# Ray Traversal for Incremental Voxel Colouring

A thesis
submitted in partial fulfilment
of the requirements for the Degree
of
Master of Science
in the
University of Canterbury
by
O. W. Batchelor

University of Canterbury
2006

# Abstract

Image based scene reconstruction from multiple views is an interesting challenge, with many ambiguities and sources of noise. One approach to scene reconstruction is Voxel Colouring, Seitz and Dyer [26], which uses colour information in images and handles the problem of occlusion.

Culbertson and Malzbender [11], introduced Generalised Voxel Colouring (GVC) which uses projection and rasterization to establish global scene visibility. Our work has involved investigating the use of ray traversal as an efficient alternative.

We have developed two main approaches along this line, Ray Images and Ray Buckets. Comparisons between implementations of our algorithms and variations of GVC are presented, as well as applications to areas of optimisation colour consistency and level of detail.

Ray traversal seems a promising approach to scene visibility, but requires more work to be of practical use. Our methods show some advantages over existing approaches in time use. However we have not been as succesful as anticipated in reconstruction quality shown by implementation of optimisation colour consistency.

Table of Contents

## Acknowledgments

We would like to thank all the hedgehogs. Without their help, this thesis would not have become a reality.

## *0.1* Common definitions

**Voxel** - A unit of volume, which for all practical purposes used here, it is a cube of unit size.
**Voxel array** - A 3 dimensional array of voxels.
**Dim** - The dimensions of the voxel array.
**SVL** - Surface Voxel List, the set of solid voxels adjacent to one or more empty voxels.
**VVL** - Visible Voxel List, the set of all voxels which are visible.
**CVS** - Changed Visibility Set, the set of voxels with changed visibility resulting from the carving of a voxel.
**CVSVL** - Changed Visibility Surface Voxel List, the set of all voxels which have changed visibility since they were last evaluated for colour consistency.
**IB** - Item buffer, an image comprising of Voxel IDs.
**LDI** - Layered Depth Image, an image comprising of ray surface voxel intersection lists.
**DSI** - Depth Stack Image - an image comprising of Depth Stacks (DS)
**DS** - A stack of all ray voxel intersections in furthest first order.
**Vis(V)** - The visible projection of a Voxel V onto an image/Item buffer.
**LOD** - Level Of Detail
**GVC** - Generalised Voxel Colouring

# Chapter I

# Introduction

Scene reconstruction is the process of generating scene geometry from input sensors. There are many varieties of scene reconstruction. Image based reconstruction from multiple views is an interesting challenge. One form of reconstruction from images is voxel colouring, which began with Seitz and Dyer in [26]. Recently varieties have appeared to make use of many cameras at arbitrary viewpoints, notably Generalised Voxel Colouring (GVC) [11].

Our work has involved investigating the use of ray traversal as an efficient alternative to purely projection and rasterization approaches, notably [11] in the process of finding visibility for voxel colouring. We have developed two approaches, one ray images, the other ray buckets for computing incremental voxel colouring - both using voxel ray traversal.

In the first approach we build on the GVC method, using projection and rasterization with an item buffer to map voxels to visible rays, using ray traversal to update individual rays changed in the item buffer. For the second approach we used a different organisation of visibility data strucutes, associating visibility information directly with voxels.

## *1.1* Motivation

There are many uses of image based scene reconstruction including 3D photography, capturing landmarks and heritage. Our interest in the topic stems from the ability to create artwork from images for writing small games. Voxel colouring [26] became a topic of interest due to its elegant simplicity. However, currently image based scene reconstruction suffers from lower quality and longer processing times compared to active capturing methods.

## *1.2* Goals

The initial goal was to provide a better means for implementing optimisation based voxel colouring, which appeared to produce some very good quality results, but with a rather heavy computational cost. The layered depth image approach of [11] required a large amount of memory and time. We have applied our approaches to this task, the first proved problematic - prompting further investigation, which yielded the second.

It was initially envisioned that the independence between rays would be a very useful property, as it could lead to parallel implementations on a GPU for example. However it proved equally useful as a means for updating visibility in a serial/incremental means, which ended up being the focus of this research.

## *1.3* Overview

Firstly, we give a summary of important background material and previous research in the area. Visual hull reconstruction including Voxel Colouring, Space carving and GVC, as well as colour consistency. We then describe our implementation of relevant algorithms for comparisons, GVC with OpenGL and the rasterization of cubes.

We describe each of our approaches in detail, beginning with some less successful methods - Depth Stack Images and how they led to Ray Images. Our difficulties with using ray images to implement optimisation colour consistency are explained, which leads to the Ray Buckets algorithm which is discussed in detail.

We implement various methods and compare them with our approaches. We make comparisons on runtime statistics and reprojection error using threshold colour consistency for a number of implementations, trials with optimisation colour consistency and comparisons between methods for use with level of detail methods. Results show that our method is useful in some situations, and compares favourably to GVC-LDI.

# Chapter II

# Background and Previous work

Different approaches to scene reconstruction can be categorised over several axes: Active vs. passive reconstruction, sparse vs. dense/volumetric and global vs. local. We focus on image based (passive), local, volumetric scene reconstruction - specifically voxel colouring.

Active acquisition involves projecting patterns or light onto a scene. The correspondences between views can be found very accurately, leading to good quality reconstruction. Active acquisition involves separate hardware - an example is using laser range finders.

Passive acquisition (image based) involves measuring the existing lighting, which is more accessible and more widely used than active capturing technologies. Image based reconstruction is often less accurate and contains more ambiguities than active reconstruction.

Sparse reconstruction involves identifying and solving for sparse scene features (eg. points, lines) and camera geometry. Two examples are point clouds from an active acquisition and structure from motion (SFM). Recent surveys on this topic include [34, 13].

Dense reconstruction focuses on reconstructing every point in the scene (or a continuous volume). Dense stereo is an example, where an algorithm attempts to assign a depth to every pixel in the view image. Volumetric reconstruction is a kind of dense reconstruction, where a scene is divided into volume elements (voxels), and every voxel marked opaque or transparent (or some level inbetween).

Global reconstruction algorithms attempt to consider the whole (or large parts) simultaneously. A good example is using global minimisation technique with stereo, where graph cuts [5] or dynamic programming [21] can be used to penalise bad matches and local discontinuity. These algorithms are used to find a global solution where constraints (prior knowledge) are provided to regularise the solution.

A local reconstruction algorithm only examines a fraction of the scene at any time. Voxel colouring is a good example of a local reconstruction algorithm. Colour consistency imposes no relationship (bar visibility) between voxels. As such, it does not make assumptions about the scene (eg. objects have smooth surfaces), except those required to simplify the problem.

## *2.1* Structure from motion

Structure from motion includes a wide range of reconstructions, from solving for epipolar–lines in two images, to solving for unknown camera parameters and points in many images. A common scenario is solving equations where many noisy but overspecified constraints are identified from known relationships. For example the camera geometry and the relationship between projected points etc. A recent publication on how to achieve reconstruction of points and cameras by solving purely linear equations is found in [25]

Sparse reconstruction is especially important as, to perform higher level reconstruction - eg. volumetric reconstruction, it is usually necessary to know camera parameters, so it may first involve some form of sparse reconstruction, eg. camera calibration with grid or markers, SFM from image features. Examples of using both sparse then volumetric reconstruction are presented in [6, 24].

## *2.2* Volumetric reconstruction

Volumetric reconstruction attempts to reconstruct every scene point (or an approximation of). Discrete, volumetric reconstruction involves representation of the scene by discrete volume elements (voxels). Volumetric reconstruction algorithms include computerised tomography, silhouette intersection, level sets [28], various types of stereo and Voxel colouring. Two surveys on volumetric reconstruction (mostly centred on Voxel colouring) are [30, 12].

Volumetric reconstruction algorithms can also be divided over how visibility is treated. Most computer vision based algorithms usually assume a

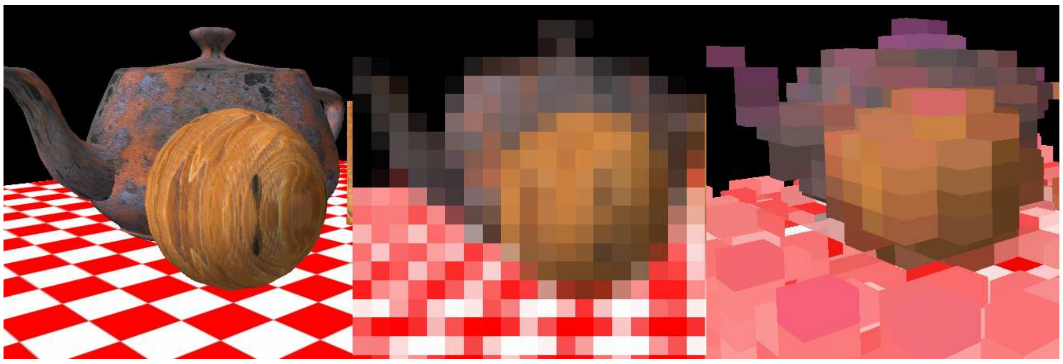

Figure 2.1: (a) An image, (b) pixels of an image at $34 \times 21$, (c) a rough voxel model at $20 \times 20 \times 20$.

scene with binary opacity and occlusions. Various types of computer tomography (typically used for medical applications), assume a translucent scene (without refraction) such as from an X-Ray. An image intensity at a point is treated as an integral of material down a ray and occlusions are usually ignored. Tomography has been applied to vision problems too [14], and some vision problems applied to scenes with transparent [3] or reflective surfaces [4].

Voxels may be interpreted in two ways. One is to consider them to be a point sample in a volume, the other is to consider them to be a unit of volume, eg. a cuboid. Tomography, Voxel Colouring and Space Carving all consider voxels to be point samples, whereas Generalised Voxel Colouring considers voxels to be volumes.

We base our work on the framework presented in Generalised Voxel Colouring and consider voxels as volumes. Figure 2.1 shows an example of a low resolution cuboid voxel representation of a model (c), and for comparison a pixel representation of an image (b), with the original image of a continuous scene at high resolution.

Additional implications arise from these interpretations, the major one being how the image of a voxel is treated. If a voxel is a point sample, then it will project to a point on each image. Whereas if a voxel is a volume, then a voxel will project to an area on each image. This has implications with regard to colour consistency.

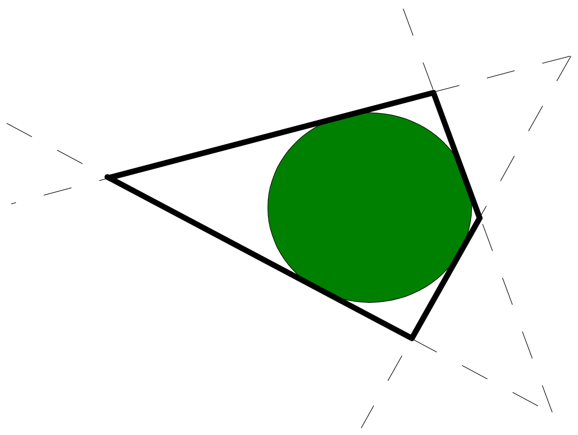

Figure 2.2: The visual hull of an object from two cameras.

## *2.3* Silhouette, Visual hull reconstruction

A silhouette gives an area (in an image) for which there exists, at an unknown depth, a non background object. Silhouettes can be created by segmenting an image, most often from a controlled environment with a solid background colour eg. a bluescreen. Figure 2.2 shows the 2D visual hull of an ellipse.

The simplest volumetric reconstruction algorithm is silhouette intersection, which involves extruding the silhouette of each input image and finding the intersection of all extrusions. Silhouette intersection is most easily performed using a discrete voxel approach. The intersection algorithm carves away all non-foreground voxels for each image.

The result of silhouette intersection is known as the *Visual Hull*. This is a conservative estimate of the shape in the foreground, it is often used as a starting point for other algorithms or in addition to other information. However, this use is restricted as it requires an accurate segmentation, usually requiring a controlled environment.

An example of early use of silhouette intersections for reconstruction is [35]. A recent example using stereo and silhouettes together, giving very good quality reconstructions is presented in [2]. Voxel colouring [26] also may make optional use of silhouettes.

## *2.4* Voxel Colouring

Voxel Colouring [26, 27] is an attempt to use the colour information in multiple views to obtain a reconstruction. Voxel Colouring can be characterised by its general scene representation, binary colour consistency measure, explicit occlusion handling and arbitrary number of input cameras. A review of recent progress is presented in [31].

In contrast to stereo algorithms which directly represent correspondences as disparities, image correspondences are represented implicitly by the projections of each voxel to images. For this, camera geometry must be fully known (up to a scale factor).

Points in the scene which are roughly 'the same' colour when projected to non-occluded images are termed *colour consistent* or *photo consistent*. Photo consistent points are deemed to be part of the shape, and those which are deemed inconsistent, are not. This assumes a lambertian scene, where, given all other conditions constant, a point should appear to be the same colour independant of view angle.

Voxel Colouring explicitly handles occlusion by identifying a class of camera configuration, given by the *ordinal visibility constraint* in which cameras are restricted in a way which allows explicit front to back iteration of the voxellised volume.

The ordinal visibility constraint is defined in [26] as:

> There exists a norm $|\cdot|$ such that for all scene points $P$ and $Q$, and input images $I$, $P$ occludes $Q$ in $I$ only if $|P| < |Q|$.

In practice this means that the voxel volume will consist of a series of voxel planes. Voxels within a plane have the same norm, thus cannot occlude each other. This means the voxel volume need only be iterated in one pass (in planes). Each voxel may be examined for colour consistency just once. Solid voxels are masked with image bitmasks, so that voxels farther away may be identified as occluded. The Voxel Colouring algorithm is outlined in figure 2.3. Vis(V) is the set of projected pixels, Proj(V), masked by the marked pixels in each image.

```
initialise Solid voxel set to null

for each voxel plane 1 to r {

    for each voxel V in plane {
        clear Vis(V)

        for each image we {
            compute footprint p of V in we
            add unmarked pixels in p to Vis(V)
        }

        s = consistency (Vis(V))
        if(s < threshold) {
            add V to Solid voxel set
            add Vis(V) to ToMark
        }
    }
    mark pixels in ToMark
    clear ToMark
}
```

Figure 2.3: Voxel Colouring algorithm

A key property of this voxel colouring algorithm is that occlusion is explicitly handled. Once a voxel is marked solid, it casts a ”shadow” over voxels in successive iterations by marking the pixels in its projection as occluded using bitmasks over input images. Note that the reason for using *ToMark* rather than directly marking voxels, is that voxels on the same plane should not occlude each other (they have the same norm, from the ordinal visibility constraint).

The ordinal visibility constraint provides a mechanism for an efficient, mechanical front to back traversal, and works in a good deal of camera configurations. Derivatives of Voxel Colouring develop algorithms for handling occlusion for arbitrary camera configurations. We refer to this family derivatives, including voxel colouring, as ”voxel colouring algorithms”.

## *2.5* Space Carving

In [16, 19] Kutulakos and Seitz set out a general framework for volumetric reconstruction based on photo consistency. They define the limits of a reconstruction based on photo consistency – the photo hull, then present an algorithm to find the photo hull, called Space Carving.

### *2.5.1 Photo hull*

The geometry which can be inferred from multiple views is ambiguous, since there are a range of photo consistent shapes which may satisfy a set of input images. It is trivial to show this is the case. For example, the geometry of a solid object could be faked by a set of precisely placed images.

Figure 2.4 shows the 2D visual hull and photo hull of a square with two different colours per side. The visual hull provides an approximation of the shape, the photo hull gives a slightly closer approximation. Note that the quality of the photo hull depends heavily on the colours of the object. Here the approximation is not much better than the visual hull because of few changes in colour. If the object was of constant colour, the photo hull is equal to the visual hull. The photo hull is always equal or a subset of the visual hull, as the real object is always equal or a subset of the photo hull. Visual hull $\supseteq$ Photo hull $\supseteq$ Real object.

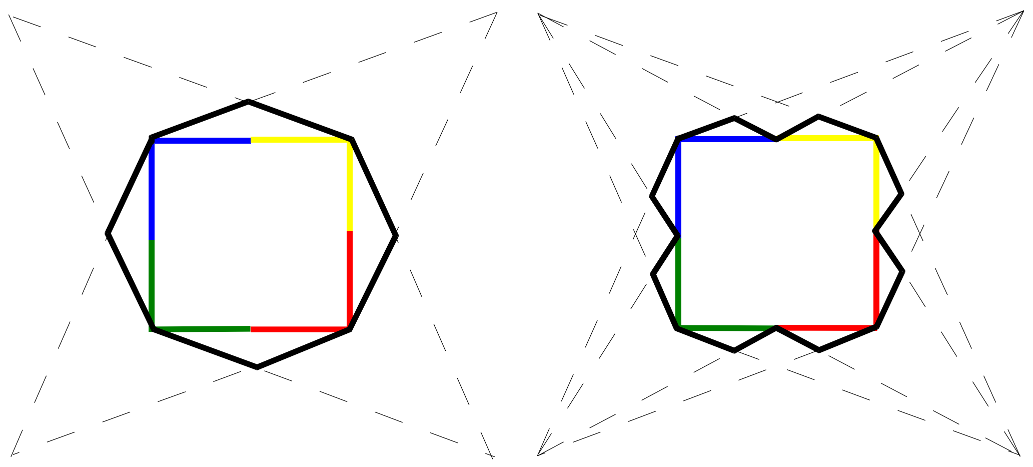

Figure 2.4: A visual hull and a photo hull (solid black lines) of a coloured square - the difference between them is the area where projections show two different colours.

Photo consistency $consist_K()$ is defined in [19] as:

1. An algorithm $consist_K()$ is available that takes as input at least $K \leq N$ colors $col_1, ..., col_K$, $K$ vectors $\xi_1, ..., \xi_K$, and the light source positions (non-Lambertian case), and decides whether it is possible for a single surface point to reflect light of color $col_i$ in direction $\xi_i$ simultaneously for all $i = 1, ..., K$.
2. $consist_K()$ is assumed to be monotonic, i.e., $consist_K(col_1, ..., col_j, \xi_1, ..., \xi_j)$ implies that $consist_K(col_1, ..., col_m, \xi_1, ..., \xi_m)$ for every $m < j$ and permutation of $1, ..., j$.

The shape which is the union of all consistent shapes, is termed the *maximum photo-consistent shape* or *photo hull.* In [19] the photo hull is defined as:

> Let $\mathbf{V}$ be an arbitrary subset of $\mathbb{R}3$. If $\mathbf{V}$ is the union of all photo-consistent shapes in $\mathbf{V}$, every point on the surface of $\mathbf{V}$ is photo-consistent. We call $\mathbf{V}$ the photo hull.

The photo hull becomes an upper bound for all shapes which may project to the same set of images. Kutulakos and Seitz then prove that given a monotonic photo consistency function (and perfect inputs), their space carving algorithm will produce the photo hull.

A key idea introduced by the Space Carving algorithm is the idea of *least commitment* reconstruction. A superset of the true scene is used as an estimate for scene visibility. Used with a monotonic colour consistency function, carving is *conservative.* No voxel evaluated with a subset of its true visibility will be marked inconsistent. Most practical colour consistency functions are not monotonic, e.g. thresholded standard deviation is not monotonic.

Despite the possibility of using non lambertian scenes, in practice it is assumed that a scene is lambertian. So the definition of $consist_K()$ reduces to checking that $col_1, ..., col_j$ are identical within bounds of calibration error and noise.

### *2.5.2 Space Carving algorithm*

The Space Carving algorithm [16, 19] generalises Voxel Colouring to allow arbitrary camera placement. It does this by performing several plane sweeps, only a subset of cameras satisfying the Ordinal Visibility Constraint are used at any point. The images/cameras which satisfy this condition are the cameras which lie behind the path of the sweeping plane.

Space Carving uses plane sweeps along axes of the volume bounding box. Each voxel on the plane is examined for colour consistency only among pixels of (non occluded) cameras which are behind the sweeping plane. Voxels which are not consistent are removed. Voxels which are consistent are marked using bitmasks over input images (in the same fashion as Voxel Colouring) for masking visibility of subsequent voxels.

