Real-Time Interactive 3D Modelling with Commodity Hardware

Peter Coetzee
Supervisor: Andrew Davison

June 2010

Abstract

A long-standing problem in Computer Vision has been recovery of scene geometry; in both controlled and (in many cases) arbitrary environments. With applications ranging from robot navigation to augmented reality in film and television, the process of taking a real-world object and representing it in 3D has enormous inherent value. Nevertheless, existing techniques all leave something to be desired; be it their cost, the limited range of their applicability, or their accuracy.

The theoretical approach outlined in this document aims to alleviate many of the aforementioned pitfalls of traditional methods, by utilising cheap commodity hardware for recognition, by creating a system that is robust in completely un-controlled environmental conditions, and by making use of both *human* and *machine* intelligence in the manners best suited to them.

Contents

1 Motivation

There are a deceptively large number of industries with an interest in producing accurate 3D models of objects in the real world. Engineers, for example, working on a CAD model of a product upgrade may want to scan the old product as a base template. The visual effects industry invests thousands of pounds in laser scans for every show they work on; it is critically important that "digital doubles" (digital replications of human beings) are identical to the people they are based on, both for immersion and often for technical reasons too. It is not unusual for a VFX studio to take a live action scene and replace the actor with their digital double, so that (for example) shadows are cast correctly, or some particularly gruesome mutilation can occur (such as for Harvey Dent's character in the recent Dark Knight movie, seen in Figure 1).

Figure 1: CGI Compositing: Digital Makeup for Harvey 'Two-Face' Dent in The Dark Knight, ©2007 Warner Bros. Pictures

Unless a studio owns the (typically expensive) equipment themselves, it can take weeks to get results back from 3rd party LIDAR scanning - if they had a means by which to create accurate models quickly and cheaply, they could do away with this expense altogether. In conversations with VFX professionals it emerges that the real challenge is to get an accurate *proportional* reconstruction; artistic modelling can fill in as much detail as is desired, but to get the model in proportion a physical scan is required, by way of a foundation.

Furthermore, an increasing number of desktop applications are being produced to allow users to add models to their favourite games and social experiences; from Second Life to Quake, more people are learning 3D modelling and texture artistry. If they had a way of getting real-life objects into their games without having to undergo the horrifically steep learning curve of tools like Autodesk Maya or Mudbox, they would jump on the opportunity.

2 Background

As mentioned above, there are a variety of approaches to this well-studied problem available today. These solutions are typically selected to fit the particular use-case for the 3D model; their various strengths and weaknesses tend to recommend one over the other in any given situation.

2.1 LIDAR

Figure 2: A Leica LIDAR Scanner

One common system used in the visual effects industry is LIDAR capture (Wong et al., 1999), in which a scanning laser is used to trace the distance between it and the target object. The object (or the scanner) is typically rotated during this process in order to generate a dense, 360° point-cloud representation of the scanned object. This technique, while expensive, is applicable in a variety of scenarios; from high-resolution facial capture through to large scale airborne ground topology scanning. This process requires little user intervention, and generates a large amount of data relatively quickly. However, much post-processing of this data is typically required; as it is a slightly noisy point-cloud, it must be cleaned up before it can be used in any real application. Furthermore, this cloud of points is not, by itself, geometry; it is more a set of candidate vertices for geometry, many of which will be discarded as they lie on surfaces. As a result, steps must be taken to generate surfaces in 3D which will fit these points.

2.2 Automated Vision-Based Reconstruction

One set of potentially valuable solutions which have received a lot of research attention recently are those around computer vision based systems. These show promise, in that they are based on much simpler hardware than LIDAR systems, instead focussing their complexity in intelligent software to analyse full colour imagery and extract the geometry from it. They are often based on using either a stereo camera, and thus using the fixed separation of the lenses to recover depth data, or augmenting a single lens with a laser rangefinder (a sort of vision-augmented LIDAR system). The details of implementation of such systems varies wildly. On one end of the scale, Photosynth (Microsoft, December 2009) employs heavy vision processing on still photographs of a given scene (potentially taken with different cameras, lenses, and in varying lighting conditions) and constructs a 3D montage of these images in order to give the effect of a textured 3D model. As part of this process, it actually constructs a coloured point cloud (as seen in Figure 3), in much the same fashion as a LIDAR scan, and matches these points probabilistically to features in the set of input images. Work has been done (Goesele et al., 2007) on a similar approach with the goal of constructing arbitrary 3D surfaces, rather than just interpolation of the views presented by a finite set of images.

Alternative approaches go to the other extreme: complete on-line processing of visual data, aiming for an interactive experience (Pan et al., 2009). In ProFORMA, the application guides the user as to which areas of the target object

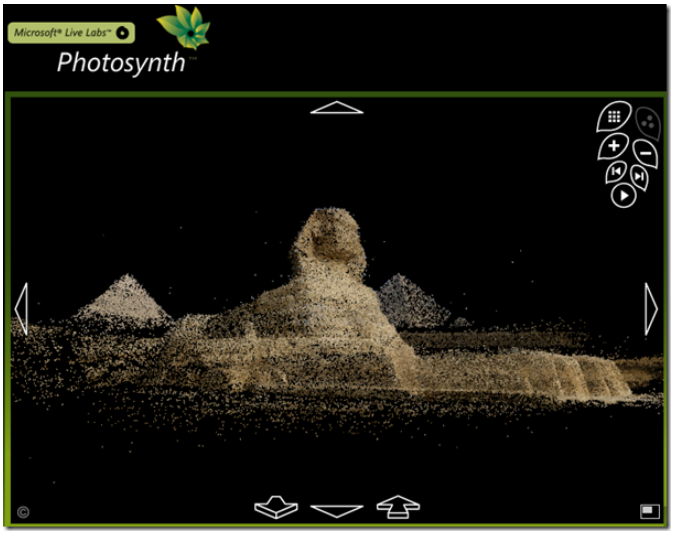

Figure 3: An Egyptian Sphinx, viewed as Photosynth's point cloud (Microsoft, December 2009)

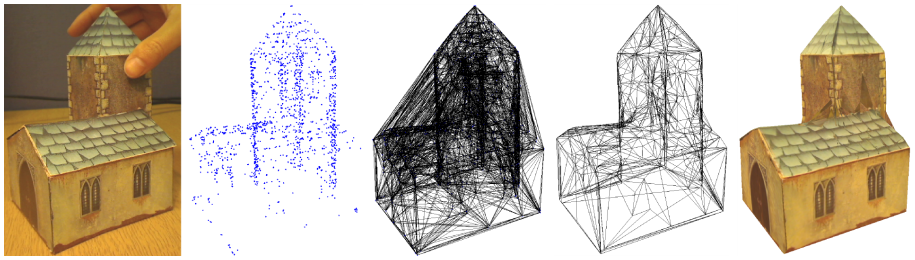

Figure 4: The stages of ProFORMA's automated modelling process (Pan et al., 2009)

it requires further data on, and shows the model in augmented reality as the scanning takes place. The approach taken by Pan is to perform structure-from-motion analysis on the rotating object in order to inform the application as to the angle at which it is viewing the object. From this, it generates and maintains a 3D point-map of the target object. This point map is used as input to a Delaunay Tetrahedralisation (Delaunay, 1934) process, which is then probabilistically carved to produce a complete mesh. Finally, texture is extracted from the video stream to suit the polygons of this mesh (see Figure 4 for an example). However, as we can clearly see in this example (of Qi Pan's own production!), the carving is imperfect; the modelled shape clearly does not quite match the input object, and extra polygons can be seen protruding from the surface of the bell-tower. Should a user wish to clean this up, they would first have to import the data into a 3D modelling package, and then attempt to understand the automated mesh that was generated, in order to alter the polygon geometry and remove these spurious constructions. Furthermore, they would have to manually modify the texture map, since ProFORMA has extracted texture for regions which do not exist in the source shape.

Debevec et al. (Debevec et al., 1996) produced an interesting, and similar,

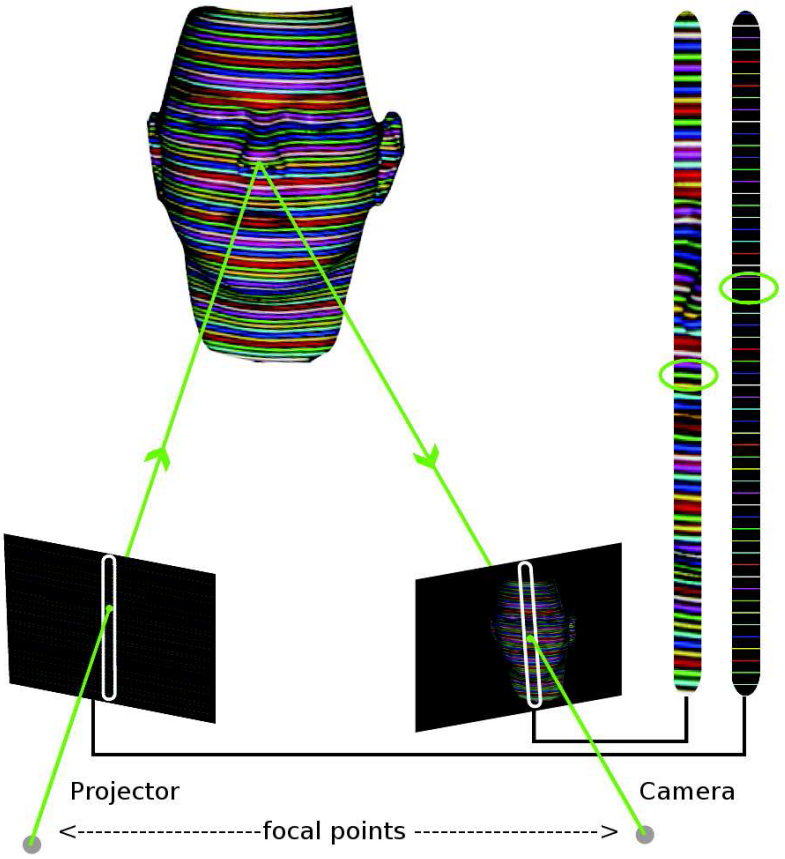

Figure 5: An example of point projection from the human face using structured light (Fechteler et al., 2007)

approach based on a series of static images and minimal user input, tailored primarily for architectural modelling. This required altogether less user input - simply marking edges on the images - but on its relatively limited test domain produced some strong results.

