# Image–Based Methods of Terrain Rendering

by

Reimar Schubert

A thesis submitted in conformity with the requirements
for the degree of Master of Science
Graduate Department of Computer Science
University of Toronto

# Abstract

Image–Based Methods of Terrain Rendering

Reimar Schubert

Master of Science

Graduate Department of Computer Science

University of Toronto

1999

Terrains are often seen in many graphics applications which require interactive walkthroughs. While very common, they can be difficult to render at interactive rates, since they may be composed of millions of polygons. This thesis presents an image–based method of rendering terrains that produces detailed impostors which can be drawn at constant frame rates, independent of the number of triangles in the terrain. A spectrum of image–based rendering methods is discussed, and four methods are implemented with increasing success. Image quality is examined, and timing results of an interactive fly–by over the final terrain impostor are compared with a fly–by over the original terrain geometry.

Acknowledgements

First, the real acknowledgements. Many thanks to James Stewart, whose ideas, knowledge, helpfulness, and deadlines allowed me to finish this thesis. I couldn't have asked for a better supervisor. Thanks also to my second reader Michiel van de Panne, who was able to come up with many useful comments even in the little time I left him with! Finally, I received help in a wide variety of areas from other students in DGP. Thanks to Tasso Karkanis, Joe Laszlo, Ryan Meredith-Jones, Petros Faloutsos, and Glenn Tsang; your help was appreciated.

I would now like to thank: Tasso, for being a slower writer than me, making me feel good about myself even as my finishing date was moving further and further back; Petros, for making me look good in bzflag and for providing me with a place to stay in Athens; Jade, for giving me fluorescent hair; Ryan, for not gloating too much about his many glorious victories in bzflag (although maybe he should thank me for not strangling him); Joe, for his amusing acrobatics and for always agreeing with me; and Glenn, for helping me kill Ryan in bzflag and for holding a party in my honour.

DGP was a great place to hang out for two years. I'm going to miss it.

# Contents

# Chapter 1

# Introduction

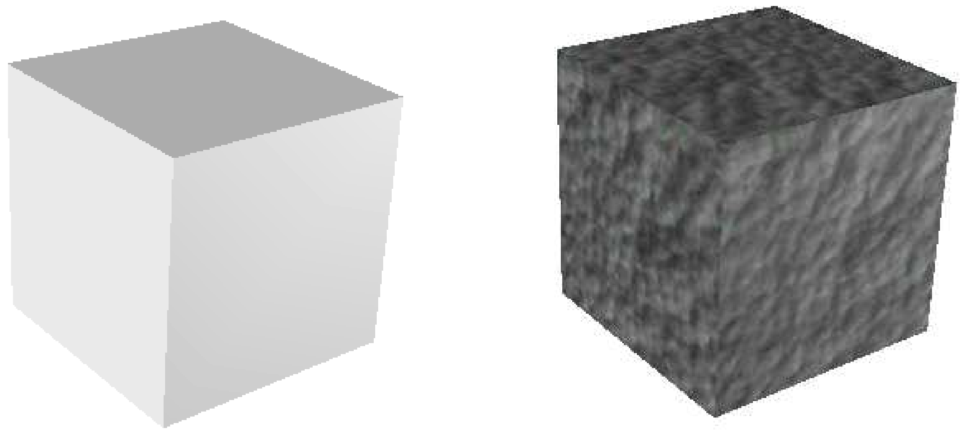
In all computer graphics research, one must always consider how long it takes to draw a frame of the scene being displayed. This is especially true when dealing with interactive graphics applications (such as video games or virtual reality). For such applications, it is desirable to produce a frame rate (the number of frames drawn per second) which will result in a smooth animation when moving through the scene. To achieve this, one needs at least 30 frames per second. This can be difficult when rendering complex scenes such as terrains, which may be composed of millions of polygons. This thesis describes a technique that accelerates the rendering of terrains.

Objects or scenes such as terrains are commonly composed of many simple polygons such as triangles. All of these pieces must be drawn individually: The screen coordinates of all vertices must be calculated, and then the pixels of each polygon must be drawn on the screen. In a complicated scene, it common to have millions of polygons which need to be drawn for each frame. When a scene cannot be drawn at interactive frame rates, methods which in some way reduce the number of polygons drawn must be explored.

Research to decrease rendering time of a scene has been done in many different directions. For example, hardware advances allow polygons to be drawn at quicker rates. Also, many algorithms have been devised which reduce the number of polygons in an object without visibly affecting geometric detail of the object. Other algorithms determine which polygons cannot be seen from the current position, so time is not wasted drawing them. This paper deals with another time–saving method known as Image–Based Rendering.

## 1.1 Image–Based Rendering

Image–based rendering (IBR) is a technique which has become increasingly popular in the last few years. The previously mentioned methods to decrease rendering time all result in a scene where objects are composed of many polygons which need to be drawn individually. IBR methods, on the other hand, replace these polygon models with one or more images of those objects from certain points of view. Therefore, instead of re–drawing possibly millions of polygons for each frame, only a much smaller set of pre–generated images need to be displayed. The largest advantage of using IBR methods is that no matter how many polygons compose the original scene, only this pre–generated set of images will need to be drawn during interactive viewing. The drawing time of this set is not dependent upon the original number of polygons, and therefore does not increase with the polygon count.



(a) (b)

Figure 1.1: (a) A simple cube (b) The cube with a texture map pasted onto the sides.

The most basic IBR method is known as texture mapping, in which an image is pasted onto a simple object in the scene. This can add detail to a scene with simple geometry, as seen in figure 1.1. As another example, consider a typical office building. Although it has a regular pattern, many polygons would be needed to accurately model all of the windows, company logos, and other such details. However, suppose we took a picture of one face of the the building and pasted it onto a single rectangle in place of the previous geometry (this is a common application of texture–mapping, known as billboarding). From the viewpoint that the picture was taken this would look completely accurate, and in fact, with features such as windows which have little depth, the building face would look quite accurate from other viewpoints. Now suppose the building had a feature with significant depth, such as a gargoyle. The image–based building would then look correct from the original viewpoint, but the gargoyle would distort as soon as one moves from that point. It would be apparent that the gargoyle is not in fact a 3D object.

It is clear from this example that IBR methods tend to be viewpoint–specific; when viewpoints are moving, it may be neccessary to use a view–dependent IBR method. View–dependent IBR methods are useful for two main reasons: IBR methods replace 3D geometry with 2D images, which can look very inaccurate from different points of view, and the scenes may be lit in a view–dependent manner. The goal of these methods is always to make the scene appear three dimensional, while still reducing the number of polygons being drawn by replacing them with pre–stored textures.

## 1.2 Motivation

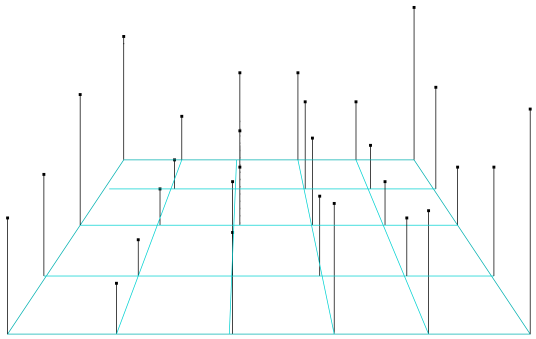


Figure 1.2: A height field, with an $x - y$ grid and associated height values.

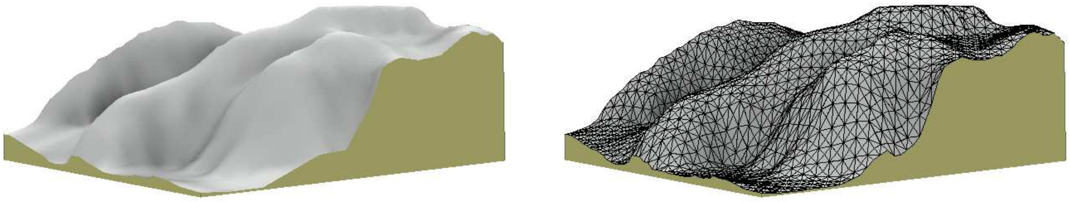
A terrain is a good example of a type of scene which can have millions of polygons, due to the high complexity seen in surface features. A terrain is usually represented by a triangulated rectangular mesh of points, each of which has an associated height (figure 1.2). Traditionally, terrain simplification methods have focused on mesh reduction. Either the number of polygons in the terrain is reduced in a preprocessing step, or the mesh is simplified in real–time as the user changes position and view direction (for example, multiple polygons that are in the distance can be combined into a single larger polygon). However, this reduced number could still be far too detailed to draw at interactive rates, unless the detail in the terrain is greatly reduced.

The goal of this thesis was to develop an IBR method for rendering terrains, which could operate at interactive frame rates, and still maintain the detail of the original terrain. Since IBR methods do not draw the actual polygons which formed the original scene, the time to draw each frame would not be dependent on the polygon complexity of the scene. Therefore, greater detail in the scene would not affect the speed of the method during interactive viewing.

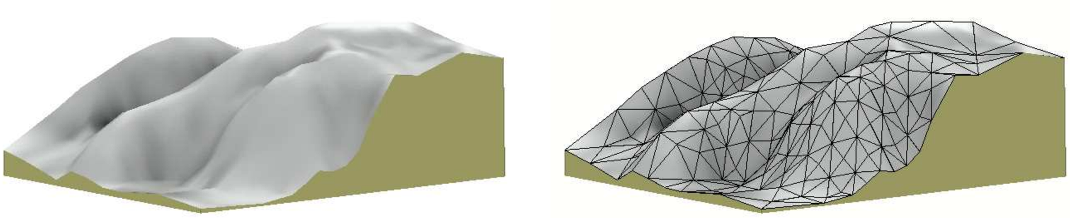
This thesis describes a spectrum of methods to achieve an IBR terrain rendering. Each subsequent method is driven by the visual artifacts of the preceding method. An overview of these methods is now presented.

## 1.3 Implemented Methods

A terrain is especially difficult to render using IBR methods. The usual structure of a terrain (hills and valleys) is difficult to simplify while retaining most of the detail, particularly at silhouette edges (such as ridges), beyond which a more distant part of the terrain is visible. The eye is particularly sensitive to simplification of silhouettes (figure 1.3). The following algorithms address the problem of rendering terrains with IBR methods, with increasing visual quality. Note that all terrains used in the implemetation were lit with pre–computed view–independent lighting data. The main goal of the impostors was to generate 2D replacements for the terrain triangles which accurately model the 3D geometry, while avoiding the "texture lifting" problem that will be discussed in Chapter 3.



(a) (b)

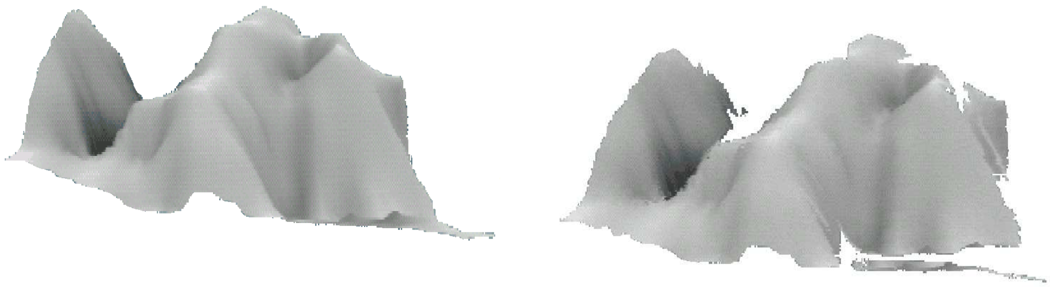


(c) (d)

Figure 1.3: (a) A terrain showing a ridge silhouette drawn at full detail. (b) The terrain with the mesh shown. (c) The simplified terrain. The loss of detail is very noticeable along the ridge silhouette. (d) With the mesh.

### 1.3.1 Sprites with Depth

This algorithm begins with a pre–processing step, in which images of the terrain are recorded from **sample viewpoints** on a hemisphere surrounding the terrain. In addition to the image, the depth of each pixel in the image is recorded. When the user is interactively viewing the terrain, a sample viewpoint is first selected, based on the current position of the viewer. The corresponding sample image is then distorted (or "warped" in IBR terminology) so that it looks correct from the current viewpoint, and the warped image is drawn.



(a) (b)

Figure 1.4: (a) A reference image. (b) The reference image backwards mapped, for a new viewpoint which is up and to the right of the sample viewpoint.

There were two major difficulties with this method. First, when an entire image is warped pixel by pixel, holes appear in the resultant image (figure 1.4 (b)). This is mostly due to the fact that segments of the terrain which could not be seen from the recorded viewpoint can now be seen (since these segments did not appear in the sample image, they cannot appear in the warped image). Several hole–filling algorithms were implemented in this thesis, but these reduce rendering speed, and the final image may still be incomplete.

Secondly, the rendering time was very slow. As mentioned, this was partly due to the hole–filling procedure, but more importantly, warping every pixel of an image is an extremely expensive operation. This motivates a different approach, in which only a small fraction of the pixels need to be warped.

### 1.3.2 Meshed Impostors

As with the previous algorithm, this method begins with a pre–processing step in which images and depths are recorded from various sample viewpoints. In addition, a mesh is generated on top of each image by placing vertices on parts of the terrain that are visually important, such as the outline of the terrain, depth discontinuities, and high–contrast boundaries (figure 1.5 (c)). When viewing the terrain interactively, rather than warping individual pixels, only the vertices of the mesh are warped. The texture associated with each triangle of the mesh is automatically stretched or shrunk accordingly by the texture–mapping hardware. This solves both of the problems associated with sprites with depth; holes do not occur since the mesh remains connected, and rendering time is very fast, since there's no hole–filling, and warping calculations are greatly reduced. The texture warp within the triangles is not as accurate as a true perspective warp (since a planar triangle provides only a linear approximation to the true pixel depths), but this usually is not perceptible, particularly if the mesh triangles are small enough.

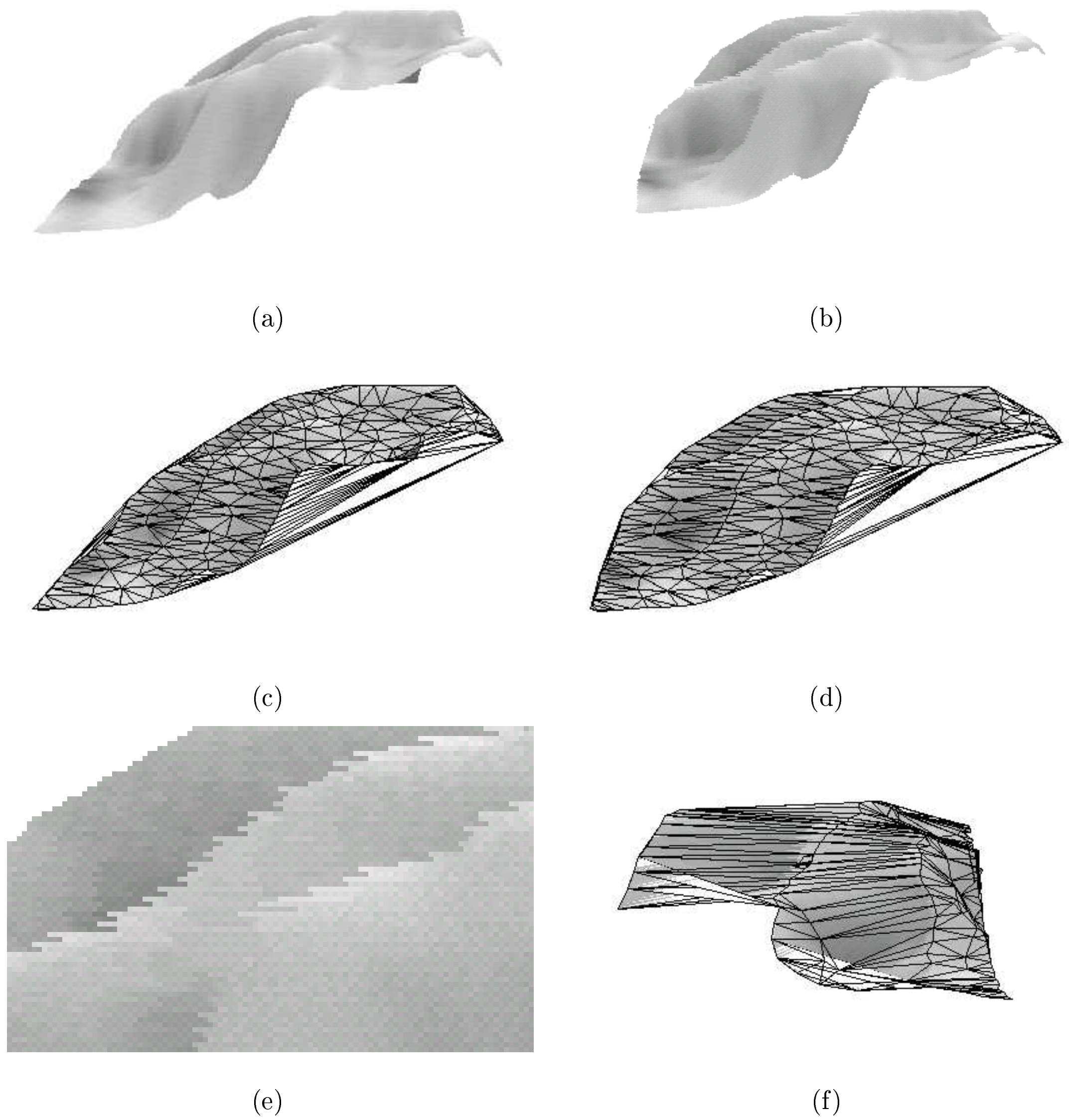


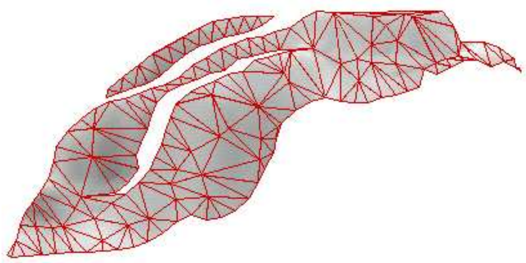

Figure 1.5: (a) An unwarped mesh impostor. (b) The warped impostor. (c) and (d): The impostors with the mesh shown. (e) Close–up of the jagged ridge lines. (f) VDTM impostor seen from side view, showing the stretched triangles caused by the depth discontinuities.

However, other artifacts arise with this method. First, extreme stretching of the textures within some triangles can cause a jagged effect on some terrain ridges (figure 1.5 (e) and (f)). Also, pieces of the terrain occluded by hills or other features in the foreground should be revealed ("**disoccluded**" in IBR terminology) as the viewpoint moves; this doesn't happen with meshed impostors.

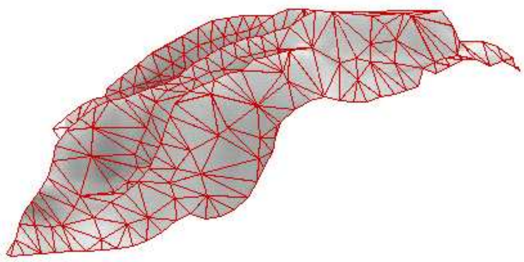
An interesting observation is that warping and drawing a textured mesh on a 2D plane is equivalent to simply drawing the mesh in 3D world space (resulting in view–dependent textured meshes, or VDTMs). This can be done since the depth of each vertex is known, and therefore the exact 3D coordinates can be calculated. In effect, this moves the software warping calculation to a much faster hardware calculation.

### 1.3.3 Unzipped Meshed Impostors

To deal with artifacts discussed in the previous section, a third element was added to the pre–processing step. After generating the mesh for a particular image, the mesh triangles are partitioned into several sets, based on distance (i.e. depth) from the viewpoint. The mesh is then "unzipped" along the boundaries between sets, and additional triangles are placed in the occluded regions that are hidden between boundaries. This avoids the jags seen in the previous method, since the large texture stretches occurred in triangles which spanned two ridges. Secondly, occluded parts of the terrain become visible as the view moves away from the recorded viewpoint (figure 1.6).



(a)



(b)

Figure 1.6: (a) A VDTM mesh after being unzipped. (b) The unzipped mesh with disocclusion triangles added.

# Chapter 2

# Previous Work

The work done in this thesis can best be classified as image based rendering. Therefore similar work in this field will be examined closely in this chapter. However, non–image based techniques also exist for simplifying terrain or polygon meshes; these methods focus on reducing the number of polygons in the mesh to reduce total rendering time, usually while keeping within certain error constraints. A brief overview of this alternative for terrain simplification will be presented. Finally, the work presented in this thesis makes use of a form of view–dependent textures. Some research has been done previously in this area, although the definition of view–dependent textures is somewhat different in those works.

## 2.1 Image Based Rendering

Image–based rendering essentially involves rendering scenes that are at least partially composed of pre–computed images (texture maps) projected onto a plane or surface. This is in contrast to pure geometric rendering, which involves scenes composed entirely of 3D geometric objects. IBR has, in a very basic form, been used for many years, but has only substantially advanced within the last six years.

### 2.1.1 Original Work

The original work in IBR began with basic texture–mapping simplifications. Simple billboards or impostors are an example of this. A **billboard** is a two–dimensional texture–mapped planar rectangle, usually centered within the object being replaced. The original object is projected onto this plane and this image is recorded as the texture map. The billboard can be fixed in place, which is only effective if the viewer is not moving much. Alternatively, the billboard can rotate to always face the viewer. A common use of this method is in representing trees in flight simulators.

Early impostors were simply texture maps pasted onto the bounding boxes of the simulated objects. Either the image of the object (or objects) is displayed on an appropriate side of the box, or all faces of the bounding box (essentially a set of fixed billboards) are texture–mapped, independent of viewer position. The latter method is effective when the object being represented is actually rectangular in shape, such as a skyscraper.

A final early method in IBR is known as environment mapping. This essentially takes a large section (typically spherical or cylindrical) of a scene and converts it to a single texture map. For example, all objects in a scene which are more than a certain distance away from the area of the viewer could be projected onto one cylindrical texture map, and displayed as a background image in front of which closer objects are drawn as geometry.

Clearly, these original methods were quite rough; they worked well with certain objects, but in most cases could not adjust well for movement of the viewer. It was often too obvious that the object was in fact an imposter. However, recent developments in image warping created a new interest in this area.