The space carving algorithm as described in [16] (slightly paraphrased to fit terms used here) is listed in figure 2.5. In addition to this algorithm they describe the multi-plane sweep algorithm, which performs the plane sweep for several planes, and then performs a post-evaluation for each voxel using visible cameras from all plane sweeps.

Space carving has been implemented using graphics hardware [8, 20], to

```
initialise Solid volume to superset of the scene
initialise Plane behind volume

intersect Plane with Solid
for each surface voxel V on Plane {
    for each camera C behind Plane {
        compute projection p of V in C
        if p is unmarked add V to Vis(V)
    }

    if (consist (Vis(V))) {
        mark projection of voxels in Vis(V)
    } else {
        remove V from Solid
    }

    move Plane foward one step

} until(Plane lies in front of volume)
```

Figure 2.5: Space carving algorithm. Note that the terminology differs from [16] Vis(V) is used in place of $col_1, ..., col_j$, and *Solid* in place of *V*.

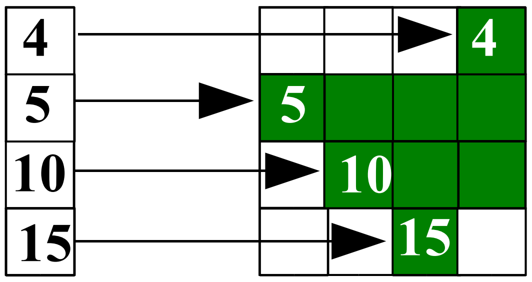

Figure 2.6: An item buffer of a voxel model, the numbers are voxel IDs.

which it is well suited because of the fast projection of an image to a plane, performed by texture mapping.

## *2.6* Generalised Voxel Colouring

The Generalised Voxel Colouring of Culbertson and Malzbender [11], (GVC) generalises Space Carving so that all views can be considered simultaneously, rather than with separate passes. GVC follows the conservative carving approach of Space Carving. It uses rasterization with depth buffering to determine the closest visible voxel projecting to a pixel.

### *2.6.1 Item buffers*

The set of solid voxels which adjoin (6-connected) one or more empty voxels is termed the Surface Voxel List, or SVL. The SVL is updated by examining neighbours as voxels are carved In [11], the SVL is implemented as a hash table, and the voxel volume is implemented as a bitfield. The term 6-connected refers to cubes which connect to their neighbours by thier faces (a cube has 6 faces).

Each voxel is assigned a unique ID. An item buffer is an image of voxel IDs, and their depth (distance from the camera). Each voxel ID corresponds to the nearest visible voxel at each pixel of an input image. An item buffer corresponds one to one with the colours in an input image. An example item buffer is shown in figure 2.6.

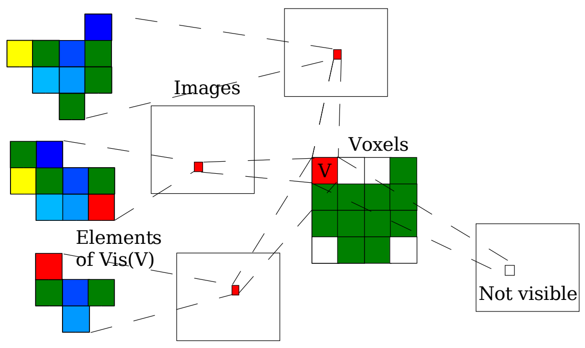

Figure 2.7: Vis(V), the set of pixels in visible projections of a voxel in each image.

An item buffer can be computed by rasterizing the set of surface voxels (SVL) to an image using depth buffering. A rasterization algorithm will produce a set of pixels and depths for a voxel. These pixels are updated in item buffers. A pixel is masked if the incoming depth is farther than the existing depth, but replaced if it is less.

An item buffer can be used to quickly establish the set of visible pixels for a voxel V. Items in the projection of V are visible if they have the same ID as V.

Given a set of item buffers, one for each input image, Vis(V) may be found. Vis(V) is the set of visible pixels (for one voxel, V) in all item buffers, an example is shown in figure 2.7. Vis(V) is a subset of Proj(V), the projection of V to each image. This relationship can be stated as, $p \in Proj(V), V \in Item(p) \Leftrightarrow p \in Vis(V)$.

The Vis(V) set establishes correspondences between pixels in multiple images, this can be used to determine colour consistency. An advantage of GVC over Space Carving is that at any point, the full colour information is used (from Vis(V)), where as Space Carving uses a limited set depending on the carving plane. Another difference is that GVC treats Voxels uniformly as a volume, with projections as an area. Vis(V) contains potentially several pixels per image.

```
initialise Solid, SVL

do {
  compute item buffers by rasterizing each voxels in SVL

  for every voxel V in SVL {
     compute Vis(V)

     if (not consist(Vis(V))) {
       carve V
       remove V from SVL

       add uncarved neighbours of V to SVL
     }
  }
} while(some voxels carved)
```

Figure 2.8: GVC algorithm, [32]

### *2.6.2 GVC algorithm*

GVC uses an iterative process of calculating item buffers and evaluating/modifying surface voxels. Results in [11] showed that this gave an improvement in reconstruction quality over Space Carving. One slightly problematic issue with GVC is that it becomes much slower as the number of voxels carved per iteration is reduced. This occurs close to convergence. GVC-LDI also [11], is given as a solution which carves each voxel in serial. The GVC algorithm is given in figure 2.8, definitions of Vis(V) and the SVL follow.

## *2.7* GVC-LDI

GVC with Layered Depth Images (GVC-LDI) [11] uses Layered Depth Images as a means of updating visibility. GVC-LDI is a serial carving process, compared to GVC's parallel carving process. The major difference lies in the granularity of updates of the item buffers. GVC-LDI uses LDIs to determine the exact change in visibility of carving one voxel.

Only voxels which change visibility require re-evaluation of colour consistency. GVC-LDI keeps track of these voxels in the Changed Visibility Sur-

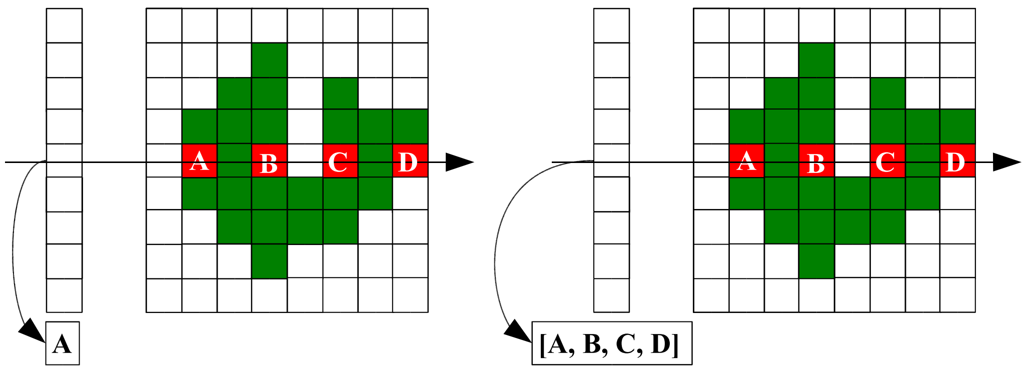

Figure 2.9: An element of a GVC item buffer, and an element of a LDI.

face Voxel List (CVSVL). Voxels are removed one by one from the CVSVL, if found inconsistent they are carved, and any voxels changing visibility as a result are added to the CVSVL for re-evaluation.

The CVSVL may be considered a carving queue. A possibility is that re-arranging the carving order (a priority queue) may improve results.

### *2.7.1 Layered Depth Images*

In the context of GVC-LDI, a layered depth image is an extension of an item buffer. For each pixel P, instead of containing just the nearest voxel ID, an LDI instead stores (at P) a linked list of all intersections (Voxel ID, depth) with surface voxels (SVL). The lists are stored in near to far depth ordering, so that the top elements of all lists may be used as an item buffer. Voxels are added to the list at a certain depth, the insertion place is found by linear search. Removing voxels from the list is by linear search of Voxel ID.

A comparison between an item buffer and a layered depth image is shown in figure 2.9. The item buffer stores only the first surface intersection at the given pixel (A), whereas the layered depth image stores every surface intersection sorted by depth. In this case there is a hole in the centre of the voxel model, so (A, B, C and D) are all surface intersections.

The Changed Visibility Set (CVS) is the set of all voxels which *become* the head of each list within changed LDIs. This includes voxels within lists

```
initialise Solid, SVL
initialise CVSVL to SVL
render SVL to LDIs

while (CVSVL not empty) {
    delete V from head of CVSVL
    compute Vis(V)

    if(not consist(V)) {
        carve V
        remove V from SVL

        remove V from each LDI(P) in Proj(V)

        for each neighbor N of V not in SVL {
            add N to SVL, each LDI(P) in Proj(N), CVSVL
        }

        for each LDI(P) with a changed head U, add U to CVSVL
    }
}
```

Figure 2.10: GVC-LDI algorithm, [32]

already when the top voxel is carved, as well as voxels which are uncovered and become a new member of a list (and the SVL).

GVC-LDI shows some advantages over GVC. These include:

- It performs many less colour consistency evaluations.
- Serial carving process allows rapid convergence towards end of process.
- Carving order becomes more flexible.
- There is potential to use the exact visibility change in the colour consistency test.

However, the implementation using LDI means there are some significant downsides, which are:

- It uses orders of magnitude more memory - O(LDI depth * pixels).
- It has a much greater constant cost per voxel. (It uses up to six voxel projections per voxel evaluation to GVC's one).

Although [11] did not report such a significant difference in running times between GVC and GVC-LDI, it has been our experience in implementing these two algorithms, that GVC-LDI is much slower on the whole. Some comparisons from our implementations are presented in section 8.

## *2.8* Colour consistency

Colour consistency is a decision. It aims to determine if it is possible for one voxel to simultaneously reflect the light, as measured by its visible projections to images. For lambertian scenes this amounts to determining if it is possible within error (eg. calibration and discretisation errors), that each visible projection is of the same voxel, of a constant colour.

Colour consistency is used with voxel colouring algorithms to determine if a voxel should be opaque or transparent. If a voxel is deemed colour consistent the voxel is considered opaque, otherwise it is considered transparent.

There has been a great deal of work put into finding a consistency function which will adequately reconstruct lambertian scenes of various texture and shape. Colour consistency functions can be divided into several categories, monotonic, non monotonic, those which operate on groups of pixels between images, those which operate on a single pixel (treat a voxel as a point), colour set matching, statistically based and optimisation based.

Colour consistency has many issues in practice. Both the nature of surface texture and scale, as well as noise, can cause errors in reconstruction. Noise mainly will result from camera calibration, image noise, and lighting issues. Any errors can accumulate and cause voxels to seem inconsistent, while solid in the true scene or consistent while empty. A reconstruction can end up with both types of error simultaneously. These errors can cascade, and cause huge errors beyond the initial misclassification.

### *2.8.1 Monotonic consistency function*

To ensure a carving process is conservative, a monotonic colour consistency function is required. A monotonic consistency function is one where if a voxel is deemed colour consistent for a certain set of pixels, then it will always be consistent for a subset of those pixels. This is a desirable property for maintaining a conservative approach, but seems awkward from the point of view of normalising a colour consistency test to be fair for a variable number of views. Colour caching [9] and histograms [10] are two examples of monotonic consistency tests.

### *2.8.2 Voxel projection sampling*

The representation of a voxel, and its projection to images is an important consideration for a colour consistency function. On one hand, a voxel can be treated as a point which projects to a point on images, as Space Carving [16]. On the other hand a voxel can be treated as a unit of volume, which projects to an area on images [11].

The method by which a voxel area is sampled is also an important consideration, GVC [11] treats a projection as a discrete set of pixels - Vis(V), where as [7] uses a weighting for pixels on the edge of a voxel's projection.

Using an area projection means that instead of having a single pixel/colour per image, one has a set of pixels/colours per image. This means that one can use the statistics of the local image region in the colour consistency test. Broadhurst and Cipolla use analysis of variance in [7].

Another approach discussed in [17] is to assume there may be some errors in inputs which effectively result in local permutation within a radius. Assuming the correct projection will be within this radius, a disc of pixels around a voxel's projected point can be used to assess colour consistency, and this will include the correct colour. This is also discussed below.

## *2.9* Colour consistency functions

A useful measure for evaluating the results of reconstructions is the reprojection error. This is the output image of voxels are reprojected and compared with input images on a pixel by pixel basis. The comparison is usually the difference squared between colour components of each pixel of an image of reprojected voxels, and the input image. This can also be computed on a per voxel basis by summing the error, for each pixel in its projection.

$$C_v = \frac{\sum_{p \in vis(V)} C_p}{|Vis(V)|} \tag{2.1}$$

$$error(V) = \sum_{p \in vis(V)} dist(C_p, C_v) \tag{2.2}$$

$$dist(C1, C2) = \sum_{\alpha=1}^{3} (C1_\alpha - C2_\alpha)^2 \tag{2.3}$$

$$consist(V) = error(V) < \tau \tag{2.4}$$

Where the subscript $X_\alpha$ refers to the channel $\alpha$ of colour $X$. $C_p$, $C_v$, refers to the colour of pixel p, voxel v.

Reprojection error can be used as a monotonic colour consistency function, as shown in equation 2.4. This provides a normalised value for the reprojection error, however it is not monotonic because of this normalisation. Reprojection error has also been used with optimisation processes as discussed in section 2.9.2.

A simple way to use multi-pixel projections is by a weighted average of (euclidian) differences of voxel colours from each view, and the overall mean voxel colour. This is shown in equation 2.5, where $pixels(V, i)$ is the number of pixels in the projection of voxel V, to image i, $C_i$ is the average colour of pixels in projection to image i, $C_v$ is the average colour of all pixels in the projection of voxel V (equation 2.1).

$$between(V) = \frac{\sum_{i=1)^n} dist(C_i, C_v) * pixels(V, i)}{|Vis(V)|} \tag{2.5}$$

Another function is presented in [32] is termed the adaptive standard deviation test. It compares the ratio of *within image variance* to *between image variance*. Where *within image variance* refers to the variance within Pixels(V, i) and *between image variance* refers to variance between the means of Pixels(V, i).

Let $s_{between}$ be the standard deviation between the means of image colours, and $\overline{s_{within}}$ be the mean of the within image standard deviations. Two thresholds are used, determined by experimentation $\tau_1, \tau_2$.

$$consist(V) = s_{between} < \tau_1 + \tau_2 \overline{s_{within}} \tag{2.6}$$

An alternative to the *average* within image standard deviation, is to use the *minimum*. We have found this efffective, as textureless regions on the input scene, which usually correspond to textureless regions on all images. If a textureless region exists on one input image then it may be reasonable to expect textureless regions on all images.

The adaptive standard deviation test is really an approximation of the full analysis of variance test as used by [7] between groups of pixels.

### *2.9.1 Colour sets*

Corresponding pixel sets can also be directly compared. One way to do this, termed Colour Caching, is presented in [9], where a voxel is consistent if there are any similar colours included in all sets. A similar method uses histograms [10], where the colours set in each image are discretised into histograms. A voxel is considered colour consistent if every pair of histograms intersect.

In [18] Kutulakos presents a framework for reconstruction where calibration and image errors are taken into consideration. The idea is that an image be treated as if it had a *shuffle transform* applied, as a model for noise and errors. A shuffle transform of size *N*, shifts each pixel in an image up to *N* pixels from where it originated.

A colour consistency function taking the shuffle transform into account is required to scan a much larger area in order to find the real occupying pixel. In practice, implementation involves searching a disc around a voxel's centre of projection. A colour consistency test requiring similar colours in each disc is used. This can be then used in a coarse to fine reconstruction algorithm, moving from a larger to smaller sized disc.

### *2.9.2 Optimisation based colour consistency*

Slabaugh et al. [29] introduced a different paradigm of colour consistency, based on carving voxels only if they improve a global function. An optimisation based colour consistency test turns voxel colouring into an energy minimisation search problem. This is intended to solve some of the problems with threshold based consistency tests. Problems are determining the ad-hoc value for the threshold, and a single threshold can often be both too large and too small in different sections of the same scene.

If a voxel's approximate colour is determined as the average of all pixels to which it projects, then its reprojection error can be computed by equation 2.2. One can compute a voxel colour, (and thus reprojection error), for every pixel in an image. Global reprojection error can then be calculated as the sum of reprojection errors over every pixel.

This global reprojection error can be examined before and after the carving of one voxel. Reprojection error will change for not only the pixels in the projection of the removed voxel, but all pixels in projections of voxels with changed visibility. Taking the difference of error in this set of pixels before and after carving, gives us the change in reprojection error. The change in reprojection error can be used as a colour consistency function. A voxel is carved only if it improves the reprojection error.

This can be made practical by reducing the problem to a local calculation, by realising that only a small subset of voxels change colour with the carving of one voxel (as voxel colours and their reprojections are independent of each other). This set is termed the CVS, the Changed Visibility set, and only the visible projections of this set will change colour, shown in equation 2.7.

Only pixels in the projection of the CVS voxels need be compared, to determine if the carving of one voxel increases or reduces the global reprojection error. The reprojection error of pixels of (2.7) are summed (2.8) before and after the carving of a voxel. If the reprojection error is improved (reduced) after carving then the voxel is colour consistent (2.9).

A method for finding the CVS is GVC-LDI, [11] which is used by Slabaugh et al. [29] in their algorithm. In GVC-LDI this set is found for voxel V, as the set of all voxels U which became the head of an LDI, as shown in figure

2.10.

$$CVRS(V) = \bigcup_{U \in CVS(V)} vis(U) \tag{2.7}$$

$$reproj(V) = \sum_{P \in CVRS(V)} distSq(C_v, C_p) \tag{2.8}$$

$$consist(V) = reproj(V) \leq carved(reproj(V)) \tag{2.9}$$

One of the problem is that it contains many local minima. Far from the global minimum, reprojection error is rather noisy. On its own using equation 2.9 as a consistency function, away from the global minimum, produces no useful decision.

Slabaugh et al. [29] treat optimisation as a post process from an existing voxel colouring algorithm, which ensures that the starting conditions are close to a global minimum. They also consider *adding* voxels, rather than a purely carving approach, using simulated annealing to overcome local minima.

Results from [29] show that this process produces voxel models which have less noise and fit the original scene more closely. Despite achieving good reconstructions, their results show that the process is very slow. Quoting figures from the results section, they used a HP J5000 workstation with two 440MHz processors and 2GB of RAM. For their shoes scene, a real scene, at $144 \times 128 \times 90$ initialised by the GVC-LDI algorithm which took 71 minutes, execution for their greedy method took 491 minutes, and their simulated annealing 709 minutes while consuming 800 MB of RAM.

An unknown aspect of optimisation colour consistency, is handling background pixels. What should be done about pixels which were originally coloured, but became background pixels on the carving of a voxel? Background pixels are those pixels in the visible projections of a voxel V, which are not in the set given by equation 2.7.

$$Background(V) = Vis(V) \setminus CVRS(V) \tag{2.10}$$

For segmented images this is not a problem. Background pixels should never occur unless holes are mistakenly introduced. For non-segmented images, it is not clear as to what part they should play. This is potentially problematic, as the "trivial solution", where all voxels are carved, will always have the lowest reprojection error! In [29], a fixed penalty is assigned to background pixels to stop holes forming in the model.

# Chapter III

# Implementation issues

## *3.1* GVC implementation

### *3.1.1 Voxel projection*

Voxel rasterization lies at the heart of GVC-based algorithms. In order to make a good comparison between them, voxel rasterization must be implemented in a reasonably efficient fashion. As we assume voxels to be unit cubes, we also use cubes to rasterize voxels. This is the only sensible choice. Although it is possible to use others, such as points, there is no guarantee that these will be watertight or be consistent between views.

Rasterization is performed as a three step process.

- Corner points are projected using the projection matrix (pinhole camera model). Clipping is done here, rather coarsely (if no corners are visible then the voxel is rejected).
- Faces are culled (as discussed below).
- Faces are rasterized as quads by a scanline or half-space quad rasterization method.