Another approach that has received recent attention is Structured Light (Scharstein and Szeliski, 2003). This entails controlling the lighting of the target object in such a way as to extract 3D data; it requires more equipment and a controlled environment, but can produce interactive performance. Typically, a known pattern of light (potentially changing over time) is projected onto the image, from a fixed source. A single lens may then pick up the way the pattern has fallen on the image, and use this to infer its 3D shape. The suitability of such a process for high-resolution complex modelling has already been shown with 3D facial scanning (Fechteler et al., 2007), as in Figure 5.

This approach is favoured by Microsoft, in their new "Project Natal" motion camera, which exemplifies structured light's suitability for real-time processing. The Natal system gets around problems inherent in controlling the light in an environment by utilising an infra-red spectrum projection and camera - thereby

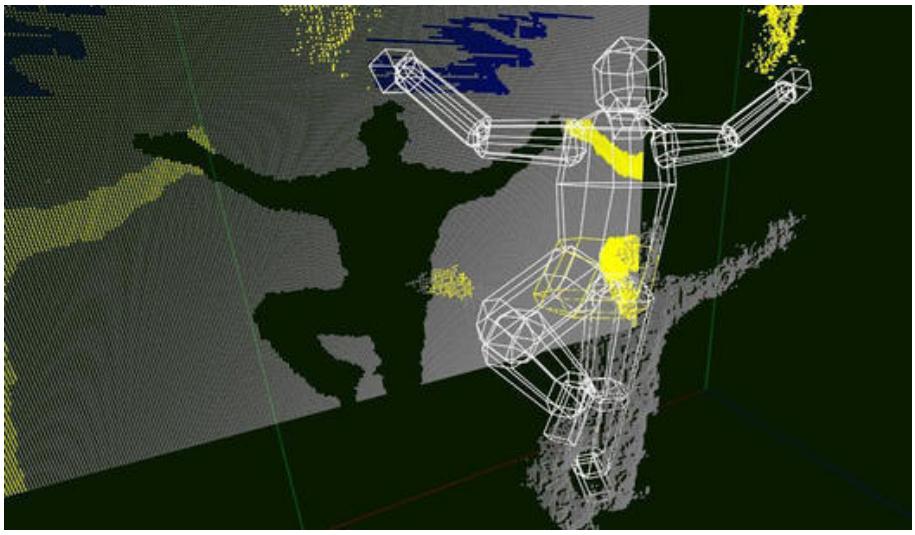

Figure 6: An example of how Project Natal projects a noisy depth-map onto the human skeleton, ©2009 Microsoft

also preventing visible projected light from dazzling the user. However, such an approach offers insufficient resolution for full 3D reconstruction; Natal extracts a depth map from the scene, and is primarily focused on using this for skeletal mapping and recognition (see Figure 6). Other structured light solutions may be more suitable for the problem described here, however the extra hardware and controlled environment required preclude it as an approach, based on the goals laid out for this system.

2.3 Surface Reconstruction

A common aspect of many of these systems is the need to construct a polygon mesh from a point cloud in an automated fashion. This problem is orthogonal to that of extracting the point cloud in the first place; many of these systems could use the same approaches, and differ only in the state-of-the-art at the time of their development. We have already mentioned Delaunay Tetrahedralisation, in the context of ProFORMA. However, this is not the only approach available.

The notion of 3-dimensional α-shapes (Edelsbrunner and Mücke, 1992) were introduced as an early attempt to fit a manifold onto a cloud of points in 3D space, based on Dalaunay Triangulation (and their dual, Voronoi diagrams). These shapes are a mathematically rigorous generalisation of the convex hull approach (Jarvis, 1977), with an additional parameter (the eponymous α), which defines the range of the family of shapes; as α tends towards 0, the shape tends towards the point cloud. As α tends towards ∞, the shape tends towards the convex hull; for values in between, the shape is somewhere between the point cloud and the convex hull - it is not guaranteed that this α-shape is fully connected, or necessarily convex, as the convex hull must be. See Figure 7 for an example of this in two disconnected tori.

An alternate approach is that of implicit surface reconstruction (Hoppe,

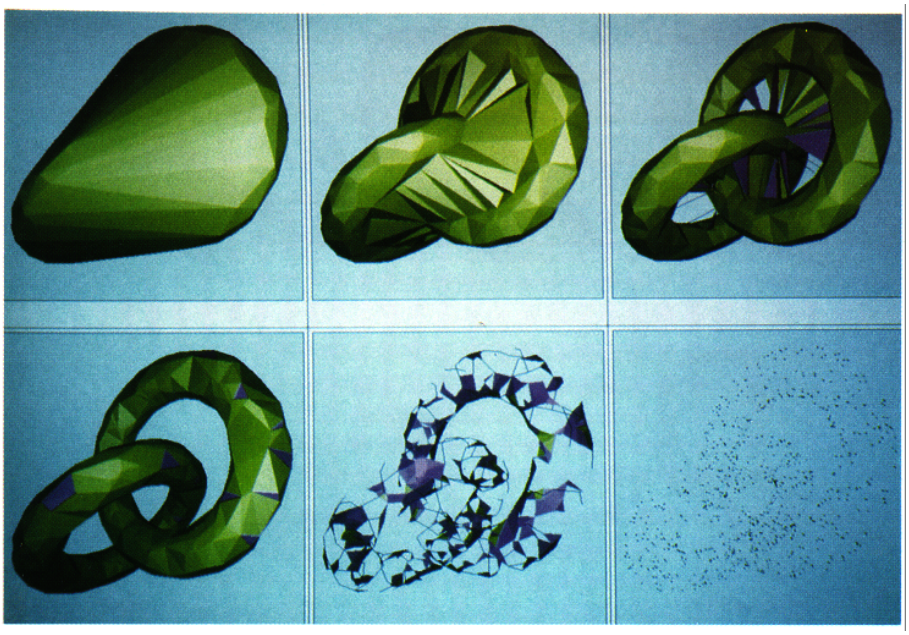

Figure 7: A progression of α-shapes, with α ranging from ∞ to 0 left-to-right and top-to-bottom (Edelsbrunner and Mücke, 1992)

1994; Hoppe et al., 1992), in which the surface is estimated as a *simplicial surface* (a linear surface with triangular faces) based on the signed distance between it and the points that comprise its cloud. At each point, a tangent plane is defined as a linear approximation to the surface at that point - how to define the orientation of the planes is characterised by the author as one of the most challenging aspects of this approach. This is done by fitting the least-squares best fitting plane to those points in the "neighbourhood" of the current point, and subsequently flipping planes to ensure that normals for sufficiently geometrically close points are pointing in similar directions. With these tangent planes constructed, the signed distance function can be estimated based on projecting points onto their nearest tangent plane. Finally, Hoppe uses a marching cubes algorithm (Allgower and Schmidt, 1985) to trace contours and perform his final surface construction.

2.4 User-Driven Vision-Augmented Reconstruction

The final approach we will examine is that taken by the likes of VideoTrace (van den Hengel et al., 2007). These approaches recognise the difficulty inherent in producing a solution that will work for "all of the people, all of the time", and instead focus on creating a toolkit that can be used to trivially craft a model from some video input.