### 2.1.2 Warping

The problem of **image warping** is as follows: Given one of more "reference" images of a scene taken from known viewpoints, derive an intermediate image for a novel viewpoint. The depths of the objects in the scene must be known or computed to achieve this (for a correct perspective warp). Also, the location and direction of the camera for each of the reference images must be available. This will be discussed in more detail in the next chapter. See Wolberg's book [34] for a comprehensive discussion.

The use of warping in image–based rendering first appeared in the graphics literature in 1993, with Chen and Williams' [4] work on view interpolation. It was the first work that warped pre–stored images to form intermediate viewpoints for an interactive walkthrough. They used coherence between frames (i.e. as a user moves slowly through a scene, one would not expect the scene to change much from one frame to the next) to pre–compute pixel correspondences. These correspondences could then be used to interpolate between two images and arrive at an intermediate view. Although this achieved promising results, the pixel warping method results in holes in the final image, when an area of the scene which cannot be seen from either reference image is visible from the new view location. Various methods of interpolation are suggested to fill these holes, but this significantly affects rendering time, since they are typically done in software. Another possibility would be to choose reference camera positions that completely cover all areas of the scene, but this is a complex problem [31] which is still open.

### Cylindrical Environment Maps

Chen and Williams were, in effect, warping between two or more environment maps, since the entire scene from one point of view is represented by a single image. This idea was used again later by Chen [3] with cylindrical environment maps. To create a cylindrical map, several areas of a scene (nodes) are chosen, and images are taken in a circle around each node, spaced correctly to cover all areas of the scene. The viewer can then rotate (on a vertical axis) within the cylinder and see an accurate representation of the entire scene (figure 2.1). To move between cylindrical maps, two possibilities are available. Either the movement of the viewer must be constrained to jump directly from one node to the next, or warping must be used to generate intermediate viewpoints between the nodes. Chen [3] used environment maps created from camera images, so no depth information is available to perform the warping operation. However, work by McMillan and Bishop [23] also uses cylindrical environment maps, and warps between existing images to generate novel viewpoints. That work will be discussed in section 2.1.4.

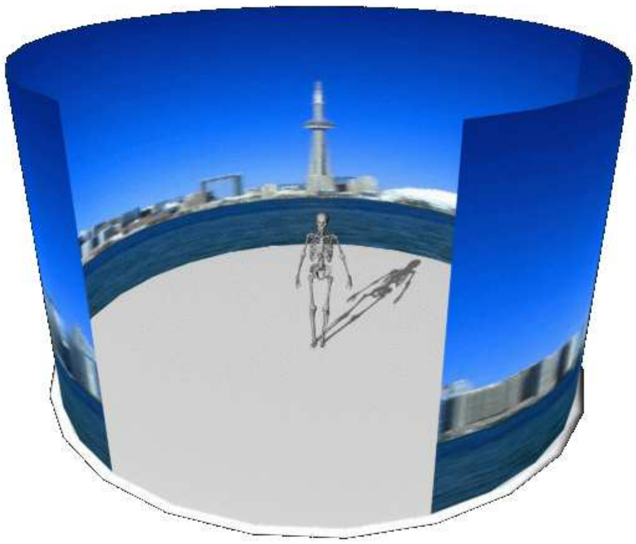


Figure 2.1: A figure inside a cylindrical environment map.

The idea of cylindrical maps was extended by Kang and Desikan [17], who used clusters of at least three cylindrical images for each discrete area of a scene, such as a room. As the viewer moves within the room, virtual cylinders are created by forward–mapping images from the recorded cylinders onto novel cylinders centered at the viewpoint. This results in holes, which are filled either with image smoothing (coloring in empty pixels with the colors of neighboring pixels) or geometric interpolation. This allows a user to move freely within an area of the scene. To move to a new area of the scene, a transition needs to be made between neighbouring clusters. This is still somewhat limited, because the user needs to explicitly request a transition, and the movement from one cluster to the next is restricted to a two–dimensional translation.

**Other Warped Environments**

Image warping is also used with several variations. Darsa *et al* [5] speed up the warping process by first performing an image–space simplification of the scenes being warped. A scene is represented with cubical environment maps, and novel viewpoints are generated by warping two adjacent environment maps and combining the results (morphing). However, image–warping is a slow procedure, and leaves holes in the final image. To solve this problem, the reference images are first simplified by polygonization; the images are triangulated, creating a mesh over the scene. The triangles are warped individually, and the resulting, overlapping triangles are blended to form the final image. This triangle combination proved to be a problem however, and the best visual results came at the cost of slow rendering time.

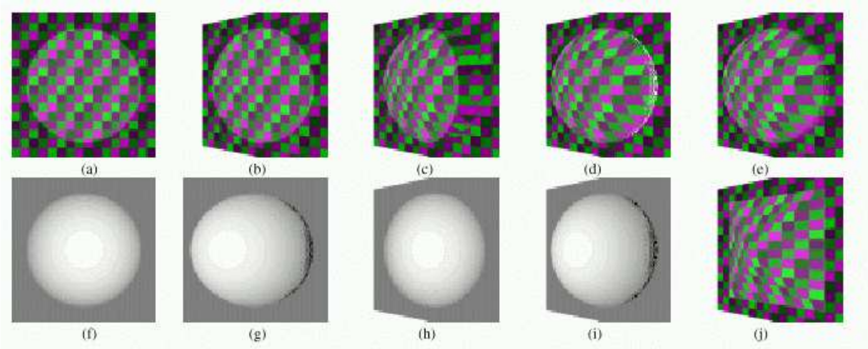



Figure 2.2: Plane with bump rendering example (Shade *et al* [27]): (a) input color (sprite) image $I_1(x_1, y_1)$; (b) sprite warped by homography only (no parallax); (c) sprite warped by homography and crude parallax; (d) sprite warped by homography and true parallax; (e) with gap fill width set to 3; (f) input depth map; (g) pure parallax warped depth map; (h) forward warped depth map; (i) forward warped depth map without parallax correction; (j) sprite with "pyramid" depth map.

Another variation is the idea of "sprites with depth" [27] (see figure 2.2). A sprite is simply a billboard: a two–dimensional image representing an object. However, sprites with depth have depth recorded, as well as the standard colour information. This depth is used to warp the sprite. A novel warping approach is suggested: Rather than forward–mapping the image, which results in holes, only the depths are forward–mapped, and the novel image can then be generated from a backward–mapping procedure that uses these forward–mapped depths. This results in fewer holes, since holes will result only from depth changes, rather than image size variations. This implementation and details of this method will be discussed in Chapter 4; however, it was found that this process is very slow, and holes in the final image still occur.

The idea of a layered depth image (LDI) is also presented in this work [27] for objects that lie closer to the viewer or have more complex geometry. LDI's are similar to sprites with depth, but rather than associating only one depth per pixel, each pixel is assigned an array of depths for each layer of the object which projects onto the pixel. This way, self–occlusions within an object can be correctly represented in an impostor. Naturally, this will still exhibit the same speed problems found in sprites with depth. Chang *et al* [2] combined LDI's with a hierarchical partition of the scene. This is done to preserve the sampling rate of the reference images, and therefore produce a constant detail level. Consider several reference images, some which are closer to an object than others. Many different LDI's can be formed from these images, from all of the different combinations. However, those reference images which are closer to the object clearly have a higher sampling rate, i.e. more detail is visible in the image. If the novel viewpoint being generated is farther away from the object than these images, then this level of detail (or sampling rate) is not necessary, and will produce unnecessary computations. Therefore an LDI tree is generated from the space partition, and can be traversed to select an LDI which has the closest sampling rate to the current virtual viewpoint. This preserves the detail and rendering time of the LDI.

### 2.1.3 Static Impostors

Not all recent research in IBR involves image warping. Some papers are extensions of the earlier forms of impostors, such as billboards or texture–mapped bounding boxes. This was first done by Maciel and Shirley [21], who first used the term "impostors." They use several types of impostors, both view dependent (billboards, images on single face of bounding box) and view independent (levels of detail, texture mapped boxes) to render a scene. To begin, the scene is hierarchically segmented with an octree. At each node, a low–cost representation of the objects within that node is stored. Cost is determined by several factors, such as the pixel size of the object on the screen, and the angle at which the viewer sees the object. Higher nodes on the tree use impostors for clusters of objects, while the lowest nodes contain the geometry of individual objects. To render the scene, a traversal of the scene which does not exceed the cost of a user–defined value is selected. A limitation of this method is that to choose the best traversal, a particular node must have information about the (viewpoint–dependent) cost of its children. This means every node must be visited for each frame, slowing down rendering.

Hierarchical representations of a scene are also seen in Shade *et al* [28] and Schaufler and Stuertzlinger's [26] work. Both use an algorithm to partition the scene, and generate impostors for each level of the partition (called "image caching"). In Shade *et al* [28], the imposter (a 2D texture image) is displayed on a quadrilateral centered on the position of the original object. Schaufler and Stuertzlinger [26] display the image on a dynamically chosen face of the object's bounding box. In each case, the impostor is updated when it is determined that the screen–space error exceeds a certain threshold. A new impostor can then be generated from the child impostors in the hierarchy or, if necessary, by refreshing the cached image. This method is effective for very large scenes which cannot be interactively viewed using full geometry. When a small screen error threshold is used, the walkthrough does not look noticeably different than using full geometry, and the rendering speed significantly increases. However, unless the error threshold is reduced, the frame rates are still not interactive, but with lower thresholds, the artifacts are noticeable. In addition, this algorithm requires large amounts of memory to store the entire hierarchical cache.

A different approach is presented in Schaufler's [25] paper on layered impostors (figure 2.3). In previous work, a single impostor was used to represent at minimum a single object. Schaufler represented an object with a layered impostor which is a pyramidal stack of textured billboards perpendicular to the view direction, with the apex of the pyramid facing the viewer. To create these layered impostors, the object is rendered and the pixels of the image are then placed on the closest billboards, which has the effect of dividing the pixels into sets, based upon depth. As the viewpoint moves, these layers move relative to each other, which creates the effect of parallax. To avoid holes showing up instantly as the viewpoint moves, the images on each layer overlap slightly. This allows parallax effects to appear without letting the background show through. Clearly, the more layers that exist, the more accurate the effect will be. Despite the overlapping, the impostors need to be updated after the user exceeds a certain angle from the original viewing direction. This algorithm has two problems. First, holes will appear as usual, when previously occluded parts of the object should be seen from the current viewpoint (since they could not be seen from the sample viewpoint), they could not be included in any of the impostor layers). Secondly, the number of layers should be chosen automatically based on some error calculation, rather than letting the user choose.

Finally, the idea of a three–dimensional meshed imposter is presented by Sillion *et al* [29], and extended by Decoret *et al* [9]. The objects (buildings) in an urban scene are first partitioned into sets, based upon depth. Each set of buildings is then drawn in pre–processing step, and the resulting image is stored and meshed. The depth information is also recorded, so in interactive viewing, the textured mesh is drawn in 3D world space. As the viewer moves, nearer objects are drawn in full geometry, while impostors are used for distant clusters. The three–dimensional impostors are valid for a longer period of time than simple image caches, so they need to be updated less frequently. However, some

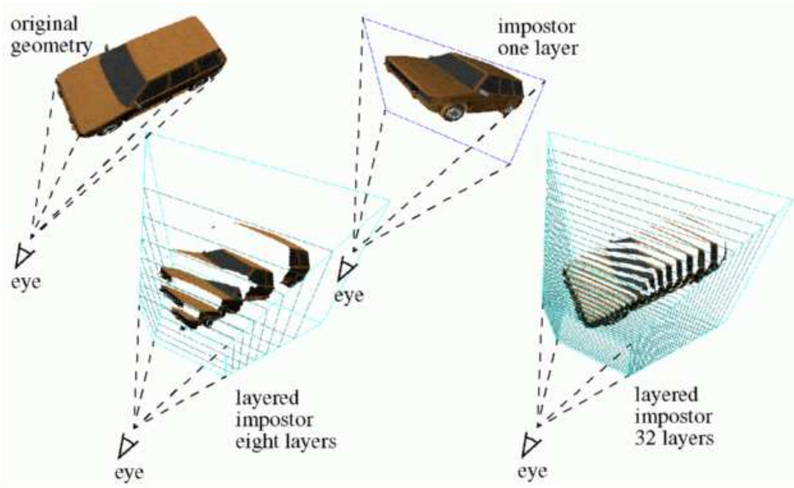



Figure 2.3: Schaufler's Layered Impostors.

problems exist, which are addressed in the more recent paper [9]. The two major problems arise as the viewer moves through the environment: (1) Areas of the scene not visible from the sample viewpoint are not filled in as they become disoccluded. (2) Meshes which stretch across two objects in the scene cause a "rubber sheet" (stretching) effect as the viewpoint moves. These two artifacts are dealt with by using multiple meshes with error checking: The environment is divided into *viewing cells*, and a suitable impostor is constructed for each cell so that the impostor will look reasonably correct from all points within the cell. The error calculation (to determine what is "reasonably correct") essentially calculates the amount of disocclusion that can occur while moving within a particular cell. When large disocclusions can occur between objects, the impostor will need to be more detailed, i.e. separate objects will need to be meshed rather than clusters.

### 2.1.4 More Theoretical Models

Several papers have presented methods which attempt to warp images using using complex physical models. In a relatively early work, McMillan and Bishop [23] present the plenoptic function, a five–dimensional function which describes all image information that can be seen for a particular view position and direction. That is, the plenoptic function describes the flow of light for a particular point (x, y, z) and viewing direction ($\theta$, $\phi$). They describe the goal of IBR as defining a continuous representation of the plenoptic function, given a set of discrete samples. In an attempt to implement this, pre–stored images are projected onto cylindrical image planes around the viewer. Intermediate views between cylinders are computed by reconstructing a complete sample of the plenoptic function. This method was limited because the sample nodes (cylinders) needed to be close together, and user input was needed to properly match nodes.

Gortler *et al* [13] and Levoy and Hanrahan [18] developed the idea of the light field (or lumigraph), which is essentially a method to capture and redraw the plenoptic function from several points in space. However, it is assumed that there are no occluders, and that the air is transparent, so the radiance along a ray remains constant and the plenoptic function is reduced to four variables. After a four–dimensional parameterization is chosen, a large database of values is generated by shooting all rays which fit the parameterization. To generate a new viewpoint, the values in the database are interpolated for the current viewpoint. While this results in accurate images when the sampling is sufficiently dense, the database filling process is very slow, memory usage is very heavy, and this has only been applied to single objects rather than entire scenes. To reduce the database size, several methods such as vector quantization and wavelet decompostion have been applied. This reduces the size by a factor of 20, with a future goal being a factor of about 200.

## 2.2 Mesh Reduction

Most objects which compose a scene in an interactive walkthrough are modeled with a polygonal mesh, usually triangles. To draw the scene, the vertices of each polygon must transformed to calculate the screen coordinates, and then pixels of the polygon must be filled. Clearly, reducing the number of polygons in the scene would help decrease the rendering time for each frame.

Early work in this area involved off–line reductions: that is, algorithms which took a mesh as input and produced as output a new mesh with fewer vertices. These algorithms usually involve collapsing edges or vertices to generate larger polygons, or removing edges to create holes, and then filling the holes with fewer polygons than were originally there. Of course, one wants the reduced mesh to closely approximate the original mesh, preferably within a user–defined error bound. Maximizing the number of polygons eliminated while minimizing the error is the primary focus in this work.

However, these methods are not relevant to interactive viewing, except of course that the scene can be pre–simplified before beginning the interactive stage. We are more interested in methods that dynamically change the level of detail of the mesh as the user moves through the environment. A fair amount of work has been done in this area in recent years, as the demand for realism in applications such as games and virtual reality increased quickly.

Most level–of–detail algorithms generate a hierarchical representation of the of the mesh and choose a level of representation that is best for the current viewing parameters [32, 14, 11, 6, 35, 19, 15, 10, 16, 12, 20]. The top of the hierarchy represents the coarsest version of the entire mesh, while the leaves represent an exact (finest detail) section of the mesh. The nodes contain intermediate levels of detail; a single node contains a simplified version of the segments of the mesh contained within its children. The primary goal of a level–of–detail renderer is to determine which nodes in the hierarchy to render, to maximize accuracy while still maintaining a certain rendering speed.

Many different methods exist for creating such a hierarchy. Hoppe's Progressive Mesh [15] representation is created by starting with the original mesh, and performing a series of edge collapse operations. Collapsing a single edge removes one edge, one vertex, and two faces from the current mesh. Note that this is an invertible operation; the same edge can be replaced to get back to the original mesh. An edge to remove is chosen by a set of criteria, such as the overall distance of all new points to the original mesh, and the preservation of discontinuity curves in the mesh. The final progressive mesh consists of the coarsest representation of the mesh, and a list of vertex split operations (inverse of edge collapse) which restore the original mesh step by step.

Lindstrom *et al* [19] use a triangle combination step, which reduces two neighbouring triangles into a single triangle. Two triangles are combined if the change in slope between them is minimal. For each frame, triangles are combined until the overall error of the scene is too large. This algorithm also makes use of an "active list" [20, 35] of all triangles that are displayed at the current level of detail. As the viewer moves through a scene, one would expect the level–of–detail from frame to frame not to differ very much. Therefore, if one keeps track of an active list, only small modifications need to be made to it for each frame, rather than recomputing the entire list each time.

Level–of–detail methods often attempt to address the following qualitative characteristics:

- Screen space error should be minimized. This is the basic priority for all mesh reduction algorithms, although the measure and methods differ.

- Silhouettes are particularly noticeable features in terrain, so detail should be greater in these regions [20, 15]. For example, a distant ridge may originally be chosen for simplification, since the distance results in small screen space error. However, even at a distance, a straight–edged ridge is much more noticeable than simplifications within the terrain. Therefore these areas should be given higher priority for triangles.

- Triangles that cannot be seen by the viewer from the current position should be culled [20, 30]. In terrains there are many cases in which triangle that are not backfacing still cannot be seen by the viewer, and clearly rendering time is decreased if these triangles can be eliminated quickly.

- Illumination should be preserved. Since illumination is determined by the triangle normals, reducing the mesh affects lighting. Therefore Xia and Varshney [35] suggest that greater detail should be used in areas where illumination changes sharply.

- Level of detail transitions should be smooth to avoid sudden pops in the mesh. Pops occur while the viewpoint is moving and the terrain suddenly switches detail levels, so there are suddenly a different number of triangles. This detail jump can be very noticeable. Several structures solve this problem, such as Hoppe's progressive meshes [15]. He defines a geomorph, which is a visually smooth transition between levels of the mesh created with linear interpolation between vertices.

## 2.3 View–Dependent Textures

When one is using texture mapping on a simple object, such as pasting a brick texture onto a wall, it is not necessary to change the texture as the viewpoint moves. However, for more complex objects, the texture may need to vary as the viewpoint moves. There are two main reasons that a texture may need to change. First, the new texture may need to show parts of the object which were not visible from the original viewpoint. Second, lighting may be view–dependent, and changes due to lighting should be reflected in the new textures. This is the basic idea of view–dependent textures; the texture on the object changes as the viewpoint moves.

The term view–dependent textures was first introduced by Debevec *et al* [8], for use in rendering buildings from photographs. After recovering geometry camera positions from the photographs, the pictures were pasted onto the buildings with texture maps. However, no picture contained the building in its entirety, so several textures needed to be blended to form the entire image. This compositing of textures was performed in a view–dependent manner: When a certain pixel was covered by a single texture, then that texture was used normally; but if more than one texture covered a pixel, the texture were blended with a weighting factor based on the angle between the current viewing position and the recorded positions. While this produced an accurate image, the per–pixel operations resulted in a rendering time of several minutes per frame.

Mark *et al* [22] and Pulli *et al* [24] also derive a virtual frame of a scene by combining pre–stored textures. However, the rendering speed is increased by first meshing the reference images and warping the mesh rather than individual pixels. Therefore, to create a new image, the relevant reference images are meshed and warped. Then for each pixel, either a decision is made of which warped triangle should be used, or the images are composited in a view–dependent manner. Using this mesh method, interactive frame rates were achieved.

Finally, Debevec *et al* [7] enhanced their original work in two ways. First, for each polygon in the scene, a pre–processing step computes a list of all reference images that contain that polygon. Then in the rendering step, the three nearest images are chosen, and weights are calculated for each. A polygon is then rendered in a three–pass process, which pastes weighted textures onto the polygon from each of the three reference images. This method was also able to operate at interactive frame rates.

## 2.4 Closely–Related Work

The work in this thesis uses ideas from work in image–based rendering and view–dependent textures. Both of these areas can be thought of as alternatives to mesh reduction methods. The implementation sections begin with sprites with depth, which is taken directly from Shade *et al* [27]. This was found to be far too slow, so ideas found in view–dependent texturing papers ([22, 24]) were incorporated (the addition of a screen–space mesh to speed up the warping process). The view–dependent textured meshes of Chapter 5, which use hardware perspective transformations rather than screen space image warping, are most closely related to the meshed impostors used for urban scenery [29]. These use depth discontinuities to generate meshes for images, which are then projected into 3D world space. Further work in this area [9] used multiple meshes to enhance parallax effects and reduce artifacts due to mesh stretching. This is somewhat similar to the unzipped mesh impostors seen in Chapter 6 in this thesis, but substantial differences exist.

The principal contribution of this thesis is the idea of unzipped mesh impostors. A secondary contribution is a classification of rendering methods, described in the next chapter.

# Chapter 3

# The Image–Based Rendering Spectrum

Most image–based rendering algorithms can be classified by the categories shown in the rendering spectrum (figure 3.1). As will be discussed later, it was found that the two major divisions in the methods (mesh based and billboard based) contained algorithms which are essentially equivalent. All of these methods are now discussed.

## 3.1 Billboard Based Rendering Methods

As described in Chapter 2, image–based rendering began with texture maps placed on planar objects such as billboards or bounding boxes. Naturally, this evolved into more sophisticated methods, but many later methods still had the same basis. These are thus classified as billboard based rendering algorithms.

### 3.1.1 Texture–mapped Billboards

As discussed in Chapter 2, IBR began with simple texture mapped billboards, such as images pasted onto faces of an object's bounding box. The use of these impostors was originally limited to only types of objects (such as a house) which can be represented relatively accurately in such a manner. However, more recent work in hierarchical image caching [28, 26] (section 2.1.3) extends this idea into a more robust system.