## *3.2* Face culling

Visible faces of a cube may be efficiently culled by determining the nearest feature to the camera. If this is a face, then that face only will be visible. If it is an edge, two faces will be visible. If it is a corner, three faces will be visible. This can be determined on a face by face basis for an axis aligned cube, by comparing the camera centre with the axis aligned planes of the

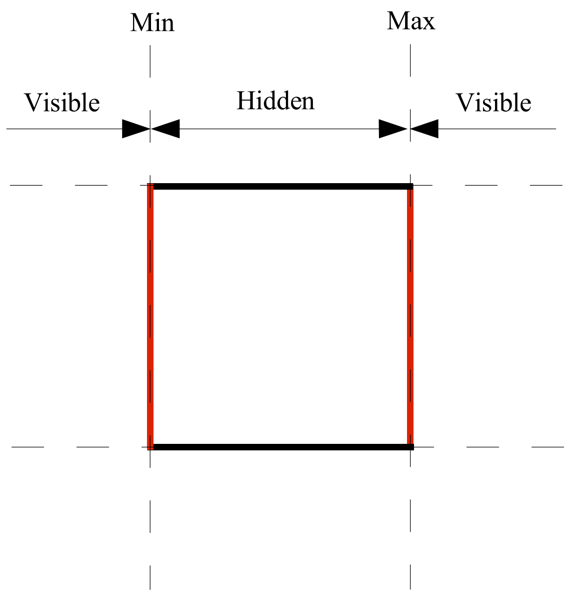

Figure 3.1: The visible faces of a square dependent on camera position along the horizontal axis.

cubes faces. Figure 3.1 shows the visible faces of a square. For example, the left face is only visible when the centre of the camera is less than the line marked *Min*.

$$visible(min, max, camera) = \begin{cases} positive\ visible & camera > max \\ not\ visible & min \leq camera \leq max \\ negative\ visible & camera < min \end{cases} \tag{3.1}$$

Equally important, interior faces should be culled (except in some cases where this is required!). This can be done by simple comparison with neighbouring values in the volume. Both forms of culling can be implemented using 6 bit bitmasks for face visibility with an *and* operation. Deciding which faces to draw is also simplified this way, by using a lookup table of 64 cubes, in a similar style to the marching cubes algorithm. These cubes contain a list of quads, which are the final visible quads to be rasterized.

## *3.3* OpenGL table based GVC

For comparison, we made a simple implementation of GVC using OpenGL for rasterization. This relied on the batch-update style of GVC. It uses a resizable array of Voxel IDs as a representation of the SVL (instead of a hashtable in the GVC). Then, instead of using an item buffer comprising of Voxel IDs, we use indices into the SVL array as "items". This facilitates use of scanning images pixel-by-pixel directly, adding to a table of colour consistency information. This would also be feasible using a hashtable, but is much faster in practice and more compact.

### *3.3.1 Creating item buffers*

Item buffers are created by OpenGL rasterization. For each voxel, visible faces are determined. Faces are accumulated (as vertices) in a large array, and rendered all at once using a call to glDrawArrays. This proved a reasonably efficient method of rasterizing large numbers of voxels, sufficient that the step of calculating Vis(V) and evaluating colour consistency became the largest component. Using a Geforce Fx 5200 on an Athlon 2600+, this process is roughly 3 times faster than software rendering for a mid size voxel grid (eg. $128^3$). However, as the number of voxels increase the benefit of using hardware rasterization diminishes.

### *3.3.2 Calculating Vis(V)*

Vis(V) is computed incrementally for all voxels, instead of rasterizing each voxel separately, as per GVC. The process is done in reverse, by scanning every pixel on every item buffer. A large table is allocated, with an entry for each voxel in the SVL. When a pixel is scanned, the (useful) information is accumulated in this table. For simple thresholding colour consistency tests, this usually does not require evaluating all voxels in Vis(V) at once, so some simplification could be used. For example, accumulating the average colour in all views, vs. the average colour in each view.

### *3.3.3 Updates*

Updates are performed in parallel, all inconsistent voxels removed and all newly exposed voxels to be added to the SVL are added in one iteration. A small difference from the vanilla GVC algorithm. Here two binary volumes are used, one for solid/empty state, the other in order to supplement the SVL for O(1) lookup – set if the voxel is currently on the surface. This is useful during the batch-update process, so that surface changes can be made without updating the SVL array incrementally, which otherwise would be expensive. The algorithm is listed in figure 3.2.

```
initialise Solid, Surface, SVL

do {

    compute item buffers by rasterizing voxels on SVL
    initialise table to size of SVL

    clear ToAdd, ToCarve lists

    for each item buffer IB {
        for each pixel P in IB add (IB, P) to table
    }

    for each voxel V in table {
        if not consist(table, V) add V to ToCarve list
    }

    for each voxel V in ToCarve {
        for each neighbor N of V {
            if not Surface(V) && Solid(V) {
                Surface(V) = True
                add V to ToAdd list
            }
        }

        Solid(V) = False
        Volume(V)  = False
    }

    filter ToCarve from SVL
    add ToAdd to SVL

} until (empty(ToCarve))
```

Figure 3.2: Simple OpenGL table variant of GVC

# Chapter IV

# Depth Stack Images

One approach to maintaining item buffers, is to precompute a list of all (occupied or empty) voxels intersecting a particular ray. This, computed for all pixels/rays in an input image, forms another kind of layered depth image, which can be used in an incremental voxel colouring algorithm.

A Depth Stack, is a stack of voxel IDs intersected along one ray. The back of a Depth Stack, is the closest voxel to the input image, as shown in figure 4.1. A Depth Stack Image (DSI), is an image of Depth Stacks. We use this name to avoid confusion with a LDI from GVC-LDI.

Unlike a LDI, a DS is not updated in the middle. Voxels are only pushed or popped off the back, as a stack.

## *4.1* Finding the Changed Visibility Set

In order to find the next visible voxel for one ray or pixel, voxels must be iteratively popped off a depth stack. Each voxel popped from the stack is a voxel which may potentially become visible, if it is a solid voxel (found by a

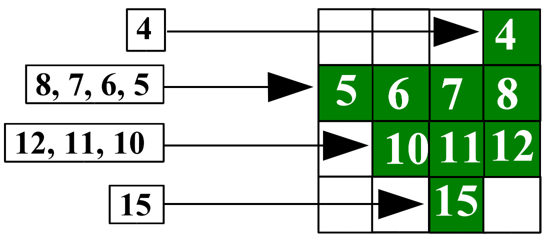

Figure 4.1: A layered depth stack image, the nearest voxel is at the back of each depth stack.

```
for each pixel P in Vis(V) {
    do {
        pop back of DSI(P)
        B = new back of DSI(P)
    } until solid(B) or (B is null)

    add B (if not null) to CVS
}

return CVS
```

Figure 4.2: Finding CVS set using Depth Stack Images

lookup to the binary voxel array), it will have changed visibility.

A simple algorithm for finding the Changed Visibility Set (CVS) of removing one voxel is figure 4.2. The CVS set is calculated as the union of voxels found to have changed visibility for each pixel/ray in Vis(V).

## *4.2* Calculating Vis(V)

The back of each Depth Stack (thus the closest voxel) in a DSI may be used as an item buffer. So Vis(V) may be efficiently calculated by finding the set of pixels in the projection of V, which have the same Voxel ID at the back of its depth stack as V.

## *4.3* Voxel colouring algorithm

A simple voxel carving algorithm in the mould of GVC-LDI, results from the ability to find precisely the set of voxels with changed visibility after a voxel carving, the CVS. However, it is simpler because each DSI is only operated on as a stack and is an otherwise static structure. In addition, there is no need to keep a Surface Voxel List (SVL) anymore, because the CVSVL may be updated without any knowledge of neighbour voxels. This simplifies the top level algorithm somewhat, as shown in figure 4.3.

```
initialise Solid, CVSVL
push voxels onto a DSI for each image

while(CVSVL is not empty) {
    remove voxel V from CVSVL
    calculate Vis(V)
    if not Consist(Vis(V)) then {
        carve V from Solid

        update DSIs to find CVS
        add CVS set to CVSVL
    }
}
```

Figure 4.3: DSI Voxel colouring algorithm

## *4.4* Pre-calculating

Pre-calculating the initial state of a DSI can be done by rasterization. Voxels may be rasterized in a furthest-first depth ordering. When a voxel V is rasterized, its ID is pushed onto the depth stack of each pixel in its projection, Proj(V).

## *4.5* Furthest first ordering

Furthest first rasterization may be done by splitting the voxel space into eighths, separated by an axis aligned plane of voxels as a pivot, this voxel plane (an interval the width of one voxel) contains the camera position for each dimension. If the camera lies outside the voxel space on any axis then the last voxel plane is used as the pivot.

Each of the eighths then has a clear back to front ordering from the pivot towards the edges (either 0, or the maximum voxel bound). Looping on each voxel region is ordered. The order of loop nesting is along the most significant axis to least significant. This is determined from the primary camera view direction vector. To simplify implementation, instead of re-ordering the loops, indices accessing the voxel array are permuted/unpermuted.

Pseudo code is listed in figure 4.4. Where cameraPos in voxel units,

```
foreach i in [0, 1, 2] {
    pivot[i] = clamp(cameraPos[i], 0, max)
}

permutation = first (sort decreasing pairs) where
pairs = zip [0, 1, 2] cameraDir

foreach d in [0, 1] {
    start[d]  = permute((not d) * max)
    end[d]    = permute(pivot.x + d, pivot.y + d, pivot.z + d)
}

foreach a in [0, 1]
for x = start[a].x to end[a].x {
    foreach b in [0, 1]
    for y = start[b].y to end[b].y {
        foreach c in [0, 1]
        for z = start[c].z to end[c].z {
            v = unpermute (x, y, z)
            rasterize v
        }
    }
}
```

Figure 4.4: Furthest first voxel ordering

cameraDir the primary view direction of the camera, start and end are a 2 element array of 3D vectors. Max is the maximum index of each dimension of the voxel array.

## *4.6* Space use

The obvious problem with this approach, is the drastic amount of memory required, which is on the order of the depth $\times$ pixels for each input image. When encoding elements of a DS as 32 bit integers, a DSI may use impractical amounts of memory. To make things more practical the voxels may be encoded using differences (using 6-connected voxels). Rasterization does not ensure this is necessarily correct, but edge or corner connected voxels can be forced into 6-connected voxels by inserting extra voxels.

When 6-connected, each voxel can be encoded with just 3 bits, which makes it nearly practical to use. Memory use is roughly equivalent to a LDI, with linked lists of 32 bit integers, for a similar scene. Both are very large. A comparison of relative memory use of techniques is given in section 8.5.

## *4.7* Problems

Aside from large memory use, the major problem with this technique is the time required to rasterize and compress the DSIs. This step required nearly two thirds of the entire execution time.

It turned out that using voxel raytracing, (as described in the next section), achieved equivalent runtimes excluding the pre-calculation step altogether. If anything, it was more expensive to unpack pre-computed 6-connected voxels on a depth stack, than to use a ray traversal algorithm.

The approach showed some promise in that it carved voxels at a much greater rate than GVC-LDI, although taking a much greater time to initialise. It achieved times of less than half, compared to our implementation of GVC-LDI (including initialisation).

## *4.8* Summary

GVC-LDI took the approach of storing and updating surface intersections down a ray. We looked at pre-calculating *every* voxel intersection (a depth stack) down rays from input images, using Depth Stack Images. The DSIs also required updating as a voxel is removed, but only those elements which are members of Vis(V), and only the back element(s) required removing – allowing the use of a static structure.

This has two major advantages over GVC-LDI. Firstly, a much simpler and direct method of finding the voxels of changed visibility (the CVS). Secondly, a much simpler data structure to update allowing for a mostly static sized data structure and O(1) updates, where as GVC-LDI requires multiple additional projections and linear updates on lists.

It has several unsolved problems however. Memory use is immense for the uncompressed data structure, storing every voxel down a ray is a large amount of data – in the order of the total number of voxels. This mandates a compressed structure (which is possible because of the coherency between voxels along a ray). Initialisation is extremely slow. Rasterizing every voxel in back to front order, to be placed in per-pixel lists ends up taking 2/3 of the total running time. It is less flexible compared to GVC-LDI when looking at the possibility of adding voxels as well as carving.

# Chapter V

# Ray Images

An alternative to pre-calculating every ray-voxel intersection, is to traverse a ray through the voxel space. In a ray traversal algorithm the exact ordering of the intersections of a ray through the voxel space is required. This requires a voxel ray tracing algorithm.

An algorithm which can be stopped and resumed is required, (otherwise rays must be traced through the entire voxel space at each iteration), while storing a minimum amount of intermediate calculations. This becomes very significant, as the number of intermediate calculations is equal to the number of pixels in all input images.

There exist several algorithms for ray-voxel traversal, which are commonly used for ray tracing, spacial subdivision and volume rendering. We chose to use the voxel traversal algorithm of Amanatides and Woo [1], for a uniform voxel grid. Alternatives exist for non uniform grids eg. an octree, for which a traversal algorithm is presented in [23].

## *5.1* Voxel traversal algorithm

The voxel traversal algorithm of Amanatides and Woo [1] computes exactly the intersections of all voxels along the path of a ray, which is precisely the information required for our purposes. It is particularly efficient, involving just 2 floating point comparisons per voxel traversed, one addition and an integer comparison.

In essence, voxel traversal consists of line drawing. The major difference of this algorithm to other line drawing algorithms, is that it is symmetrical with regard to the axes. There is no primary stepping axis, which simplifies the algorithm considerably. It consists of an initialisation step, followed by

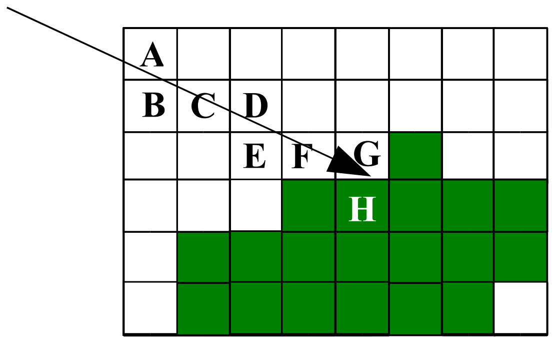

Figure 5.1: Ray voxel traversal, to find the nearest solid voxel, the ray should intersect [A, B, C, D, E, F, G] in that order, and stop at H

iteration.

Values calculated in initialisation include:

- (x, y, z) - The current voxel
- tDelta(x, y, z) - The distance (in ray units t) to step the (width/height) of one unit, for each (x, y, z) axis.
- tMax(x, y, z) - The distance (in ray units t) until the next crossing of the (x, y, z) axis.
- step(x, y, z) - The direction to step in each axis.
- justOut(x, y, z) - The value for which the voxel is just out of bounds.

The iterative part of the algorithm can be expressed most simply, as finding the axis with the least distance to travel (in ray units t), at each iteration. This is shown in figure 5.2. However using array lookups to the current axis is not as efficient as explicit if statements. It is more practically implemented as figure 5.3; and it is presented in this form in [1].

## 5.2 Incremental voxel traversal

The traversal algorithm shown above may be stopped and resumed. It requires storing 6 floating point numbers to do so, for *tDelta* and *tMax*. The

```
do {
    axis            = min tMax (x, y, z)
    tMax[axis]      = tMax[axis]    + tDelta[axis]
    current[axis]   = current[axis] + step[axis]

    if(current[axis] == justOut[axis]) return

    processVoxel (current)
}
```

Figure 5.2: Simplified ray voxel traversal

```
do {
    if(tMax.x < tMax.y) {
        if(tMax.x < tMax.z) {
            tMax.x += tDelta.x
            x = x + step.x
            if(x == justOut.x) return
        } else {
            tMax.z += tDelta.z
            z = z + step.z
            if(z == justOut.z) return
        }
    } else {
        if(tMax.y < tMax.z) {
            tMax.y += tDelta.y
            y = y + step.y
            if(y == justOut.y) return
        } else {
            tMax.z += tDelta.z
            z = z + step.z
            if(z == justOut.z) return
        }
    }

    processVoxel (x, y, z)
}
```

Figure 5.3: Ray voxel traversal [1]

```
struct RayStepper {

    float mx, our, mz;   //tMax
    float dx, dy, dz;   //tDelta

    unsigned short x, y, z;
    unsigned short dirMask;  //Bitmask for sign of dx, dy, dz
};
```

Figure 5.4: Ray stepping structure

current voxel is represented by 3 integers, and step. *justOut* may be precomputed and looked up by the sign of *tDelta*.

The C++ data structure used for storing this is shown in figure 5.4. Given space useage of 4 byte floats, and 2 bytes per unsigned short, this results in a 32 byte structure. This structure consumes a rather large amount of memory when stored for each pixel on each input image. But none the less, it is similar to the use of a DSI or LDI. A comparison of representation is given in chapter 8.

## 5.3 Ray generation

We use ray generation typical to ray tracing. The pinhole camera model is used. Rays may be generated by taking an inverse-projection for a given pixel and depth. A slightly better approach, involves taking the finding the 4 corners of the image plane with inverse-projection and interpolating to find rays inbetween.

In equation 5.1, $P$, is the combined OpenGL projection matrix, extended with a 4th row to form the usual 3x4 projection matrix, used for calibration. This is done in order to facilitate display (depth buffering). It makes $P$ invertable, which is useful for ray generation or reverse projection. $X$ is a $4\times 1$ homogeneous vector representing a scene point. $x$ is a $4\times 1$ homogeneous vector of a projected point where its $x, y$ components are image pixels and the component $z$ is a normalised depth used for depth buffering.

$$PX = x \tag{5.1}$$

```
initialise CVS(V) to the empty set

for each pixel P in Vis(V) {
    do {
        step ray (P)
        R = current ray (P)
    } until solid(R) or (R is null)

    add R (if not null) to CVS(V)
}
```

Figure 5.5: Finding CVS set using ray traversal

$$unproj(x) = X = P^{-1}x \tag{5.2}$$

$$ray(x) = (camera, unproj(x)) \tag{5.3}$$

## *5.4* Ray initialisation

Rays are generated, then intersected with the bounding box of the voxel space. This gives an initial voxel. The ray stepper is then initialised. The combined ray generation, and ray initialisation, takes about a tenth of the total time of a the Ray Images algorithm. If starting on a more complicated, or half-carved voxel model, then tracing each ray is required in addition. This process can be quite slow, illustrated by figure 9.13 where we apply these algorithms to level of detail, requiring re-initialisation multiple times.

## *5.5* Finding the Changed Visibility Set

The changed visibility set, is found by an equivalent method to the DSI method. The CVS, is the set of solid voxels found from traversing each ray in Vis(V), as shown in figure 5.5.

## *5.6* Calculating Vis(V)

A ray image may also be used as an item buffer, with one notable difference. The pixels of a rasterized voxel will not match the pixels in a ray image perfectly, because the ray traversal process does not follow fill conventions used by rasterization. To get around this, we use an over approximation – a bounding box around a cube's projection. This results in scanning a few extra pixels, which ends up being more or less identical in terms of efficiency, compared to using an optimised rasterization routine.

## *5.7* Voxel colouring algorithm

We use an identical voxel colouring algorithm to that used with a DSI, as shown in 4.3, replacing DSI with ray image in each instance.

## *5.8* Comparing traversal methods

A parallel (stepping all rays at once) front to back traversal of a voxel volume (from an arbitrary viewpoint) is often faster than an equivalent operation using the (entirely precomputed) Depth Stack Image.

We have performed some small trials, using two implementations (compressed, uncompressed - using a C++ std::stack) of DSI, as well as ray traversal [1], shown in figure 5.3. The test is to traverse a voxel space entirely from the viewpoint of one image, with rays generated for each pixel. For a DSI this involves popping one element off every stack (unless empty) in a loop. Ray traversal steps every ray once, in a loop. Results are presented in chapter 8.

