0, 1.5>
  angle 60
  up <0, 1, 0>
  right <4/3, 0, 0>
  look_at <0, 1, 0>
}

light_source { <100.0, 400.0, -600.0> color White }

//   a back-light to create a highlight on the board
light_source { <12.0, 4.0, 12.0> color White }

//
//   Pawn
//
#declare pawn = union {
  difference {
    object { Disk_Y scale <8, 12.7468, 8> }
    quartic {
      < 1.0,  0.0,  0.0,   0.0,    2.0,  0.0,  0.0,  2.0,  0.0,-738.0,
        0.0,  0.0,  0.0,   0.0,    0.0,  0.0,  0.0,  0.0,  0.0,   0.0,
        1.0,  0.0,  0.0,   2.0,    0.0, 162.0,  0.0,  0.0,  0.0,   0.0,
        1.0,   0.0, -738.0,   0.0,   6561.0>
        sturm
    }
  }

  quartic {
    < 1.0,  0.0,  0.0,   0.0,    2.0,  0.0,  0.0,  2.0,  0.0, -132.5,
      0.0,  0.0,  0.0,   0.0,    0.0,  0.0,  0.0,  0.0,  0.0,   0.0,
      1.0,  0.0,  0.0,   2.0,    0.0, 123.5,  0.0,  0.0,  0.0,   0.0,
      1.0,  0.0, -132.5,  0.0,  3813.0625 >
      sturm
```

```
    translate -11.2468*y
  }

  quartic {
    < 1.0,  0.0,  0.0,   0.0,    2.0,  0.0,  0.0,  2.0,  0.0, -132.5,
    0.0,  0.0,  0.0,   0.0,    0.0,  0.0,  0.0,  0.0,  0.0,   0.0,
    1.0,  0.0,  0.0,   2.0,    0.0, 123.5,  0.0,  0.0,  0.0,   0.0,
    1.0,  0.0, -132.5,  0.0,  3813.0625>
    sturm
    translate 11.2468*y
  }

  //  Base
  intersection {
    object { Disk_Y
      scale <12, 3, 12>
      translate -15.7468*y
    }
    object { QCone_Y
      translate -2*y
    }
  }

  //  Ball on top
  sphere { <0, 17.7468, 0>, 7 }

  bounded_by { object { Disk_Y scale <14, 26, 14> } }

  translate 18.7468*y
  scale 0.06
}


//  Now let's put the pieces together

//  Pawn 1
object { pawn
  texture {
    Yellow_Pine
    finish { phong 0.8 }
  }

  rotate 60*y
  translate <-1, 0, 1>
}

//  Pawn 2
object { pawn
  texture {
    Yellow_Pine
    finish { phong 0.8 }
  }

  rotate 30*y
  translate <1, 0, -1>
}

//  Pawn 3
object { pawn
  texture {
    Rosewood
    finish {
      phong 1.0
      ambient 0.5
      diffuse 0.7
    }
  }

  rotate 30*y
  translate <0.72, -0.24, 0>
  rotate 96.2052*z
  translate <0, 0, 0>
}

//  a background glow to add interest
sphere { <0, 0, 0>, 1000
  inverse
  hollow on