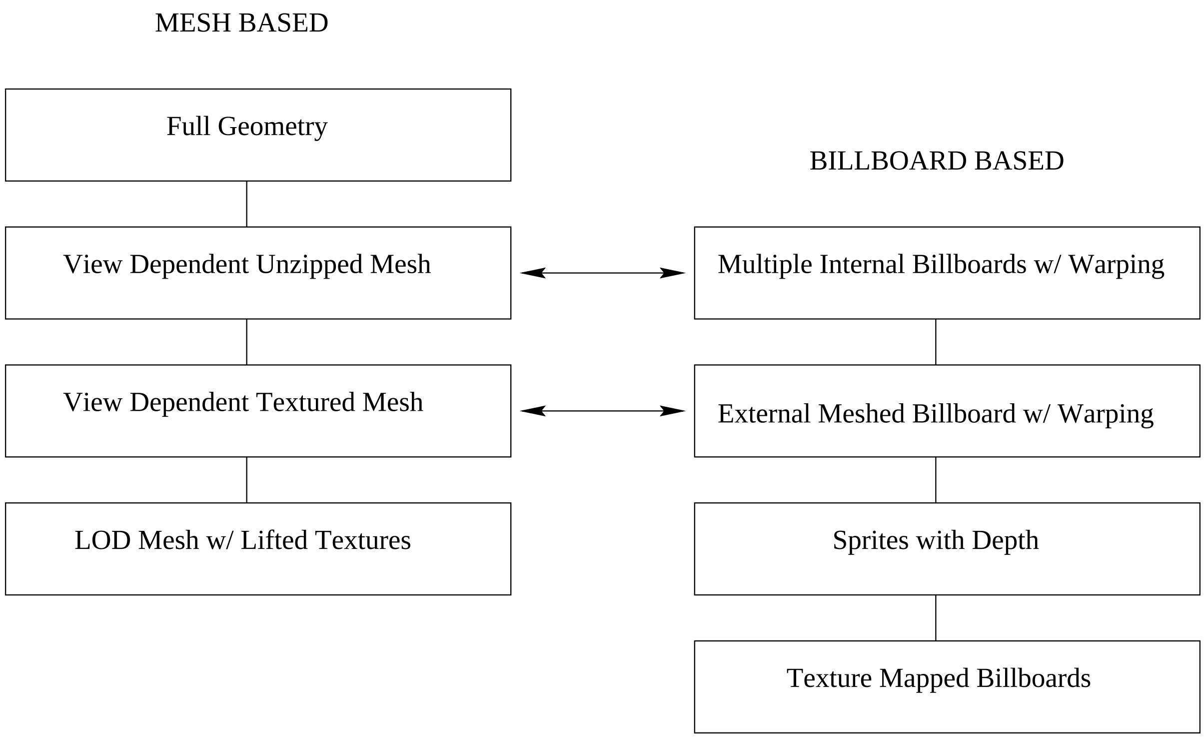



Figure 3.1: The image–based rendering spectrum, ordered by quality of the resulting images. Currently, this quality measure ignores the "popping" seen in VDTMs and unzipped meshes, since this is a correctable problem (see section 8.2). In a small area around a sample viewpoint, these two methods produce more accurate images than LOD terrains with lifted textures. Arrows indicate that the methods are equivalent.

### 3.1.2 Sprites With Depth

Sprites with depth are billboards with depth information [27]. They are created in the following manner. First, a position and direction in world space (the coordinate system of the objects being viewed) is chosen. The object is then projected onto an image plane and recorded as a texture. The depths can then be recorded from the z–buffer. This is done for several viewpoints around the object.

To use the sprite, an optimal image for the current viewpoint is projected onto a billboard in world space. As the viewpoint moves, the image on the billboard is perspective–warped, using the depth information to create the intermediate view. The details of warping will be discussed in the following chapter, which describes the implementation of sprites with depth.

Any IBR method which uses full pixel warping of a scene (or a part of a scene) can be thought of as a sprite with depth. This includes Chen and Williams' view interpolation work [4] as well as methods such as cylindrical environment maps [3, 23, 17].

### 3.1.3 External Meshed Billboard with Warping

External meshed billboards address the two main problems associated with sprites with depth (slow rendering time and image holes). The full implementation issues are discussed in Chapter 5.

This method is initially the same as sprites with depth: An object is projected onto image–planes from several viewpoints, with the images being saved as textures, and the depth (z–buffer) information being stored for each view. However, rather than simply warping all pixels in the image, a mesh is first created (in a pre–processing step) for each texture (an idea used by Darsa *et al* [5]). The method for creating this mesh is highly dependent on the type of object being represented. Mesh creation for terrains will be discussed in section 5.1.

Once the image plane mesh is created, it is then warped on the image plane to create novel images as the user moves through the environment. This warping procedure is similar to that used with sprites with depth, but greatly simplified. Rather than warping each individual pixel and filling holes, only the vertices of the mesh are warped. The textures within the mesh triangles are then warped in hardware, with OpenGL texture mapping. This mesh warp greatly increases rendering speed, and avoids the the problem of holes. However, while the mesh vertices are correctly perspective–warped, the texture warp is affine in the 3D plane of the 3D triangle, resulting in less accurate images within the triangles.

As with sprites with depth, this method does not allow for disocclusions within the image–based object. Only the parts of the object visible from the reference viewpoint will be seen in the warped object.

### 3.1.4 Multiple Internal Billboards with Warping

Multiple internal billboards can be used to solve the two main deficiencies of a single external meshed billboard. First, the inaccuracies resulting from an affine warp of the textures can be somewhat alleviated with multiple billboards. With multiple billboards, each billboard represents a certain range of depth values. The object is partitioned by depth into these billboards: Therefore, within a particular billboard, there are no large depth disparities and an affine warp is a good approximation of the true perspective warp.

The second problem is that of occlusions within the object. With a single billboard, no disocclusions will occur when the viewpoint moves. However, with multiple billboards, a billboard which is occluded by a nearer one can become visible as the viewpoint moves. In this way it is possible for areas of the object which were not previously visible to appear as the viewpoint moves.

This method was not implemented, since it was found to be virtually equivalent to unzipped meshes (see sections 3.2.2 and 3.2.3). The closest work in the literature is Schaufler's layered impostors [25]. This divides objects into multiple internal layers (with overlapping areas), but not perform any meshing or warping (section 2.1.3). This is in fact the norm; internal billboards are usually used only to simulate a warping effect. An object is divided by depth into slices, and the parallax effect seen between these slices will approximate a warp or normal geometric transformation. However, this cannot be as accurate as warping, since the slices will just represent an average for all of the pixels drawn on the slice.

## 3.2 Mesh Based Rendering Methods

Billboard based methods of IBR project objects onto a 2D plane and then use some form of 2D warping to create intermediate views of the object. Mesh based methods focus on simplifying the original polygonal mesh of the object, creating an "imposter" which is still three–dimensional and lies generally inside the space occupied by the original object. Warping is not necessary, since the imposter is 3D; parallax effects result from the hardware perspective transformation (i.e. the projection of 3D points onto the 2D image plane).

### 3.2.1 Level of Detail Mesh with Lifted Textures

Level of detail meshes ([15, 19, 20, 35]) were discussed in Chapter 2. This method involves simplifying the mesh by reducing the number of triangles, often dependent on view position and direction. However, it was not mentioned how the new larger triangles should be textured. It is possible to simply use the colours on each vertex and interpolate the entire triangle colour from this information. This is not very accurate, and is not in fact an image–based method. More commonly, a texture is created for the new triangle by *vertically* projecting the textures of the replaced triangles onto the new one (figure 3.2). The texture on the original triangles is created from a single top view image of the terrain, therefore the projection onto the new triangle is a simple vertical "lifting" of the texture (independent of view position). As one can see in figure 3.2, this results in inaccurate textures for all positions other than directly above the scene.

### 3.2.2 View Dependent Textured Mesh

Although LOD meshes often simplify the mesh in a view–dependent manner, the new textures generated are not dependent on the viewpoint. View dependent textured meshes (VDTM) solve this with both view–dependent meshes and view–dependent textures [29]. To create a VDTM, the object is projected onto the image plane from a series of sample viewpoints. For each, a simplified mesh on the image

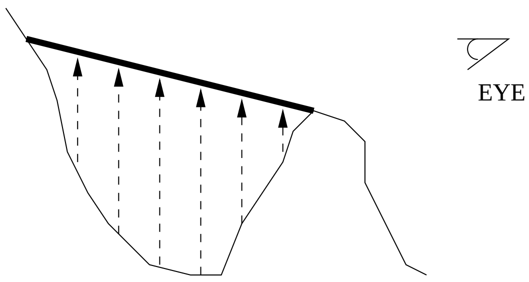



Figure 3.2: An LOD mesh with lifted textures. The texture for the new triangle is created by lifting (shown by the arrows) the lower textures onto the larger triangle (bold line). From the shown viewpoint, areas of the valley can be seen on the larger triangle, although they should be occluded by the nearer ridge.

plane is created for the object. (Creation of this mesh will be discussed in Chapter 5.) Textures are stored for each sample viewpoint, as well as depths at the mesh vertices. This simplified mesh and associated texture is then projected into world space using the depth information to calculate the 3D vertex positions.

As a viewer moves around the object, an optimal sample view point is chosen, and the imposter generated for that view is displayed in 3D. This imposter thus contains a view–dependent mesh coloured with view–dependent textures.

*This method is equivalent to that of external meshed billboards with warping.* Note that the impostors for both methods are created in an identical fashion: The mesh and textures are generated for sample viewpoints around the object. The only difference is in the way that the depth information is used: In external billboards, the depth is used to warp the mesh vertices on a 2D plane, to create the correct parallax effect; in VDTM's, the depth is used to project the mesh vertices into world space coordinates. This changes the warping operation into a hardware–supported perspective projection, with the same final effect. Using VDTM's is generally a better choice for impostors, since the hardware perspective transform is faster than the software warp.

### 3.2.3 View Dependent Unzipped Mesh

VDTM's exhibit the same problems as external billboards with warping: incorrect affine warps of the textures within the mesh triangles, and lack of disocclusions. The goals of unzipped meshes are therefore the same as those of multiple billboards. *In fact, unzipped meshes are equivalent to multiple billboards with warping.*

To create an unzipped mesh, a VDTM is created, and the mesh triangles are partitioned by depth.

The mesh is then separated into these discrete depth sets ("unzipped"), and new triangles are added for areas which are occluded from the sample viewpoint. These new triangles are assigned 3D coordinates, as is the entire simplified mesh. Therefore, as with VDTM's, warping is replaced by perspective transformations in hardware. This unzipped extension of VDTMs is somewhat similar to the multi–layered [9] extension of Sillion *et al*'s meshed impostors [29].

Unzipping the mesh is equivalent to splitting a single billboard into multiple billboards. Filling in occlusion information is accomplished in the same manner as well; it is only the display method which differs. Again, unzipped meshes are more efficient than multiple billboards, because of hardware versus software calculation time.

### 3.2.4 Full Geometry

The final, most accurate mesh–based method of displaying an object is to simply draw the entire, fully–detailed geometry. Of course, this can possibly take a long time for very complex objects, which is what the other methods attempt to overcome.

## 3.3 Conclusions

There are two basic types of image–based rendering methods for object simplification: billboard based and mesh based. These two methods result in two different types of impostors: 2D impostors projected onto a plane, which may use image warping, and 3D impostors which are projected into world space and substitute hardware perspective transformations for software warping. Within each of these two groups, there is a spectrum of methods which can be used, which generally build upon each other to generate more accurate object impostors. It was shown that some of these methods from different groups are theoretically equivalent, with only minor practical differences. The next chapters will discuss the four methods which were implemented: Sprites with depth, external meshed billboards with warping, view–dependent textured meshes, and view–dependent unzipped meshes.

# Chapter 4

# Sprites with Depth

The first method to be implemented was sprites with depth. The general algorithm has been discussed in previous chapters, but to briefly reiterate, there are two preprocessing steps:

- Select reference viewpoints from which to record images of the object.
- For each reference viewpoint, project the object onto one or several image planes, recording the image texture and all information necessary to perform a warp on the texture (depth buffer and camera matrix).

As the user moves through the scene, the scene is rendered from a set of novel viewpoints. At each such viewpoint, there are three steps:

- Choose a reference viewpoint which will be used to generate the image for the novel viewpoint.
- Warp all pixels in the texture associated with the reference viewpoint.
- Apply a hole–filling algorithm to the warped image to fix gaps due to disocclusions or an increase in image size.

## 4.1 Selecting Sample Points

The sample view points are evenly placed on a hemisphere surrounding the terrain (figure 4.1). The size of the hemisphere was chosen to be large enough to encompass the whole terrain. Eventually (see section 8.2), these impostors are meant to be incorporated into a system that uses geometry for close parts of the scene, and impostors for the more distant parts. The radius for the sampling hemisphere represents the minimum distance at which the current quad would be represented as an impostor rather than using the geometry. Viewpoints that are farther away than those on the hemisphere will be able to see more of the terrain than that which was visible from the sample viewpoints, but these disoccluded areas will be handled in a later implementation (see Chapter 6). Therefore, it is sufficient to take sample images from only this one hemisphere, which represents the closest possible viewpoints to the impostor.

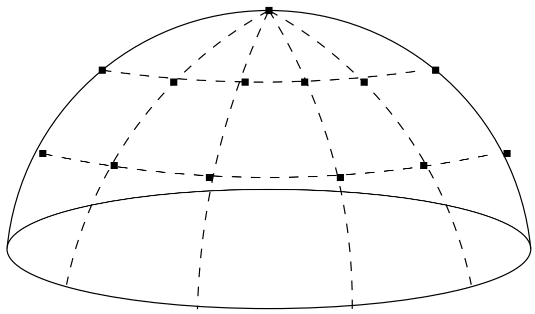


Figure 4.1: Sample view points evenly spaced on the hemisphere

Naturally, only one image needs to be recorded at the top of the hemisphere. Ideally, one would like to place the points in such a way that guarantees total coverage of terrain as one moves both horizontally and vertically. For example, consider a valley in the terrain. It is possible that the view of the valley is obscured by a valley wall from one position, and then obscured by the opposite wall from the adjacent position. However, this is a complex problem which was not dealt with in this thesis (see Stuerzlinger [31] for some initial steps in this direction). These areas of occlusion were dealt with in a different fashion with unzipped meshes, discussed in a later chapter.

## 4.2 Recording Images

A texture, the depth buffer, and the camera matrix need to be stored from each reference viewpoint: To begin, an image plane (or planes) is chosen to project the terrain onto. The original choice was a cube of planes surrounding the terrain (figure 4.2). It was first determined which sides of the cube were visible from the current viewpoint, and the terrain was then projected onto those planes one at a time. This cubic design is useful since, when mixing impostors and geometry, a cubic impostor fits exactly into a terrain quad (figure 4.3). However, this method led to complications at the seams of the cube (e.g. the warped texture would move outside the boundaries of the cube). It proved unneccesary to use such a cube in implementing the meshed based impostors (VDTM's and unzipped meshed), which are preferable. Therefore, only a newer method will be discussed below.

In the newer method, an image plane is chosen which is perpendicular to the line between the

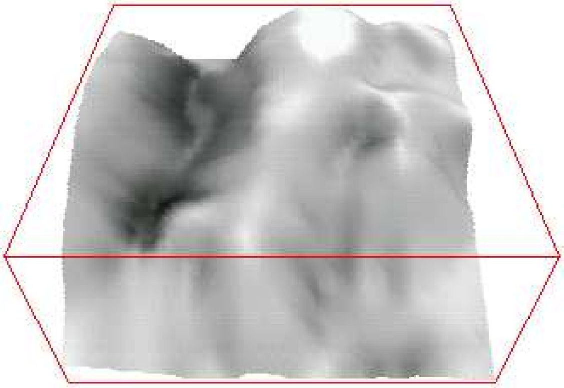


Figure 4.2: Original image planes form a cube surrounding the terrain

current eye position and the centre of the terrain (this is done with OpenGL calls to `gluLookAt` and `gluPerspective`). The terrain is rendered with full geometry (i.e. all triangles are perspective projected onto the image plane, so that an image is generated that is exactly accurate for the current viewpoint). Then the framebuffer is read and stored as a texture, the depth buffer is captured, and the camera matrix for the current view position and direction is recorded.

The camera matrix is formed by multiplying the OpenGL projection matrix (generated with the `gluPerspective` call) with the modelview matrix (generated with `gluLookAt`). This matrix transforms points in world coordinates to points in the clipping coordinate system, (i.e. four–dimensional points of the form $(wx, wy, wz, w)$). When this matrix is inverted, it can be used to transform points in normalized device coordinates (which lie in the range ([-1,1], [-1,1], [-1,1])) into points in world coordinates. This will be used in image warping, and later to reproject the screen–space mesh into world coordinates (for VDTM's and unzipped meshes).

## 4.3 Selecting a Reference Point

After all relevant information from the sample viewpoints is recorded, the interactive viewing of the terrain can begin. An impostor must be generated for the current viewing position of the user. To begin, a reference viewpoint is chosen. The obvious choice would be the nearest point on the hemisphere of sample points (figure 4.4). However, this is not necessarily the best choice. It was found experimentally that choosing a sample point *above* the virtual viewpoint results in better images. There are two reasons for this: First, when the virtual viewpoint is below the sample viewpoint, warping the sample image

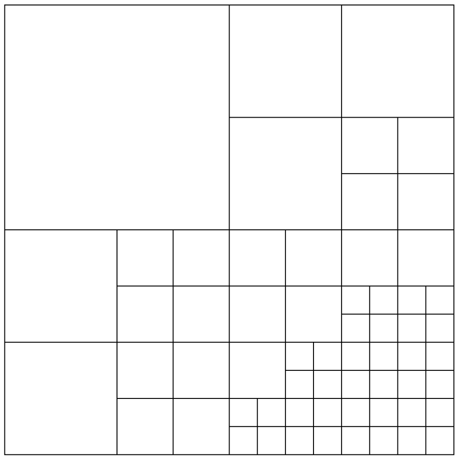


Figure 4.3: Terrain simplification is often done by hierarchically dividing the height field into quads. At each step, a choice is made to draw the terrain at the current level of detail with a certain method, or to continue to recurse.

contracts it vertically, whereas the warping process stretches the image if the virtual viewpoint is above the sample. The stretched images had much more obvious artifacts than the contracted images. Second, the higher sample images are able to see more of the occluded areas in the terrain, and therefore one wishes to switch to this view as soon as possible.

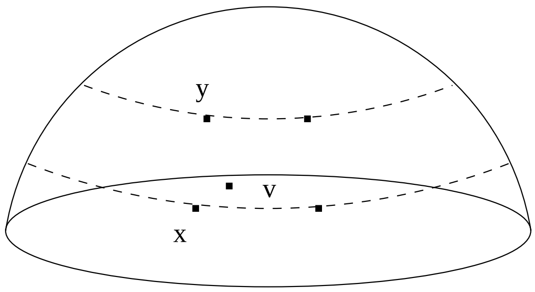



Figure 4.4: a reference image must be chosen for virtual viewpoint $v$. Point $x$ is the closest, but $y$ is chosen since it lies above $v$. Note: If $v$ does not lie on the hemisphere, $v$ is redefined as the nearest point on the hemisphere to the actual virtual viewpoint.

However, we cannot make such predictions about horizontal movement, and therefore the sample point chosen is that which is nearest horizontally, and higher.

## 4.4 Warping

Once the sample viewpoint has been selected, an image for the novel viewpoint can be generated with a warping algorithm. There are two basic warping procedures: forward–mapping and backward–mapping. Forward–mapping involves mapping each pixel in the reference image $I_1$ to a new pixel location in the generated image $I_2$. This is not, however, a one–to–one mapping, and the resultant image has holes which then need to be filled. This issue is addressed with backward–mapping, which maps each pixel in $I_2$ to a pixel in $I_1$. This guarantees that all pixels in the new image are filled. However, this cannot be done as readily, because calculating the mapping from a pixel $p_2$ in $I_2$ to the corresponding pixel $p_1$ in $I_1$ requires that the depth at $p_2$ be known.

The method used is similar to that developed by Shade *et al* [27]. The basic idea is to forward map the depths, fill in holes in the resulting depth image, and then use these new depth values to backward map the image.

### 4.4.1 Forward Mapping

A reference image has been chosen, which has an associated camera matrix $C_1$ that maps points in world coordinates, $(X, Y, Z, 1)$ to points in normalized device coordinates (NDCS), $(x_1, y_1, z_1, 1)$:

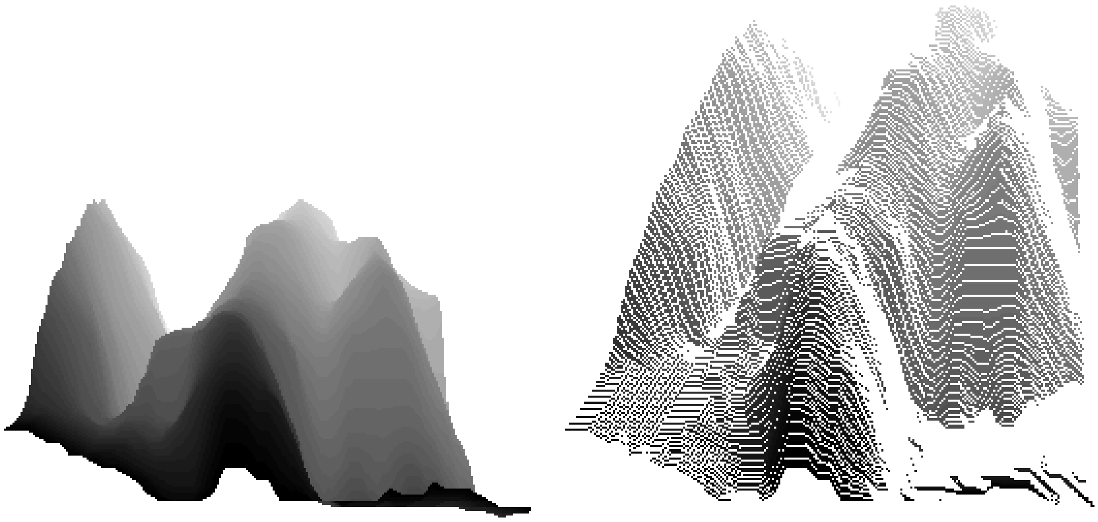
$$\begin{bmatrix} w_1 x_1 \\ w_1 y_1 \\ w_1 z_1 \\ w_1 \end{bmatrix} = C_1 \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} \tag{4.1}$$

Note that $w_1$ needs to be divided out to transform from the clipping coordinate system (CCS) to NDCS. Similarly, if $C_1$ is inverted, then $C_1^{-1}$ maps points $(x_1, y_1, z_1, 1)$ in NDCS into points (X, Y, Z, 1) in WCS.