## *5.9* Summary

While the Depth Stack Image approach pre-calculates and stores every intersection down a ray, the Ray Image approach calculates each ray intersection on the fly, using a ray-voxel traversal algorithm. This requires storing intermediate stepping variables with each input image pixel, instead of popping elements off the back of a Depth Stack. The ray traversal algorithm is resumed and stopped again.

This has the same advantages as the DSI, but negates many of the difficulties. The main issue, the initialisation period is solved. It's initialisation involves intersecting rays with the starting voxel volume, this is somewhat slower than GVC and GVC-LDI, however it is not significant. The memory use is not significantly reduced from using compressed DSIs. It is a little more flexible with regard to adding voxels. A voxel may be directly tested for ray intersection, compared to its (approximate) projection, however such avenues have not been examined closely.

# Chapter VI

# Incremental Voxel Statistics

The Ray Image algorithm (figure 4.3), has been applied to optimisation based colour consistency. Colour consistency relies on calculating certain statistics about each voxel. These statistics are often expensive to calculate via rasterization. The key point of this method, is that these voxel statistics can often be updated incrementally as the reconstruction proceeds. This builds on the spirit of GVC-LDI of maintaining incremental visibility information.

The methods considered in this section presented some difficulties, and were not very successful in themselves. As a result, the more general Ray Buckets approach (discussed in chapter 7), was developed with these ideas in mind, and is much more successful for the tasks discussed in this chapter.

It was found that a useful way of computing statistics required for colour consistency, (and ordering the carving queue), was to perform incremental updates. Some way of finding a voxel's colour quickly was required. It turns out that the voxel colour can be easily calculated incrementally, by adding colour as new rays are found to be visible.

We use a hashtable, mapping voxels to their statistics, to store this new information. The hashtable is termed the Visible Voxel List (VVL). The VVL is a subset of SVL. It is possible to have voxels on the surface (in the SVL) which are not visible to any view (thus are not in the VVL).

When a ray is traced to a solid voxel using figure 5.5, the colour and visibility count is accumulated and stored in the VVL.

## *6.1* Tentative carving

In order to evaluate how the carving of a voxel will change the global and local reprojection error, we use a tentative carving procedure. This involves creating a set of candidates, along with their changed visibility statistics, (these are the same voxels as the CVS). The reason for the term candidate, is that once a voxel is found inconsistent, the candidates are confirmed, and used to update old visibility information. This procedure differs from tentative carving as described in 2.9 in that they focus on the ability to undo a change rather than computing the change and deciding whether to apply it, as described here.

In order to perform a tentative carving procedure, first a set of candidate voxels is computed by ray traversal for each ray in Vis(V). These candidates are the voxels with changed visibility (the CVS). Each ray is stepped until it meets a solid voxel. This voxel will be in the CVS, and is recorded as a candidate.

Rays are then placed in bins for each candidate. The bins are implemented as lists, and a hashtable maps candidates to rays. A fixed array with linear search is a more practical alternative to a hashtable for small numbers. The candidates inherit colour/count statistics looked up from the VVL. Together with the new set of rays a new voxel colour statistics are calculated.

In order to apply optimisation colour consistency, it needs to be determined if the carving is an improvement using equation 2.9. The information required is the colour of each CVS voxel U, and its visible projection Vis(U), before and after carving.

The colour of the CVS voxels are round from the VVL and candidates. Identifying the visible projection of each of these voxels involves projecting and filtering with each item buffer as usual.

## *6.2* Computing change in reprojection error

The procedure is listed below for the simplest case of evaluating the change in reprojection error. The reprojection error before and after carving, is calculated using the previous and updated colour of each candidate voxel

```
initialise Solid, VVL, CVSVL, Ray Images

while(CVSVL is not empty) {
    remove voxel V from CVSVL
    calculate Vis(V)

    for each pixel P in Vis(V) {
        U = next Solid voxel on Ray(P)
        add (P if not null) to Candidates(U)
    }

    for each voxel U in Candidates {
        add Proj(U) to CVRS(V)
        calculate before/after Colours(U)
    }

    if not consistent (V, Colours, CVRS(V))
        carve V from Solid
        update or add Candidates to VVL
        update or add Candidates to to CVSVL
    }
}
```

Figure 6.1: Optimisation colour consistency using ray traversal

(and the voxel being carved). A dash represents the property after carving (eg. $Vis'(U)$ will replace Vis(U) if V is carved). $C_r$ is the colour of ray r.

$$Vis'(U) = Vis(U) \cup Rays(Candidate(U))$$

(6.1) Updated Vis(U) set.

$$Colour(U) = \frac{\sum_{r \in Vis(U)} C_r}{|Vis(U)|}$$

(6.2) Colour before carving.

$$Colour'(U) = \frac{\sum_{r \in Vis'(U)} C_r}{|Vis'(U)|}$$

(6.3) Colour after carving.

$$E(U) = \sum_{r \in Vis(U)} dist(C, Colour(U)$$

(6.4) Reprojection error before carving.

$$E'(U) = \sum_{r \in Vis'(U)} dist(C, Colour'(U)$$

(6.5) Reprojection error after carving.

$$FG(V) = \bigcup_{U \in Vis(V)} Vis(U)$$

(6.6) Set of foreground voxels.

$$change(V) = \sum_{U \in FG(V)} E'(U) - (E(V) + \sum_{U \in FG(V)} E(U))$$

(6.7) Change in reprojection error.

$$consist(V) = change(V) \leq 0$$

(6.8) Colour consistency function.

A similar approach can be taken for the probabilistic approach given in [15]. The probability associated with a voxel occupying a pixel is transformed by the negative log, which is summed and compared in the same way.

### *6.2.1 Problems*

The motivation for using ray traversal was that it would apply nicely to Optimisation based colour consistency measures. It was thought that once the CVS was efficiently obtained, that an algorithm to utilise Optimisation based colour consistency would be easily found.

It proved slightly more difficult and less practical than first thought. The difficulty lies in the fact that one must end up calculating the projections of each voxel in the Candidates anyway. The ability to make use of the incremental statistics is therefore negated. A key advantage over GVC-LDI is avoiding excessive rasterization. However, this advantage is lost.

In order to address these issues, another approach was taken, as described in section 7, where Vis(V) is updated incrementally, and the use of item buffers and rasterization are avoided entirely.

## *6.3* Carving order

Using a monotonic colour consistency function, carving does not matter. Using a non monotonic carving order, the order matters makes a difference, a conservative carving is not guaranteed. Using an ordering from low probability to high probability, was discussed in [15], but made no mention of how

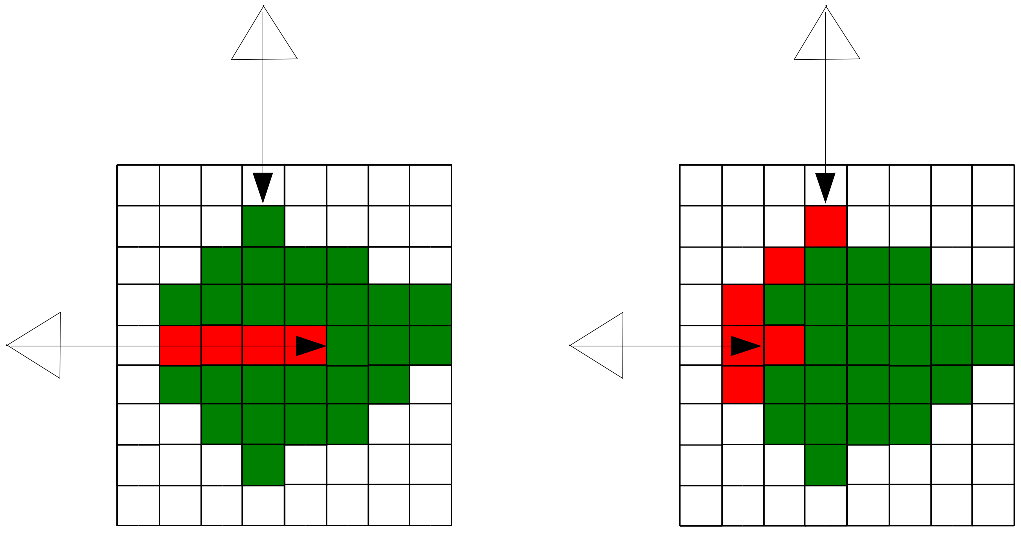

Figure 6.2: Least visible first, most visible first carving order.

it influenced results, or how it was implemented (using GVC-LDI).

Take an extreme case, shown in figure 6.2, where two examples are shown. The orderings are analogous to depth first vs. breadth first search. Assuming voxels are never colour consistent, using a least visible first ordering burrows a one voxel wide hole through the centre of the volume. This happens because the newly exposed voxels almost always also have the least visibility.

The problems here are obvious, the information from the second camera is never used, as the voxels in the tunnel are never visible. In the second, voxels on the surface and closest to the two cameras are carved first, which results in much better conditioning as voxels are only examined when the maximum information is used.

Although least visible first is an absurd ordering, a more natural ordering such as the GVC ordering may also result in a sub par ordering. Extreme cases may be avoided by applying some kind of minimum visibility for consistency checks. Some examples of different carving orders are shown in figure 6.3. A hashtable ordering results from using a hashtable for the CVSVL, where the hashtable is iterated removing the previous element each time, giving a fairly arbitrary ordering but occasionally showing an uneven pat-

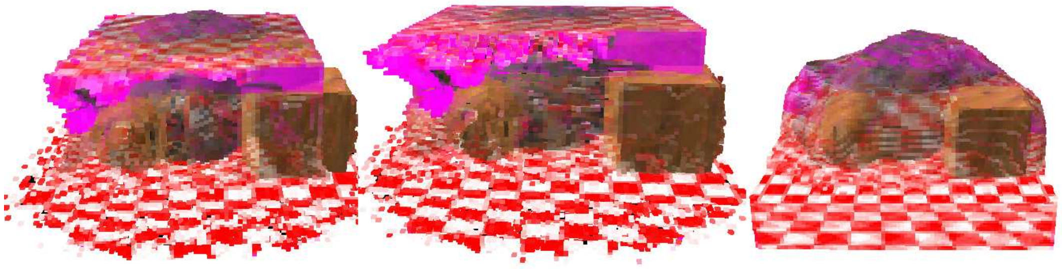

Figure 6.3: Three different carving orders: GVC order, hashtable order and most visible first order.

tern.

Practical results using a most–visible first ordering are compared in chapter 8. It can be seen that using a most-visible first order, uses a much larger Vis(V) on average, than the GVC ordering, or an arbitrary ordering (as with Ray Images).

### *6.3.1 Carving queues*

An advantage of incremental methods is that a priority queue, rather than a systematic or randomised carving order may be used. To achieve this a queue is needed which allows elements to be indexed by key/voxel, and updated in the queue. Additionally, we need to be able to remove the first element in the queue, and link it back to voxel statistics.

To do this there are two ways which seem acceptable. The first uses a stable tree for the CVSL priority queue, specifically a C++ (STL) multimap, where elements are ordered by some property of the voxel (in this case we use most-visible first). The elements in the map contain the voxel ID. A hashtable stores visible voxels, containing the visibility count, the accumulated colour and also an iterator into the CVSVL priority queue.

The first element in the multimap can be removed to find the highest priority voxel. Queue items may be updated by looking up the visible voxel hashtable, then using the iterator to manipulate the queue.

The second way, is to use a heap where heap elements are doubly linked with the visible voxel hashtable elements. The hashtable is setup as before, only each element now has an additional index into the heap array (which may be invalid/null, as the CVSVL is a subset of the VVL set). Each heap element contains the priority, and an iterator into the hashtable. A slightly annoying issue is that when implemented as an array, the heap is manipulations (eg. during a sift up or sift down), move making it hard to keep an index.

The disadvantage of the second way is that it accesses memory which is sparsely distributed. Whenever a sift up or sift down operation is carried out on the heap, many hashtable items are updated. It seems to work well in practice, being a little faster than the tree approach using standard C++ containers. It is significantly faster on updates which occur much more frequently than insertions and deletions. For each voxel carved, several will be updated, one deleted and few inserted. Time complexity is the same for both methods, on average $O(logn)$ for inserts deletions and updates.

## *6.4* Summary

We have looked at an approach to apply the Ray Images algorithm to optimisation colour consistency. We associated voxels with statistics incrementally gathered. This introduced the Visible Voxel List structure as a hashtable mapping voxel IDs to voxel statistics. A tentative carving approach was taken whereby a set of candidate voxel statistics were first computed, colour consistency evaluated using those statistics, then if found inconsistent the candidates are committed.

These methods worked well for applying an ordering to the carving process (as discussed in section 6.3), but a deficiency in the Ray Image approach was found leading to another approach, discussed in chapter 7.

We have looked at keeping incremental statistics on voxels. These statistics can then be used to order the carving process. We discussed briefly the potential problems associated with carving voxels in a poor ordering – though results discussed in section 8.6 tend to show that it does not matter a great deal with regard to quality in practice, but perhaps more in algorithmic complexity.

Two methods of keeping implementing carving priority queues are given, using either a stable tree or a binary heap. In either case the structure is doubly linked with the VVL hashtable. Both methods do not add significant overhead to the algorithm, but the binary heap is a little faster with regard to updating voxel visibility.

# Chapter VII

# Ray Buckets

In (section 6.1), the approach of updating voxel statistics incrementally was introduced. Previously ray stepping information has been stored in an image, to use as an item buffer, in conjunction with rasterization. A natural generalisation is to avoid the use of images and rasterization entirely, and use incremental updates for both visibility information/Vis(V) as well as voxel statistics (colour, reprojection error etc.).

The common factor required for colour consistency tests is Vis(V). Generalised Voxel Colouring [11] uses rasterization to determine Proj(V). Vis(V) contains each element Proj(V) equal to its corresponding entry in an item buffer. Instead of looking to evaluate Vis(V) efficiently, we look at an approach where Vis(V) is *updated* incrementally for all visible voxels.

This turns the VVL set (a hashtable), into the primary data structure. Voxels of this hashtable set contain containers termed buckets - due to the nature of their use. Buckets are filled as rays pass through and become visible. When the voxel is carved the bucket is tipped out and the rays pass through to buckets further down.

Voxel ray traversal is used to update the VVL structure. Each ray stored (each member of vis(V)) also stores ray stepping information. When a voxel V, is carved, rays in Vis(V) are accumulated to other voxels which are, or will become, members of the VVL. These voxels are the CVS(V), found by ray traversal.

A queue is used to maintain the voxels which require evaluation. Only those which change visibility during a carving operation require re-evaluation. This is the same role as Changed Visible Surface Voxel List (CVSVL) from GVC-LDI [11]. The data structures used are outlined in figure 7.1.

The VVL is implemented as a hashtable, mapping Voxel IDs to a structure

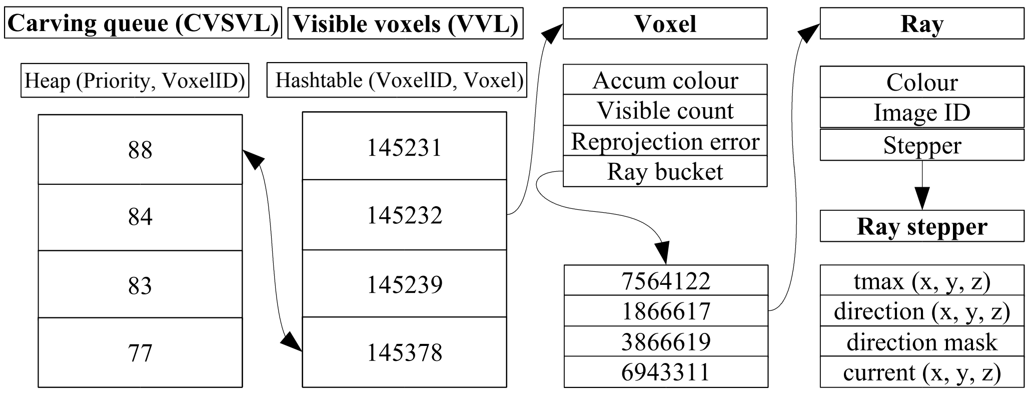

Figure 7.1: The major data structures and their connections.

containing Vis(V). We use the term *bucket* for this structure, as it behaves as water poured into a bucket. When the voxel is carved, the water is tipped out and "falls" into other buckets. The bucket may also be used to update useful statistics, for example current reprojection error of a voxel.

One further step is taken, in that ray stepper information is also merged with colour information into one record. This is convenient from an implementation point of view, as they are both likely to be used together. It also cuts down on memory, as background rays will no longer be stored at all. This helps with segmented images and using a small region of interest as a large portion of unneeded information may be discarded.

## *7.1* Accumulation

As an entirely carving process is taken, then visibility is never reduced for a voxel, only increased. This means updates can be performed by accumulating rays to voxel buckets. When a voxel is carved, several rays pass through to solid voxels further on.

One implementation decision which must be made is how voxel buckets and rays are represented. The most straightforward approach is to use a structure for rays (copying by value), and voxel buckets are simply an array or list of these rays. The other potential approach is to use an indirect pointer into a giant table of rays. The first approach is simpler, but using an indirect pointer seems to work slightly better. This is perhaps due to lessening the large quantity of data being copied.

We use a simple C++ vector to implement ray buckets. New rays are grouped and added all at once (like candidates as listed in figure 6.1). Buckets are efficiently copied by using swaps rather than copies to conserve allocations. Given the large amount of memory being moved, this is particularly important. Other alternatives were tried, including linked lists and linked lists of vectors.

## *7.2* Initialisation

Initialisation is performed by accumulating all generating rays for input images, and accumulated into an empty VVL hashtable. In practice, this occurs by generating one image of rays (as per the Ray Image algorithm) at a time, pairing it with an input image and extracting each ray, accumulating it to the VVL hashtable. This is done image by image, so that only a small part of the dataset needs to be loaded twice at any one time.

Effectively the input to the algorithm is no longer a series of images, but a large set of rays (associated with a colour/other statistics). This could be useful in providing a level of detail approach, where only the most important rays are added. In its most basic form this means that background rays can be immediately culled. Potentially different sampling could be used dependent on image data. For example it might be useful to sample more densely around object contours.

Capturing input data becomes more flexible. For example, instead of 15 images with $800 \times 600$, it could become feasible to use 60 images with resolution at $400 \times 300$. This would still use approximately the same number of rays (thus using approximately the same resources), but cover a much more comprehensive set of view angles. Given that voxel colouring techniques work most successfully on occluding contours, it may be a promising approach. We made a comparison using some generated images of the teapot scene. Results are shown in section 9.4.

## *7.3* Voxel colouring algorithm

A voxel colouring algorithm using ray buckets is shown in figure 7.2. It operates in much the same way as shown in figure 6.1, however the differences are in the detail, ie. initialisation, finding Vis(V) and data structures.

One point is that the rays are stepped before evaluating colour consistency. This is only required for evaluating an optimisation based colour consistency test. Otherwise, it is more efficient to evaluate consistency first.

One of the issues mentioned in section 6.1, was that despite being able to easily calculate the CVS set, in order to properly evaluate a change in reprojection error, one needed to project each voxel in the CVS again. The advantage of ray buckets approach, is that each voxel maintains its visibility set, so that it does not need to be recalculated to evaluate voxel colour consistency. This means that calculation of the voxel's projection is shared between evaluations, which allows optimisation colour consistency to operate in a much more efficient manner.

## *7.4* Optimisation colour consistency

We have applied the Bucket algorithm to optimisation voxel colouring as [29, 15]. We have looked at a purely carving method, as opposed to [29], which involved a process of adding and removing voxels. Adding voxels would complicate the bucket algorithm because it would involve removing rays from existing buckets, as well as forcing a mapping from images back to rays. This is one area where GVC-LDI is more flexible.

From each candidate all the required information is available for implementing an optimisation based consistency function. The old visibility information is looked up from the VVL, and the new visibility information is stored with each candidate.

