Investigating the Feasibility of Volumetric Billboards in Games

by

Rashid Bhamjee, BSc Computer Science

Dissertation

Presented to the

University of Dublin, Trinity College

in fulfillment

of the requirements

for the Degree of

Master of Science in Computer Science

(Interactive Entertainment Technology)

University of Dublin, Trinity College

August 2011

Declaration

I, the undersigned, declare that this work has not previously been submitted as an exercise for a degree at this, or any other University, and that unless otherwise stated, is my own work.

Rashid Bhamjee

August 29, 2011

Permission to Lend and/or Copy

I, the undersigned, agree that Trinity College Library may lend or copy this thesis upon request.

Rashid Bhamjee

August 29, 2011

Acknowledgments

I would like to thank my supervisor John Dingliana and everyone who helped me during this project.

Rashid Bhamjee

University of Dublin, Trinity College
August 2011

Investigating the Feasibility of Volumetric Billboards in Games

Rashid Bhamjee
University of Dublin, Trinity College, 2011

Supervisor: John Dingliana

Real time applications typically render simplified versions of distant or unimportant objects to create scenes that appear more detailed than they actually are. Such objects might be rendered using low resolution polygon meshes or 2D images, called billboards. Volume rendering is a technique usually found in the domain of medical imaging but has recently been proposed for use in interactive entertainment applications. Volumetric billboards is a technique described by Decaudin and Neyret in which volumetric representations of objects are used for simplified rendering. With recent advances in GPU speeds, on board memory, and programmable pipelines, real time volume rendering is possible.

The feasibility of using volume billboards in games is investigated by implementing a volume and polygon rendering application. Volumes are evaluated against polygonal

meshes in terms of rendering performance and memory usage. Considerations for a practical implementation of the technique and integration into a polygonal rendering pipeline are discussed.

Contents

Acknowledgments **iv**

Abstract **v**

List of Tables **x**

List of Figures **xi**

Chapter 1 Introduction **1**

1.1 Background 1

1.1.1 Level of Detail Rendering 1

1.1.2 Volume Data 2

1.2 Motivation 3

1.3 Contributions 4

1.4 Dissertation Layout 4

Chapter 2 State of the Art and Related Work **6**

2.1 Volume Rendering 6

2.1.1 3D Slice Based Techniques 7

2.1.2 Ray Casting 8

2.1.3 Empty Space Removal 9

2.2 Impostors 10

2.2.1 Billboards 10

2.2.2 Billboard Clouds 11

2.2.3 3-View Impostors 11

2.2.4 Volumetric Billboards 12

2.3 Mesh Voxelisation 12
2.4 Level of Detail 14
2.4.1 Polygon Mesh Level of Detail 16
2.4.2 Automatic Mesh Simplification 16

Chapter 3 Implementation **17**
3.1 Technologies Used 17
3.2 Scene Objects 18
3.3 Model Loading and Rendering 19
3.4 Volume Rendering Pipeline 20
3.5 GPU Prism-Plane Intersection 22
3.6 CPU Prism-Plane intersection 23
3.7 GPU Volume Compression 24
3.8 Slicing Rate 25
3.9 Slicing Optimisation 26
3.10 Volume Generation 27
3.11 Errors With Flat Opaque Surfaces 28

Chapter 4 Evaluation **30**
4.1 Test Setup 30
4.2 Volume and Mesh Performance 30
4.3 Volumes on Screen 32
4.4 Texture Switching Cost 33
4.5 CPU vs GPU Implementation 34
4.6 Volume Memory Requirements 35
4.6.1 GPU Storage 35
4.6.2 Disk Storage 37

Chapter 5 Conclusions **39**
5.1 Future Work 40

Appendix A Appendix **41**
A.1 Vertex Program 41
A.2 Fragment Program 41

A.3 Geometry Program 42

Appendices **41**

Bibliography **44**

List of Tables

4.1 Disk space required for a set of test volumes. 38

List of Figures

1.1 Level of detail example 3

2.1 Object space vs image space slicing 7
2.2 Proxy geometry box. 8
2.3 Volume empty space removal. 10
2.4 GPU mesh voxelisation overview. 14

3.1 Scene in the volume billboards system. 18
3.2 Polygonal and volumetric scene objects. 19
3.3 Slice polygon winding order. 24
3.4 Undersampling, ideally sampling, and oversampling a volume 26
3.5 Volume generation from polygonal models. 29
3.6 Shimmering artifacts. 29