VideoTrace (Figure 8) uses pre-recorded footage as input to structure from motion analysis, and extracts (in a pre-processing step) a 3D point cloud across the field of view of the camera throughout the sequence. At this time, it caches the results of a number of other computationally expensive vision tasks, such

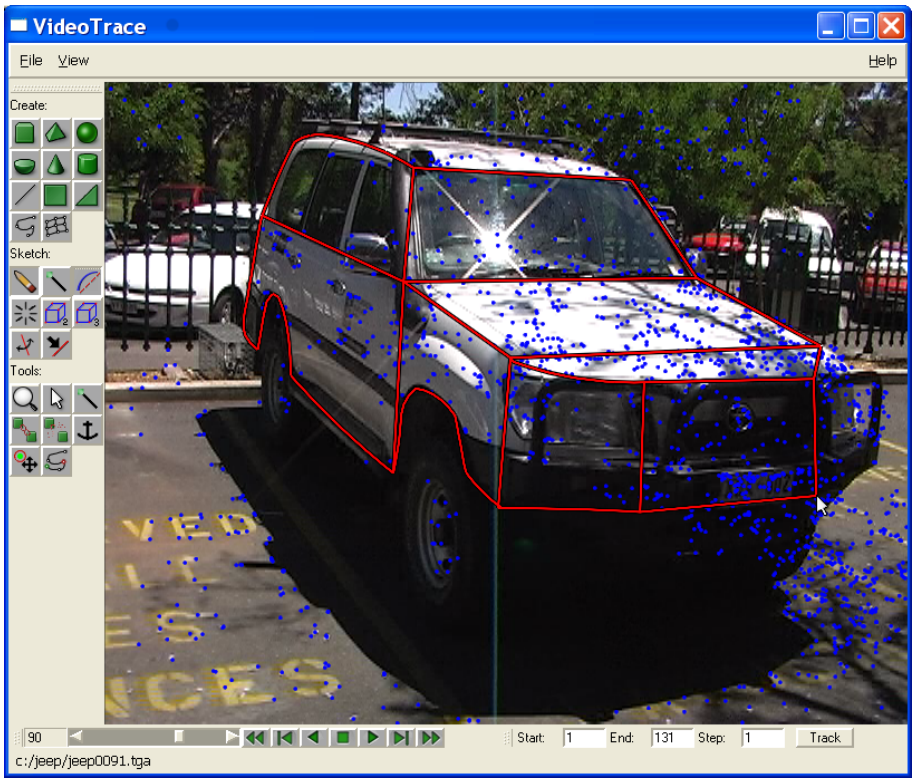

Figure 8: An example of the VideoTrace User Interface in action (van den Hengel et al., 2007)

as superpixel segmentation, and thus edgelet delineation (along the boundaries of superpixels). Once this is complete, it presents a highly simplified modelling interface, not entirely dissimilar to SketchUp (Google, December 2009).

VideoTrace uses the vision data already calculated to augment the user's "tracing" of the video; the user is encouraged to "scrub" the timeline back and forth, freezing the video on a given frame and then tracing shapes with the given tools (for example straight lines, free-hand drawing, and NURBS curves). As the user does this, VideoTrace snaps these lines to the nearest edgelets, permitting the user's inaccurate interactions to be enhanced by visual structure in the image. VideoTrace offers a suite of tools to help model occluded portions of objects too; for example, by extruding an existing shape or by mirroring geometry along a given plane. All of this processing is informed by visual cues in the image; for example, a user may draw the line of symmetry directly onto a frame of video, and VideoTrace will project that onto its geometry model and construct a fairly accurate mirroring plane. Finally, once the user moves from a given frame of video, VideoTrace processes the traced geometry once more, in order to fix perspective-related inaccuracies; using the continuity of frames to fit the trace to the edgelets in the video, continually enhancing its accuracy.

One obvious pitfall to a VideoTrace-style approach is that one must have enough recorded footage of the object to be modelled a priori; if the available

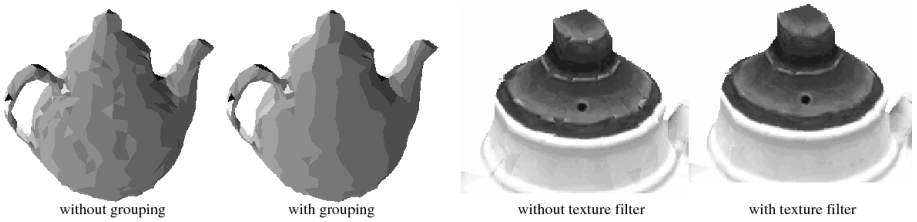

Figure 9: A comparison of the various stages of multi-view texture extraction (Niem and Broszio, 1995)

footage doesn't quite show a part of the model, or the texture is occluded or in shadow, new video must be recorded - and this can't be known until one attempts the modelling process. By contrast, interactive systems using live video can show the user in real time what information is lacking, and how they should perhaps tweak their environment to obtain better results.

2.5 Texture Extraction

The final key aspect of these approaches we have not studied in detail is the problem of extracting texture from the input video, and mapping that onto the output geometry. There are a number of elements to this; selection of video frames for texture, compensation for lens distortion and projection of the texture into 2D space, and mapping of this texture onto the generated model such that it forms a contiguous view. A popular technique is to use texture from multiple camera views (Niem and Broszio, 1995) in order to minimise distortions at the boundaries. The approach outlined by Niem and Broszio requires calibration of the camera (Pedersini et al., 1993) in order to remove distortion and reduce aberrations in the input image due to lens shape. It assigns triangles on the surface of the mesh a camera image (selected according to best resolution). Groups of triangles which are textured by the same image (within some tolerance) are then considered to be *surface regions*. These regions are refined so as to group them together as much as possible, reducing the number of region boundaries. Finally, it filters textures at the boundaries by linearly interpolating the texture from one view to the other across the boundary edge. The result is a smoothly varying texture over the surface of the shape; however, it does necessitate choosing sub-optimal video frames for some polygons in the texture-map. The end result of this is a texture map which requires less space to store, and varies smoothly, and thus the trade-off is probably acceptable. Niem and Broszio also describe a technique for synthesising texture for polygons which lack information, by reflecting texture from adjacent regions and blending the results.

The approach taken by Debevec et al. (1996, discussed earlier) sidesteps the issues around perspective quite neatly; the interactive output of their system dynamically selected the texture frame to use based on the current position of the camera. Their approach uses a similar blend technique at polygon boundaries, to reduce rendering artefacts. What is not made immediately clear is how this technique handles a smoothly changing camera; whether it sticks with

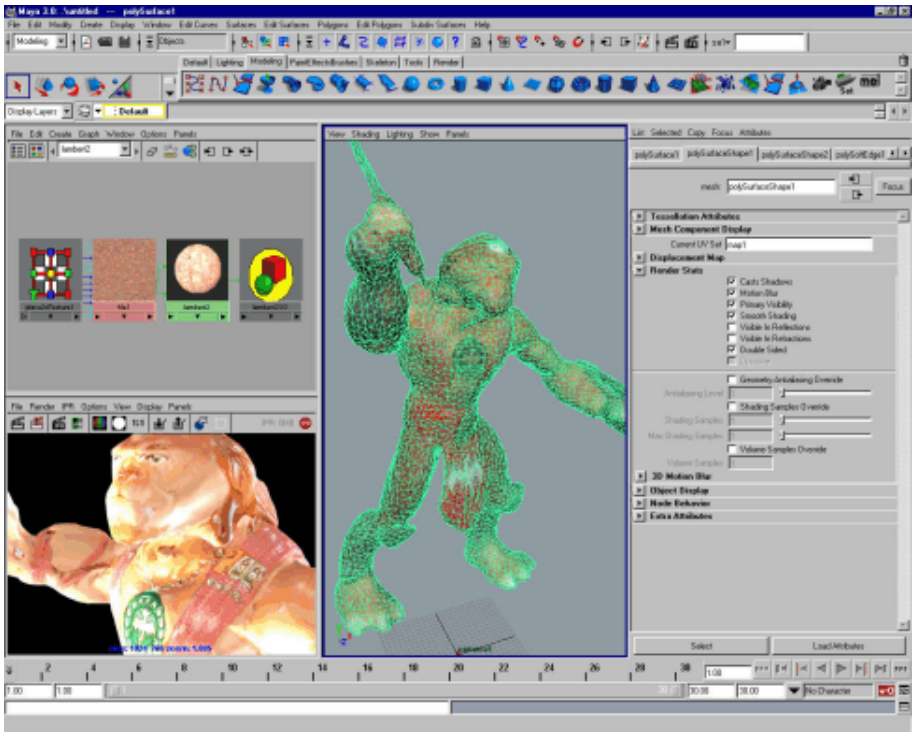

Figure 10: An example of Maya's complex user interface (autodesk.com, 2009)

what is potentially a later non-ideal texture, blends between two target textures smoothly (the problem of knowing which texture to blend *to* seems non-trivial, as it involves predicting the user's interactions), or simply "jumps" from one texture frame to another at some point in the interaction is not made clear.

3 Approach

3.1 Overview

Figure 11: A Unibrain Fire-i™

Broadly speaking, the approach described herein is to carefully delineate responsibilities between human and machine, so as to make the best possible use of their own varieties of intelligence. For example, heavy *mathematical* reconstruction is an ideal problem domain for a computer; whereas more *interpretive* problems are better solved by the human. The goal of this system is to minimise the complexity of user input (in particular when compared to the likes of Autodesk Maya, Figure 10) by augmenting it with vision technologies. The primary input device is to be a single-lensed commodity webcam, such as the Unibrain Fire-i™ seen in Figure 11.

The goal of the system is to permit the user to move the webcam in space around the object they wish to "scan", and identify key features to the scanner.

Throughout this process, the model created will be updated and continually shown as an augmented reality overlay, so that the user may easily tweak the result and examine its accuracy; this continual feedback loop is required to make the most of the interactivity of the system.