The Bucket approach substantially improves on the Ray Images approach, as the visible projections are pre-calculated when the projections of each voxel in the CVS is required. An extension of this is possible. The reprojection error can be incrementally calculated. It might also be useful to use the metric as a priority for ordering. Kim and Kweon [15] used an ordering from

```
initialise Solid

Generate rays from input images
For each ray R {
    Find first solid voxel V
    Acumulate R to VVL(V)
}

Set CVSVL to voxels in VVL

while(CVSVL is not empty) {
    remove voxel V from CVSVL
    lookup Vis(V) from VVL

    for ray R in Vis(V) {
        step R
        U = next Solid voxel on R
        if not background(U)
            add R Candidates(U)
    }

    if(not Consist(Vis(V), Candidates) {
        carve V from Solid
        remove V from VVL

        for each Voxel U in Candidates {
            add rays in Candidates(U) to VVL(U)
        }
    }
}
```

Figure 7.2: Ray Buckets algorithm

high probability to low probability.

## 7.5 Summary

We have looked at an alternative to Ray Images by throwing out input images and item buffers entirely, and relying on a purely incremental approach where all ray statistics including input colour, and stepping information are associated with rays. Rays are associated directly with voxels in the Visible Voxel List hashtable. Rays are "tipped" from one bucket as a voxel is carved, to end up being accumulated in another, eventually residing in either a colour consistent voxel or reaching the background.

We have discussed a voxel colouring algorithm and implementations for thresholds, and optimisation colour consistency. The bucket approach solves the earlier problems of Ray Images, by having the Vis(V) set for any voxel available for constant time lookup (by the VVL hashtable).

The bucket approach is less flexible than GVC-LDI with regard to adding voxels. Another mechanism associating image pixels to rays would be needed for adding voxels. It is much more flexible than GVC-LDI in the representation of input images, (images are transformed into a set of independent rays). This also may be a drawback, as noted it becomes harder to find the source of a ray. For example, a ray may need to store the "image number". A use could be level of detail in input images.

# Chapter VIII

# Comparison

We have compared several different aspects of the proposed approaches. We compare runtime statistics for threshold colour consistency with implementations of all proposed algorithms and variations, as well as controls consisting of GVC and OpenGL accelerated GVC, for threshold colour consistency.

We make use of four datasets, a set of rasterized images "Teapot", calibrated photo sets Cactus, Gargoyle, and "Violet" courtesy of Pr. Kyros Kutulakos (University of Toronto). We use one system for most comparisons, an Athlon 2600+, with 768MB of RAM and a Geforce FX 5200 graphics card. For the gargoyle dataset with threshold colour consistency, another system was used with more memory to allow GVC-LDI to run.

We have compared our algorithms to several others (at a common level) using the threshold colour consistency test given in equation 2.5. We compare time vs. reprojection error, as well as runtime statistics of memory use. The implementations compared are GVC, GVC-LDI, a version of optimised GVC using OpenGL for rasterization (GVC table), and earlier work using ray traversal with item buffers (Ray–image). We have endeavoured to use optimised versions of each algorithm, most sharing significant common code for rasterization and volume representation. We have recorded statistics on runtime behaviour for each algorithm where possible. (GVC-LDI exhausted memory for some data sets).

Reprojection error is a problematic measure here and can't be compared between methods adequately, given that it always favors a reconstruction with a fewer number of rays. Reprojection error per ray suffers similarly, in that a model comprising of only a tiny part of the original scene may still have a good reprojection error for the parts which do exist. A better measure

| Method | Memory use (MB) |
|---|---|
| Images alone | 54.5 |
| GVC table OpenGL | 70.7 |
| GVC | 87.6 |
| Bucket | 225.0 |
| Ray–image | 311.8 |
| DSI | 316.4 |
| GVC-LDI | 538.3 |

Table 8.1: Memory use for teapot scene.

would have been to prepare reference voxel models and compare the volume error.

## *8.1* Teapot scene

The first dataset consists of a teapot scene with 15 800 $\times$ 600 generated images (by rasterization). The reconstructed resolution of the teapot scene was at $120^3$.

All six methods successfully reconstruct the teapot to a similar quality shown in figure 8.2. There are minor differences due to carving order. Vis(V) differs between rasterization, OpenGL rasterization, and ray traversal. Figure 8.1 shows that the GVC table method is faster than the others (though converges slowly), followed closely by the Bucket method, GVC, DSI with an extremely long initialisation period, and GVC-LDI significantly slower.

To show the differences resulting from ordering, we have shown two carving orders for the Bucket carver. The reason for the hashtable ordering (Also used by implementations of GVC-LDI and Ray–image) being faster, is simply that it creates more surface noise and uses less rays per voxel. This results in smaller Vis(V), with a larger ray traversal at each iteration (ie. each ray skips over more voxels at a time, so is involved in less colour consistency checks).

Of the two classes of algorithm, it can be seen that incremental methods {GVC-LDI, Ray–image, Bucket} carve at a nearly constant rate, but GVC based methods trail off rapidly towards convergence.

Maximum memory use (table 8.1) was significantly higher for incremental

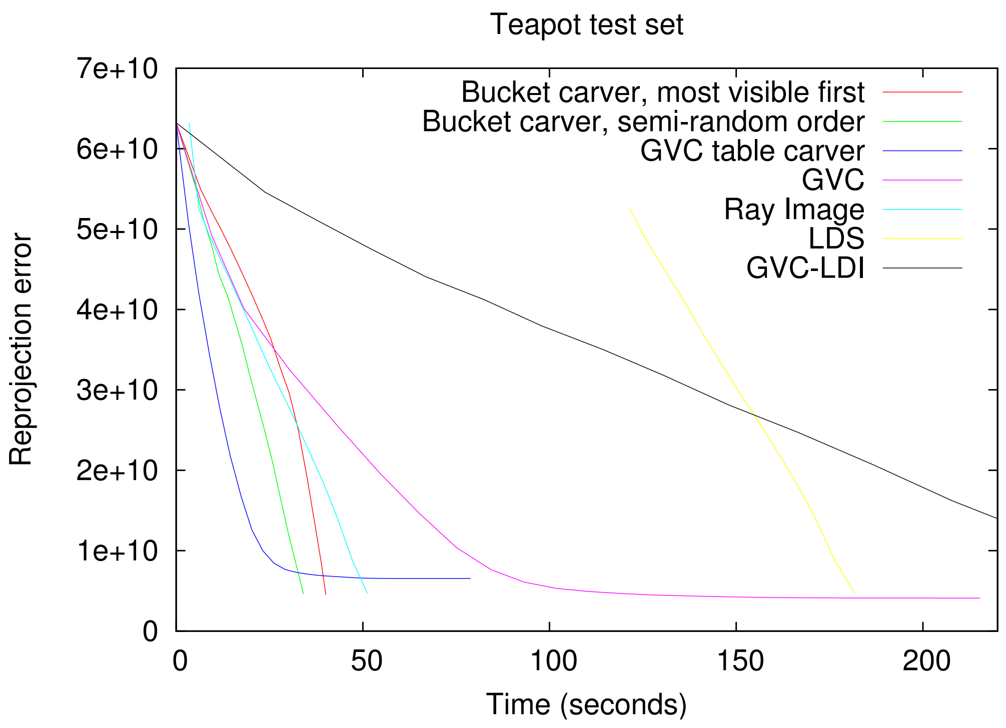

Figure 8.1: Reprojection vs. time for several algorithms

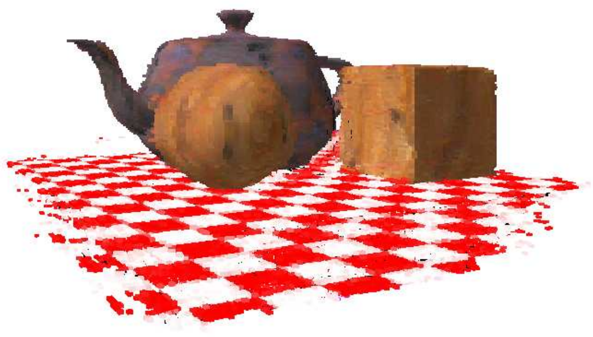

Figure 8.2: Reconstructed teapot scene.

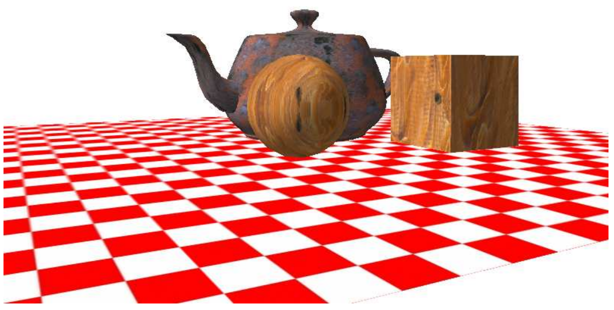

Figure 8.3: Rasterized image of teapot scene.

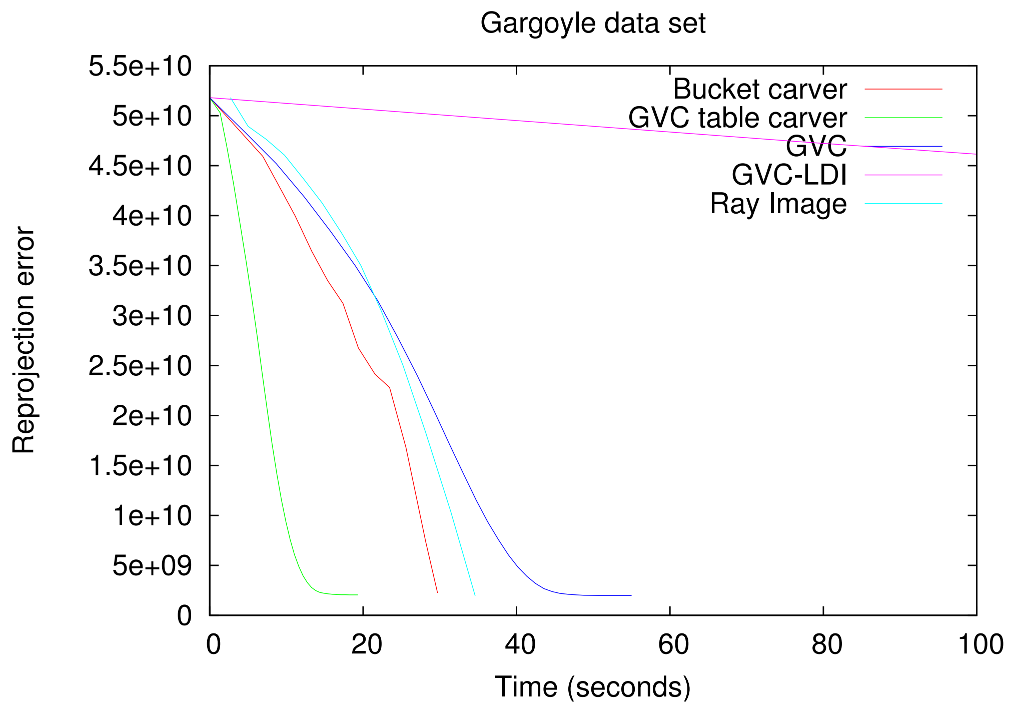

Figure 8.4: Reprojection vs. time for several algorithms

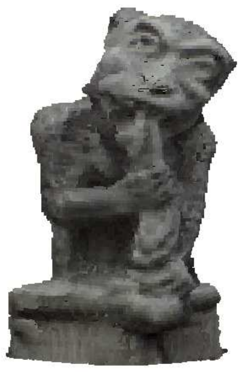

Figure 8.5: Reconstructed gargoyle scene.

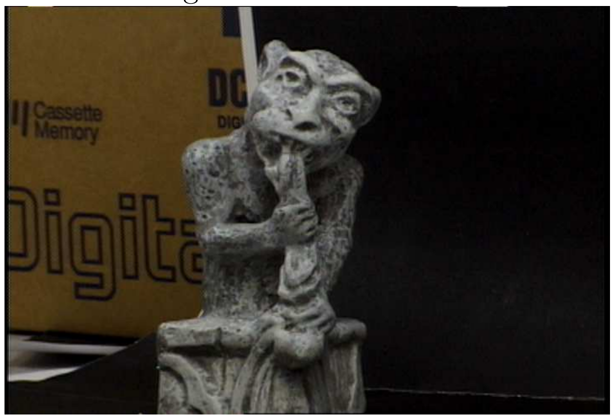

Figure 8.6: Photograph of gargoyle scene.

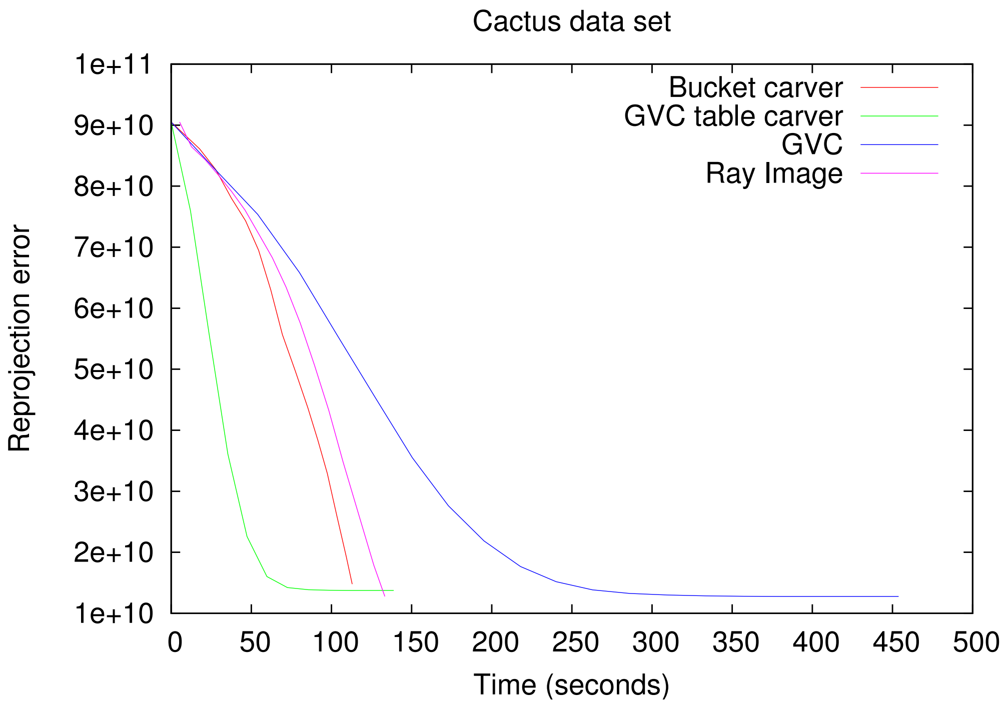

Figure 8.7: Reprojection vs. time for several algorithms

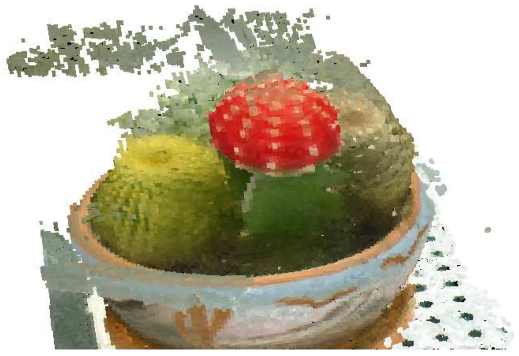

Figure 8.8: Reconstructed cactus scene.

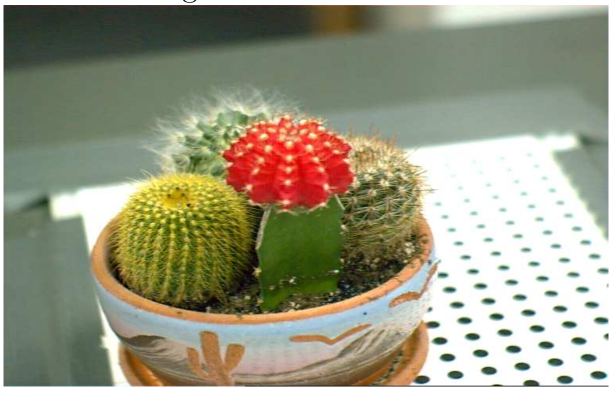

Figure 8.9: Photograph of cactus scene.

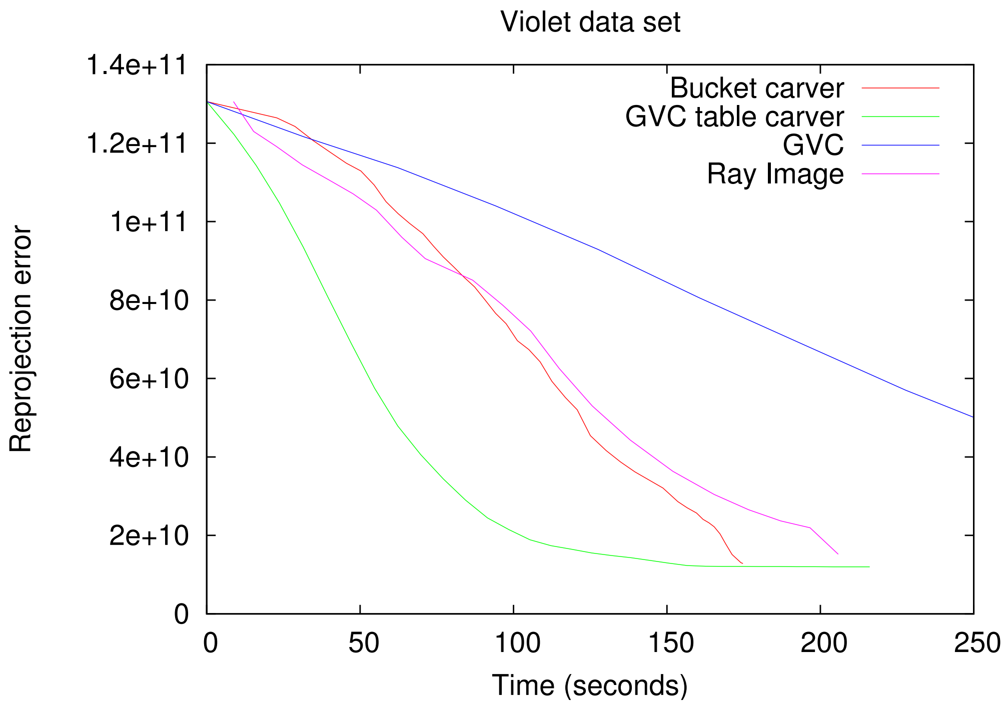

Figure 8.10: Reprojection vs. time for several algorithms

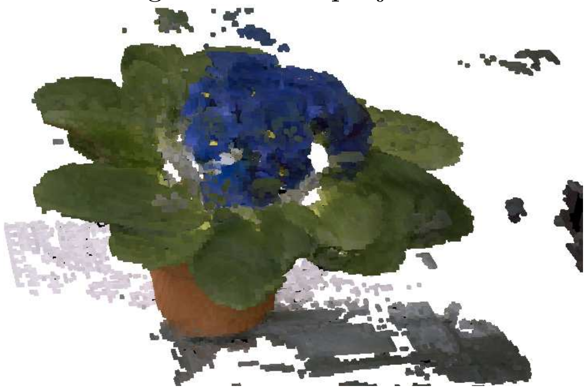

Figure 8.11: Reconstructed violet scene.

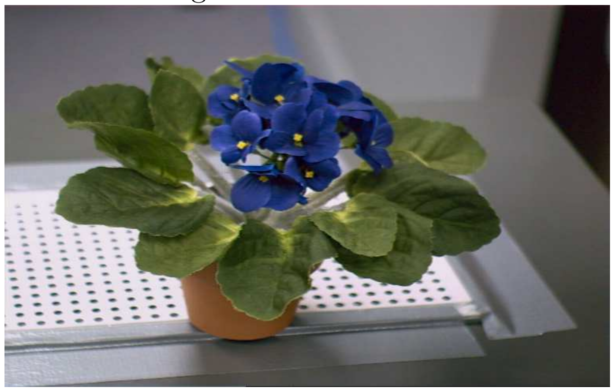

Figure 8.12: Photograph of violet scene.

methods. GVC variations used insignificant amounts of memory in comparison, using barely more than the memory taken up by the input images, the voxel volume and other constant costs. Memory use for the incremental methods is dominated by an order of the number of rays, which diminishes as carving proceeds. GVC-LDI memory use increases as time progresses, if surface noise increases. A surface with more noise creates more surface intersections which increases the list size per pixel. This could be resolved by using a priority queue for ordering.