A camera matrix $C_2$ is generated for the virtual viewpoint by multiplying the current projection matrix with the current modelview matrix. The transformation matrix $T_{1,2}$ is generated by multiplying $C_2$ and $C_1^{-1}$: $T_{1,2} = C_2 C_1^{-1}$. This maps NDCS points in $I_1$ to NDCS points in $I_2$:

$$\begin{bmatrix} w_2 x_2 \\ w_2 y_2 \\ w_2 z_2 \\ w_2 \end{bmatrix} = T_{1,2} \begin{bmatrix} x_1 \\ y_1 \\ z_1 \\ 1 \end{bmatrix} \tag{4.2}$$

Therefore, we can now apply this transformation to each pixel $(x_1, y_1)$ with associated depth $z_1$ in $I_1$, to generate a new set of points in $I_2$. We could, at this point, simply fill all resultant pixels $(x_2, y_2)$ with the colour at $(x_1, y_1)$. However, as reported by Shade *et al* [27], performing hole–filling in the image can have poor results, compared to hole–filling in the depth buffer. This is because objects generally vary more smoothly in depth than in colour. Therefore, we use the forward–mapping operation only to generate a set of depths (which has holes) consisting of a depth, $z_2$, at each pixel, $(x_2, y_2)$. An example of a forward–mapped depth image is shown in figure 4.5. Clearly the depths for the new image $I_2$ need to be filled in.



(a) (b)

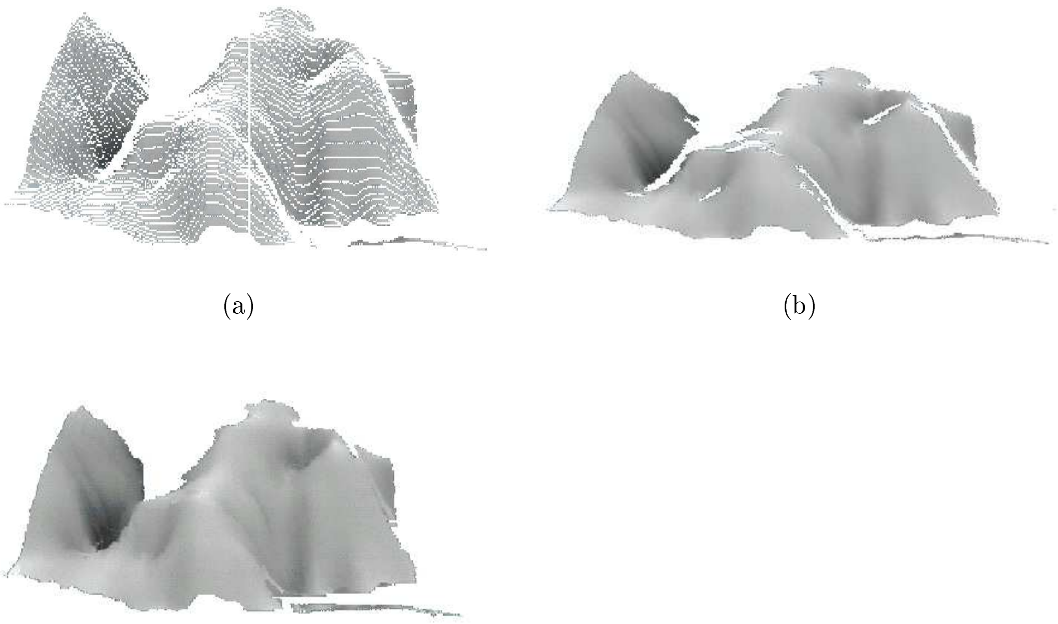
Figure 4.5: (a) The depth buffer of a reference image. Darker areas correspond to closer depths. (b) The buffer forward–mapped for a new, higher viewpoint.

### 4.4.2 Hole Filling

Two algorithms were implemented for hole–filling:

#### Eight Neighbour Average

This algorithm scans through each pixel in the depth image of $I_2$. If the pixel has no depth yet, all eight neighbouring pixels are examined, and the depth of each, if it exists, is recorded. If five or more of the neighbouring pixels have a depth, then the current pixel is set to the average of those. Otherwise nothing is done. It is necessary to set this lower limit of five neighbours with depth, because without this, the entire depth image would be filled. The result of this is that small holes, which are at most several pixels wide, are filled, but larger ones are untouched. This is shown in figure 4.6 (b).



(a) (b)

(c)

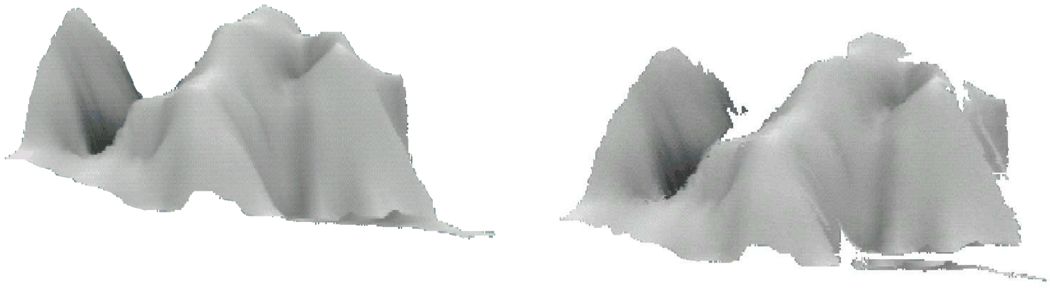
Figure 4.6: (a) The backward–mapped terrain image when no hole–filling was performed on the depths (after forward–mapping). (b) After using the eight neighbour algorithm. (c) After using the outline algorithm.

**Image Outline**

This algorithm first attempts to determine the outline of the depth image, by scanning in from each side of the image until a pixel with a depth is found. Until a such a pixel is found, each pixel encountered (along the current scanline) is said to be outside of the image. After this process, the border of the image is completely defined. Now the eight neighbour algorithm is run again, this time only on unfilled pixels within the border. These pixels can be filled regardless of the number of neighbours which have a depth. Therefore the larger holes can be filled. However, as seen in figure 4.6(c), only internal holes are filled. Holes which do not have a complete border are indistinguishable from the actual edge of the terrain, and cannot be filled.

### 4.4.3 Backward Mapping

Assuming that we have all depth values, $z_2$, for the new image, $I_2$, we can backward–map into $I_1$ to determine the colours of all pixels in $I_2$. This is done in a similar fashion as before, but with a new transformation matrix $T_{2,1} = C_1 C_2^{-1}$. This transforms points $(x_2, y_2, z_2, 1)$ in $I_2$ to points $(w_1 x_1, w_1 y_1, w_1 z_1, w_1)$ in $I_1$. With this, we can now calculate a pixel $p_1 = (x_1, y_1)$ in $I_1$ for each pixel $p_2 = (x_2, y_2)$ in $I_2$, and set colour$(p_2)$ = colour$(p_1)$. This generates the intermediate image (figure 4.7).



(a) (b)

Figure 4.7: (a) A reference image. (b) The reference image backwards mapped, for a new viewpoint which is up and to the right of the sample viewpoint. Note the holes, discussed in section 4.4.2.

## 4.5 Results

Table 4.1 shows the run times for the three stages of the warping process, on an image of size 256 x 256. Clearly, warping all pixels does not allow for interactive frame rates. It was therefore concluded that sprites with depth could not be used as we required: The pixel warping process is far too slow, and even with a hole–filling procedure, some larger holes cannot be eliminated. Both of these problems are addressed in the next chapter with meshed impostors.

Table 4.1: Warping time for 256 x 256 image, on a 333 MHz Pentium2. .

| **Algorithm** | **Time (seconds)** |
|---|---|
| Forward Mapping the Depths | 0.13 |
| Hole Filling | 0.08 |
| Backward Mapping the Image | 0.11 |
| Total | 0.32 |

# Chapter 5

# Meshed Impostors

Warping each pixel, as is done with sprites–with–depth, results in unacceptably slow frame rates and holes in the final image. A method is needed which addresses both of these problems. "External meshed billboards with warping" solve both the speed and hole issues by creating a mesh on the image plane (i.e. external to the terrain) and performing a perspective warp only on the mesh vertices.

External meshed billboards, which lie on the image plane, are equivalent to view–dependent textured meshes, which lie in object space (see section 3.2.2). The creation and artifacts of both of these impostors are identical, so the following sections apply to both. However, the rendering methods are different, and are therefore dealt with separately.

## 5.1 Impostor Creation

The creation of meshed impostors begins with the same steps used to create sprites–with–depth. From regular sample viewpoints, the geometry is projected onto an image plane, and the image, depth buffer, and camera matrix are stored for each. Another pre–processing step is now added; this step creates a mesh for each sample viewpoint. Vertices for the mesh are selected in a four–step process:

- Vertices are first placed along the outline of the terrain image
- A filter is applied to the depth map to detect depth discontinuities. Vertices are then placed along these discontinuities.
- A filter is applied to the image to detect luminosity discontinuities, and vertices are placed in these areas.

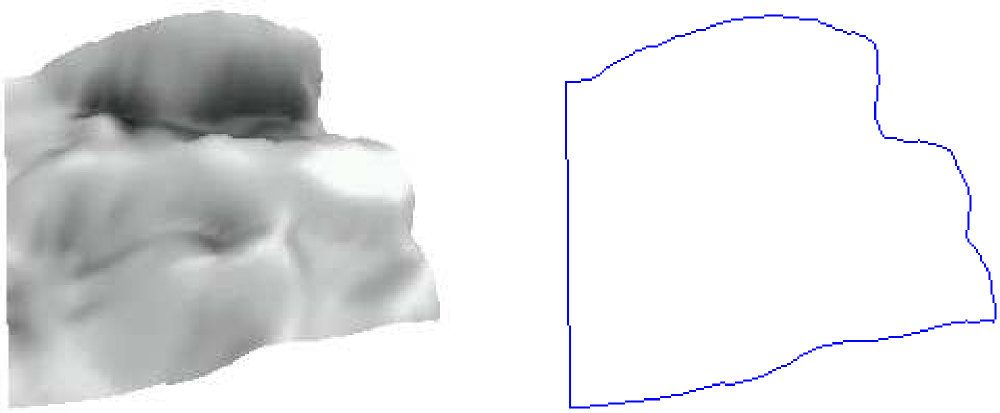
- The remaining vertices are placed randomly within the terrain image, constrained only in that they be sufficiently far from previously placed points.

This mesh is created specifically to handle the types of structures (hills and valleys) seen in terrains. While all of these algorithms could be used for other types of scenes as well, these other scenes would likely have other mesh generation techniques that would be better suited for their specific features.

After all vertices are selected, they are triangulated. The vertex placement and triangulation will now be described in more detail.

### 5.1.1 Outlining the Terrain

The outline of the terrain is created by scanning through all pixels in the depth map. All pixels in the terrain which are immediately adjacent to background pixels (those pixels with a depth of 1.0) are part of the outline (figure 5.1). This outline, which is one pixel wide, is traced from a random starting point, and vertices of the mesh are placed evenly throughout. For future work, it would be preferable to concentrate vertices in areas of greater curvature or change in the outline on the image plane. This would better preserve details in the terrain outline, but is not currently done.



(a) (b)

Figure 5.1: (a) A terrain. (b) The outline of the terrain.

### 5.1.2 Filtering for Depth Discontinuities

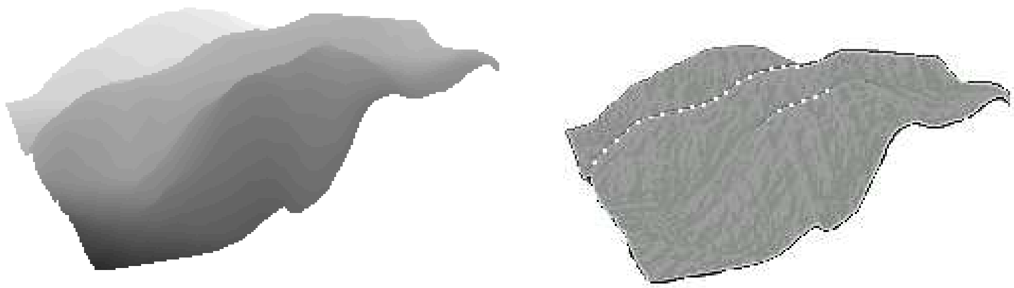
A simple two–dimensional "difference of Gaussians" filter is applied to the depth map to detect depth discontinuities . The weighting of the filter (which considers only immediate neighbours) is shown in

| | | |
|---|---|---|
| -1/8 | -1/8 | -1/8 |
| -1/8 | 1 | -1/8 |
| -1/8 | -1/8 | -1/8 |

Figure 5.2: The filter used to detect depth and illumination discontinuities.

figure 5.2.

The values in the filtered depth map are then scanned, and all pixels which have values below a certain cutoff number are considered to be depth discontinuity pixels. Filtered pixels with negative values are chosen rather than those with positive values, because we wish to place vertices on the nearer section of the discontinuity (i.e. the ridge) rather than on the background section. This ensures that ridges will move correctly as the eyepoint moves, since all vertices of the mesh are accurately perspective–warped. Ridges are particularly noticeable features in terrains, and preserving their shape and position is important in making the impostor look as accurate as possible. An example of the filtered depth map is shown in figure 5.3.



(a) (b)

Figure 5.3: (a) The depth map from a sample viewpoint. (b) The filtered depth map, with the resultant mesh vertices shown.

### 5.1.3 Filtering for Luminosity Discontinuities

The texture images are filtered in exactly the same manner as the depths, also using the filter seen in figure 5.2. As with depth discontinuities, luminosity discontinuities are naturally eye–attracting areas in a terrain. With more mesh vertices in these areas, these features will be better preserved, both as the eyepoint moves, and when one mesh is replaced by a new one (figure 5.4).

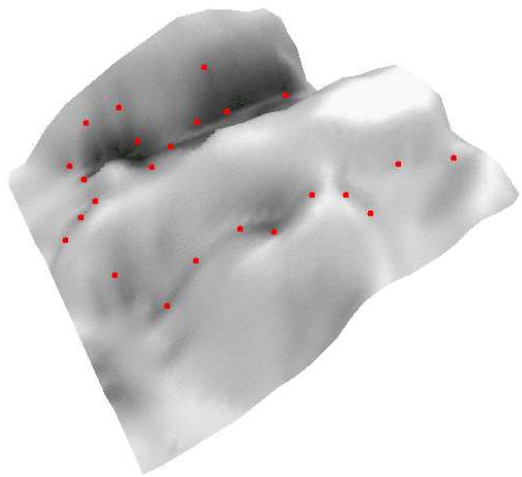


Figure 5.4: A texture image with the luminosity discontinuity vertices shown. Note that some obvious illumination discontinuities have no vertices. This is because there are already other vertices in these areas, due to the depth discontinuities.

### 5.1.4 Semi–Random placement

All remaining vertices (up to a user–defined limit) are placed semi–randomly within the terrain (using Poisson–disc sampling): A random pixel is chosen and a vertex is placed there only if there are no other vertices within a certain radius. This radius starts with a large value, but is decreased as the program begins having difficulty finding any spots to place vertices.

### 5.1.5 Triangulation

Once all vertices have been placed on the image, the mesh is formed by connecting the vertices with a Delaunay Triangulation. The Delaunay Triangulation has the property that its triangles are rarely long and thin (figure 5.5).

## 5.2 Displaying the Impostor

Once a mesh is created for each sample point, a user can begin interactively viewing the terrain. From the current viewpoint, an optimal sample viewpoint must first be chosen. The same method is used as described in section 4.3. The impostor is then displayed. For external meshes, a mesh warping procedure is executed, and the result is displayed on a 2D plane in object space. For view–dependent meshes, the

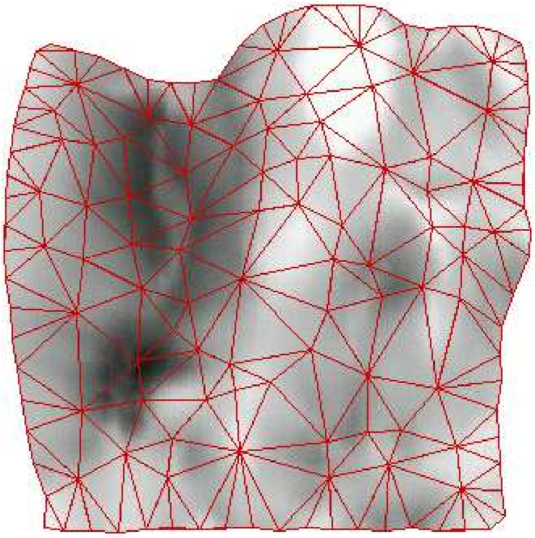


Figure 5.5: An impostor with a mesh created by Delaunay triangulation.

world–space coordinates of each vertex in the mesh have been pre–calculated using the information from the depth buffer, and the textured triangles are rendered in 3D world space.

To execute the mesh warp for external meshes, each vertex is simply forward–mapped on the image plane using Equation 4.2. (It is not necessary to follow the same forward and backward–mapping procedure seen in sprites–with–depth, since that was done only to make the hole–filling process somewhat more accurate). This warp results in an image that has no holes, and only a small fraction of the original warping calculations need to be performed (figure 5.6).

One major difficulty with this method of external meshed billboards is that the mesh vertices, which lie on a 2D image plane, might be warped a great distance on that plane, especially as the viewing angle moves far away from original recorded position (figure 5.6 (e)). If the planar impostor were to be used in one quad of a terrain and actual geometry in other, closer quads, difficulties would arise when the warped mesh would overlap the surrounding terrain geometry. However, this problem does not exist in view–dependent textures meshes, described next. A second problem with external meshed billboards is that the mesh triangles can overlap after the individual vertices have been warped. Therefore, a depth test must be used to determine which triangle should be displayed in front. With VDTM's, this is easily taken care of by the OpenGL z–buffer depth test.

With view–dependent textured meshes (VDTMs), a simplified geometric model of the terrain is created. Rather than using the depth of the mesh vertices to warp them on the image plane, the depths

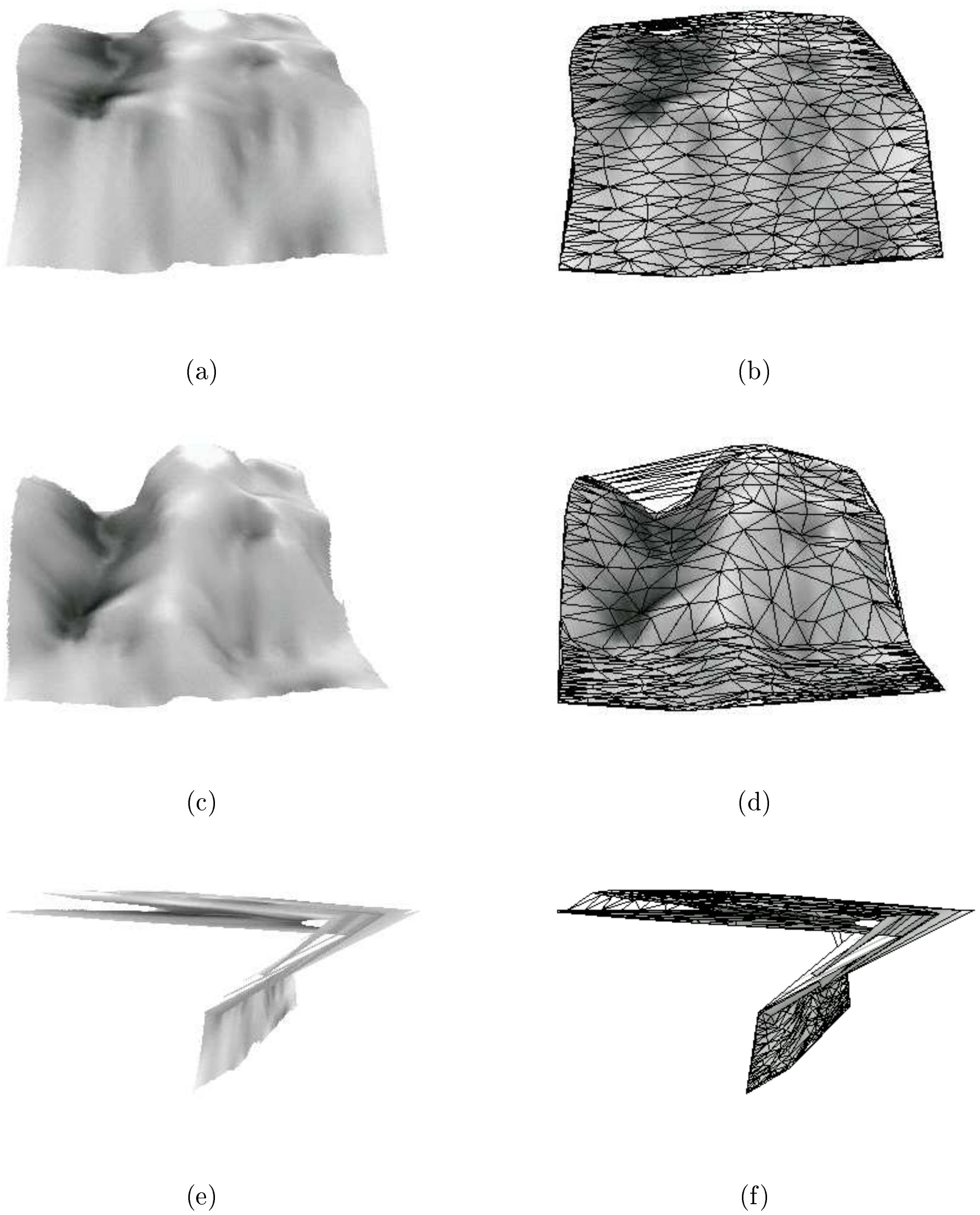
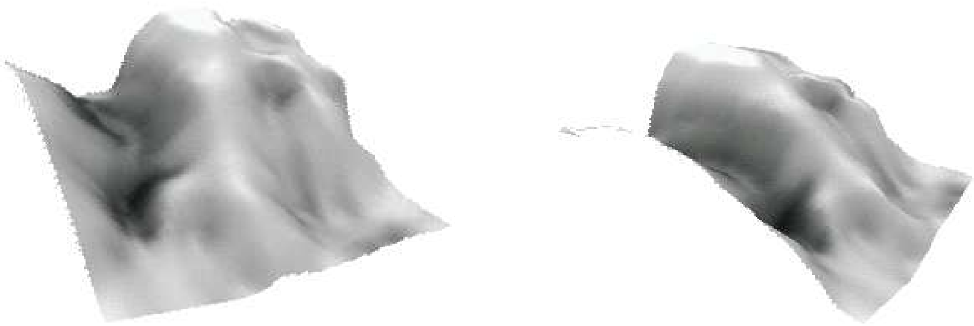


Figure 5.6: (a) A sample image which has not yet been warped. (b) With the mesh shown. (c) The same image after the mesh–warping procedure. (d) With the mesh. (e) The warped impostor from a side–view. The mesh has been warped off the original face of the impostor. (f) When the mesh is shown, it is easier to see the original two planes of the impostor.

are used to calculate the world coordinates of the vertices. As discussed in chapter 4, the camera matrix $C_i$ is stored for each sample viewpoint, and the inverse $C_i^{-1}$ is calculated. $C_i^{-1}$ transforms points from the image plane into world coordinates:

$$\begin{bmatrix} w_i X \\ w_i Y \\ w_i Z \\ w_i \end{bmatrix} = C_i^{-1} \begin{bmatrix} x_i \\ y_i \\ z_i \\ 1 \end{bmatrix} \tag{5.1}$$

Once the world coordinates of the mesh vertices are calculated, the mesh triangles can be drawn in world space with their associated textures.