3.2 Ego-Motion Recovery

The first problem to be considered in relation to this solution is that of working out where in space the camera is located, and directed, at any given time. Some augmented reality systems, such as Layar, do this by using positioning sensors like GPS in tandem with a magnetometer for orientation. This is a relatively simple way of performing approximate localisation, however the problem of scale rapidly arises; if you desire a resolution of greater than approximately one meter, GPS will quickly fail you. Thus, whilst well suited to (for example) modelling the layout of a whole city, it is less suitable for a user wishing to record a detailed scan of an object on their desk. 3rd party *local* positioning systems operate on a similar principle to GPS, and can provide much greater resolution over small areas. However, they are expensive and seem an unnecessary extra burden for the user to own and maintain. Ideally, the ego-motion recovery problem should be solved without the application of further hardware.

Systems like VideoTrace (van den Hengel et al., 2007) are able to make use of processor-intensive offline tracking applications to perform camera motion recovery - in the case of VideoTrace, the Voodoo Camera Tracker (Thormälen, 2006). These are commonly used in the visual effects industry as well, and thus much effort has gone into research and development on these. They tend to produce excellent and smooth results, even to the extent of being able to automatically fix operator-induced visual defects such as camera shake. However, their processing is designed for batch runs on high-performance machines; they are wholly insuitable to a real-time processing application.

To this end, a number of computer vision researchers have worked on solving the problem of real-time *monocular SLAM*; *S*imultaneous *L*ocalisation *A*nd *M*apping using a single camera lens, with high enough throughput to be used as part of a larger real-time system (Leonard and Durrant-Whyte, 1991). In essence, a SLAM system works by estimating the movement of the camera between frames based on the motion of the intersection-set of detected feature points in each frame. This approach can produce very accurate results, assuming non-degenerate camera motion. There are instances in which a SLAM-based system can get "confused", however; it is important that the localiser can detect these cases and re-localise based on the presented frame

A recent development on existing SLAM approaches is PTAM; *P*arallel *T*racking *A*nd *M*apping (Klein and Murray, 2007). PTAM improves the robustness of SLAM by using a faster approach to motion estimation, thus allowing it to consider far more points in a given iteration of recognition. It can maintain its 3D point-map in a separate thread to the actual motion processing, further reducing the overhead in the tracking thread - making excellent use of common multi-core processing systems available in desktop and laptop computers today. PTAM requires a stereo initialisation at the start, but once this has been processed it will provide a reliable transformation to describe the translation and orientation of the camera in 3D space, relative to an arbitrary zero-point, on an arbitrary scale (both of which are constructed relative to the position

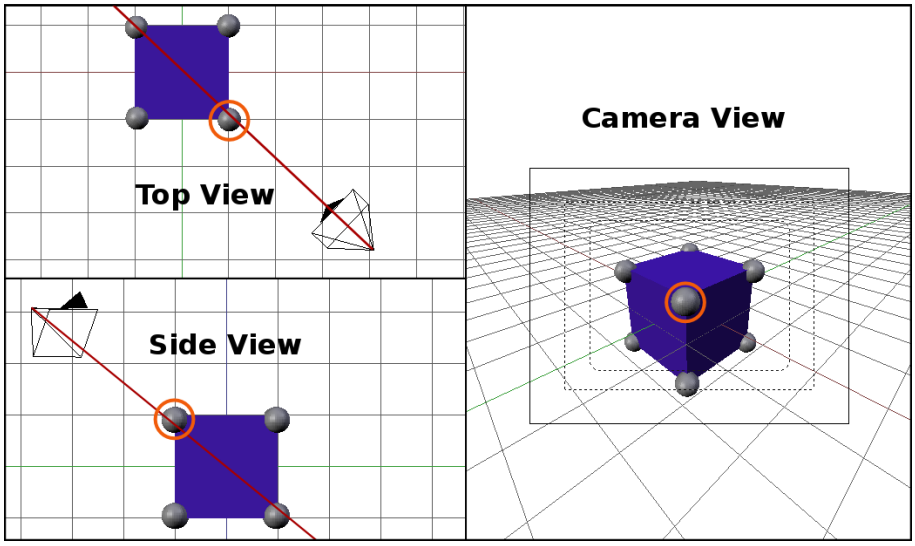

Figure 12: An example of the look-vector of the camera aiming towards a set of candidate feature points (marked as spheres), with the closest to that vector ringed

and motion of the camera during stereo initialisation). As an added benefit, PTAM maintains a map of the features it recognises in 3D-space. It is highly challenging, given only the position and orientation of the camera, to determine precisely what point in space the user is attempting to point to; there are insufficient bounds on the problem, rendering the system with an infinite vector of possible points. This PTAM point-map can be used to augment such decisions, by deciding which point in the map is closest to the camera's look-vector (see Figure 12).

PTAM (as with many SLAM systems) works best with camera lenses exhibiting a wide field-of-view; in this way, it can gather even more points with which to localise. However, this does mean that the images provided to the processing system are distorted by the wide angle lens. In order to account for this, PTAM requires a one-time camera calibration process, which feeds the parameters on its ATAN-Camera model (Devernay and Faugeras, 2001). It is therefore important that, when considering the user interface, this distortion is also considered.

3.3 User Interface

As already discussed, the primary user input is to be a simple webcam. This device is to be augmented with a maximum of two other digital inputs (i.e. buttons). These inputs could be positioned on the camera itself - indeed, some cameras already have these. However, in the interests of the most generally usable solution possible, these inputs will be assumed to be on the keyboard.

It is envisaged that the user would interact with the model on the basis of "tools"; different processors to operate on the scene. One of these input buttons should interact with the selected tool, whilst the second is used to select the

tool the user wishes to make use of. Potential examples of tools might be to move points in space, to bisect an edge, or to delete a given vertex in the model.

Feedback is to be provided to the user on a continuous basis. Using the tracking information from PTAM, as well as its camera calibration data, it is possible to project a 3D render of the current model on top of the current camera frame such that it provides a perspective-correct overlay of the current model on the actual object being modelled. To get this right, it will be best to render the model at each stage relative to the co-ordinate system set up by PTAM, then use OpenGL's projection matrix to project those from 3D-space into camera-space. In this way, the projection may also be accelerated by graphics hardware available on the user's computer. However, the scene will need to be rendered twice; once onto an alpha-channel framebuffer in order to do the projection from world-space into screen-space, and once from that onto the camera frame to take into account camera lens distortion. In this manner, an accurate augmented reality scene will be projected atop the camera's image, permitting the user to visualise and improve their results throughout the modelling process.

3.4 Modelling

The modelling process itself should be as general as possible; models should consist of triangular polygons assembled into a valid 3D model. As far as possible, problems such as degenerate faces, co-occurrent points, isolated vertices, etc should be avoided. It is not *necessary* that the produced model be a closed hull; but it should be *possible* to create such a model with this system. The precise details of the modelling process are open to experiments in user-interface design, however it is expected to be a gradual process. The user should work in one of two ways; either they identify vertices in space, and connect them to make the faces of their model, or they may place "primitives" in the scene (such as planes, cubes, and the like) and manipulate them to best fit the model. Furthermore, there may be cases where a hybrid approach is needed - the user may construct an initial shape that approximates their model, and then refine it over time to improve its accuracy.

It should be possible for the system to use some of the vision techniques above to improve its results - for example, texture extraction may be fully automated. It is important that any vision techniques that are used do not interfere with the will of the user; it must be possible to override their results. A likely example is that of vertex positioning - augmenting the user's input with information about features in the scene can make the task of placing points in 3D space much easier. However, the user *must* have the option of placing points at features that are not detected by vision algorithms. Another useful example of where vision techniques may enhance the modelling process is in edge-fitting; if the user joins two vertices, and the edge is found to run along an edge in the image, the edge that is created in the model may, perhaps, insert extra points in order to better follow the contours of the image. The precise vision techniques that may be applied should be investigated and evaluated for their validity and utility in their particular use cases.

4 Implementation

4.1 Architecture

In order to realise the above goals, a carefully planned and executed architecture is of prime importance. Broadly speaking, the system is realised as a collection of plugins in four primary areas, with a shared Model at its core. This vision is outlined in Figure 13; here, the Model is represented as an Environment - lists of points in space, the edges connecting them, along with details of the current camera pose and the like.

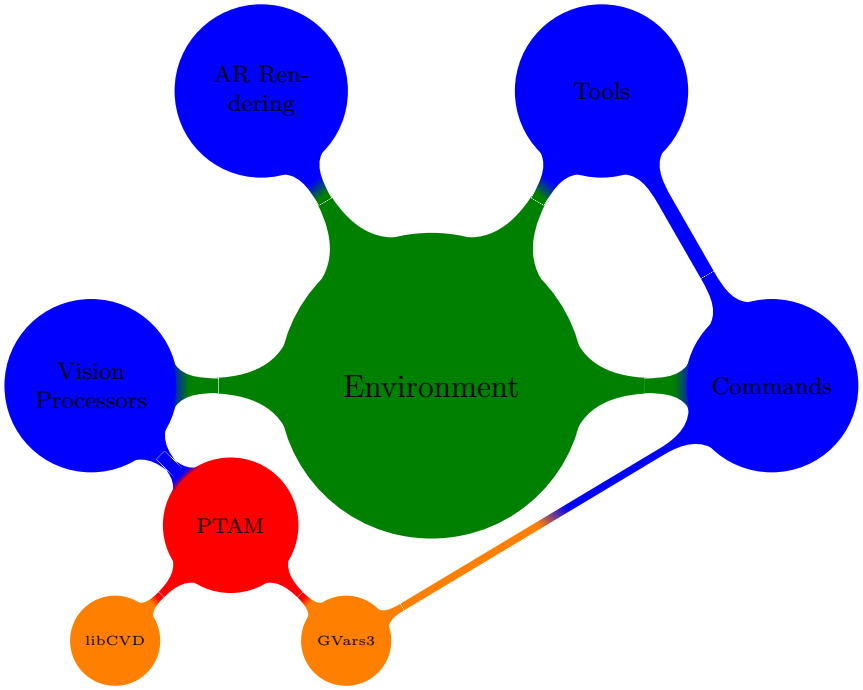

Figure 13: Architectural Vision

These plugin areas, in blue, are the extension points for the system. In this way, the dependency on PTAM is isolated to the Vision Processors - responsible for any non-interactive ongoing computer vision tasks. "Commands" in this vision are the direct means of manipulating this model; whether creating new geometry or editing existing polygons. "Tools", by way of contrast, are the interface between the user's graphical frontend and the rest of the system's internals, interacting with these commands.