## *8.2* Gargoyle scene

The gargoyle dataset consists of 16 $719 \times 485$ photographs. The reconstructed resolution of the gargoyle scene was lower than others at $80^3$, in an attempt to reduce memory use, to compare more algorithms.

This data set was run with different parameters in attempt to run the GVC-LDI program without running into memory swapping issues. The test system was a Pentium 4 2.8GHz, with 1GB of memory and a Geforce Fx 5200 graphics card. It was noted that despite these measures, GVC-LDI was using close to 95 % of system memory. It is probable that performance is severely degraded by memory swapping.

Statistics from the gargoyle scene show a similar general pattern to the teapot scene. However the incremental methods are all relatively slower compared to the GVC methods than for the teapot scene. The OpenGL GVC program ran much faster than any of the others, and GVC-LDI was exceptionally slow.

One reason for GVC doing well is a simplified surface, as the gargoyle has a relatively convex shape (with small concavities). The serial algorithms (Ray Buckets, and GVC-LDI) will be relatively faster when dealing with shapes with uneven surfaces and deep concavities, as they will be processed in a serial manner. For each voxel-step into a concavity, GVC will also re-evaluate all the other voxels. This is further discussed in section 8.6.

The reconstruction quality of the gargoyle is close to the visual hull, as the background varies greatly in contrast to the gargoyle. The gargoyle itself, does not vary much in colour. The threshold colour consistency test does not distinguish some finer points, for example the pattern on the base seen in figure 8.6 is not captured in the reprojection 8.5.

## *8.3* Cactus scene

The cactus dataset consists of 30 $768 \times 484$ photographs. The reconstructed resolution of the cactus scene was at $120^3$.

The cactus scene showed typical runtime statistics (figure 8.7), which are very similar to the gargoyle. The GVC-LDI program was not used, as it used more memory than we had available in test machines (more than 1GB). While others used a lot of memory (more than 500MB), they were not affected by swapping issues. The OpenGL GVC was fastest. Both ray traversal methods were similar, although a little slower than the OpenGL GVC.

The cactus scene is a tricky example for voxel colouring algorithms. Strictly speaking it contains mostly diffuse reflectors and no transparency, the fine needles of the cactuses provide a similar effect to transparency and result in some surface noisy surfaces in the reconstruction. The blurry grey background also results in some noisy voxels in the reconstructions. Despite these issues the plants are all very colourful which gives a good silhouette for each interior plant, though fairly noisy.

## *8.4* Violet scene

The violet dataset consists of 40 $768 \times 484$ photographs. The reconstructed resolution of the violet scene was at $120^3$.

The violet scene showed the same general pattern (figure 8.7). The GVC-LDI implementation was again not used as it used more memory than we had available in test machines. The ray traversal methods also used a large amount of memory, and were using close to 85 % of the 768MB in use, and slow to start as a result (swapping was occurring). The OpenGL GVC was fastest again, though with a smaller margin, it took some time to converge. While both ray traversal methods were similar, they show a different curve due to the different carving order used.

| Method | Memory used (MB) | Init time (s) | Traversal time (s) |
|---|---|---|---|
| Ray Image | 16.1 | 0.18 | 3.6 |
| DSI (Compressed) | 18.4 | 9.2 | 5.9 |
| DSI | 165 | 12.1 | 7.6 |
| Ray Image | 16.1 | 0.20 | 4.2 |
| DSI (Compressed) | 21.5 | 11.1 | 7.3 |
| DSI | 227 | 11.6 | 8.4 |

Table 8.2: Runtime statistics for different ray traversal methods and structures.

The violet scene contains many thin leaves with nearly constant colour. As a result the threshold consistency function failed to reconstruct these very adequately - as can be seen in figure 8.11. The light green texture is not shown on the leaves, and the geometry of the leaves consist of thick slabs.

## *8.5* Ray traversal

Here we compare the difference between pre-computing ray traversals with Depth Stack Images vs. using ray traversal algorithms, shown in table 8.2. The first set of data is from head-on to a voxel volume. The second is a view from a corner. A ray which travels diagonally will pass through more voxels on average, than a ray aligned with the voxel axes. The traversal times are recorded by stepping one layer at a time through the volume.

It is fairly conclusive to say that the Ray Image approach is superior to the DSI approach, given that it is better in memory use, initialisation time and traversal time. It is interesting to see the compressed DSI traversal takes less time, presumably because it involves accessing a much smaller memory size. Probably for the same reason the Ray Image approach is faster than either DSI lookup traversals.

## *8.6* Discussion

Trends, interesting statistics and complexity of the various algorithms is informally discussed here. The time complexity for algorithms examined, is dominated largely by the number of voxels (evaluation loop) $\times$ pixels per

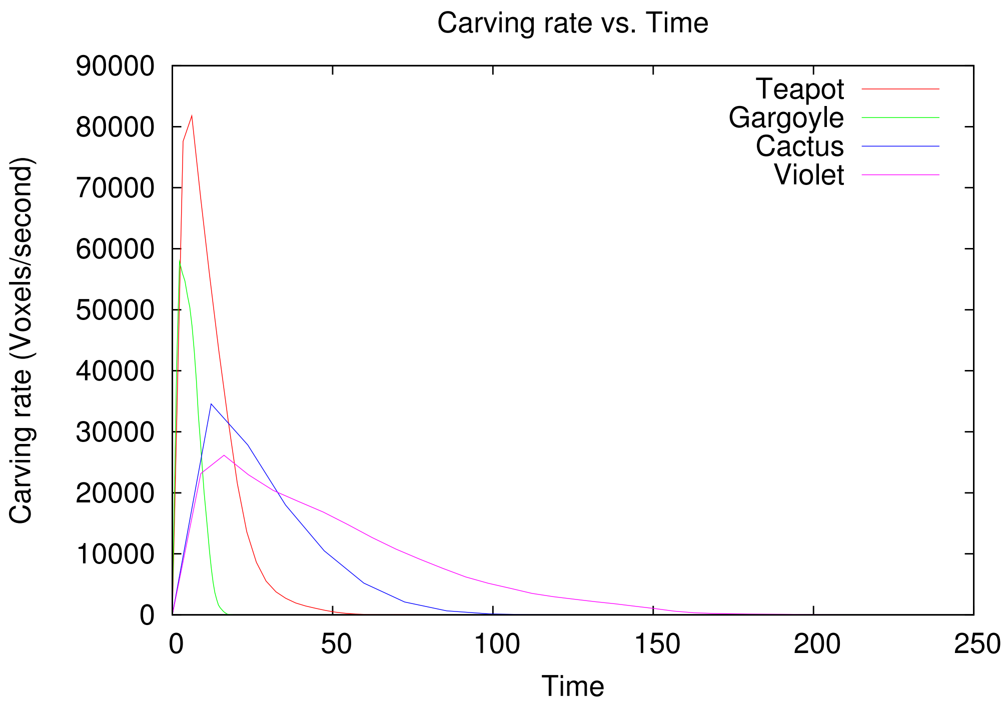

Figure 8.13: Carving rate vs. time for GVC (OpenGL)

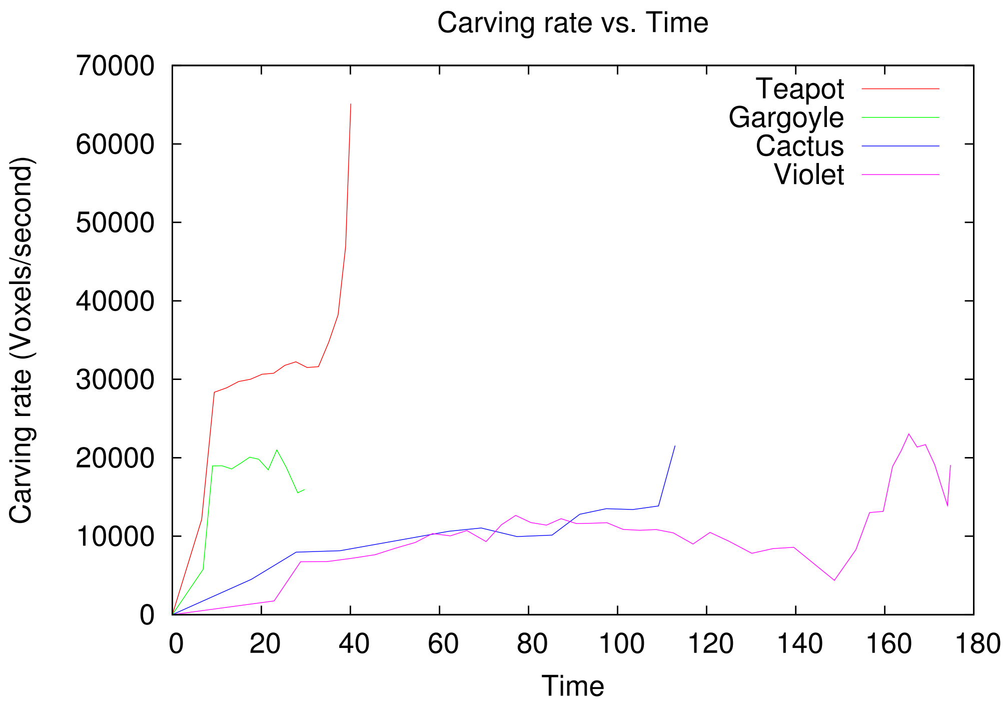

Figure 8.14: Carving rate vs. time for Bucket carver

voxel (consistency function). Memory complexity however, varies between them.

The colour consistency function plays a large part, which has complexity linear with the size of the Vis(V) set. For a typical case, the size of Vis(V) will vary depending on visibility and proximity from the camera centres. So we approximate the complexity of colour consistency to a roughly constant complexity equal to the average size of Vis(V).

The average cost per voxel-iteration is given by $c$ which varies between algorithms and colour consistency algorithm. This cost is mostly dependent on the average size of the Vis(V) set, which depends on the relative voxel, image resolution and the carving order, especially.

Statistics on how time for evaluation varies within each scene are shown in figures 8.15, 8.16. The same general trend is shown (larger Vis(V) takes longer). Of particular note, the Bucket algorithm shows significantly larger average Vis(V) than GVC. This is not surprising as the statistic is a running mean over time. The carving queue specifically orders carving voxels with the largest Vis(V) first, whereas GVC evaluates all surface voxels at each iteration, so is bound to include a wider distribution.

The time complexity of all voxel colouring algorithms come from the total voxel evaluations $e$, by the cost of evaluating each voxel $c$. We assume a voxel volume of size $n^3$. Given some voxels will be re-evaluated, (other factors ignored), the complexity will be $O(c * k * n^3)$ where $k$ is the average number of times each voxel is evaluated. This factor is dependent on scene and algorithm factors.

### *8.6.1 GVC*

The number of voxels evaluated by GVC is the sum of of surface area at each iteration. The number of iterations and the surface area are both dependent on the scene. Some statistics resulting from the four test scenes used above are shown in table 8.3.

Statistics on iterations and surface area for GVC can be seen in figure 8.17. For a typical scene the surface area reduces only a little, and the number of iterations are about $0.75n$ in 3 of 4 cases. Note that the gargoyle used an

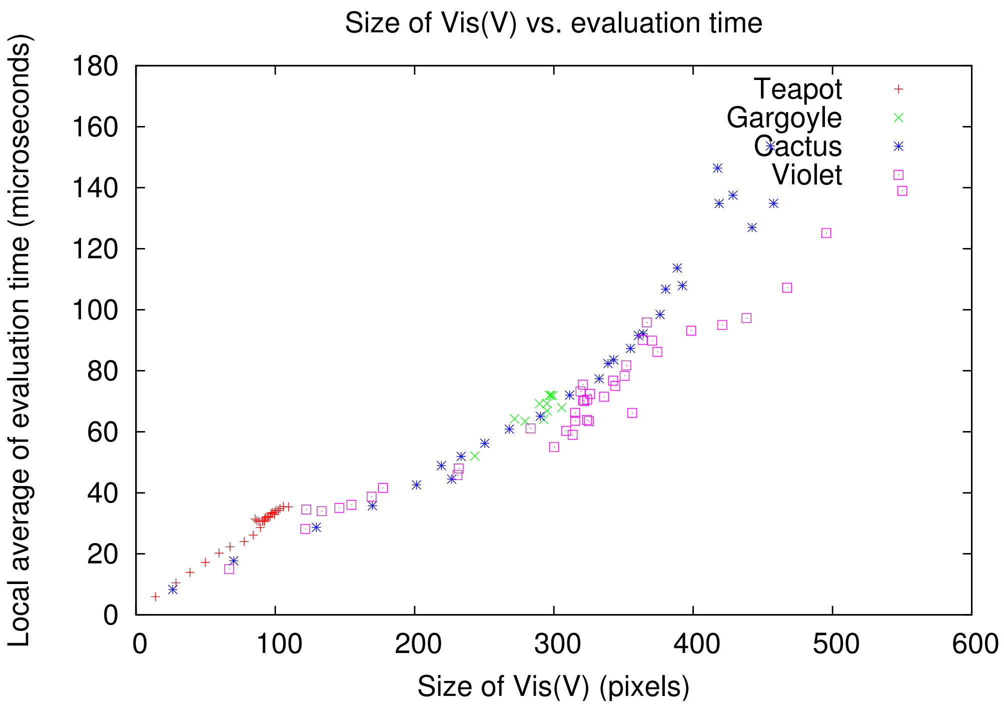

Figure 8.15: Vis(V) vs. Carving time for Bucket algorithm

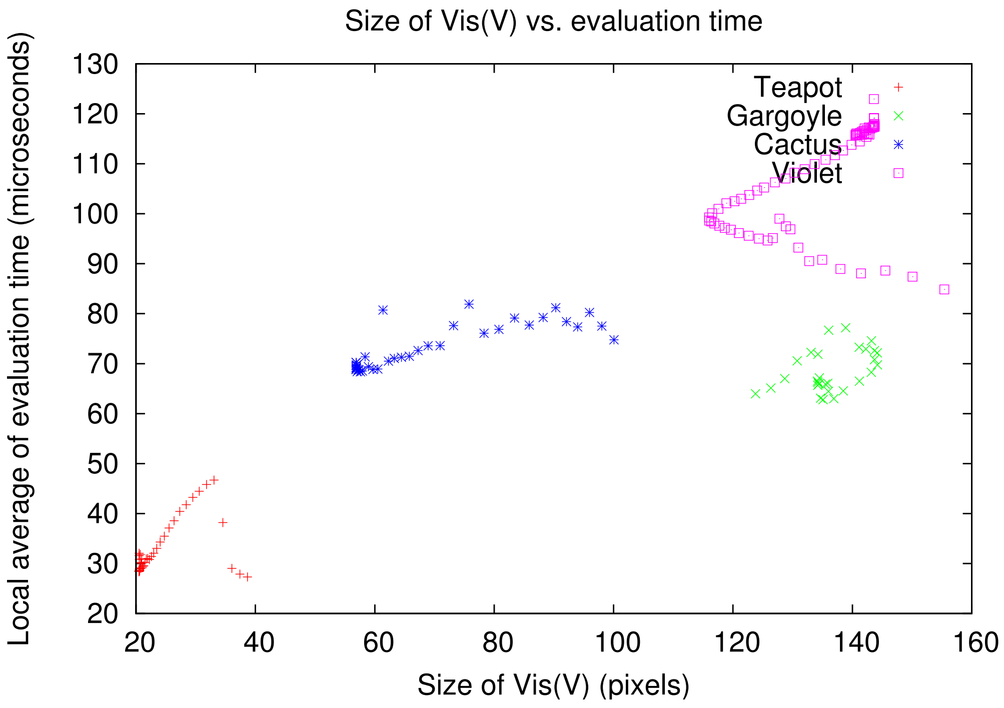

Figure 8.16: Vis(V) vs. Carving time for GVC algorithm

| **Scene** | **Iterations** | **Evaluated** | $k$ |
|---|---|---|---|
| Teapot | 91 | 6867691 | 3.97 |
| Gargoyle | 59 | 911211 | 1.78 |
| Cactus | 77 | 6569071 | 3.80 |
| Violet | 148 | 8972587 | 5.19 |

Table 8.3: Average evaluations per voxel, for GVC.

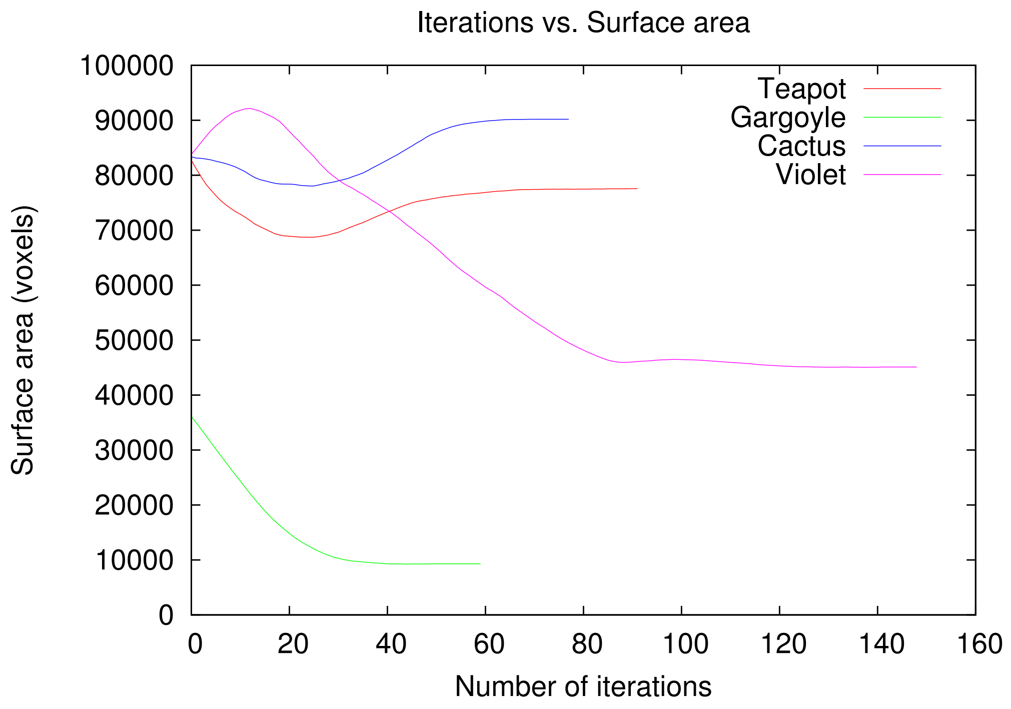

Figure 8.17: Iterations vs surface area, GVC

| **Scene** | **Evaluated** | $k$ |
|---|---|---|
| Teapot | 1356920 | 0.79 |
| Gargoyle | 911211 | 0.95 |
| Cactus | 6569071 | 0.88 |
| Violet | 8972587 | 1.38 |

Table 8.4: Average evaluations per voxel, for Ray Buckets.

$80^3$ volume resolution, the others used a $120^3$ resolution. Assuming constant surface area at $O(n^2)$ and the iterations $O(n)$ an approximate complexity is $O(n^3 \times c)$. This does seems to be a reasonable approximation, as $k$ varies between 1.78 and 5.19.

### *8.6.2 Bucket, GVC-LDI*

A major difference between incremental/serial (GVC-LDI, Bucket carver) and parallel (GVC) is that in the incremental methods, carving is at a nearly constant rate as shown by 8.14. Where as (necessarily) parallel methods evaluate an almost constant number of voxels per iteration, and carve at a decreasing rate per iteration. This can be seen in figure 8.13.

Statistics from test scenes using the bucket carver are given in table 8.4 below. Of note, overall a serial algorithm performs greatly fewer evaluations, and $k$ varies between 0.79 and 1.38. There is no direct correspondence between the statistics for GVC and the statistics below. Note that for the gargoyle set, $k$ is relatively low for GVC, but relatively high for Buckets. This illustrates the relative disparity seen for the gargoyle, but not for other datasets.