4.1 Volume vs triangle rendering performance. 31
4.2 Number of pixels a volume projects to. 32
4.3 Rendering performance for a number of distant volumes. 33
4.4 Texture switching overhead. 35
4.5 Performance of CPU and GPU volume slicing. 36
4.6 Performance of CPU and GPU volume slicing when no sliced polygons are rendered. 36
4.7 GPU memory required for volumes and meshes. 37

Chapter 1

Introduction

Volume billboards are an image based representation of three dimensional objects proposed by Decaudin and Neyret[1] as an alternative to polygonal meshes. Such a representation could be useful as a replacement for low resolution meshes and two dimensional billboards commonly used to render simplified versions of complex objects. This chapter introduces the concepts of level of detail and volume rendering which lead on to the motivation behind this dissertation and the contributions made.

1.1 Background

1.1.1 Level of Detail Rendering

Rendering scenes in real time involves a compromise between the complexity of the rendering and the time it takes to produce the final image. When rendering a large number of objects on screen at once, performance can be significantly increased by drawing less detailed versions of objects at certain times. The metric used to decide when to use a simplified version of an object is application specific. Making the choice based on distance to the viewpoint is just one metric that could be chosen. Such a choice is made since distant objects will map to fewer screen pixels than when they are viewed up close, and might not be as important in their contribution to the overall scene. This allows for a less detailed version of the same object to be rendered and appear very similar to how the more detailed would.

This is referred to as level of detail. Simplified versions of an original object are

commonly referred to as impostors. The term impostor conventionally refers to a 2D representation of a 3D object. In this report, we use the term impostor to refer to any simplified representation of an object. There are many different techniques used for representing and generating impostors; some common approaches are discussed in section 2.2.

How different levels of detail for a single object differ depends on how the object is represented and rendered. For polygonal meshes, a simplified version of the mesh is created or automatically generated with the goal of representing the surface of the original mesh as accurately as possible using fewer vertices, reducing the the amount of data that gets sent through the rendering pipeline. Simplification is not restricted to geometric approaches and can involve anything that requires less work to be performed. For example, using a less detailed shading model can speed up the fragment shader and using smaller textures can improve texture fetch speeds by allowing better cache coherency.

1.1.2 Volume Data

Volumetric data is used to represent discretely sampled data in three dimensional space. The data stored is arbitrary and depends on the desired use of the volume. A volumetric dataset used for rendering can be thought of as the 3D analog of a 2D image. The volume data consists of a regular three dimensional grid of sample points, and can be visualised as a stack of 2D images. Each sample point is called a voxel, similar to an image, where each sample point is called a pixel. Volume rendering is used in medical imaging to visualise the results obtained from body scans and is also used in offline visual effects to represent phenomena such as smoke or water. The information stored in each voxel is determined by the rendering technique used, for example, it might be useful for voxels to store colour and normal information. Different volume rendering techniques are discussed in section 2.1.