The following sections will detail the implementation details of each of these sections in turn, including more formal diagrams as well as details and justification of the design decisions taken along the way. The organisation of these sections reflects the traditional delineation of *Model, View, Controller* exhibited in well architected systems.

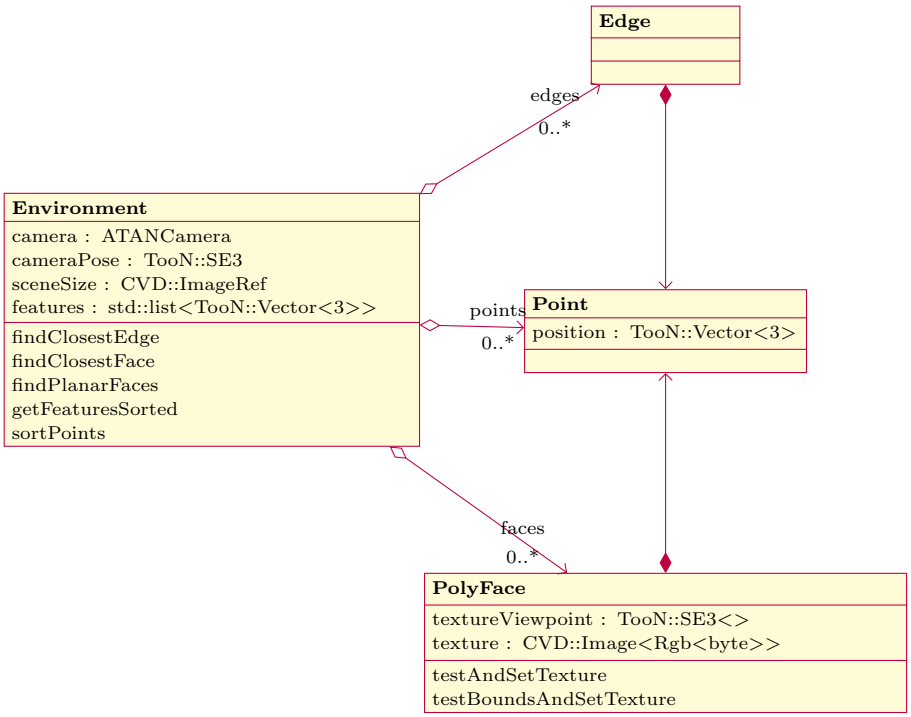

Figure 14: Model Class Diagram

4.2 Model

The aforementioned notion of an *Environment* is carried through from vision to implementation more-or-less verbatim. As seen in Figure 14, there are four classes of note. The main `Environment` is a single object that is passed to almost all objects in the Scanner system; it contains lists of the `PolyFace`s, `Edge`s, and `Point`s in the system. The classes of `Edge` and `Point` are little more than data containers, identifying the lowest level of geometry. Early on it was decided that the core form of geometry in the system should be a triangular polygon; it is the most flexible primitive, and readily maps to the geometry used in 3D modelling packages. Thus, the notion of a `PolyFace` encodes a triangular polygon as a function of three vertices (`Point`s from the `Environment`), with associated texture data. This texture is stored as a full frame from the camera, for later processing. A given `PolyFace` is capable of deciding whether a given texture frame is better than the currently recorded one, as well as for mapping from the given camera frame to a UV co-ordinate for texture rendering. Finally, the `PolyFace` may also compute useful quantities, such as the face centre and normal vector, for use in other calculations.

The `Environment` class is more than just pointers to data, however; it contains a variety of common methods useful for interacting with the model at its lower levels. For example, it contains a list of "features" extracted by vision algorithms (the details of which are discussed later), along with methods to extract a sorted list of these features based on the current camera position and orienta-

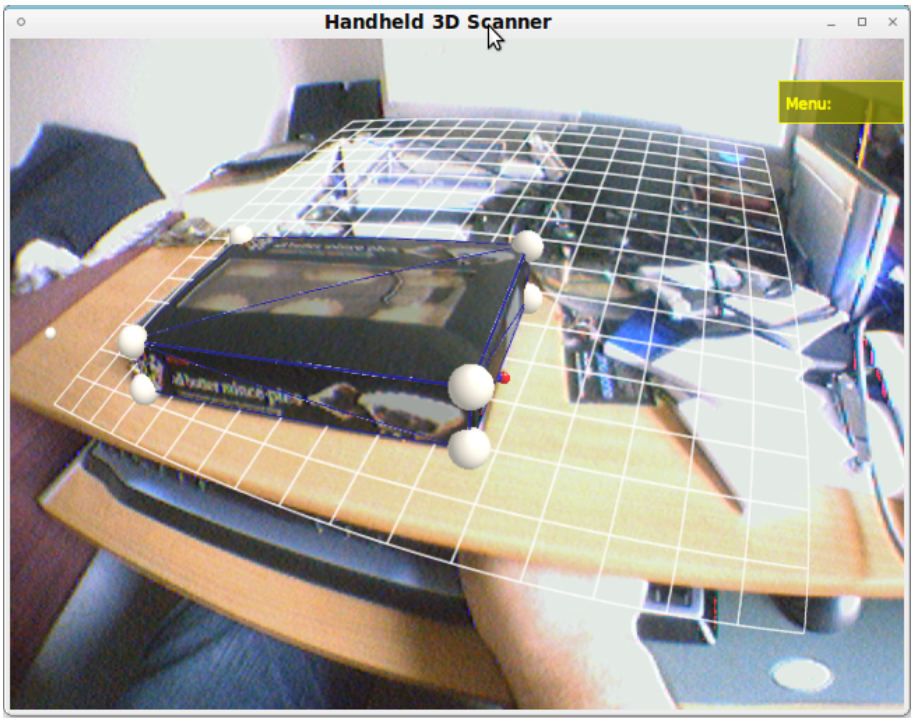

Figure 15: An early modelling test, showing a textured AR box of mince pies overlaid on the camera plane **TODO: Better image showing highlighting, points around, etc?**

tion. Likewise, it can identify which `PolyFace` or `Edge` the camera is aimed at, and where on that target the camera is pointed. Other more obvious methods provide interfaces for adding to the model; a method `addEdge(Point*,Point*)` for example will ensure that the edge is not a duplicate, and automatically construct any new `PolyFace`s created by this new `Edge`.

4.3 Augmented Reality Rendering & User Experience

The approach taken for rendering the scene in augmented reality is built on top of that in PTAM's demonstration application. This uses two passes of OpenGL rendering; first, it renders the scene onto a regular framebuffer in memory, and then in a second pass uses the PTAM camera calibration data to distort and render that framebuffer onto the screen. In order to render an Augmented Reality view of the model atop this camera plane, the UFB Linear Frustum Matrix from PTAM is applied as an OpenGL Projection matrix, permitting points in `Environment`-space to be projected to image-space en masse.

For the purposes of usability, vertices are rendered with an exaggerated size, to facilitate using the camera to "grab" them. All aspects of rendering are configured using variables in the GVar3 system. For example, a single keypress can be used to toggle rendering of various elements (edges, faces, vertices, camera plane, etc). Further settings include scale settings for things like vertices, or the

sensitivity and accuracy of the camera "pointer" system. Most interaction is done using the camera as a pointer, as described above, seeking feature points around the camera's look-vector; this setting simply adjusts how tolerant this should be of deviation from the precise vector. As seen in Figure 15, the UI is kept to an absolute minimum of clutter (even in early tests such as this). Other usability enhancements present in the system include highlighting of currently targetted objects; these present themselves as orange shading. When creating points it is useful to be able to see the feature points extracted by PTAM's analysis, so the renderer displays the nearest few points, scaled according to their distance from the look-vector of the camera. Usability tweaks like these permit the user to fully predict the behaviour of the system; which face is selected will never be a surprise, and points will always go where the user expects. As this is an interactive modelling system, rather than a fully automated solution, transparency of operation is of the utmost importance.