The evaluation count statistics for GVC-LDI, and ray images are directly related, and differ only by two factors. Firstly, the carving order is different. Secondly (in the case of GVC-LDI), it is possible for a voxel to be removed on one side of a voxel model, causing a voxel on the other to rise to the head of its linked list. While not actually changing visibility, this results in fractionally more voxels being evaluated.

The cost for evaluating one voxel is vastly different between GVC-LDI, Bucket and Ray Image algorithms. Bucket and Ray Image algorithms are

| **Scene** | **List modifications** | **Average list length** |
|---|---|---|
| Teapot | $473 \times 10^6$ | 1.44 |

Table 8.5: Average list length, for GVC-LDI.

```
    remove V from each LDI(P) in Proj(V)

    for each neighbor N of V not in SVL {
        add N to SVL, each LDI(P) in Proj(N), CVSVL
    }

    for each LDI(P) with a changed head U, add U to CVSVL
```

Figure 8.18: Voxel modification portion of GVC-LDI algorithm listed in figure 2.10

considerably faster on a per voxel cost, and use a little less memory. A summary of each is given below.

### *8.6.3 GVC-LDI*

Adding or removing a given voxel V, for GVC-LDI requires modifying the LDIs. This requires finding Proj(V) first, then updating each linked list in Proj(V). This has complexity $O(|Proj(V)| * length)$, where length is the average length of a linked list in an LDI.

Updating visibility structures for GVC-LDI requires first removing the current voxel from LDIs. Each newly visible neighbour voxel U, is added to the LDIs. The overall cost per voxel, will depend on the number of new voxels added, the length of the LDI lists, and the number of times the lists are modified. Statistics for the teapot scene are given. The list lengths are relatively short, but there are a large number of list modifications compared to ray steps for the Ray Image and Bucket algorithms, statistics are listed in table 8.5.

The set of changed visibility voxels (a superset of CVS), is found by gathering the head of each linked list in LDIs which has changed. The relevant portion of the GVC-LDI algorithm is listed in figure 8.18.

| **Scene** | **Ray traversals** | **Voxels traversed** | **Average depth** |
|---|---|---|---|
| Teapot | $539 \times 10^5$ | $195 \times 10^6$ | 3.62 |
| Gargoyle | $645 \times 10^5$ | $278 \times 10^6$ | 4.32 |
| Cactus | $162 \times 10^6$ | $672 \times 10^6$ | 4.14 |
| Violet | $267 \times 10^6$ | $119 \times 10^7$ | 4.48 |

Table 8.6: Ray traversal statistics, for Ray Images.

### *8.6.4 Ray Image*

Updating visibility structures using ray images requires finding the CVS (shown in figure 5.5), then adding the CVS to the CVSVL (adding elements to the CVSVL hashtable is O(1)). This involves iterating each ray and stepping until the next solid voxel is found. So, the complexity of $c$ is $O(|Vis(V)| * depth)$, where *depth* is the average depth stepped per ray traversal. Some statistics on this factor are listed in table 8.6. The depth stepped is almost identical for all four scenes despite having quite different geometry.

### *8.6.5 Bucket*

Updating visibility structures using buckets is a little more complicated. It splits finding the CVS and adding the CVS to the CVSVL before and after evaluating colour consistency (though in the case of using a threshold it works either way).

Finding the set of candidates is done by stepping each ray and storing the intermediate ray in a small hashtable. This step is $O(|Vis(V)| * depth)$ (*depth* is the average depth stepped per ray traversal). Adding the candidates to the CVSVL involves appending each ray from candidates to the equivalent bucket in the CVSVL. This is also an $O(|Vis(V)|)$ process (appending to a list is constant time).

The cost of adding/removing voxels is constant and negligible for (GVC, GVC-LDI) but incurs some cost when an ordering queue is involved (Buckets). If the carving queue is on the order of the surface area which we approximate to $O(n^2)$ - adding and deleting from the binary heap is an $O(logn^2)$ operation.

| Scene | Ray traversals | Voxels traversed | Average depth |
|---|---|---|---|
| Teapot | $101 \times 10^6$ | $192 \times 10^6$ | 1.90 |
| Gargoyle | $135 \times 10^6$ | $277 \times 10^6$ | 2.04 |
| Cactus | $354 \times 10^6$ | $651 \times 10^6$ | 1.83 |
| Violet | $475 \times 10^6$ | $117 \times 10^7$ | 2.46 |

Table 8.7: Ray traversal statistics, for Ray Buckets.

Carving queues and visibility updates are considered $c = O(|Vis(V)| * depth) + O(logn^2)$. However, in practice updating carving queues is dominated by the other factors.

The Bucket algorithm has very similar statistics (shown in table 8.7) arising from ray traversal, to that of the Ray Image listed above. The difference is a result of the carving order, otherwise they would be identical. In this case the Ray Image algorithm is using an arbitrary (hashtable) ordering, whereas the Bucket algorithm is using a most visible first ordering. The result is that the average depth is halved, but number of traversals are doubled compared to the Ray Image statistics. Unsurprisingly, nearly the same number of voxels are traversed in total for both cases, given that they both arrive at a very similar final model.

## *8.7* Summary

The GVC table method accelerated with OpenGL is consistently fastest with the lowest memory use, however it is the most restricted of the methods compared. Both Bucket and Ray Image algorithms perform similarly, in all cases faster than software GVC and by a large factor over GVC-LDI.

Each algorithm tested (bar GVC-LDI, due to memory constraints) reconstructed each of the test scenes using a threshold colour consistency function. Despite large differences in carving order, and the way Vis(V) is calculated, reconstructions were very similar. This can be seen from looking at the reprojection error achieved by each algorithm. Only minor differences in the final result occur.

The complexity of each can be approximated to $O(n^3) * c$, however there are many factors involved. The number of evaluations are different, but nearly constant in practice within algorithms (as can be seen from tables above for parameter $k$). The parameter $c$ is dominated by $O(|Vis(V)|)$, where the average size of Vis(V) depends on voxel resolution and camera geometry. Additional factors arise for GVC-LDI and ray traversal methods. Such as the average linked list depth and the average depth of voxels traversed.

# Chapter IX

# Applications

Here we examine the application of the Bucket approach to optimisation colour consistency, and compare the reconstruction quality given by implementations of two prior works. We look at the feasibility of using level of detail with the bucket approach, and compare statistics with those given by using GVC for the same task. We also examine the flexibility of the Bucket approach by comparing the usefulness of being able to use many low resolution images vs. few high resolution images.

## *9.1* Optimisation colour consistency

We have implemented two optimisation voxel colouring methods, as described in [29], and [15]. We have made some comparisons to demonstrate the feasibility, as well as demonstrate versatility, and compare each with regard to reconstruction quality.

We have tested both implementations on the cactus and the violet. We first ran a thresholding carver over the data set to reduce the time taken, though using level of detail is also equally useful for this purpose. We used a combination of threshold and optimisation to further carve the voxel model and compared the results. The reason for using both threshold and optimisation was to reduce local minima which seem to be the main problem with this type of consistency function. However this somewhat defeats the point of avoiding the need for an ad-hoc threshold.

### *9.1.1 Cactus*

Both reduced reprojection error well, statistics are shown in table 9.1. However as the depth maps, show the reconstructions produced contain very

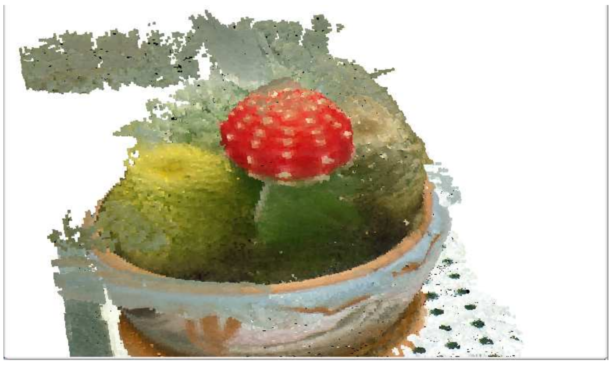

Figure 9.1: Threshold.

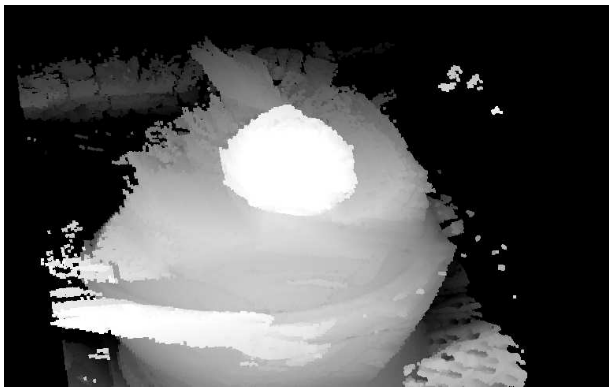

Figure 9.2: Threshold. (depth)

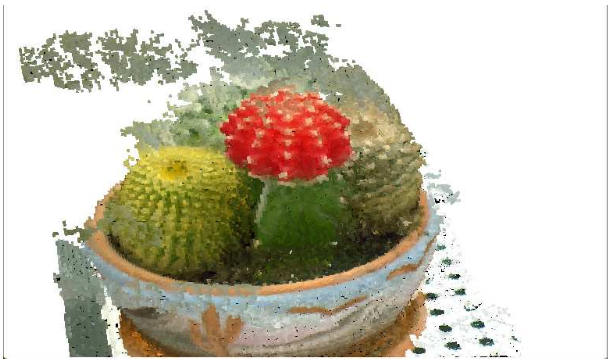

Figure 9.3: Reprojection error.

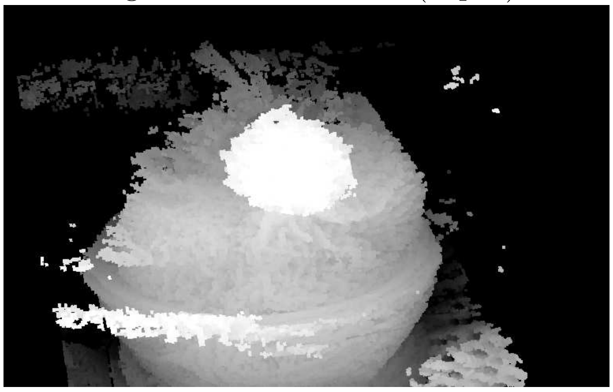

Figure 9.4: Reprojection error. (depth)

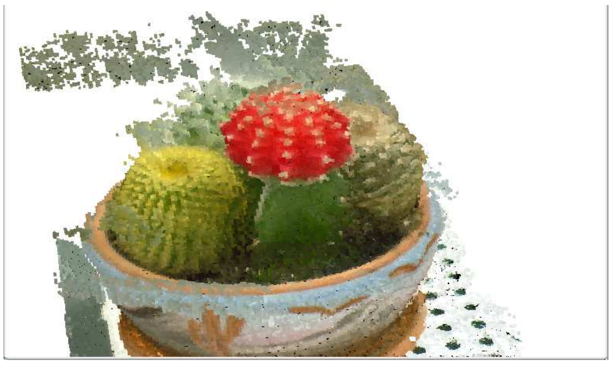

Figure 9.5: Probabilistic.

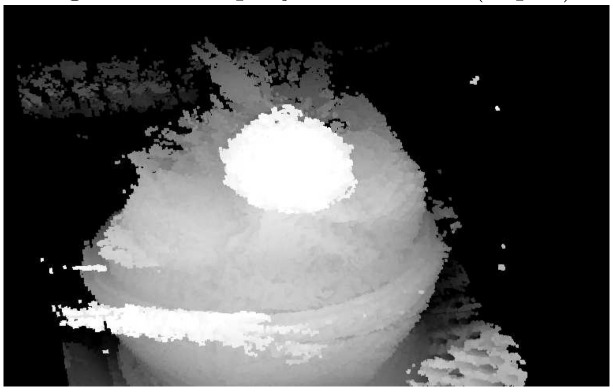

Figure 9.6: Probabilistic. (depth)

| **Method** | **Time (s)** | **Reduced reproj.** |
|---|---|---|
| Reprojection error | 82.0 | 39.7% |
| Probabilistic | 221.7 | 28.1% |

Table 9.1: Reprojection error reduced for cactus scene.

| Method | Time (s) | Reduced reproj. |
|---|---|---|
| Reprojection error | 148.3 | 39.1% |
| Probabilistic | 478.2 | 38.3% |

Table 9.2: Reprojection error reduced for violet scene.

noisy surfaces. Results of each are shown in figures 9.3, and 9.5. Both take considerably longer than threshold consistency functions, but show a small improvement in reconstruction quality.

### *9.1.2 Violet*

The reconstruction of the violet shows a similar pattern. Reprojection error is reduced, shown in table 9.2, but shows very noisy surfaces. Results of each are shown in figures 9.9, and 9.11. Again, both take a very long time but show small a improvement in reconstruction quality. The optimisation colour consistency also does not do any better with the leaves, leaving large slabs. The reconstructions still do not show up the lighter green colour of the inside of the leaves at all.

## *9.2* Summary

We have implemented and compared two optimisation consistency functions using the Ray Buckets approach. Overall, reprojection error, but not quality improved over threshold colour consistency. In each test scene, and consistency function, reprojection error is reduced but results in very noisy surface reconstructions. Some improvement can be made by adding ad-hoc constraints on a minimum number of visible cameras, or in combination with level of detail to avoid local minima.

## *9.3* Level of detail

We have looked at the applicability of using level of detail to reduce the large reconstruction period at high resolutions, for both GVC and the bucket method. We have used the method first shown by Proc and Dyer in [22]. It begins with a low resolution voxel volume and repeats steps outlined below.

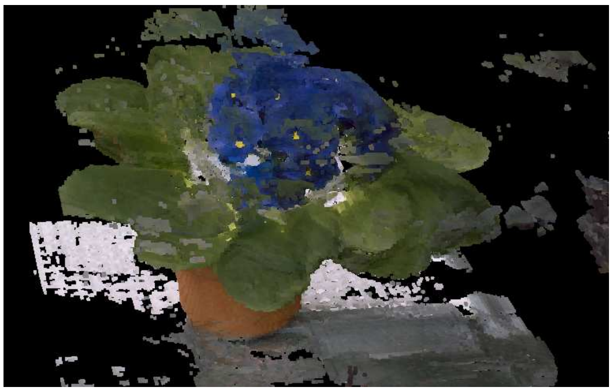

Figure 9.7: Threshold.

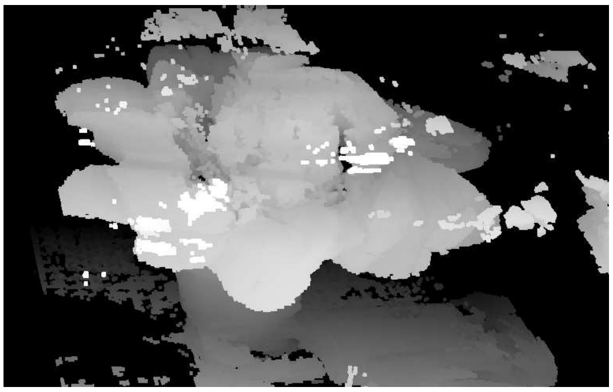

Figure 9.8: Threshold. (depth)

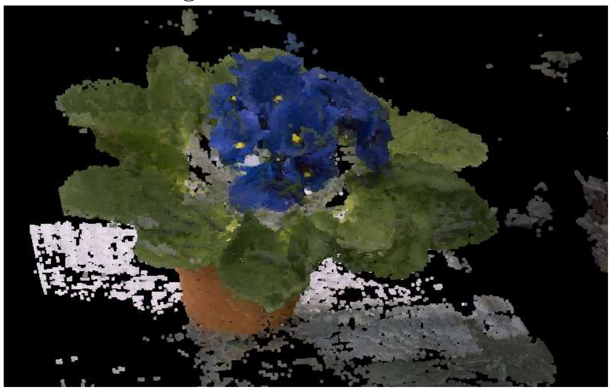

Figure 9.9: Reprojection error.

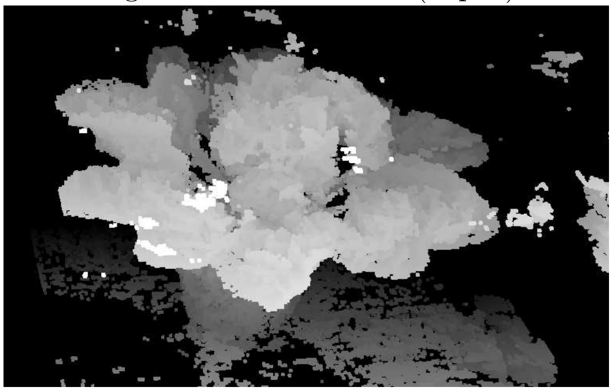

Figure 9.10: Reprojection error. (depth)

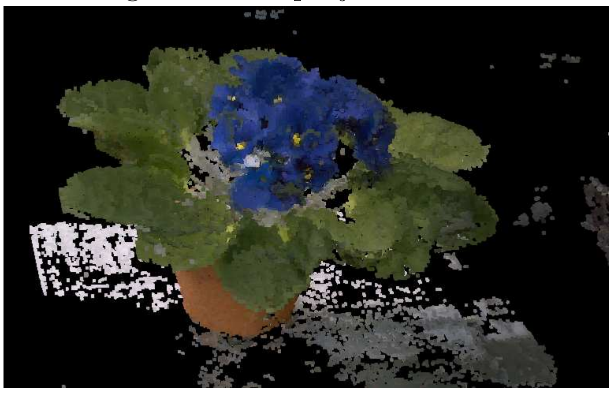

Figure 9.11: Probabilistic.

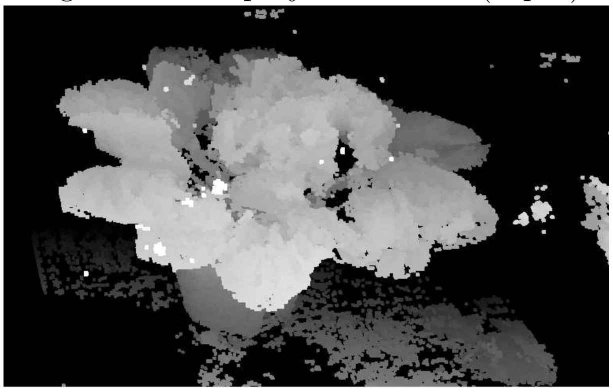

Figure 9.12: Probabilistic. (depth)

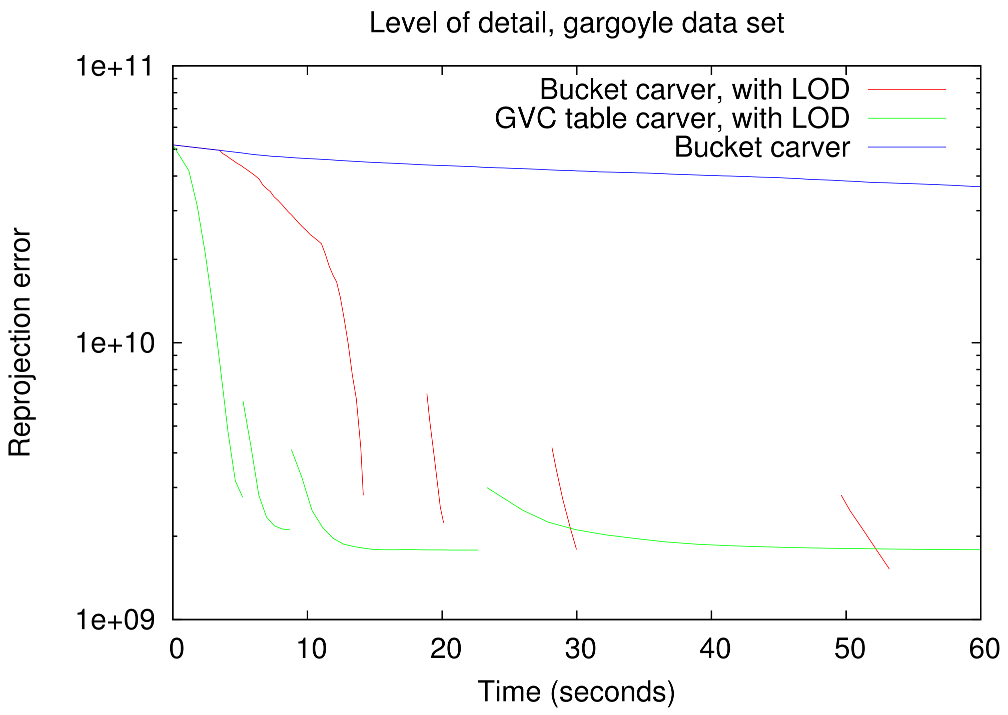

Figure 9.13: Reprojection vs. time for LOD sequence

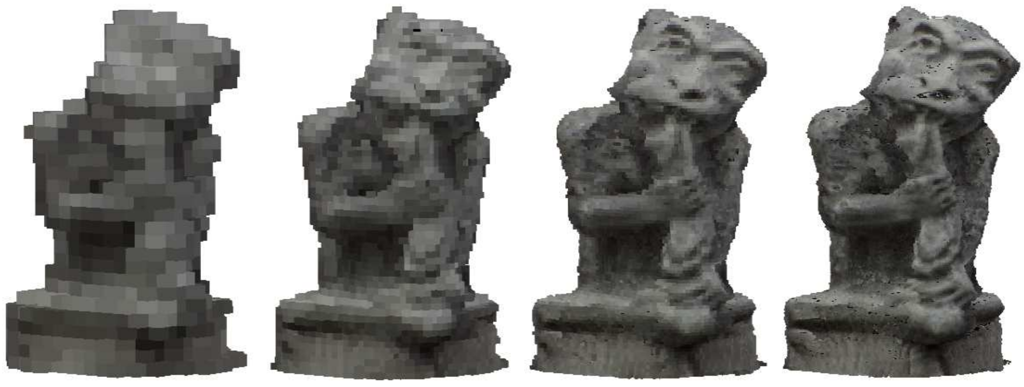

Figure 9.14: Level of detail sequence $30^3$ to $240^3$.