(a) (b)

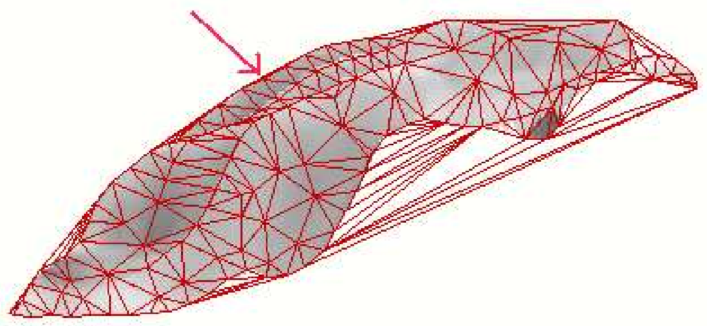
Figure 5.7: (a) The VDTM displayed in 3D coordinates. Note that this looks identical to the external meshed billboard. (b) A side view of the mesh drawn in 3D coordinates. Compare this to the side view of figure 5.6 (e).

There are several advantages to this method. First, time is saved by avoiding the software warp on mesh vertices in an image plane; a hardware perspective transformation is faster. Second, the overlapping of warped triangles, and hence the need for triangle sorting, is not an issue with VDTMs. Third and finally, since a VDTM does not use warping, the impostor never moves to points outside of the quad that contained the original mesh (figure 5.7 (b)). Therefore there is no possibility of overlapping any surrounding terrain.

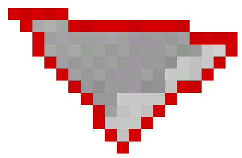
## 5.3 Artifacts

Both external meshed billboards and VDTM's exhibit two major artifacts, both due to the inherent inaccuracy of using mesh warping (or mesh simplification). "Jags" result when a mesh triangle (usually spanning a depth discontinuity) is stretched, and small inaccuracies expand, becoming noticeable. The lack of disocclusions is a related problem; rather stretching the triangles, new background triangles should become visible.

For example, consider a triangle which has two vertices on a nearby ridge, and the final vertex lies on the terrain behind this ridge. If the ridge is sufficiently curved in the image plane, several pixels of the ridge will appear in the texture of this triangle (figure 5.8). As this triangle is stretched, these pixels become noticeable as jags along the ridge edges (figure 5.9). Also, new areas of the terrain do not become exposed, since those areas were not included in the impostor. There is no simple way to solve this, but it will be dealt with later in the next chapter on "unzipped meshes."

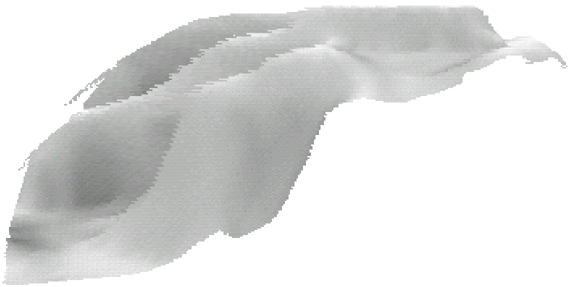


(a)

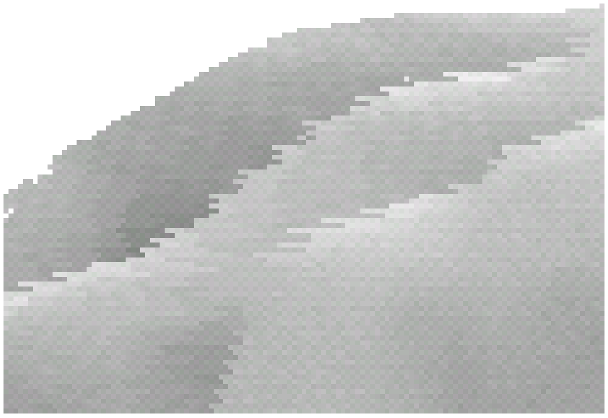


(b)

Figure 5.8: (a) Pixels of a ridge in the foreground are stored in the primarily background triangle. (b) Close–up of one of these triangles. One can see the lighter coloured foreground pixels at the bottom of the triangle.



(a)



(b)

Figure 5.9: (a) Jags along the ridge edges due to texture stretching. (b) A close–up of the ridges.

| **Terrain Triangles** | **Mesh Triangles** | **Method** | **Time (sec)** |
|---|---|---|---|
| 4600 | | Geometry | 0.016 |
| 4600 | 374 | Imposter | 0.046 |
| 120000 | | Geometry | 0.330 |
| 120000 | 348 | Imposter | 0.047 |

Table 5.1: Frame rendering times comparing VDTM impostors and geometry for different terrain sizes. Imposter times are independent of terrain size.

## 5.4 Results

The program was run on a Pentium III 500Mhz processor with hardware texture mapping, on two different terrains (with a window size of 256 x 256). These times show that when the terrain becomes sufficiently large, the rendering time of an impostor frame becomes significantly faster compared to rendering the geometry. One can see that the rendering time of the geometry is dependent upon the number of triangles in the scene, while the rendering time of the impostor is not. This is the primary advantage of image–based rendering.

# Chapter 6

# View Dependent Unzipped Meshes

It was seen with external meshes and view–dependent textured meshes that using only a single–meshed impostor results in artifacts along depth discontinuity lines: Disocclusions (parallax effects) cannot occur, and stretching of the triangles results in noticeable jags in the image. It is necessary, then, to use multiple meshes and multiple textures to create impostors that allow for hidden areas becoming exposed, and that do not exhibit the rubber–sheet effect seen with VDTM's.

## 6.1 Multiple Mesh Generation

Creating an "unzipped mesh" begins with the same steps used in creating a view–dependent textured mesh (see section 5.1). First, a mesh is created on the image plane. Then there are three new pre–processing steps to create the unzipped mesh:

- The triangles of the image–plane mesh are partitioned into several sets, each containing triangles in a range of depths. Intuitively, the mesh is "unzipped" along those edges that separate image–plane triangles of different depth–sets. Thus, the mesh is automatically split into "depth layers" (which are vaguely similar to those of Schaufler [25]).
- The terrain triangles (i.e. the original 3D geometry) are also partitioned into corresponding depth–sets. This is a subtle process, since some of these terrain triangles are occluded and the depth–sets created in the first step may have overlapping depths.
- A texture and new mesh is created for each depth–set. The mesh is identical to the original mesh, except in areas which were occluded in the original mesh. Each texture is created by rendering the corresponding terrain triangles.

These steps are now described in detail.

### 6.1.1 Unzipping Based on Depth Partitions

Each triangle of the VDTM is first assigned a depth, which is the maximum depth of all vertices in the triangle. The depth of a vertex is the value in the depth buffer at the pixel occupied by the vertex. The maximum depth of all vertices is used (rather than, for example, minimum or average) because a triangle which stretches from the top of a ridge to background terrain (see figure 5.8) should be classified in the depth–set corresponding to the background terrain (since most pixels in this triangle belong to the background).

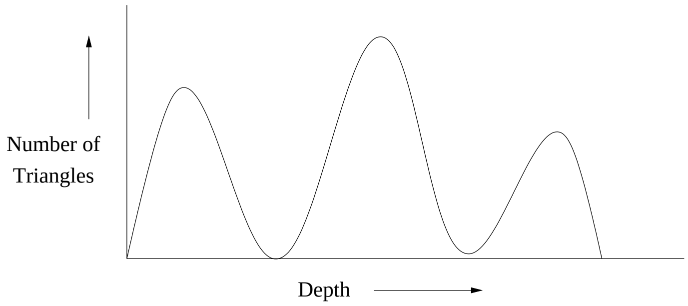
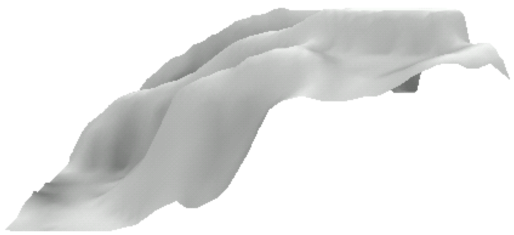



Figure 6.1: The expected distribution of triangle depths in a scene with depth discontinuities. The peaks represent visible areas of the terrain, while the valleys represent areas covered by closer ridges (depths are scaled to the range [0,100]).

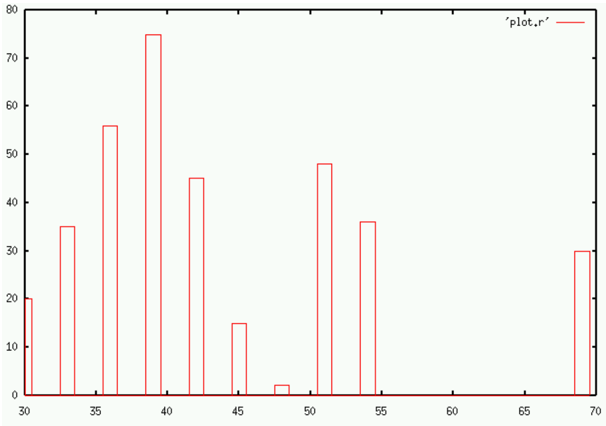
After depths have been assigned to all mesh triangles, they need to be partitioned. In scenes which have multiple depth discontinuities, one would expect to see a pattern of depths such as the one seen in the histogram of figure 6.1. Therefore, we partition the mesh triangles into sets based on peak values of the histogram.

Let $n$ be the number of desired depth–sets. The triangle depths are scaled to integers in the range [0,100] and a histogram is created with buckets of size one. If the number of histogram peaks (i.e. local maxima) is greater than $n$, the histogram is rebuilt with buckets of size two. This continues with increasing bucket size until the number of peaks is at most $n$. Figure 6.2 shows the histogram for one view, which uses buckets of size three to find three peak values.

Each mesh triangle is then associated with the peak that has a value closest to that of the depth of the triangle, and this defines the partition. The goal of this partition is that sections of the terrain which lie on opposite sides of a depth discontinuity will be partitioned into different sets. This has been found to work effectively when the upper limit on the number of sets is larger than the number of depth



(a)




(b)

Figure 6.2: (a) The terrain from a sample viewpoint. (b) The depth histogram, showing the number of triangles for each depth bin of size 3.

discontinuities (figure 6.3).

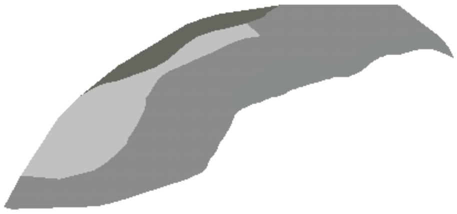


Figure 6.3: The terrain from figure 6.2 (a) partitioned into three sets based on depth.

### 6.1.2 Partitioning the Terrain Triangles

After the mesh triangles on the image plane have been partitioned, the actual terrain triangles need be divided correspondingly, since the terrain triangles of a depth set will be used to create the texture for the mesh of that depth set. Figure 6.4 illustrates the goal of this second partition.

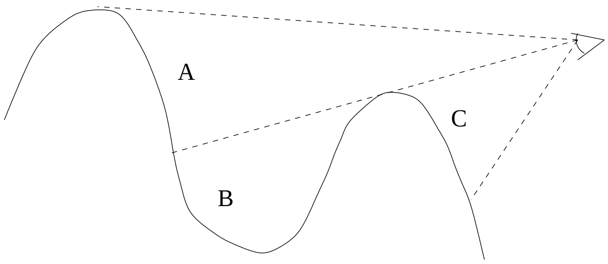



Figure 6.4: From the shown viewpoint, section A and section C can be seen, and the mesh triangles for these regions belong to different sets. Section B cannot be seen, but the terrain triangles which lie in this region should be grouped with those of section A, so they will then be part of the mesh for A. Section B will then become correctly disoccluded as the viewpoint moves upward.

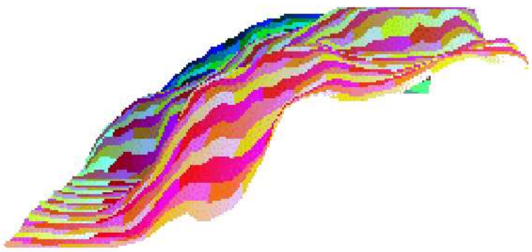
Partitioning the terrain triangles is a difficult process. As was shown in figure 5.8, pixels on the top of the ridges can appear in a mesh triangle on the image plane which is considered part of the background. This can lead to the corresponding foreground terrain triangles (i.e. those that project to these pixels) being classified into the incorrect background depth–set. Also, terrain triangles which are not visible from the sample view point must be classified.

The partition of terrain triangles is done in two steps. First, the partition of the mesh triangles is used to assign each visible terrain triangle to a depth–set. However, occluded terrain triangles are not assigned to any depth–set and a few visible terrain triangles might be assigned to the wrong depth–set (as explained below). Second, a series of heuristics is applied to improve the initial assignment.

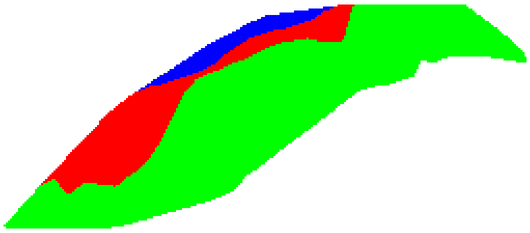
### Creating the Initial Partition

Two item buffers[1] are used to partition the terrain triangles. Into the first item buffer are rendered the terrain triangles, each with a unique identifier; into the second item buffer are rendered the mesh triangles, each with the identifier of its depth–set. A terrain triangle that appears in the first item buffer is assigned the depth–set from the corresponding location in the second item buffer.

Figure 6.7 (a) shows a terrain after this initial assignment. Clearly this is not sufficient, since several triangles (which lie on depth discontinuities) are mislabeled, and many (occluded) triangles remain unclassified. This initial assignment to depth–sets needs to be improved to obtain a usable result.



(a)



(b)

Figure 6.5: (a) The item buffer for the terrain triangles, and (b): The item buffer for the depth sets.

### Improving the Initial Depth–Set Assignments

Consider the terrain triangles in an orthographic projection viewed from above, as shown in figure 6.7. Each triangle belongs to zero, one, or more depth–sets (e.g. triangles on the border between depth groups can belong to multiple depth–sets since a single terrain triangle might project onto multiple mesh triangles belonging to different depth–sets, while triangles occluded from the sample viewpoint belong to no depth–set). Two triangles are said to be **adjacent** if they touch anywhere, even if only at a vertex.

[1] An item buffer determines which polygon occupies each pixel: Each polygon of the scene is drawn with a unique color which can be used to identify the polygon. After all polygons have been drawn, the framebuffer is read, which records the color (i.e. polygon identifier) at every pixel of the image [1, 33].

Connected components of terrain triangles are first computed. Two triangles are part of the same **connected component (CC)** if and only if (a) there is a path of adjacent triangles connecting them, and (b) all these triangles are in exactly the same depth–set(s). The **depth–set(s) of a CC** is the depth–set(s) of any one of the triangles of the CC (which is well defined, since all triangles of a CC have the same depth–set(s)).

After the CCs have been computed, each CC is classified into one of the following types:

- *Single–group*: the CC has a single depth–set.
- *Multi–group*: the CC has more than one depth–set.
- *Unknown–group*: the CC has no assigned depth–set.
- *Backfacing–group*: all triangles of the CC are backfacing.

Three stages of heuristics are now applied to eliminate errors in the initial assignment, and to assign depth–sets to CCs which have no assigned depth–set.

**Stage 1**

The first stage applies three heuristics that eliminate obvious errors, and assign groups to unknown CC's if the decision is unambiguous. There are two types of errors which need to be corrected, which usually occur along ridge lines (see figure 6.6). The unknown CCs are those sets of triangles which are not visible from the sample viewpoint. We would like them to be grouped with the appropriate depth–sets to eventually be able to fill in disocclusion areas (see figure 6.4). Specifically, the heuristics are:

1. An unknown–group CC is assigned the depth group of an adjacent single–group CC if:
   - all adjacent single–group CCs belong to the same group.
2. Multi–group CCs often lie on a ridge between two depth–sets and are separated from the more distant depth–set by backfacing triangles. These multi–group CCs should be placed in the closer depth–set if possible. Therefore, a multi–group CC is *re–assigned* the depth–set of an adjacent single–group CC if:
   - this multi–group CC has fewer than 30 triangles, and
   - all adjacent single–group CCs belong to the same group.

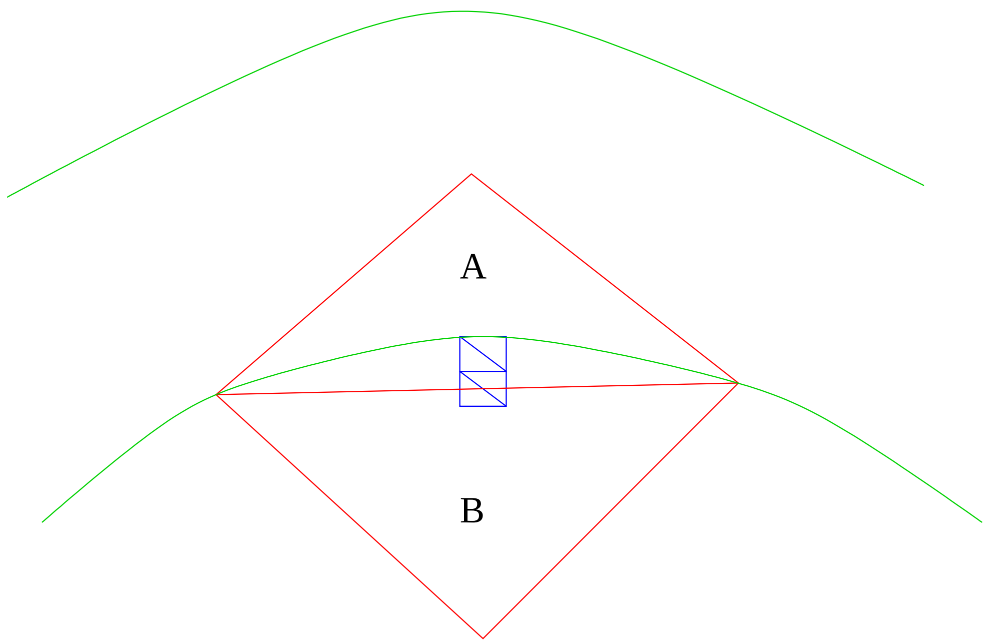



Figure 6.6: Two ridges in a terrain are shown (green). The red triangles are image–space triangles which have been put into two different depth–sets, A and B. The blue triangles are terrain triangles (from the original geometry) which need to be classified. The top two terrain triangles are entirely contained within triangle A, so they are assigned depth–set A with the item buffer. The lower two terrain triangles are in both image–space triangles A and B, so these two terrain triangles are assigned both depth–sets, A and B (therefore making them part of a multi–group CC). However, all four of these terrain triangles should be assigned depth–set B only, because in fact they all lie on the nearer part of the terrain. This needs to be fixed in one of the three stages.

3. On the boundary between two depth–sets, a triangle will occasionally be assigned to the wrong depth–set. This occurs, for example, on a curved ridge which is approximated in the mesh by a chain of straight segments. Therefore, a single–group CC is *re–assigned* the depth–set of an adjacent single–group CC if:
   - this CC has fewer than 30 triangles,
   - there is an adjacent backfacing CC, and
   - all adjacent single–group CCs belong to the same group.

After any one heuristic is applied, the CCs are recomputed and another heuristic is applied. This continues until no heuristic can be applied. This first stage has dealt only with cases which are entirely unambiguous. Further stages are needed to complete the assignments.

### Stage 2

The second stage applies two more heuristics, which assign depth–sets to CCs when the decision is somewhat ambiguous, but very likely to be correct. This occurs when an unknown CC is next to two or more single–group CCs. If it is almost totally adjacent to one of these CCs, and the others are just barely touching, then this unknown CC should be assigned the depth–set of the dominant neighbour. If there is no such dominant neighbour, then the unknown CC should be assigned all depth–sets of the neighouring CCs (therefore becoming a multi–group CC). The specific heuristics are:

1. If the depth–set of a CC is unknown, then assign it to depth–set $S$ if the number of triangles adjacent to the CC which are in depth–set $S$ is *at least 80 percent* of the total number of adjacent triangles.

2. If the depth–set of a CC is unknown, and if the ratio of

$$\frac{\text{size of largest adjacent depth–set}}{\text{size of smallest adjacent depth–set}}$$

   is at most $n$, then assign to the CC the union of the depth–sets of all adjacent CCs. $n$ is initially 1.0, which is essentially impossible to satisfy. If no heuristic is triggered (either 1 or 2) then $n$ is multiplied by 1.1 and heuristic 2 is tried again. This is repeated until $n$ exceeds 2.0, at which point we give up. The idea here is that the adjacent CCs are evenly distributed around the unknown CC, so the unknown CC is assigned the union of the adjacent CCs.

After this stage, all CCs which can safely be assigned to a depth–set or depth–sets have been so assigned (results seen in figure 6.7 (c). However, there can remain CCs which do not have a clear choice, and need to be separated in order to complete the group assignments. This takes place in the third and final stage.

#### Stage 3

The final stage ignores the CCs, and attempts to assign a group to the remaining unknown triangles on an individual basis. For each triangle, $T$, that has not been assigned a depth–set and which is adjacent to some other triangle that *has* been assigned a depth–set, $T$ is assigned the depth–sets of all adjacent triangles. This is performed in multiple passes, until only totally isolated unclassified triangles remain.