4.4 Plugins

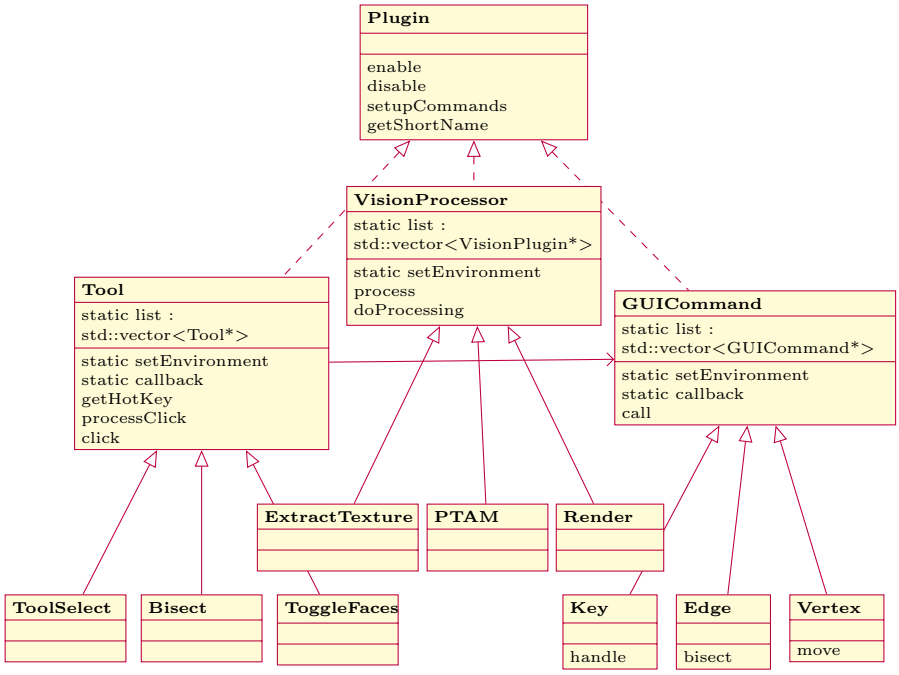

Figure 16: Plugin Class Diagram

Broadly speaking, each `Plugin` object is a singleton instance of some subclass of the root `Plugin` type. This core type offers some basic structure as to what makes a "plugin" in code. All plugins must have a "short name", which is used to refer to it in configuration. Plugins are also given object-oriented style methods in GVars3, such that a plugin named *"foo"* will automatically have functions registered with GVars3 such as *`foo`*`.enable`, *`foo`*`.disable`, or others based on their sub-type.

As seen in Figure 16, there are three sub-types in use in the system. `Tool`s, for example, are used to process the clicks in the user interface (from any hotkey on the keyboard), invoking `GUICommand`s to perform some action on the `Environment`. Finally, `VisionProcessor`s are invoked in a constant processing loop to perform vision tasks, and provide the "main loop" of the system.

As a coding standard, it is desirable that plugins are as easy to develop as possible. In order to encourage lots of small plugins, rather than monolithic chunks of code, much of the C++ syntactic overhead is removed through the use of preprocessor macros. For example, to declare a `VisionProcessor` all that is required is an indicator of the type, and optionally any `protected` variables it should require, thus:

```
MK_VISION_PLUGIN( MyVisionAlgorithm, pthread_t workerThread; );

void MyVisionAlgorithm::doProcessing( CVD::Image<CVD::byte>& bw,
        CVD::Image< CVD::Rgb<CVD::byte> >& rgb ) {
    // My code goes in here
}
```

4.4.1 Vision Processors

1. **cam**: Extracts camera frames from the compiled-in `VideoSource`, which converts them to their RGB and Black & White variants for passing on to other vision processors.

2. **ptam**: Invokes the PTAM camera tracking system to track the current video frame. Once tracking is complete, this plugin also records the camera pose into the current `Environment`, as well as updating the list of features that PTAM is tracking. These features are used by many of the Commands, outlined below.

3. **guiDispatch**: Dispatches any GUI events, and renders the current view. This extends the double-buffered `GLWindow2` and `ARDriver` classes from PTAM; in this way, they may reuse the camera calibration for AR projection, as discussed previously, while using our custom `ARPointRenderer` for actual object rendering.

4. **commandList**: Synchronised plugin to execute commands in the main thread. This uses `pthread_mutex` locks to present a thread-safe way of invoking commands from any thread in the system, and ensuring they are executed sequentially, without any interleaving. It offers a static method, `commandList::exec( string )`, to quickly add a command to the execution list, and then clears this entire list on each call to `doProcessing`.

5. **textureExtractor**: Test the current texture frame against each polygon in the image to see if it is a better match for texturing than the viewpoint it previously had. Most of the program logic for this is encapsulated within the `PolyFace` class.

6. **profiler**: Prints out hierarchical profiling timing information to the console. The run-time profiler used throughout the codebase is Shiny**??**, a

fast and low-overhead C++ profiler. Profiling points can be defined arbitrarily; either as an entire method, or for sub-components, to allow more specific behaviours to be timed. An example of the profiler's output can be seen below:

```
call tree                hits       self time      total time
<root>                    1.0      8 us    0%     54 ms  100%
  cam                     1.0     29 ms   53%     29 ms   53%
  ptam                    1.0     13 ms   24%     13 ms   24%
  gui                     1.0      5 ms    9%     13 ms   23%
    DrawStuff             1.0     15 us    0%      8 ms   14%
      DrawPoints          1.0      7 ms   12%      7 ms   12%
      DrawFeatures        1.0     20 us    0%     20 us    0%
      DrawTarget          1.0    842 us    2%    842 us    2%
      DrawPolys           1.0     65 us    0%    200 us    0%
        textureTidyup     9.0     19 us    0%     19 us    0%
        retrieveTexture   9.0     12 us    0%     12 us    0%
        drawTriangles     9.0    104 us    0%    104 us    0%
  commands                1.0     19 us    0%     19 us    0%
  textures                1.0     34 us    0%     34 us    0%
```

4.4.2 Commands & Tools

- **key.handle**: This is a keypress handler, invoked via GVars3 from the GUI key Dispatch method. It searches the list of tools for any tools which have a matching hotkey to the passed in key, and invokes them; any which are enabled will receive a "click".

- **edgelist.dump**, **pointlist.dump**, **polylist.dump**: These three debugging commands can be used to dump a list of `Edge`s, `Point`s, and `PolyFace`s to the console, respectively.

- **text.draw**: This command is designed to be invoked by other commands; it will cause text to be drawn as a "status update" to the user using OpenGL. This ensures that consistent styling is used, and makes it easy to execute the drawing in the correct thread.

- **vertex.create**: Creates a `Point` on the feature closest to the look-vector from the camera. If invoked with parameters, it can create a `Point` at an arbitrary position (defined by the 3D co-ordinates given in the parameters). The closest point is found by sorting the vector of features with the `closerVec` function, which works between two vectors thus:

  Given a camera pose C, let $O = C_{translation}$

  Given therefore that $C_{rotation} = \begin{bmatrix} x_1 & x_2 & x_3 \\ y_1 & y_2 & y_3 \\ z_1 & z_2 & z_3 \end{bmatrix}$, let $V = \begin{bmatrix} z_1 \\ z_2 \\ z_3 \end{bmatrix}$.

Therefore, with arbitrary vectors A and B, $A < B$ iff $(A - O \times A - V) < (B - O \times B - V)$.

- **vertex.remove**: This command simply removes the targeted vertex. This vertex is identified from the look-vector using the same `closerVec` function as described previously.
- **vertex.move**: This command has two operation modes; the first is a free move, which moves the targeted vertex based on its fixed distance from the camera. The second mode is much the same, but it fixes the motion to be in only one of the three axes; this is simply done using the `lock` vector, which is 0 when movement in the given axis is to be ignored, and 1 otherwise. The actual movement is performed in a separate `p_thread`, and updates the position of the target point as often as the OS' threading model permits according to the following equations:

  Given an initial camera pose C, and that $C_{rotation} = \begin{bmatrix} x_1 & x_2 & x_3 \\ y_1 & y_2 & y_3 \\ z_1 & z_2 & z_3 \end{bmatrix}$, the new vertex position at camera pose C' is given by

$$\begin{bmatrix} lock_x \times (C'_x + z'_1 \times \frac{Target_x - C_x}{z_1}) + (1 - lock_x) \times Target_x \\ lock_y \times (C'_y + z'_2 \times \frac{Target_y - C_y}{z_2}) + (1 - lock_y) \times Target_y \\ lock_z \times (C'_z + z'_3 \times \frac{Target_z - C_z}{z_3}) + (1 - lock_z) \times Target_z \end{bmatrix}$$

- **edge.connect**: This simple command is used to identify two vertices, and joins them together automatically with an `Edge` - automatically avoiding duplication and inserting `PolyFace`s as edge loops are completed.
- **edge.remove**: This command is the functional inverse of the connect tool; it disconnects two vertices, destroying any `PolyFace`s that included these edges.
- **edge.bisect**: The last of the edge manipulation commands, this will insert an extra point on a given edge, splitting any `PolyFace`s on the way. The edge identification algorithm is the same as for removal; it seeks the point of intersection between the look-vector from the camera and the vector defined by the edge, within some given tolerance. The goal is to minimise $|TargetPoint - EdgePoint|$ in the following equations;

  Let $u = \begin{bmatrix} z_1 \\ z_2 \\ z_3 \end{bmatrix}$, from the camera rotation, as before, and v be the vector described by the edge in question. Therefore, we can prepare some variables; first, let p_0 be the camera's position, and the start point of the current edge be q_0. With these definitions in hand,

$$\begin{aligned} w_0 &= p_0 - q_0 \\ a &= u \cdot u \end{aligned}$$

$$
\begin{aligned}
b &= u \cdot v \\
c &= v \cdot v \\
d &= u \cdot w_0 \\
e &= v \cdot w_0 \\
Therefore: & \\
EdgePoint &= q_0 + v \times \frac{a \times e - b \times d}{a \times c - b \times b} \\
TargetPoint &= p_0 + u \times \frac{b \times e - c \times d}{a \times c - b \times b}
\end{aligned}
$$

Once we have minimised the above constraint, and ensured that $EdgePoint$ is on the edge, by ensuring that $0 \leq v \cdot (EdgePoint - q_0) < |v|$, the target point for the bisected edge is at $EdgePoint$. The tool for this command also then automatically invokes **vertex.move**, so that the user may position the point as desired.