1. Carve voxels.

2. Augment voxel model.

3. Double resolution (along each axis)

We anticipated that the Bucket method may be particularly suited to this technique because it is fast when convergence is close. We ran a comparison between GVC with LOD, the bucket method with LOD, and the Bucket method without LOD as a control. Resolution began at $30^3$ and increased in resolution 4 times to reach $240^3$. A sequence of the intermediate reconstructions at each resolution is shown in figure 9.14. The original image from the same view is shown in figure 8.6, and a graph of reprojection error in figure 9.13.

Results show that the technique worked better with GVC. The bucket method proved somewhat slower because of the slow re-initialisation period involving raytracing the intermediate voxel model. This could be solved by using additional spacial acceleration structures. Clearly LOD pays off for both methods however, as they are much faster than the control. The only disadvantage is a possible loss of fine detail, however this also helps reduce noise.

## *9.4* Resolution vs View distribution experiment

As mentioned in section 7.2 a promising approach may be to trade off image resolution with the number of input images. This is in order to test the hypothesis that voxel colouring works best at determining occluding contours and therefore should work best with a wide distribution of input image angles. Using Ray Buckets images are not (directly) used, so reconstruction quality may improve while memory and time complexity remain roughly equivalent.

Two sets of data were used for this comparison, the teapot dataset with strongly contrasting objects, and the hand dataset with just one object and less contrasting colours. An Athlon 2600+ with 768 MB of ram was used. In both cases the Ray Buckets algorithm given in figure 7.2 were used and, the resolution of the voxel volume was $120^3$.

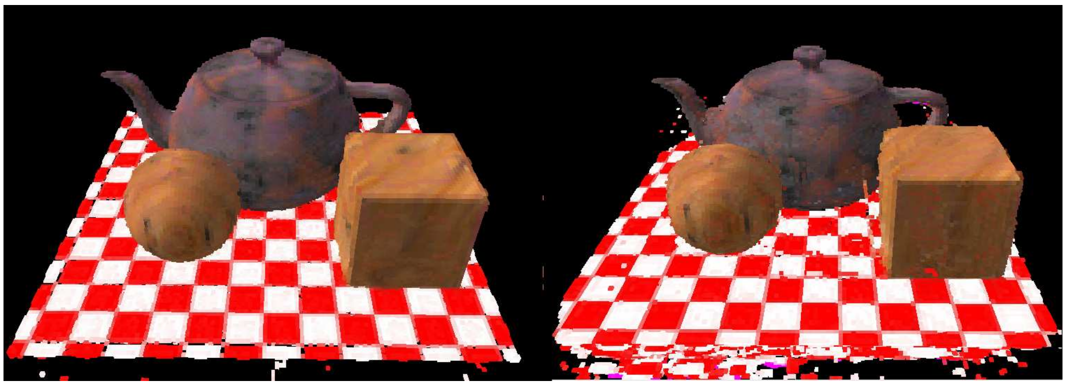
Figure 9.15: Independent views of the result using 60 × 400 × 300 images, and 15 × 800 × 600 images.

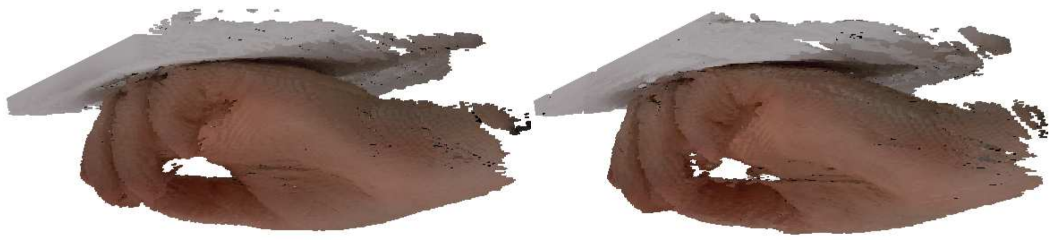
Figure 9.16: Reprojection of one view using 99 images at 384 × 242 vs. 25 images at 768 × 484.

For the teapot dataset 15 images, with 800 × 600 surrounding the voxel volume were captured. In the other case 60 images at 400 × 300 were used. For the hand dataset 99 photos at 768 × 484 of a hand were used, in one case all 99 photos were scaled to quarter resolution of 384 × 242 in the other, every fourth image was used (25 of 99 images).

For each data set ,both cases used the same threshold colour consistency function, with the same thresholds - using weighted central differences as given by 2.5. Rasterized images of the teapot scene were used. Segmentation information was not used in reconstruction.

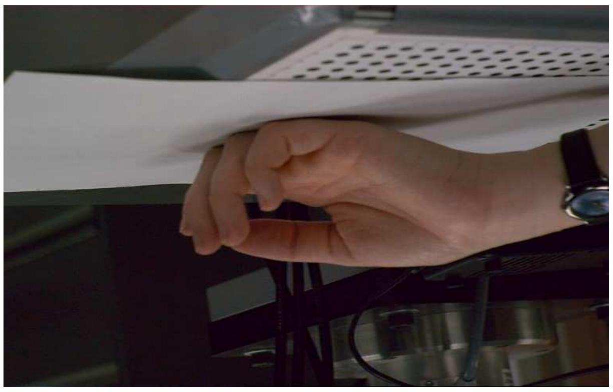

Figure 9.17: Input image for reconstruction in figure 9.16

| Case | Total rays | Time taken (s) | Carved |
|---|---|---|---|
| **teapot - 15 images** | 5145943 | 64.58 | 1074472 |
| **teapot - 60 images** | 5208912 | 76.25 | 1351763 |
| **hand - 25 images** | 3232679 | 58.07 | 1332153 |
| **hand - 99 images** | 3264641 | 75.81 | 1379089 |

Table 9.3: Runtime statistics, comparing different input image placement and resolutions.

### *9.4.1 Results*

Subjectively, using 60 images produced a less noisy reconstruction than with 15 images, as shown in figure 9.15. It should be also noted that it took many more attempts at re-arranging the cameras to cover the scene ”evenly” in the case with fewer images.

It was noted that in setting up the comparison the position of the cameras in the 15 image case affected the result considerably. In the 60 image case, the arrangement of particular images/cameras seemed to matter very little, provided they very roughly covered the space around the voxel volume.

For the hand dataset 9.16 the reconstruction was very similar in both cases. This is probably because extra camera angles provided very little extra background information when the visual hull is adequately reconstructed with a smaller number of images.

The runtime statistics (table 9.3) show that nearly the same total number of rays were used for both datasets (at the beginning – some may be discarded as the process continues). The case using fewer high resolution images was slightly faster. This is almost certainly because they evaluated and carved less voxels in total. Reprojection error does not compare directly, as the input images are different in each case, so it was not looked at.

Overall, this small experiment confirms somewhat the hypothesis that voxel colouring works best at determining occluding contours, and shows that the idea of using voxel colouring with many lower resolution images at wide viewpoints could be beneficial at least using colourful high contrast scenes as the teapot dataset.

# Chapter X

# Summary

We have investigated the use of using ray traversal as a means for incremental, serial voxel colouring. We have looked at three main approaches, Depth Stack Images (DSI), Ray Images and the Bucket approach. Each looked at improving a problem found in prior approaches. All three use a form of ray traversal. These algorithms have been implemented and compared in detail. Applications have been investigated, for threshold/optimisation colour consistency and level of detail.

It was initially thought that as ray tracing is seen as a very slow process, so directly computing ray traversal would work poorly. So we looked at an approach pre-computing ray traversals using rasterization (seen as efficient). This resulted in a different kind of Layered Depth Image, which we termed a Depth Stack Image. Ray traversal amounted to simply popping elements off the back of each depth stack. This had several problems, notably using a large amount of memory and having a slow initialisation period.

In comparison to GVC-LDI, the DSI approach was promising. It is simpler and is much faster at carving voxels. However, it created a couple of issues. Firstly uses even more memory than GVC-LDI unless some form of compression is used, and secondly the initialisation period is extremely slow. Compared to both variations of GVC, the DSI approach is considerably slower.

It turned out that pre-computation of the ray traversals provided absolutely no gain over direct ray traversal. Direct ray traversal has been a superior solution. An image of rays was used in place of the DSI, and an equivalent voxel colouring algorithm was found. Traversal took place by resuming and stopping a ray traversal algorithm. Compared to GVC-LDI and the DSI approach, it is much faster in all aspects. It is comparable but a little slower than the OpenGL GVC implementation.

It turned out that implementing optimisation colour consistency was difficult using Ray Images. The Ray Buckets approach was developed in an attempt to overcome these shortcomings. Item buffers were removed, and in their place a structure associating rays directly associated with their current. As a voxel is carved rays are "tipped" (like a bucket) and "fall" into another voxel by ray traversal. The major benefit with the Ray Buckets approach is that Vis(V) is incrementally updated and immediately available on demand, as opposed to computing Vis(V) when required.

Use of incremental statistics and tentative carving allowed the application of a Ray Bucket algorithm to optimisation colour consistency. Reconstruction quality has not improved over threshold colour consistency significantly. We have compared two functions, both reduce reprojection error but result in a very noisy surface reconstructions.

Application to level of detail highlights a weakness in that re-initialisation is slow. GVC appears to work a little better for this purpose, due to a more efficient re-initialisation. All carving methods benefit from marked improvement in running time at high voxel resolution from the level of detail approach, however.

## *10.1* Conclusion

In conclusion ray traversal seems like a promising method for maintaining global scene visibility when carving voxel models. We developed two successful approaches, one based around using Ray Images as item buffers, and the other maintaining visible ray lists for each voxel. (Ray Buckets.)

Overall, using ray traversal compares favourably with GVC-LDI greatly in time and space use for incremental voxel carving. It is more flexible in input representation, but trickier, to implement a scheme involving both adding and carving voxels than GVC-LDI.

Use of ray traversal also compares well with GVC. However the use of hardware rasterization is clearly beneficial resulting in faster running times with GVC as evidenced by the OpenGL GVC running times. Ray traversal is much more flexible than vanilla GVC for searching or determining visibility change.

We have looked at applying these methods to optimisation colour consistency, with mixed success. Reconstruction resulted in noisy surfaces, despite reducing reprojection error. Level of detail was shown to be a useful method for reducing time taken, but not as well suited to our approaches as anticipated.

## *10.2* Future

In future, it would seem to be useful to investigate use of ray traversal for parallel approaches where graphics hardware can be used. Ray traversal is currently used for displacement mapping in real time, so a proof of concept already exists in regard to performing voxel ray traversal on graphics hardware. Implementation would need to consider the type of a consistency function and routing of inputs and outputs carefully. Partially serial methods for example, processing the first N voxels from a carving queue, may be of use.

There are many avenues to explore with regard to optimisation colour consistency. Stereo methods have shown much higher quality reconstructions than voxel colouring – but typically approximate visibility in a much more crude manner, using visual hulls, thresholds or the best N views, for example [2]. Wet may be useful to use an algorithm like those presented here in combination with stereo. One notable example in this vein uses a low resolution voxel grid with GVC-LDI, followed by using stereo with graph cuts [33].

# References

[1] John Amanatides and Andrew Woo. A fast voxel traversal algorithm for ray tracing. In *Eurographics '87*, pages 3–10. Elsevier Science Publishers, Amsterdam, North-Holland, 1987.

[2] C. Andez and E. Schmitt. A snake approach for high quality image-based 3d object modeling. In *2nd IEEE Workshop on Variational, Geometric and Level Set Methods in Computer Vision*, pages 241–248, Nice, France, 2003.

[3] Jeremy S. De Bonet and Paul A. Viola. Roxels: Responsibility weighted 3d volume reconstruction. In *ICCV (1)*, pages 418–425, 1999.

[4] Thomas Bonfort and Peter Sturm. Voxel carving for specular surfaces. In *Proceedings of the 9th IEEE International Conference on Computer Vision*. IEEE CSP, October 2003.

[5] Yuri Boykov, Olga Veksler, and Ramin Zabih. Fast approximate energy minimization via graph cuts. In *ICCV (1)*, pages 377–384, 1999.

[6] Felicia Brisc and Paul Whelan. Creating virtual models from uncalibrated camera views. In *Proceedings of the Irish EuroGraphics Workshop*, 2004.

[7] A. Broadhurst and R. Cipolla. A statistical consistency check for the space carving algorithm. In *Proc. 11th British Machine Vision Conf.*, pages 282–291, 2000.

[8] Zach Ch., Karner K., Reitinger B., and Bischof H. Space carving on 3d graphics hardware. Technical report, VRVis Technical Report, Graz University of Technology, 2004.

[9] Vikram Chhabra. Reconstructing specular objects with image based rendering using color caching. Master's thesis, Worcester Polytechnic Institute, 2001.

[10] Bruce Culbertson. A histogram-based color consistency test for voxel coloring. In *ICPR '02: Proceedings of the 16 th International Conference on Pattern Recognition (ICPR'02) Volume 4*, page 40118. IEEE Computer Society, 2002.

[11] W. B. Culbertson and T. Malzbender. Generalized voxel coloring. In *Proceedings of the ICCV Workshop*, volume Vision Algorithms Theory and Practice, September 1999.

[12] C. R. Dyer. Volumetric scene reconstruction from multiple views. In L. S. Davis, editor, *Foundations of Image Understanding*, pages 469–489. Kluwer, 2001.

[13] A. Fusiello. Uncalibrated euclidean reconstruction: A review. *Image and Vision Computing, 18(67)*, pages 555–563, May 2000.

[14] David T. Gering and William M. Wells III. Object modeling using tomography and photography. In *MVIEW '99: Proceedings of the IEEE Workshop on Multi-View Modeling & Analysis of Visual Scenes*, page 11, Washington, DC, USA, 1999. IEEE Computer Society.

[15] Ho-Won Kim and In So Kweon. Optimal photo hull recovery for the image-based modeling. In *The 6th Asian Conference on Computer Vision (ACCV), Jeju, Korea*, January 2004.

[16] K. N. Kutulakos and S. M. Seitz. What do n photographs tell us about 3d shape?, January 1998. Computer Science Dept. U. Rochester.

[17] Kiriakos N. Kutulakos. Approximate n-view stereo. In *ECCV '00: Proceedings of the 6th European Conference on Computer Vision-Part I*, pages 67–83, London, UK, 2000. Springer-Verlag.

[18] Kiriakos N. Kutulakos. Approximate n-view stereo. In *ECCV '00: Proceedings of the 6th European Conference on Computer Vision-Part I*, pages 67–83, London, UK, 2000. Springer-Verlag.

[19] Kiriakos N. Kutulakos and Steven M. Seitz. A theory of shape by space carving. *Int. J. Comput. Vision*, 38(3):199–218, 2000.

[20] N. Bagherzadeh M. Sainz and A. Susin. Hardware accelerated voxel carving. In *1st Ibero-American Symposium in Computer Graphics (SIACG 2002), Guimaraes, Portugal.*, pages 289–297, 2002.

[21] Y. Ohta and T. Kanade. Stereo by intra- and inter-scanline search using dynamic programming. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 7:139–154, 1985.

[22] Andrew C. Prock and Charles R. Dyer. Towards real-time voxel coloring. In *Proc. Image Understanding Workshop*, pages 315–321, 1998.

[23] J. Revelles, C. Ureña, and M. Lastra. An efficient parametric algorithm for octree traversal. *Journal of WSCG 8(2)*, pages 212–219, 2000.

[24] Bagherzadeh N. Sainz M. and Susin A. Carving 3d models from uncalibrated views. In *5th IASTED International Conference Computer Graphics and Imaging (CGIM 2002), Kauai, Hawaii, USA*, pages 144–149, 2002.

[25] Bagherzadeh N. Sainz M. and Susin A. Recovering 3d metric structure and motion from multiple uncalibrated cameras. In *In IEEE Proc. International Conference on Information Technology: Coding and Computing*, pages 268–273, 2002.

[26] Steven M. Seitz and Charles R. Dyer. Photorealistic scene reconstruction by voxel coloring. In *CVPR '97: Proceedings of the 1997 Conference on Computer Vision and Pattern Recognition (CVPR '97)*, page 1067. IEEE Computer Society, 1997.

[27] Steven M. Seitz and Charles R. Dyer. Photorealistic scene reconstruction by voxel coloring. *Int. J. Comput. Vision*, 35(2):151–173, 1999.

[28] J. Sethian. *Level Set Methods and Fast Marching Methods: Evolving Interfaces in Computational Geometry.* Cambridge University Press, 1998.

[29] G. Slabaugh, B. Culbertson, T. Malzbender, and R. Schafer. Improved voxel coloring via volumetric optimization. Technical report, Center for Signal and Image Processing, Georgia Institute of Technology, 2000.

[30] G. Slabaugh, B. Culbertson, T. Malzbender, and R. Schafer. A survey of methods for volumetric scene reconstruction from photographs. In *International Workshop on Volume Graphics 2001, Stony Brook, New York*, pages 81–100, june 2001.

[31] G. Slabaugh, B. Culbertson, T. Malzbender, and R. Schafer. A survey of methods for volumetric scene reconstruction from photographs. Technical report, Center for Signal and Image Processing, Georgia Institute of Technology, 2001.

[32] Gregory G. Slabaugh, W. Bruce Culbertson, Thomas Malzbender, Mark R. Stevens, and Ronald W. Schafer. Methods for volumetric reconstruction of visual scenes. *Int. J. Comput. Vision*, 57(3):179–199, 2004.

[33] Gang Zeng, Sylvain Paris, Long Quan, and François Sillion. Progressive surface reconstruction from images using a local prior. In *International Conference on Computer Vision*, 2005.

[34] Ye Lu Zhang, J.Z. Wu, and Q.M.J. Ze-Nian Li. A survey of motion-parallax-based 3-d reconstruction algorithms. *Systems, Man and Cybernetics, Part C, IEEE Transactions on*, 34(4):532–548, November 2004.

[35] A. Zhirkov. Binary volumetric octree representation for image based rendering. In *Proc. of GRAPHICON'01, September 2001.*, September 2001.