This has the effect of filling in the unassigned regions with "waves" from the adjacent assigned regions, and getting multiply-assigned triangles extending out from the boundaries between different adjacent (already-assigned) regions. Figure 6.8 and figure 6.9 show an example of group–assignment in which stage three is necessary to satisfactorily complete the partition. There is a large unknown group remaining after stage two is completed, which was not assigned a group since it is adjacent to two single–group CCs. After stage three, one can see that the triangles belonging to the unknown CC have been divided into both of the adjacent single–group CCs, and the border has been extended as well.

After stage three has been completed, most of the terrain triangles have been grouped into one or more depth–set. Only those that were completely isolated after the original item buffer–based assignment remain unclassified, but these triangles tend to be rare and relatively unimportant (in areas of the terrain which are very unlikely to be seen from the associated sample area). Triangles which were classified incorrectly using the item buffers (primarily triangles along ridge edges) are reclassified correctly, and triangles hidden from the sample viewpoint are also classified appropriately. With the partitioning of terrain triangles completed, the unzipped mesh can be generated, by separately drawing the terrain triangles of each depth–set, storing the image as a texture, and creating a new mesh for each depth–set.

### 6.1.3 Remeshing the Terrain

After the terrain triangles have all been partitioned into depth–sets, a new mesh is created for each depth–set. However, since the different meshes are connected to each other, they must be created in a manner which guarantees that there will be no gaps in the terrain when the final impostor is displayed. Previous work in multiple mesh impostors was done in an environment which lends itself easily to creating separate meshes. Consider the work by Decoret *et al* [9], which uses multiple mesh impostors in an urban environment. Buildings can be meshed separately without difficulty, because there is not

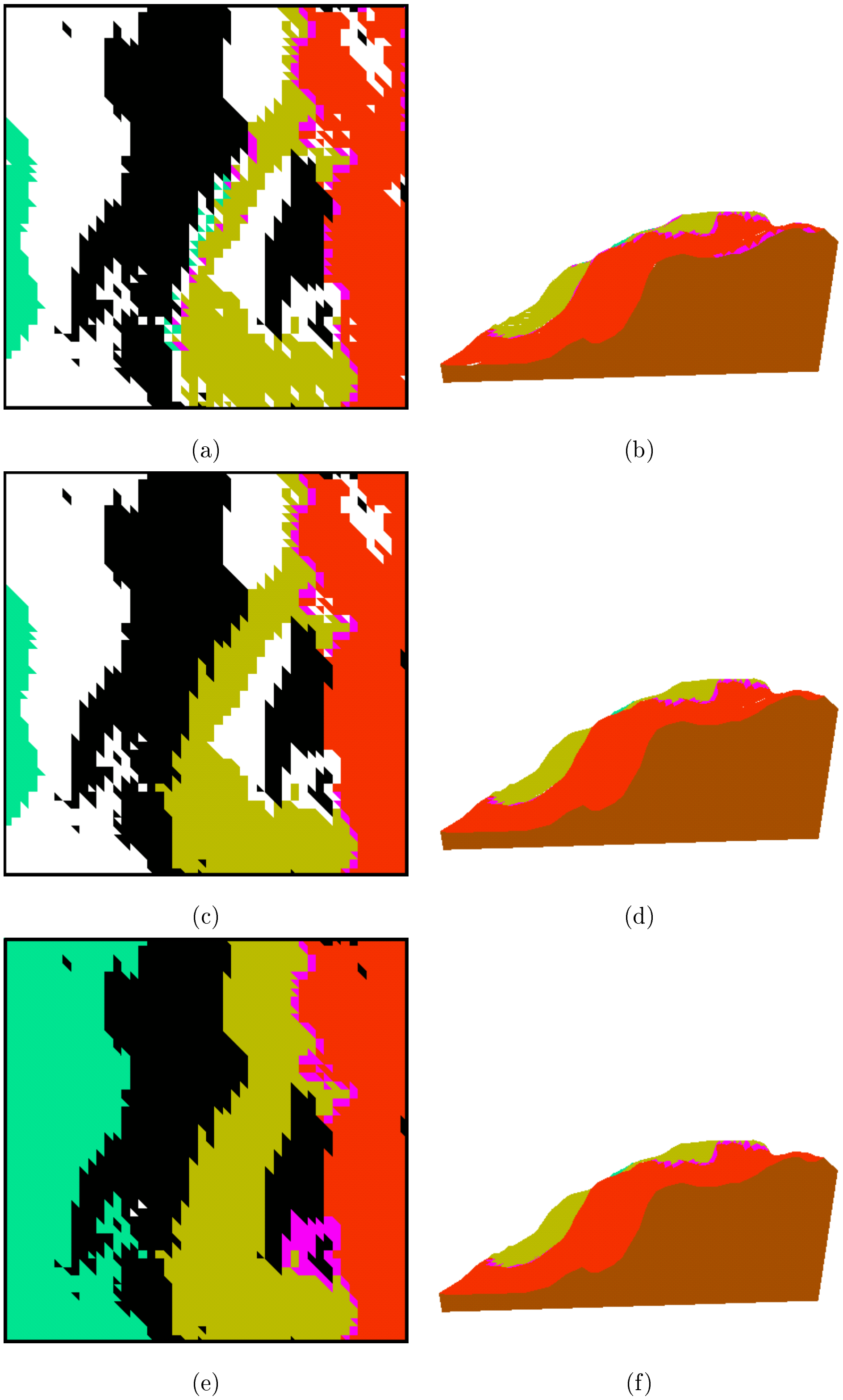


Figure 6.7: (a) The initial partitioning of terrain triangles into sets (based on the partitioning of the mesh triangles). (b) Shown from the side view which was used to generate the initial partition. (c) Terrain triangle grouping after Stage 1. (d) Side view. (e) Terrain triangle grouping after Stage 2. (f) Side view.

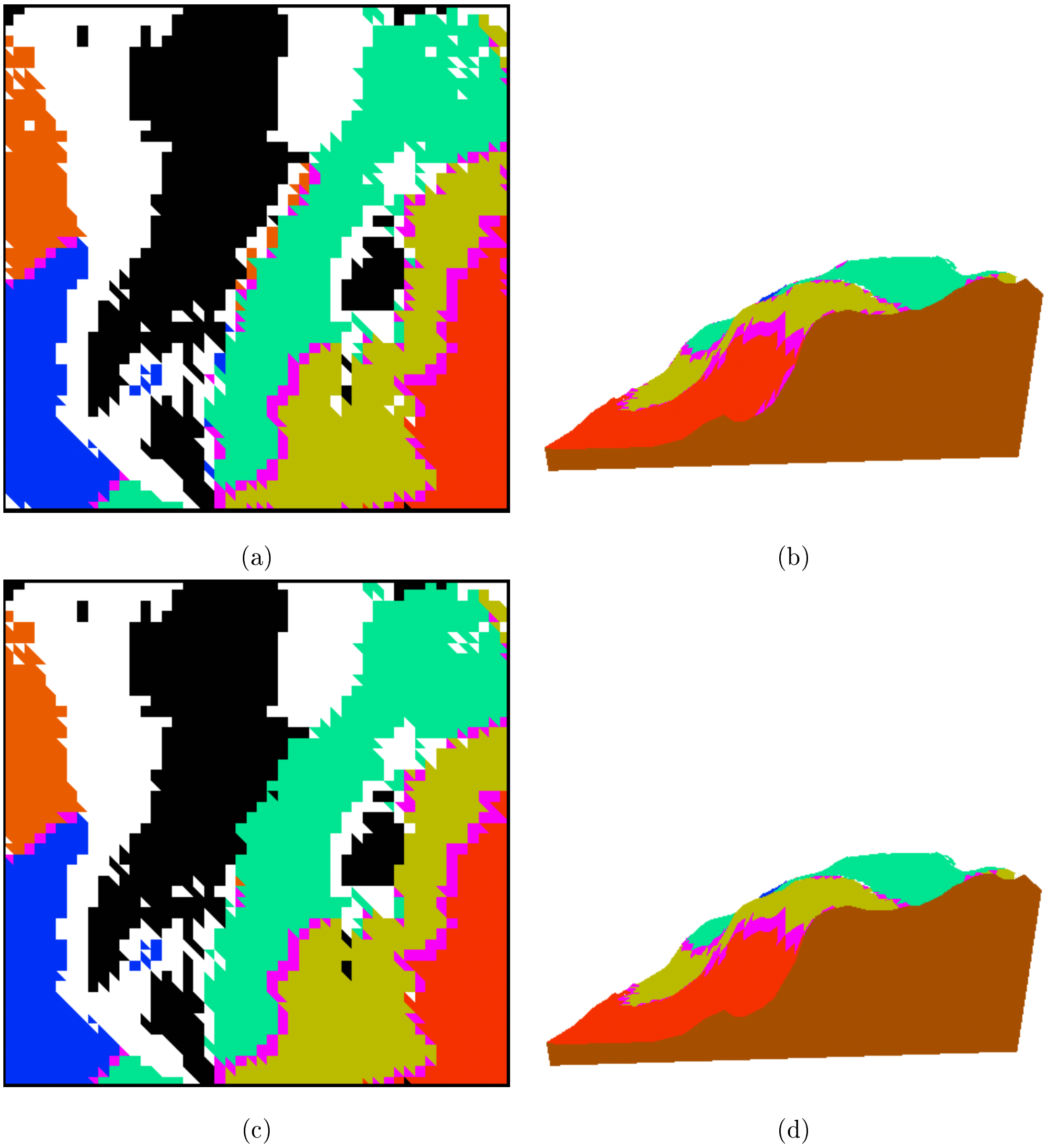



Figure 6.8: (a) The initial partition from the same viewpoint as in figure 6.7, but with a greater number of groups. (b) The same side view from which the initial partition was created. (c) After Stage 1 is completed. (d) Side view. (Continued in figure 6.9.)

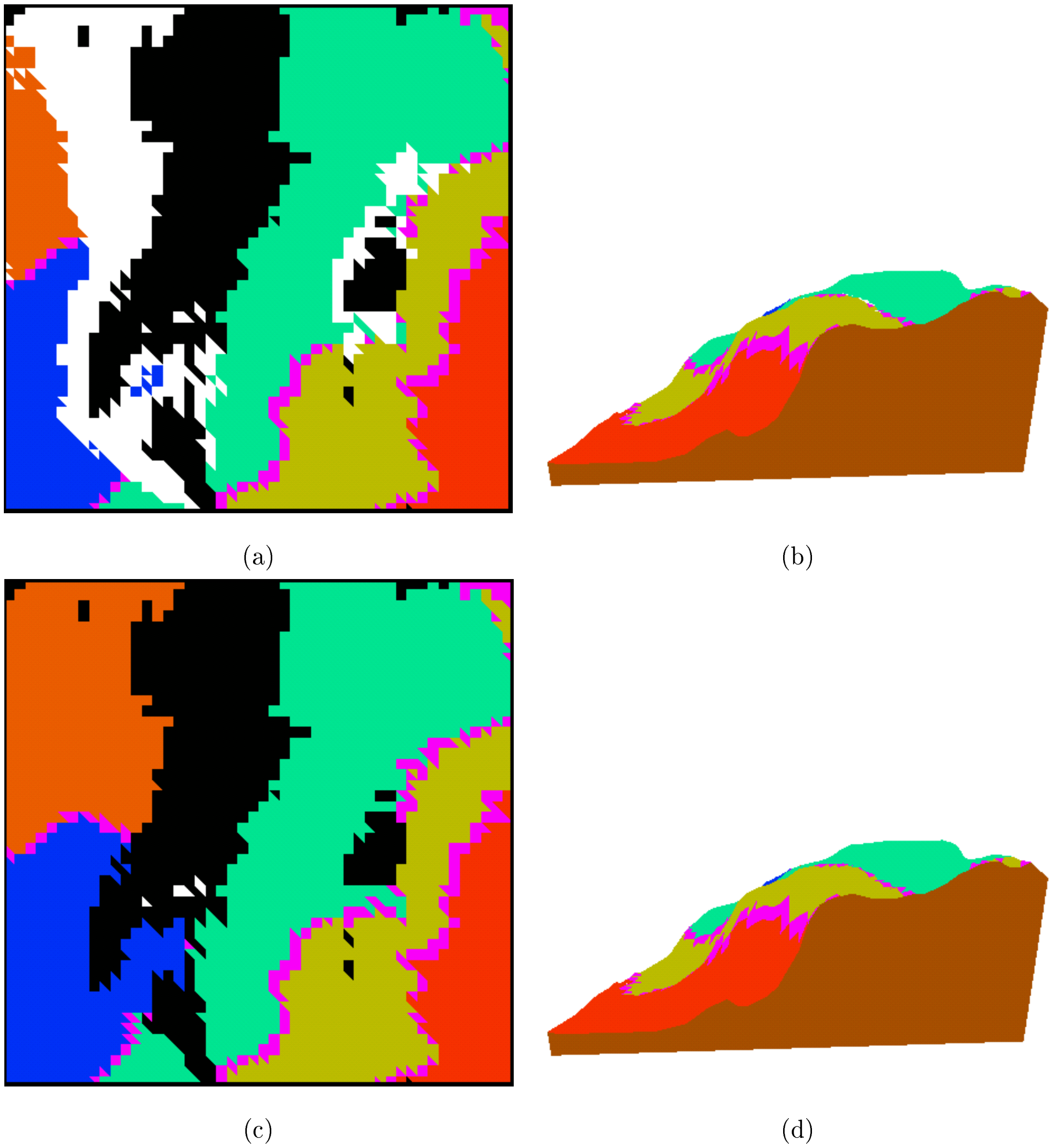



Figure 6.9: (a) Stage 2 is completed. At this point there are still large unknown CCs which have not been classified, since this cannot be done unambiguously. (b) Side view. (c) After Stage 3, the large unknown CCs are divided into all of the neighbouring CCs. (d) Side view.

interconnectivity between the separate structures. With terrains, the depth groups to be remeshed are in fact part of a continuous object, so the final set of meshes must also be continuous.

*Therefore, a mesh is created for each group using only the vertices of original image–plane mesh created for the initial VDTM.*

Consider a particular depth–set. The terrain triangles of this depth–set must be rendered and an image–plane mesh must be created that covers them. The imposter for this depth–set will consist of the image–plane triangles projected back onto the 3D terrain.

To build the image–plane mesh, each triangle of the initial, image–plane VDTM mesh is examined, and it is determined if this triangle should be added to the mesh for the current depth–set (with the depth–sets being examined in increasing order of depth). An image–plane triangle is added if it meets either of the following criteria:

- The image–plane triangle belongs to the depth set currently being drawn.

- The image–plane triangle belongs to a closer depth–set (and has therefore been used at least once already in the remeshing of that depth–set), at least one vertex of the triangle projects onto a terrain triangle of the current depth set, and all such projections result in different world coordinates than those calculated the previous time this image–space triangle was used (figure 6.10).

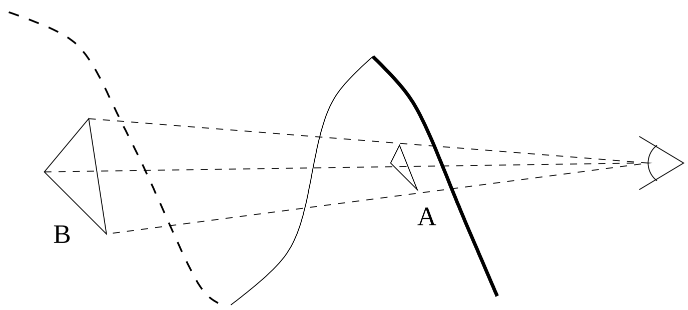
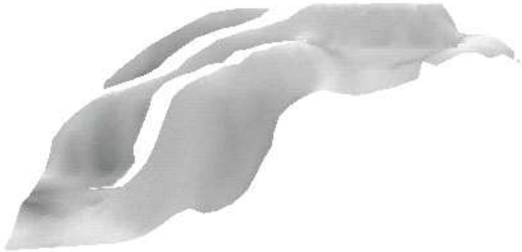
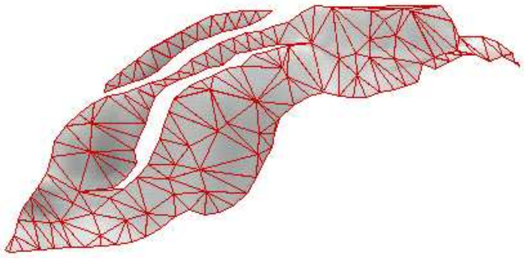



Figure 6.10: The dashed line and the bold line represent two different depth sets of terrain triangles. Suppose a new mesh is being created for the dashed set. Triangles A and B are two different projections of an original mesh triangle. We are attempting to determine if this triangle should be included in the new mesh. This triangle does not meet criterion 1, since the dashed area of the terrain is not visible from the shown viewpoint (and therefore the triangle cannot belong to the current depth set). However, it does meet criterion 2: The triangle belongs to a closer depth–set (A), at least one vertex projects onto a terrain triangle of the dashed set, and the calculated world coordinates of the vertices of B are all different than those of A.

The first criterion puts in mesh triangles which were already present in the original VDTM. The only difference is that these are now not necessarily a connected set of triangles (in 3D world coordinates). Across depth discontinuities, these meshes have now been separated (figure 6.11 (a)).

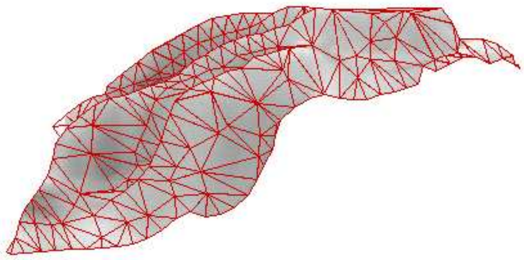


(a)



(b)

(c)



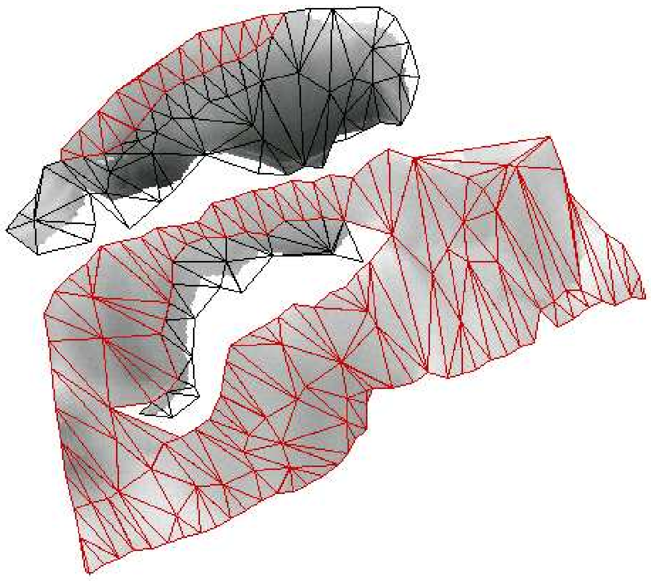
(d)

Figure 6.11: (a) An unzipped mesh after only original triangles have been inserted. (b) The unzipped mesh with the mesh drawn. (c) The unzipped mesh with disocclusion triangles added. (d) With the mesh.

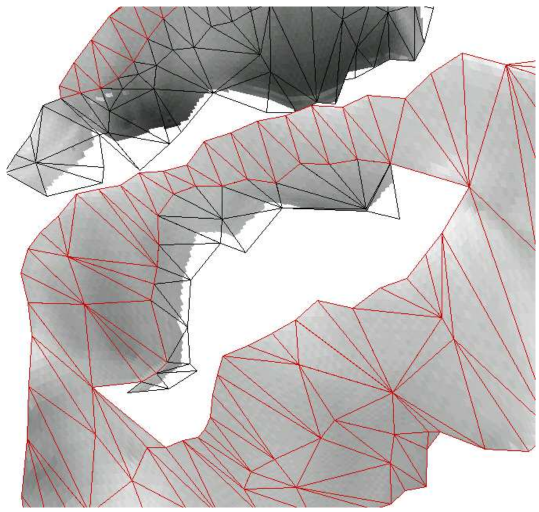
The second criterion adds in “disocclusion triangles:” those which fill in occluded areas which can become exposed as the viewpoint moves. These new triangles can be seen in figure 6.11 (d).

There are two important details to be noted about these disocclusion triangles. As discussed, they are only considered for inclusion in the mesh if at least one of the vertices projects onto the terrain triangles of the current depth set. If not all of the vertices project onto terrain triangles, then the depth of the vertices that do not is calculated by averaging the depths of the other vertices (figure 6.12 (a)). Also, the remeshing must be done in front–to–back order (by depth–set). Therefore mesh triangles belonging to closer depth sets will used earlier, and world coordinates will be calculated. Then when this triangle is considered for a further depth set, previous world coordinates will exist to compare to (for the second criterion).

A mesh is created in this fashion for each depth–set, with the world coordinates being stored for each vertex of the new meshes (the coordinates are calculated using equation 5.1. This concludes the pre-processing phase which builds the view–dependent unzipped meshes.



(a)



(b)

Figure 6.12: (a) Some disocclusion triangles (outlined with dark lines) have some vertices that lie on the background. The depths of these vertices are determined by averaging the depth of the other vertices. (b) The lower disocclusion triangles are only partially filled with textures, and the background can show through for certain points of view.

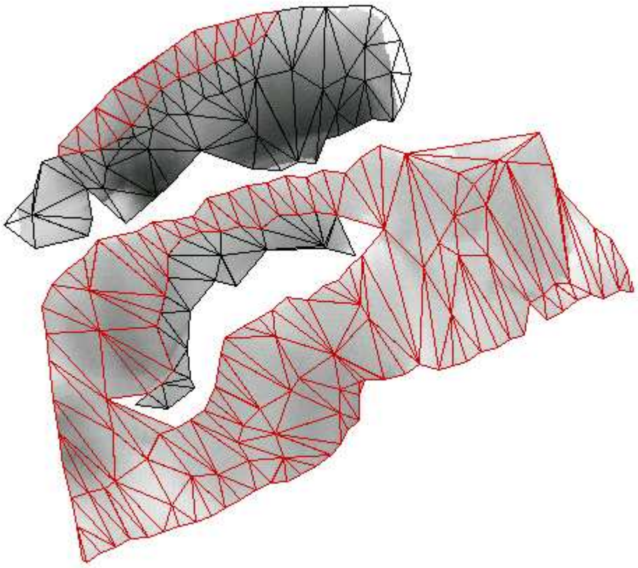


Figure 6.13: The disocclusion triangles (dark lines) filled with textures taken from the higher viewpoint.