- **plane.split**: In a similar fashion to `edge.bisect`, this tool will take a `PolyFace` and split it at the targeted location. As above, it first finds the targetted face and the point on that face. Next, it removes the targetted `PolyFace` from the environment, creating a hole in the model. It then places a new vertex in the model at the targetted position, and connects it to the vertices of the old `PolyFace`, automatically constructing new faces in the process. Finally, it invokes the standard `vertex.move` tool to permit the user to manipulate the split and tweak its exact location.

- **plane.move**: With all of the "primitive" manipulators complete, the next few tools analyse a higher level of geometric structure; namely, permitting the user to treat a collection of faces as a single plane. The constraint for "planarity" is essentially defined by a tolerance of the angle across an edge; if it is sufficiently close to 0°, it will consider the attached faces to be part of a single plane. The easiest way to do this, is to consider the angle subtended between the surface normals of these faces, trivially found using the dot-product. The `Environment` class offers a recursive solution to this problem, which ensures that zero-area faces are ignored (often important when the method is used in the midst of a series of transformations).

  The actual movement of this face is handled in a similar fashion to the vertex movement already discussed. A separate worker thread first calculates the face that is currently being "pointed at", along with the point of intersection between it and the camera's look-vector. To do this, each face is tested first to find its point of intersection: given the camera's look-vector, z, a face normal n, the camera's position p_0 and a point on the face, p_1, the shortest vector from the camera to the face can be found as the absolute magnitude of the vector:

$$
d = \frac{n \cdot (p_1 - p_0)}{n \cdot z}
$$

  Finally, a fast bounds-check is used to see if the intersection point with the face's mathematical plane, $p_0 + d$, lies within the bounds of the tested face's actual geometry.

Next, the command's mover thread pre-computes the set of target points to move, and caches their positions before the projection. It then updates positions by calculating the movement the point of intersection on the target face (as in `vertex.move` and moving all the target points by the same amount.

- **plane.extrude**: Many common candidates for modelling exhibit symmetrical structures; it thus makes sense for tools to support this mode of modelling. For those objects with lines of symmetry, this extrusion tool will permit the user to:

  1. Model a single planar face of the object
  2. Create a duplicate plane
  3. Automatically connect the vertices of the two planes to form complete polygons
  4. Invoke the `plane.move` tool to manually adjust the distance and direction of extrusion

- **plane.revolve**: Not all symmetry is reflectional; a whole host of objects display rotational symmetry instead. Thus, it is desirable to support this in the modelling as well. To do this, the user need only model half of the rotational face as a plane, and ensure that one of its edges is the axis about which to revolve. Once this is complete, they must simply aim the camera at this axis edge, and invoke this tool to duplicate the original face and rotate it around the axis. The rotation is done with a simple transformation matrix, using a rotation to and from the axis of the revolve, provided by `TooN::SO3`, thus:

$$RotationToAxis \cdot \begin{bmatrix} 1 & 0 & 0 \\ 0 & cos(\theta) & sin(\theta) \\ 0 & -sin(\theta) & cos(\theta) \end{bmatrix} \cdot RotationToAxis^{-1}$$

  The number of "steps" to use in this revolve is determined by a configurable GVars3 variable; more steps will increase the quality of the finished geometry, but with it the complexity of the model.

- **mesh.move**: Commands based on `mesh.foo` manipulate the entire mesh as a whole. This command works in exactly the same manner as `vertex.move`, but applies the vector of movement to *all* vertices of the mesh, instead of just that which is targetted. This command, and `mesh.scale` (below) are of particular use if larger meshes are to be inserted into the scene - whether through a file import, or by generating primitive shapes.

- **mesh.scale**: Along with being able to move an entire mesh, it makes sense to use the camera to scale it also; the scale of the axes in world-space are not fixed relative to any given objects, and thus it is unpredictable what size should be used for insertion of an object. This scale command works by scaling the points in the mesh according to the distance of the camera from its original pose. The scale is uniform in all axes.

- **mesh.subdivide**: An alternative experiment into ways of improving a mesh's quality, the goal of this command is to improve the resolution of the mesh, smoothing its hard edges, without losing too much detail. The method used is based on Catmull-Clark's mesh subdivision algorithm(Catmull and Clark, 1978), thus:

  1. Add new points, on the centre of each face
  2. Clear all existing `PolyFace`s in the model
  3. For each original point in the mesh, calculate the average position of the points connected to it
  4. Calculate also the average face centre of the faces joined to the current point
  5. Update the position of the current point as:

$$(1 - smoothFactor) \cdot CurrentPosition + smoothFactor \cdot$$
$$(FaceAvg + 2 \cdot EdgeAvg + (EdgeCount - 3) \cdot \frac{CurrentPosition}{EdgeCount})$$

  6. Construct new `PolyFace`s by adding new edges to the model, completing the reconstruction of the model

  The variable *smoothFactor* above is in fact a GVars3 variable, `catmullSmooth`, to permit the user to configure how much smoothing should be applied in the linear interpolation; as it approaches zero, the positions of the points converge on their original positions. By default, this is kept fairly low so as to introduce some smoothing, but without as much mesh collapse as Catmull-Clark usually causes. This tool can be invoked as many times as desired, to increase the mesh resolution as needed.

- **mesh.smoothDeform**: This partner to the `mesh.subdivide` method is designed to make it easier to manipulate higher resolution models in a smooth fashion; particularly useful when using a high-res model to approximate smooth curves on an objects' faces. To use it, the user simply "grabs" a point on the mesh, and as they pull it away it will attract the points connected to it, which will in turn attract their neighbours less strongly, et cetera. At its core, this command is very similar to `vertex.move`, but with an added step on each update; the smooth. GVars3 variables are used to define attraction and inertia factors, while the sterngth of the final pull is calculated as a weighting, w, based on the size of the movement of the target point, over the inertia - clamped such that $0 \leq w \leq 1$. The maximum and minimum weights, w_{max} and w_{min} for the other points are then found as the maximum and minimim values of:

$$\frac{||TargetPos - PointPos||}{Attraction}$$

  Finally, point positions of all points except the target are updated. A per-vertex weight w_{vtx} is calculated, and clamped between 0 and 1.

$$w_{vtx} = \frac{||TargetPos - PointPos|| \div Attraction - w_{min}}{w_{max} - w_{min}}$$

$$NewPos \;=\; OldPos \;+\; w \cdot (1 - w_{vtx}) \cdot (TargetPos - PointPos)$$

This smoothing is calculated in real-time, as the user pulls the target point, so that they can see how the deformation alters the entire mesh.

- **texture.clear**: As PTAM occasionally and temporarily loses the stability of its tracking, it can capture bad frames of texture - similarly, texture frames can be heavily blurred if the camera was in fast motion when the frame was captured. In this event, it is desirable to capture a new texture frame. The simplest way to do this, is simply to clear the existing texture, replacing it with a blank red image, and setting the camera pose for that frame to have been taken from "infinity". As soon as the camera captures a suitable frame for texturing, which will cover the targetted face, the `textureExtractor` Vision Processor will update, per its usual operation.

- **texture.clean**: Often, textures are extracted from camera frames which are very close together, or which could cover more than one `PolyFace`. Due to the slight inaccuracies of both the PTAM tracking and the texture projection, coupled with the human eye's ability to detect discontinuites in an image, it is desirable to (within reason) choose shared frames of reduced quality, so as to visually improve the result. Furthermore, using fewer frames of video for texturing has the added bonus of reducing file sizes when saving the textures for export. This operation is completed in two passes. The first pass compares the camera pose for each pair of texture frames; if these are within a given tolerance, and the texture frame will cover both polygons, it will be shared between the two faces.

  In the second pass, the code examines adjacent pieces ot texture; if the frames are taken from sufficiently similar camera angles, it will choose whichever image gives the most coverage - if the currently examined face is bounded on at least two sides by the same texture, the angle is within the tolerance, and the texture frame can cover the current face, then the replacement will be performed.

- **code.save**, **code.load**: This simple pair of debugging commands demonstrate the power of separating Commands from Tools; they provide a simple way of exporting model geometry, and reloading it later. The save command simply streams a series of primitive commands - only `vertex.create` and `edge.connect` are required to reconstruct the model; GVars3 will then load the file later, and the model will be reconstructed. Both commands can take a filename as a parameter, to identify which file to use to save / load (otherwise it will use the default, "scanner_model.out").