  pigment {
    gradient y
    color_map {
      [0.25 0.46  color Red color Red]
      [0.46 1.001 color Red color Red]
    }
    scale 2000
    translate -1000*y
  }
}
```

Project Proposal

Gordon Williams
Downing College
GW235

CST Part II Project Proposal

Textured 3D Model Reconstruction from a series of Calibrated Images

20th October 2003

Project Originator : Gordon Williams

Project Supervisor : James Srinivasan

Signature : .....

Director of Studies : Richard Stibbs

Signature : .....

Overseers : Gavin Bierman and Steven Hand

Introduction

The aim of this project is to develop a system to produce a likeness of a physical object that may be viewed from a website. It will involve using a digital camera to take pictures of an object from known positions, and then applying computer vision techniques to construct a three-dimensional representation of that object inside the computer. This will then be textured with the captured pictures to add realism and hide imperfections in the reconstructed model. The model will be saved, and a Java applet will be made that allows it to be viewed from a web browser at arbitrary angles.

Work to be undertaken

The project will consist of the following main parts:

- Finding an automated means taking a series of images of an object from known positions
- Producing a polygon mesh from the series of images
- Projecting images onto the mesh to produce a textured model
- Saving the mesh and texture in a compact form
- Producing a Java applet to view the mesh and texture from a web browser

Starting Point

At the start of the project, I will have a digital camera, and a method of taking images from known positions. I have attempted a similar project before, and i have some knowledge of the algorithms required and problems that may be encountered. The project will not use source code that was previously written, although it will make use of libraries such as OpenIL and OpenGL. It will also require the use of third-party software to control the digital camera.

I have done some research into the subject (although more remains to be done). Some research papers i have that i currently think are of merit are:

- Polygonising a scalar field - Paul Bourke
- Polygon Reduction (Game Developer Magazine Nov 98) - Stan Melax
- 3D human body model acquisition from multiple views- I.I. Kakadiaris, D. Metaxas
- Roxels: Responsibility Weighted 3D Volume Reconstruction - Jeremy De Bonet, Paul Viola
- Efficient Volumetric Reconstruction from Multiple Calibrated Cameras - Manish Jethwa
- On Combining Shape from Silhouette and Shape from Structured Light - Srdan Tosovic, Robert Sablatnig, Martin Kampel
- Fast Volume Carving - Soon Hyoung Pyo

Structure of Project

The first part of the project involves automatically taking a series of pictures of an object from known positions. The ideal way of doing this seems to be to use a camera mounted on a robot arm that can move around the object. I already have a robot arm capable of doing this which I plan to use, however if this proves to be infeasible I will mount the object on a rotating platform and keep the camera stationary.

Once the pictures have been gathered, a series of algorithms will be used to create a 3-dimensional volume representing the object, which will then be converted to a mesh of polygons. As an extension, this will then be reduced to a lower-detail mesh.

To make the mesh look more realistic, a texture will be applied to it, which will be derived from the pictures of the object. The final part of the project will consist of making a Java applet for a website capable of viewing the textured model from any angle the user chooses.

The Image processing code will have to be written in C++ since there will be a very large amount of data being processed, and the overhead Java places on structures would make most structures exceed the amount of RAM on most desktop computers. The robot arm controller contains a PIC Microcontroller which needs programming, and this will be best done in PIC Assembler due to the limited resources of the PIC. The web-browser based viewer will be written in Java because at the moment it is the most compatible method of embedding user-defined code in web browsers.

Success Criteria

The project will be judged to be a success when:

- Pictures of a test object (left) can be taken automatically from known viewpoints.
- A series of these pictures can be used to construct a textured polygon mesh which is a likeness of the test object.
- This likeness may be viewed at different angles via a website.

Resources Required

For the project I will require the following items, all of them have already been acquired and there are no issues about availability.

PC	Since external hardware will need to be connected to a computer to retrieve the sequence of pictures it is not feasible to use a PWF machine where i would not have sufficient access privileges. I propose to use my computer which I have full control of.
Digital Camera	This will be needed to take pictures of the object. The camera model i plan to use can be told to take pictures via a serial link.
Robot Arm	This is a 3-jointed robot arm that may only move in one plane. The camera will be mounted on it, and images will be taken without altering the illumination on the subject.
PIC Development Kit	The robot arm is controlled by a controller board containing a PIC Microprocessor. This needs to be programmed to control the arm.

Backup Resources

In the event that something fails, i have the several strategies. The

work on the PC will be backed up to Pelican at the end of each day, and every week will be written onto a CD. If there is a hardware failure I can obtain another PC within a week, and can write my dissertation and software on the PWF in the mean time.

The main camera I plan to use is an Olympus 990Z, but if this fails i have an Olympus C2020Z, which will be a drop-in replacement. If the Robot Arm fails, i have a turntable which can be used instead. If the PIC development kit fails, i also have a spare.

Plan of Work

Work will be divided into 2-week slices, starting from the Monday after the proposal is submitted. Vacations will be used to complete any milestones in the event they were not reached in the given timespan, and the last Vacation will be used to write part of the dissertation in.

Since this project relies mostly on algorithms, and doesn't require vast amounts of coding, i am confident that the code can be written in the relatively short timespan i have given myself to write it. However, parts of the project that are non-essential to the finishing criteria (such as a User Interface and polygon reduction) have been marked 'extension' and may be omitted if i run out of time.

The work to be done in the last few weeks of the timetable has been left intentionally low to leave time to correct problems, and to allow me extra time for other activities not related to this project.

Week	***Work to be Undertaken***	***Milestones***
27 Oct – 9 Nov	• Research into methods of 3D reconstruction and other techniques of use • Assuring the robot arm works correctly (in the event it doesn't a turntable will be used) • Installing and setting up necessary development tools	• Papers of use • Working means of getting pictures • All software needed installed
10 Nov – 23 Nov	• Code for PIC controller on Arm • Producing a set of test images via a ray tracer	• Working Arm (or turntable) Controller • Set of test images
24 Nov – 7 Dec	• PC-end code to control Robot Arm (or turntable) and Digital Camera • Work on polygon model data structure, 3D recognition, and polygonization • Java code to read polygon model	• a method of taking pictures of the test object from known positions
Vacation		
12 Jan – 25 Jan	• Finish model reconstruction and polygonization • Work on texturing code • Java viewer that will display the model	• Code to create a 3D model from 2D images • Java viewer for standard model (without texture)
26 Jan – 8 Feb	• Progress Report • Polygon reduction code (extension) • Java viewer to view model with texture	• finished Java Viewer • finished Converter
9 Feb – 22 Feb	• User Interface (extension) • Test reconstruction of different Models	• Graphical user interface to command-line tools (extension)

Week	***Work to be Undertaken***	***Milestones***
23 Feb – 7 Mar	• Testing of Viewer • Dissertation Writeup	• Dissertation 'introduction' and 'preparation' sections
Vacation	• Dissertation Writeup	• Dissertation 'implementation', 'evaluation', and 'conclusions' section
19 Apr – 2 May	• Dissertation ToC, Index, etc. • Proof-reading	• Finished, printed dissertation
3 May – 14 May (Deadline)	• Tidying up of files, code, and finishing off • Proof-reading of Dissertation by others	• Final print of dissertation and binding