### 6.1.4 Displaying Unzipped Meshes

Unzipped meshes are displayed in the same manner as VDTM's. The world coordinates of each vertex in the multiple meshes have already been calculated in the pre–processing phase, so these simply need to be drawn in world space. First, the best sample viewpoint is chosen, using the procedure described in section 4.3. The impostor created from this viewpoint is then rendered in world space, eliminating the need for any software warping procedure.

## 6.2 Re–texturing Disocclusion Triangles

Even after disocclusion triangles have been added to each mesh, it is still possible for the background to show through the set of meshes, in part because the meshes are not connected at depth discontinuities, and in part because many disocclusion triangles are only partially filled with textures (figure 6.12 (b)). This partial filling of triangles occurs simply because the textures can only be filled with the triangles that are not back–facing or occluded from the sample viewpoint; the lower boundary of the texture in a disocclusion triangle occurs where the valley floor changes from front– to back–facing with respect to the sample viewpoint.

This problem can be alleviated by incorporating texture information taken from a higher viewpoint. Clearly, as one moves higher, more of the valley floor becomes visible, and this could be used to fill the disocclusion triangles. It would be a very difficult problem to determine how high the new viewpoint would have to be to ensure that all disocclusion triangles could be completely filled; therefore the sample viewpoint above the current sample viewpoint on the hemisphere is used.

After rendering from the viewpoint above, the correct texture coordinates need to be calculated. Since the world coordinates of the disocclusion triangles are known, this is just a matter of transforming these 3D points to screen coordinates with respect to the new viewpoint. This is simply done using equation 1 in section 4.4.1, with the camera matrix of the new viewpoint. This calculates the new texture coordinates, so the disocclusion triangles can be filled with the texture from a higher viewpoint (figure 6.13).

Currently, the original textures in the disocclusion triangles are simply replaced with the textures from the higher viewpoint. This results in reasonable images when one is using view–independent lighting, but would be noticeable with view–dependent lighting. Some form of texture blending would need to be used in this case.

With this addition, it is very rare that holes in the impostor are visible. However, this can occur in certain disocclusion areas. There are possible solutions to this, but they have not been implemented. This will be discussed further in the final chapter.

# Chapter 7

# Results

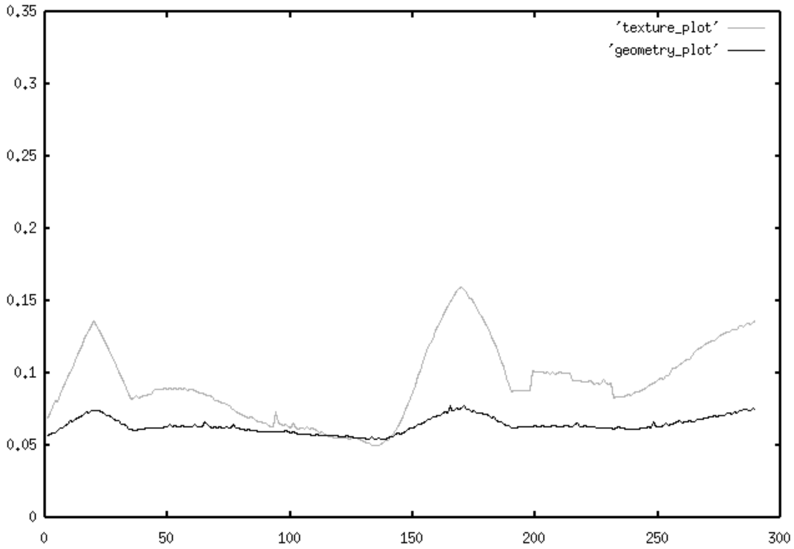
By unzipping VDTM's and creating a multiple mesh impostor, the two major problems seen in VDTM's are eliminated with unzipped meshes. Parallax effects are seen as expected due to disocclusion triangles, and the separation of the mesh along lines of disocclusion has eliminated the problem of triangle stretching and jags.

Figure 7.1 shows the timing results of the fly–by over two different terrains, with a window size of 512 x 512. This fly–by includes a full 360 degree path around the terrain, and two sequences of moving up and down. These up and down sequences generate the peaks seen in the graph; as the viewpoint moves up, more area of the terrain is visible and more of the screen space is occupied by the terrain. This results in increased rendering time.

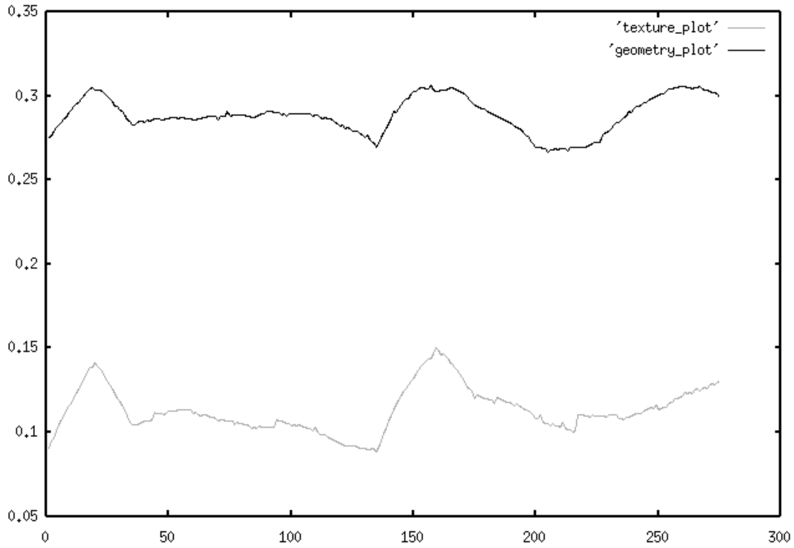
Figures 7.2 through 7.4 show a sequence of snapshots in the fly–by of the larger terrain. The final shot in figure 7.4 shows one of the higher points of view that generated a peak value in the graph.

Figures 7.5 and 7.6 show two sets of adjacent frames in which a transition between reference images occur. When viewed interactively, this transition results in a "pop" in the image. This popping is due to using a different reference image that contains somwhat different information, as well as using a different mesh. However, one can see that the images do not differ very much, partially due to the method used to construct the mesh (placing points on more visually noticeable areas). Other possibilities of improving this popping are discussed in the next chapter.

Tables 7.1 and 7.2 summarize the time results of the two fly–bys, and show the memory usage required for the different methods and terrains. The rendering speed of the unzipped meshes is not dependent upon the number of triangles in the terrain. Although the average frame rates for the two terrains is not exactly the same, this difference is only caused by the greater amount of space on the screen that the larger terrain occupied. One can see that the rendering time of the geometry increased much more




(a)




(b)

Figure 7.1: (a) Timing results for a fly–by over a terrain with 4600 triangles. The "geometry–plot" line shows results for drawing the entire geometry, and the "texture–plot" line shows results for the unzipped meshes. The x–axis is the frame number, and the y–axis is the rendering time for the frame (seconds) (b) Timing results for a terrain with 20000 triangles.

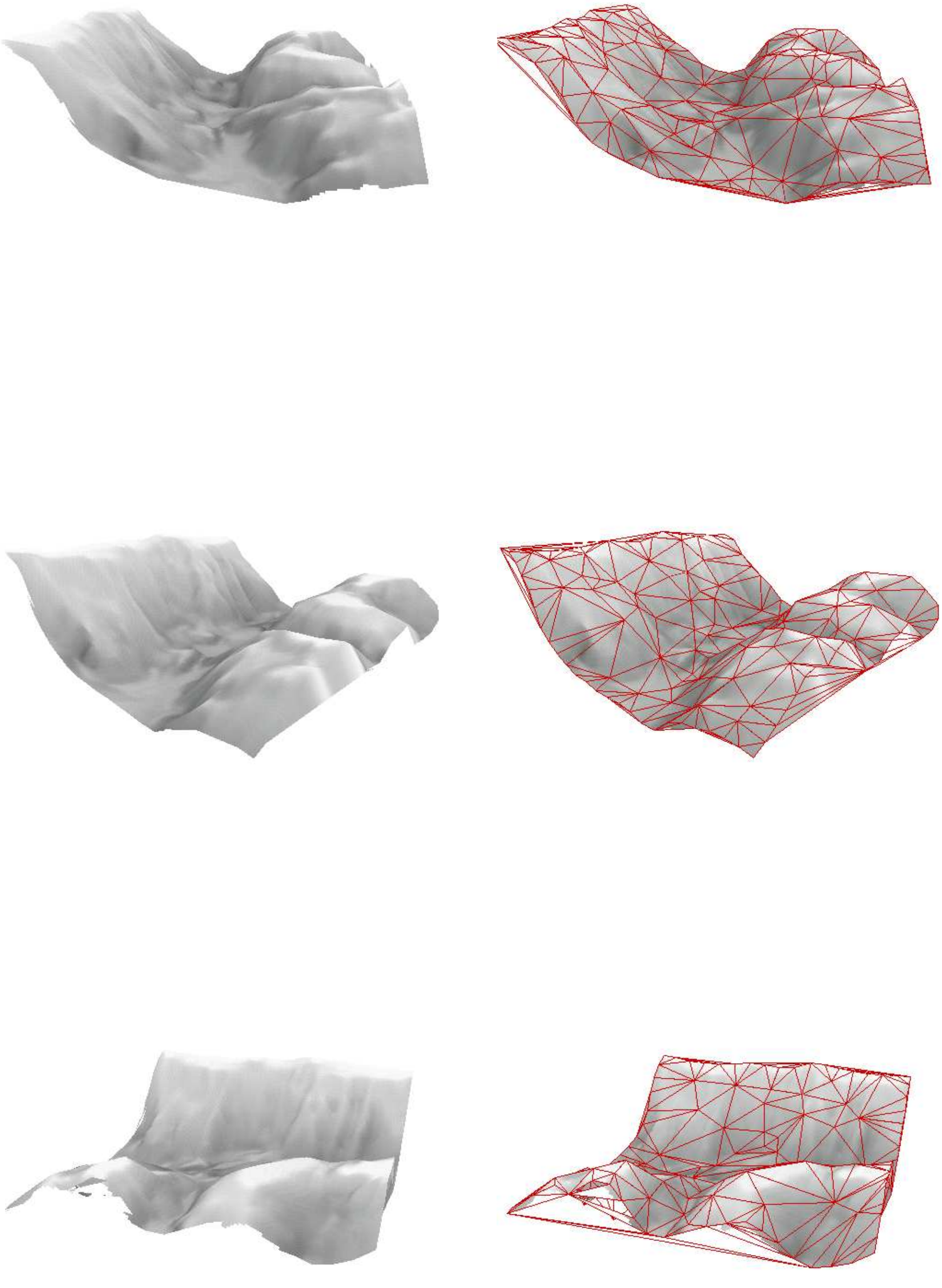


Figure 7.2: Fly–by of the terrain with 20000 triangles.

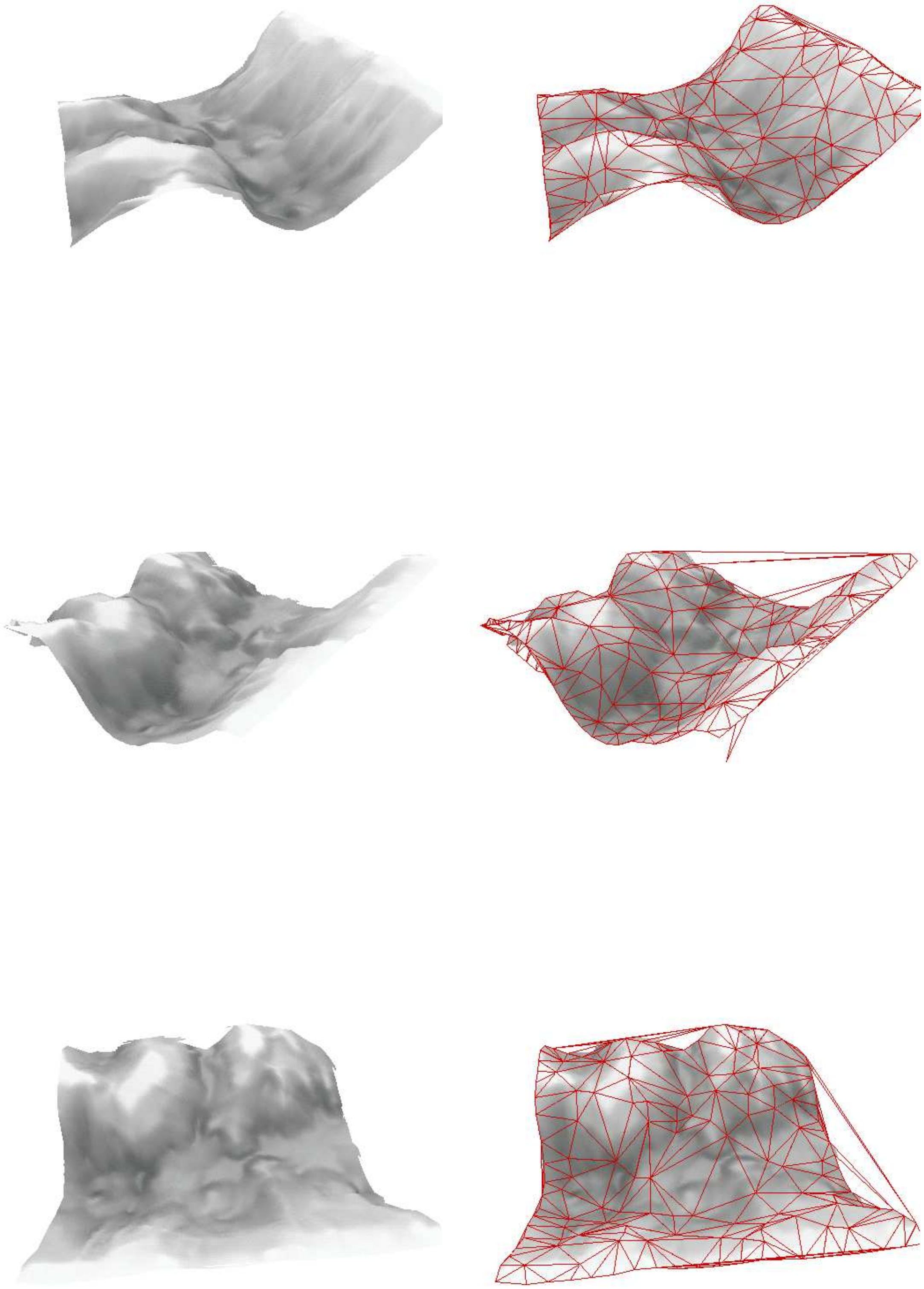


Figure 7.3: Fly–by continued.

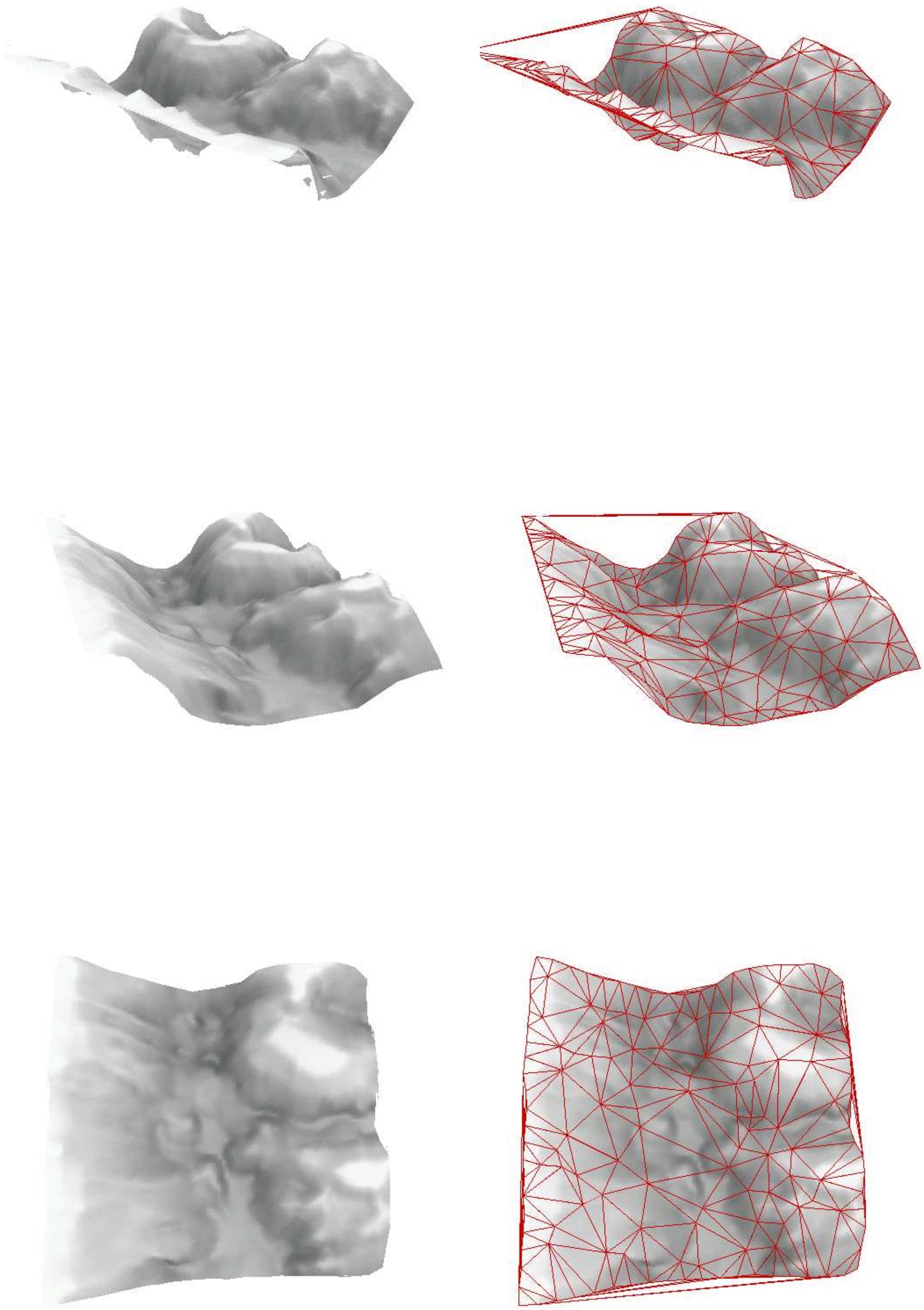
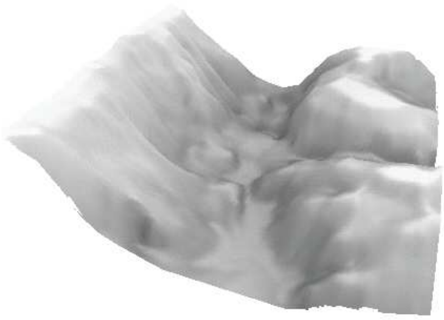
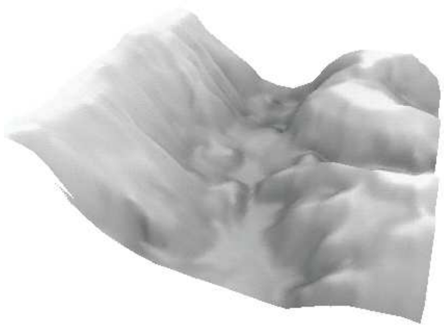


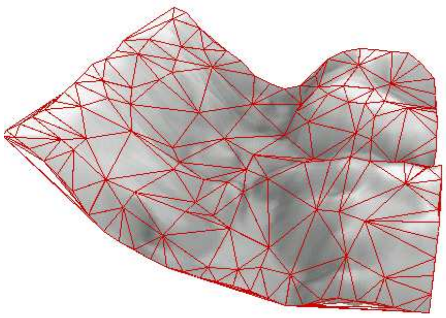
Figure 7.4: Final fly–by shots.



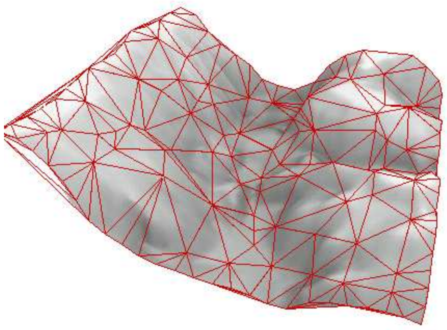
(a)



(b)



(c)



(d)

Figure 7.5: Figures (a) and (b) show a frame–to–frame transition between two different sample viewpoints along a horizontal line. (c) and (d) show this transition with the meshes.