- **obj.save**: While the above provides one form of save utility, it does not satisfy the need to load models generated in this tool into any other industry standard tools. In order to do that, the Wavefront OBJ format was selected as the export format of choice. When invoked, this command will save using the specified file's basename. The Wavefront format allows for saving geometry, material data, and texture data as three separate files. As the texture is stored uncompressed, this command will first invoke the `texture.clean` command above before it saves.

The reduced set of texture frames are then stored, as a series of consecutive images in a single 24-bit RGB TARGA file. The material data is stored in a `.mtl` file, which is essentially a plain text format defining the ambient, diffuse, and specular components of the named material (fixed at 1, 1, and 0 respectively). It also permits selection of alpha transparency, and the illumination model to be used, along with naming the TGA file to use for texturing. This file is more-or-less the same regardless of the model. Finally, the `.obj` file encodes the details of the model's geometry; the set of vertices, their texture co-ordinates, and the vertex normals. The last section of this file links the OBJ geometry to the material file, and applies the texture to the set of face definitions which follow. These definitions are made in such a way as to ensure that the points are connected in an order which will preserve their surface normals - implicit in the OBJ definition. With all three of these files serialised and linked, most standard modelling tools used in industry today can import without loss of information.

5 Platform Showcase

5.1 The Game

Architecture capability
- Examples of AR games
- Premise of game

5.2 Real-Time Physics

Bullet
- Bullet architecture - physics pipeline
- Terrain generation

5.3 AI Navigation

Navmesh generation
- A* search

6 Evaluation

In order to ascertain the extent to which the finished product meets these criteria, carefully defined evaluation must be undertaken. It is not sufficient that the system simply accomplishes the above goals - there must be a methodology defined to ascertain how *well* it accomplishes those goals. Therefore, two different types of evaluation will be undertaken; end-to-end acceptance testing, and numerical analysis of the accuracy of the result.

6.1 Acceptance Testing

Acceptance testing should consist of attempting to model a collection of real-world objects, of varying complexity; for example, ranging from a book to a more complicated shape, such as a detailed model or a pair of shoes. If the

finished product is capable, testing it on a variety of scales would be advantageous; say, the view from a balcony as well as the aforementioned articles. This model construction should take place in a noisy environment; with uncontrolled lighting, other objects in the area, and the like, in order to best simulate the environment in which the system is designed for use.

Once the modelling of each of these has been completed, they should be exported to disk, and imported into a 3D package for further analysis and raytracing; thus demonstrating the suitability of the system for producing data of real use. If the 3D package supports model checking, these static checks should be run on the produced model to check for invalidities such as degenerate faces and isolated vertices (as discussed in Section 3.4).

6.2 Accuracy

The numerical analysis of the system's accuracy is altogether less straightforward, as there is no perfect "gold standard" approach against which the final product can be measured. Furthermore, an artistic "from scratch" model against which to compare would be both hard to find and not necessarily created with accuracy in mind. If the hardware were available, the geometry that the system outputs can be compared against the point-cloud generate by a LIDAR scanner. This has the distinct advantage of testing the geometry separately from the texture. To some extent, the correctness of texture mapping can be inspected visually; however, a numerical approach would be desirable. To this end, an approach which compares the augmented-reality frame and the source camera frame from a few different perspectives would test the accuracy of the complete system; geometry, texture mapping, AR rendering, and even camera calibration. Of course, this means that if any one of these is deficient then the whole system will be deemed inaccurate. The obvious way of doing that would be to simply compare the pixel values of the two frames in the framebuffer. This would provide a quick and easy way of telling how accurate the system is overall; the average percentage pixel difference across the entire image should give an index of accuracy. Another approach would be to perform feature detection on the source and AR camera frames, and estimate the motion of co-occurrent features between the two; any features that are not within the model would not have moved, but those which *were* modelled, would be expected to exhibit some form of motion.

6.3 Usability

A third factor that remains unmeasured is the extent to which a new user can pick up the software and learn to use it; one aspect of this is documentation, the other a highly qualitative "user friendliness". This can hypothetically also be tested, by giving the system to a small cross-section of different users and observing how readily they learn to use it. Worthy of note here is the relative difficulty with which most users learn 3D packages, such as Maya or 3D Studio Max - most 3D tools are highly complex, with many months of learning curve. If a user can pick this system up and start using it to model and texture objects inside of an hour then it should rate quite highly on the usability scale!

6.4 Summary

Combining these three approaches it should be possible to give both a qualitative and quantitative analysis of the suitability of the solution for purpose. If it can complete the above tasks, be found to be of reasonable accuracy, and can be picked up by new users with relative ease, it will be deemed to be a success.

7 References

Allgower, E. L. and Schmidt, P. H. (1985). An algorithm for piecewise linear approximation of an implicitly defined manifold, *SIAM Journal of Numerical Analysis* **22**: 322–346.

Catmull, E. and Clark, J. (1978). Recursively generated b-spline surfaces on arbitrary topological meshes, *Computer-Aided Design* **10**(6): 350 – 355.
URL: *http://www.sciencedirect.com/science/article/B6TYR-481DYSP-1GC/2/786ce8cb247ef51eeac9f793556654b9*

Debevec, P. E., Taylor, C. J. and Malik, J. (1996). Modeling and rendering architecture from photographs: a hybrid geometry- and image-based approach, *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, ACM, New York, NY, USA, pp. 11–20.

Delaunay, B. N. (1934). Sur la sphère vide, *Bulletin of Academy of Sciences of the USSR* **7**: 793–800.

Devernay, F. and Faugeras, O. (2001). Straight lines have to be straight, *Machine Vision and Applications* **13**(1): 14–24.

Edelsbrunner, H. and Mücke, E. P. (1992). Three-dimensional alpha shapes, *VVS '92: Proceedings of the 1992 workshop on Volume visualization*, ACM, Boston, Massachusetts, United States, pp. 75–82.

Fechteler, P., Eisert, P. and Rurainsky, J. (2007). Fast and High Resolution 3D Face Scanning, *Proceedings of the 14th International Conference on Image Processing (ICIP2007)*, San Antonio, Texas, USA. ICIP 2007.

Goesele, M., Snavely, N., Curless, B., Hoppe, H. and Seitz, S. (2007). Multi-view stereo for community photo collections, *Proceedings of IEEE 11th International Conference on Computer Vision*, IEEE, Rio de Janeiro, Brazil, pp. 1–8.

Google (December 2009). Sketchup.
URL: *http://sketchup.google.com/*

Hoppe, H. (1994). *Surface Reconstruction from Unorganized Points*, PhD thesis, University of Washington, Seattle, WA, USA.

Hoppe, H., DeRose, T., Duchamp, T., McDonald, J. and Stuetzle, W. (1992). Surface reconstruction from unorganized points, *SIGGRAPH '92: Proceedings of the 19th annual conference on Computer graphics and interactive techniques*, ACM, New York, NY, USA, pp. 71–78.

Jarvis, R. A. (1977). Computing the shape hull of points in the plane, *Proceedings of the IEEE Computing Society Conference on Pattern Recognition and Image Processing*, IEEE, New York, NY, USA, pp. 231–241.

Klein, G. and Murray, D. (2007). Parallel tracking and mapping for small AR workspaces, *Proceedings of the Sixth IEEE and ACM International Symposium on Mixed and Augmented Reality (ISMAR'07)*, Nara, Japan.

Leonard, J. J. and Durrant-Whyte, H. (1991). Simultaneous map building and localization for an autonomous mobile robot, *IEEE International Conference on Intelligent Robot Systems*, Osaka, Japan.

Microsoft (December 2009). Photosynth.
URL: *http://photosynth.net/about.aspx*

Niem, W. and Broszio, H. (1995). Mapping texture from multiple camera views onto 3d-object models for computer animation, *in Proceedings of the International Workshop on Stereoscopic and Three Dimensional Imaging*, pp. 99–105.

Pan, Q., Reitmayr, G. and Drummond, T. (2009). ProFORMA: Probabilistic Feature-based On-line Rapid Model Acquisition, *Proc. 20th British Machine Vision Conference (BMVC)*, London.

Pedersini, F., Tubaro, S. and Rocca, F. (1993). Camera calibration and error analysis. an application to binocular and trinocular stereoscopic system, *in Proceedings of the 4th International Workshop on Time-verying Image Processing and Moving Object Recognition*, Florence, Italy.

Scharstein, D. and Szeliski, R. (2003). High-accuracy stereo depth maps using structured light, *CVPR '03: Proceedings of 2003 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR 2003)*, Madison, WI, USA, pp. 195–202.

Thormälen, T. (2006). *Zuverlässige Schätzung der Kamerabewegung aus einer Bildfolge*, PhD thesis, University of Hannover.

van den Hengel, A., Dick, A., Thormählen, T., Ward, B. and Torr, P. H. S. (2007). Videotrace: rapid interactive scene modelling from video, *SIGGRAPH '07: ACM SIGGRAPH 2007 papers*, ACM, New York, NY, USA, p. 86.

Wong, E., Martin, D., Chudak, D., Lasky, A., McKinney, G., Simon-Parker, L., Wolff, B., Cutting, G., Uchida, W., Kacyra, B. and Kacyra, B. (1999). Lidar: reality capture, *SIGGRAPH '99: ACM SIGGRAPH 99 Electronic art and animation catalog*, ACM, New York, NY, USA, p. 158.