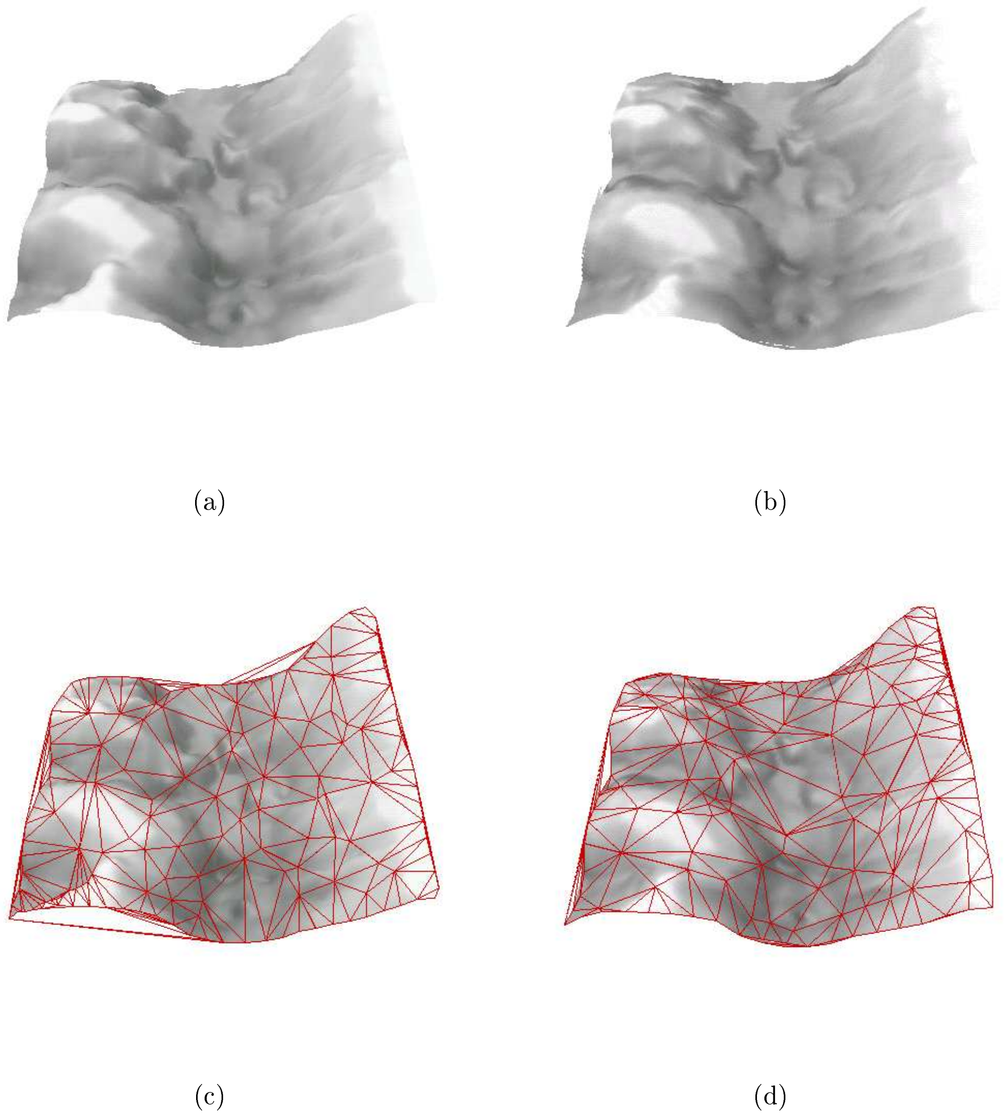



Figure 7.6: Figures (a) and (b) show a frame–to–frame transition between two different sample viewpoints along a vertical line. Figures (c) and (d) show the transition with the meshes.

| Method | Number of Triangles | Average Frame Time | Maximum Frame Time |
|---|---|---|---|
| Geometry | 4600 | 0.064 | 0.077 |
| Unzipped Mesh | 4600 | 0.094 | 0.159 |
| Geometry | 20000 | 0.289 | 0.306 |
| Unzipped Mesh | 20000 | 0.128 | 0.150 |

Table 7.1: Time results for the two interactive fly–bys.

| Method | Number of Triangles | Actual Memory Usage (Mb) | Usage with Compression | Usage with Single Texture |
|---|---|---|---|---|
| Geometry | 4600 | 0.28 | N/A | N/A |
| Unzipped Mesh | 4600 | 317 | 22.3 | 3.34 |
| Geometry | 20000 | 0.120 | N/A | N/A |
| Unzipped Mesh | 20000 | 317 | 22.3 | 3.34 |

Table 7.2: Memory usage for both of the methods and terrains.

with the larger terrain.

In this example, the amount of memory used by the impostors is enormous. Almost all of this memory is used to store the texture maps. This particular example used 25 different sample points, and unzipped the mesh into four depth–sets. This means that 100 colour textures need to be stored, and all have size 512 x 512. This accounts for 314.5 Mb of the memory seen in the table under "Actual Memory Usage". Using textures of 256 x 256 is usually sufficient, and this quarters the amount of memory needed (so texture memory is reduced to 78.6 Mb). Also, it may be possible to combine the textures for each viewpoint into a single texture (instead of one texture per depth–set). This will be discussed further in the final chapter. In this example (with four depth–sets per sample viewpoint) the texture memory usage would be quartered again, with a final texture memory usage of 19.7 Mb, which is much more reasonable. The data structures themselves use 2.5 Mb, which could be reduced, but effort was not put into this, since it is insignificant compared to the textures. The "Usage with Compression" column in table 7.2 shows these estimates, using both forms of texture compression.

The final column in the table shows memory usage if only a single texture (sized 256 x 256) is used. This is what would be used with view–independent methods, such as LOD methods with lifted textures. All meshes from every point of view would use only this texture, which is typically a top–view image of the terrain. Clearly this will not be as accurate as view–dependent textures, but reduces memory usage.

# Chapter 8

# Conclusions and Future Work

## 8.1 Conclusions

Chapter 7 showed the timing results of unzipped meshes compared to geometry for two different terrain models. Clearly, as the terrain gets larger, using unzipped meshes will generate better frame rates. Even the larger terrain (20000 triangles) is not very large compared to many of those that exist, so one can see that frame rates using simple geometry can quickly become low enough so that an interactive walkthrough becomes impossible. This cannot happen with unzipped meshes, since the frame rate stays constant regardless of the number of triangles in the terrain (aside from small variations due to the position of the viewer with respect to the terrain).

However, the size of the terrain does affect the pre–processing time of unzipped meshes. This happens in two ways. First, to generate the textures, the full geometry of the terrain has to be drawn for each viewpoint. Naturally, this takes longer when the terrain is larger. Secondly, and more importantly, the partitioning of the terrain triangles (using the connected component method) depends on the number of triangles. This process was significantly slower in the large terrain than in the smaller terrain. However, this only needs to be done once for each viewpoint, and then the interactive frame rates are not affected.

The other issue (besides rendering speed) which was addressed is image quality. Obviously drawing the full geometry produces the best results, but this is not feasible for large terrains. Unzipped meshes produce the best usable results in most cases. They result in clearer impostors (no jags), and in accurate parallax effects. However, as will be discussed in the next section, unzipped meshes still produce artifacts which can make it apparent that an impostor is being used. In fact, it is possible (although rare) with unzipped meshes that the background can show through the impostor, since it is not completely connected. In this way, VDTM's have an advantage over unzipped meshes. They are composed of a single connected mesh, so the background cannot appear through the impostor. VDTM's also have the advantage that since the terrain is never partitioned, the pre–processing time is greatly reduced.

Therefore, although unzipped meshes generally produce the best results, there are two cases in which VTDM's may be the preferred choice: if the terrain is very large, or if it varies slowly. In the former case, the pre–processing time for unzipped meshes could be unusably slow, making VDTMs the only reasonable choice. In the latter case, a slowly varying terrain would tend to result in fewer depth discontinuities. Since unzipped meshes are created specifically to deal with these discontinuities, it would be unnecessary and wasteful to use them for such a terrain.

If available memory is very low, then it may be necesary to use an LOD method, which uses only a single texture for all viewpoints. This view–independent texturing leads to more inaccuracies in the impostor, but as discussed in Chapter 7, clearly reduces memory needs.

Finally, there is the issue of interactive frame rates. As mentioned in chapter 1, approximately 30 frames per second are needed for a smooth animation. One can see in table 7.1 that while unzipped meshes did produce results significantly faster than rendering the geometry, they were still only rendered at about 10 frames per second. However, this was run with a window size of 512 x 512, on a machine which uses a software implementation of OpenGL. The frame rates of VDTMs and unzipped meshes are virtually the same, since unzipped meshes differ mostly in the pre–processing step. Table 5.1 shows the results for a VDTM in a 256 x 256 window. This produced a frame rate of almost 22 frames per second, which is near the goal.

## 8.2 Future Work

### 8.2.1 Mesh Construction

There are several ways the mesh construction could be improved. There are two main steps in mesh construction: creating the original VDTM mesh, and then creating the unzipped mesh. The first step could be improved by changing the method in which mesh vertices are chosen. Currently, the vertices are spaced evenly along the image outline and along depth discontinuities. This ignores that some areas have more detailed or complex features that could be lost in the simplified mesh. Therefore, some method could be employed which concentrates vertices in these areas, and reduces the amount in less detailed areas (for example, a section of the outline which is relatively straight).

In unzipped meshes, gaps are created at the depth discontinuities. This allows for the possibility of having the background show through the terrain. Disocclusion triangles are created to fill in these gaps, and they can be textured from an image taken from a higher viewpoint, from where more of the terrain is visible. While this worked well in most cases, there were still certain viewpoint from which the gaps in the mesh were visible. This could be fixed by connecting the mesh to ensure that there are no more gaps. This would involve creating a “floor” in the mesh; new triangles could be added which connect disocclusion triangles to triangles in the nearer ridge. This is seen in figure 8.1. These floor triangles could be filled in with a texture taken from a viewpoint directly above the terrain, to maximize the chance that the bottom of the valleys could be seen. With this floor in place, there would be no more gaps visible in the unzipped terrain.

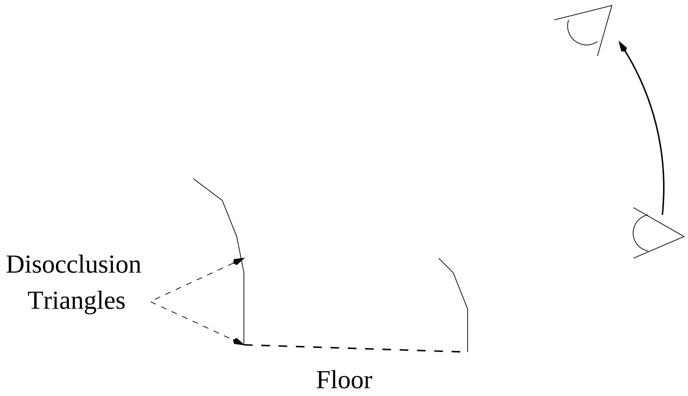



Figure 8.1: A floor is added to avoid gaps showing in the unzipped mesh as the viewpoint moves up from the sample viewpoint.

### 8.2.2 Texture Blending

As the viewpoint moves around an unzipped mesh impostor, a “popping” artifact is seen: When the viewer crosses a threshold between two sample viewpoints, the old imposter is replaced with the impostor created from the new sample viewpoint. This means that from one frame to the next, the mesh and textures of the impostor are replaced. Since the two impostors are created separately, there is no real correspondence between the two (other than that they are representing the same object from similar points of view!), so this transition is noticeable as a “pop” from one impostor to the next.

This popping could be improved by determining a correspondence between the two impostors. It would be very difficult to create a one–to–one mapping between vertices of the two meshes. Rather, the vertices of each mesh could be mapped to new locations on the other impostor image. Therefore for each triangle $t1$ in mesh $m1$, a triangle $t2$ on mesh $m2$ (not necessarily one of $m2$'s mesh triangles) could be computed. For each triangle in $m1$, we now have two different texture which could be used to fill it in. A weighted blending method could then be used (such as those seen in the papers on view–dependent textures [8, 22, 24, 7]), that would create a texture which varies smoothly as the viewpoint moves. When the threshold is crossed and the impostors are swapped, the textures in the new triangles would still be composed of images from both reference images, so the popping effect would be minimized.

This blending method could also be used in texturing the disocclusion triangles. Currently, the original texture within these triangles is simply replaced with a texture generated from a higher viewpoint. This texture would be more accurate if the two textures could be smoothly blended.

### 8.2.3 Efficiency

Creating multiple impostors for each sample viewpoint inherently uses up a fair amount of memory. However, this is increased greatly when using unzipped meshes rather than VDTMs. This is because in unzipped meshes, rather than having a single texture image for each impostor, each impostor uses one texture image for each depth set. This could be improved, since a single texture image for one depth set is very sparse; most of the texture is simply the background, with very little actual terrain information (figure 8.2). Also, these separate textures are often composed of neighbouring parts of the terrain, so the edges of the separate terrain images will often match quite closely. It could therefore be possible to combine all of the textures for one impostor into a single texture image; then the memory usage for textures would be that same as that of VDTMs. Of course, the mapping of texture coordinates would be more difficult, but it could be done, and would not affect the rendering time of each frame.

### 8.2.4 Hierarchical Methods

Finally, the methods implemented in this thesis render the entire terrain as a single impostor. As was seen in much of the related work, it is often useful to use a space partitioning algorithm to divide the scene, and then render different parts of the scene at different levels of detail. In a terrain, for example, the height field could be divided into a quadtree, with impostors being generated for only certain levels of detail. From a certain viewpoint, the nearer quads are drawn with full geometry, and while more distant quads are rendered with impostors. These could be either VDTMs or unzipped meshes, depending on the type of terrain. Using this type of idea, terrain impostors could also be combined with other types of scenes. For example, a building standing in the terrain could be drawn with an urban–specific impostor, while the farther terrain is rendered with a terrain impostor. This combination of methods is viable, since the meshed impostors stay within the region that contained the original geometry. Therefore the different representations of the terrain or other objects would not intersect.

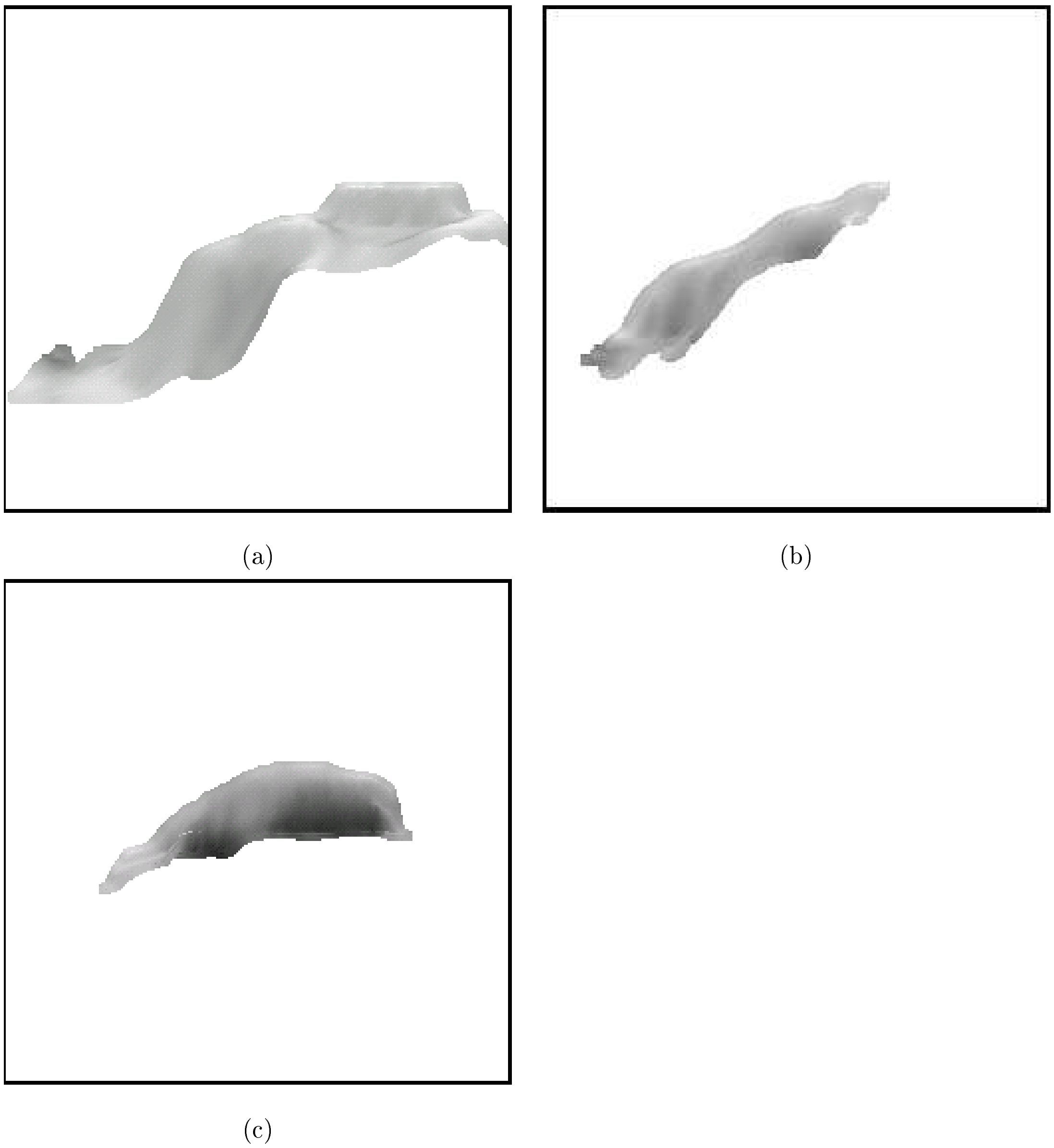


Figure 8.2: Textures for three depth–sets of a sample viewpoint. One can see that these textures could be combined into a single texture, since there is so much wasted space in each.

# Bibliography


[1] Greenberg Atherton, Weiler. Polygon shadow generation. *Computer Graphics (SIGGRAPH '78 Proceedings)*, pages 275–281, 1984.

[2] Chun-Fa Chang, Gary Bishop, and Anselmo Lastra. Ldi tree: A hierarchical representation for image-based rendering. *Computer Graphics (SIGGRAPH '99 Proceedings)*, August 1999.

[3] Shenchang Eric Chen. Quicktime vr - an image-based approach to virtual environment navigation. *Computer Graphics (SIGGRAPH '95 Proceedings)*, pages 29–38, August 1995.

[4] Shenchang Eric Chen and Lance Williams. View interpolation for image synthesis. *COMPUTER GRAPHICS Proceedings, Annual Conference Series*, pages 279–288, 1993.

[5] L. Darsa, B. C. Silva, and A. Varshney. Navigating static environments using image-space simplification and morphing. *Proceedings 1997 Symposium on Interactive 3D Graphics*, pages 25–34, 1997.

[6] Mark de Berg and Katrin T. G. Dobrindt. On levels of detail in terrain. *11th ACM Symposium on Computational Geometry*, June 1995.

[7] Paul Debevec, Yizhou Yu, and George Borshukov. Efficient view-dependent image-based rendering with projective texture-mapping. 1998.

[8] Paul E. Debevec, Camillo J. Taylor, and Jitendra Malik. Modeling and rendering architecture from photographs: A hybrid geometry-and image-based approach. *Computer Graphics (SIGGRAPH '96 Proceedings)*, pages 11–20, August 1996.

[9] X. Decoret, G. Schaufler, F. Sillion, and J. Dorsey. Multi-layered imposters for accelerated rendering. *Proceedings Eurographics '99*, 1999.

[10] Leila De Floriani, Paulo Magillo, and Enrico Puppo. Building and traversing a surface at variable resolution. *IEEE Visualization '97*, pages 103–110, 1997.

[11] Leila De Floriani and Enrico Puppo. Hierarchical triangulation for multiresolution surface description. *ACM Transactions on Graphics*, pages 363–411, October 1995.

[12] Michael Garland and Paul S. Heckbert. Surface simplification using quatric error metrics. *Computer Graphics (SIGGRAPH '97 Proceedings)*, pages 209–216, August 1997.

[13] Steven J. Gortler, Radek Grzeszczuk, Richard Szeliski, and Michael F. Cohen. The lumigraph. *Computer Graphics (SIGGRAPH '96 Proceedings)*, pages 43–54, August 1996.

[14] Paul S. Heckbert and Michael Garland. Multiresolution modeling for fast rendering. *Proceedings Graphics Interface '94*, pages 43–50, 1994.

[15] Hughes Hoppe. Progressive meshes. *Computer Graphics (SIGGRAPH '96 Proceedings)*, pages 99–108, August 1996.

[16] Hughes Hoppe. View-dependent refinement of progressive meshes. *Computer Graphics (SIGGRAPH '97 Proceedings)*, pages 189–198, August 1997.

[17] Sing Bing Kang and Pavan K. Desikan. Virtual navigation of complex scenes using clusters of cylindrical panoramic images. *Proceedings Graphics Interface '98*, pages 223–232, 1998.

[18] Marc Levoy and Pat Hanrahan. Light field rendering. *Computer Graphics (SIGGRAPH '96 Proceedings)*, pages 31–42, August 1996.

[19] P. Lindstrom, D. Koller, W. Ribarsky, L. F. Hodges, N. Faust, and G. A. Turner. Real-time, continuous level of detail rendering of height fields. *Computer Graphics (SIGGRAPH '96 Proceedings)*, pages 109–118, 1996.

[20] David Luebke and Carl Erikson. View-dependent simplification of arbitrary polygonal environments. *Computer Graphics (SIGGRAPH '97 Proceedings)*, pages 199–208, August 1997.

[21] Paulo W. C. Maciel and Peter Shirley. Visual navigation of large environments using textured clusters. *Symposium on Interactive 3D Graphics*, pages 95–102, April 1995.

[22] William R. Mark, Leonard McMillan, and Gary Bishop. Post-rendering 3d warping. *1997 Symposium on Interactive 3D Graphics*, pages 7–16, 1997.

[23] Leonard McMillan and Gary Bishop. Plenoptic modeling: An image-based rendering system. *Computer Graphics (SIGGRAPH '95 Proceedings)*, pages 39–46, August 1995.

[24] K. Pulli, M. Cohen, T. Duchamp, H. Hoppe, L. Shapiro, and W. Stuetzle. View-based rendering: Visualizing real objects from scanned range and color data. *Proceedings of 8th Eurographics Workshop on Rendering*, pages 23–34, June 1997.

[25] Gernot Schaufler. Per-object image warping with layered imposters. *Proceedings of the 9th Eurographics Workshop on Rendering '98*, pages 145–156, June 1998.

[26] Gernot Schaufler and Wolfgang Stuerzlinger. A three dimensional image cache for virtual reality. *Computer Graphics Forum, 15(3), Proceedings Eurographics '96*, pages 227–236, September 1996.

[27] J. Shade, S. Gortler, L. He, and R. Szeliski. Layered depth images. *Computer Graphics (SIGGRAPH '98 Proceedings)*, pages 231–242, July 1998.

[28] J. Shade, D. Lischinski, D.H. Salesin, T. DeRose, and J. Snyder. Hierarchical image caching for accelerated walkthroughs of complex environments. *Computer Graphics (SIGGRAPH '96 Proceedings)*, pages 75–82, August 1996.

[29] Francois Sillion, George Drettakis, and Benoit Bodelet. Efficient imposter manipulation for real-time visualization of urban scenery. *Proceedings of Eurographics '97*, pages 207–218, September 1997.

[30] A. James Stewart. Hierarchical visibility in terrains. In *Eurographics Rendering Workshop*, pages 217–228, June 1997.

[31] Wolfgang Stuerzlinger. Imaging all visible surfaces. *Proceedings Graphics Interface '99*, pages 115–122, June 1999.

[32] David C. Taylor and William A. Barrett. An algorithm for continuous resolution polygonalizations of a discrete surface. *Proceedings Graphics Interface '94*, pages 33–41, 1994.

[33] H. Weghorst, G. Cooper, and D. Greenberg. Improved computational methods for ray tracing. *ACM Transactions on Graphics*, 3(1):52–69, January 1984.

[34] G. Wolberg. Digital image warping. *IEEE Computer Society Press*, 1999.

[35] Julie C. Xia and Amitabh Varshney. Dynamic view-dependent simplification for polygonal models. *IEEE Visualization '96*, pages 327–334, 1996.