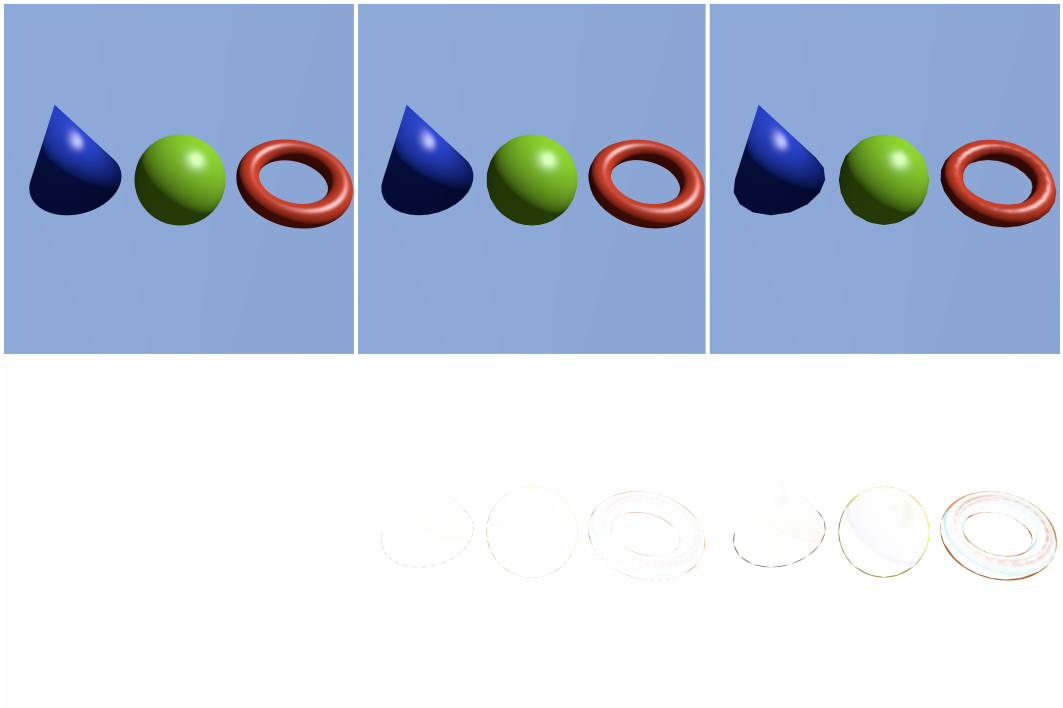

Figure 1.1: The top row shows three level of detail representations of the same scene. The left image contains the most detailed objects and has a total of 4356 vertices, the middle image uses 1092 vertices, and the right image uses 258 vertices. The bottom two images show the difference between the full detail rendering and the level of detail renderings. Both simplified rendering show differences around the edges of smooth shapes. Shading differences cannot be seen in the middle image, while the most simplified representation shows noticeable differences on the torus.

1.2 Motivation

The introduction of programmable stages in modern graphics processing unit (GPU) pipelines make it possible to implement rendering techniques other than rasterisation that take advantage of the GPU's parallel architecture. Volumetric billboards relies on a fast plane/prism intersection algorithm that can be implemented using the GPU's geometry shader. With the increase in GPU speeds, real time volume rendering for distant objects may be a better alternative to discrete polygonal mesh level of detail. It has been shown that volume billboards can be created from meshes and rendered in real time. The question of whether there are benefits to integrating volume billboards

into a game's level of detail system should be answered.

1.3 Contributions

This dissertation aims to investigate whether volumetric billboards could be used as impostors in games as a replacement for low resolution polygonal meshes. Only static impostors are considered as voxel animation is still an open research topic. Specifically, the following topics are discussed:

- Implementation of a volume billboard system to visualise polygonal models and volume billboards as described by Decaudin and Neyret:
  - This includes detailed information extrapolated from related literature on how to implement such a system.
- Considerations to be taken into account for a practical implementation:
  - The number of triangles a volume is equivalent to based on rendering time.
  - Choosing a slicing rate when multiple volumes with different voxel sizes are in the same scene.
  - Details a level of detail metric should take into account when deciding to switch to a volumetric impostor.
- Evaluation of volume billboards against polygon meshes in terms of:
  - Determining when the performance of volume rendering is better than a polygonal mesh.
  - Memory usage during rendering.

1.4 Dissertation Layout

The remainder of this dissertation is organised as follows:

Chapter 2 reviews related work and both seminal and state of the art literature on the topics of level of detail, impostors, volume rendering, mesh simplification, and mesh voxelisation.

Chapter 3 discusses implementation of the volume billboards system. Details are given on how the system was implemented. Issues and considerations that arose are highlighted.

Chapter 4 evaluates the performance of the system and compares the rendering speed of volume billboards against polygonal meshes. Memory usage during rendering and offline storage is discussed and contrasted.

Chapter 5 discusses the conclusions drawn from the implementation and evaluation and summarises the results obtained. Possible future work is mentioned.

Chapter 2

State of the Art and Related Work

2.1 Volume Rendering

Volume rendering creates a two dimensional visual representation of a volumetric data set. Volume data consists of a uniform three dimensional grid of samples, where each sample stores some data required for rendering, such as colour and normal information. Each sample is referred to as a voxel, which can be thought of as a volumetric (3D) pixel, or volume element. Polygonal techniques represent objects by modeling the object's surface, while voxels easily allow the modeling of interior details. Techniques such as texture mapping are used to create the illusion of additional details and geometry in rasterisers. In a voxel based renderer such techniques are not required as each voxel primitive stores its own rendering related information, at the expense of a larger memory overhead [2], but allows voxels to be the basic building block used to create scenes.

Recent research shows that it is possible to render large voxel data sets in real time on current consumer hardware [2, 3]. In terms of performance, voxels could be used as an alternative to polygons, but other limitations such as large memory requirements and inefficient animation techniques are areas which still need further research for voxels to be viable for large, dynamic scenes. The following sections outline some techniques used to render static volume data in real-time.

2.1.1 3D Slice Based Techniques

Slice based volume rendering produces a visualisation of volumetric data by rendering a number of textured polygons using rasterisation [4, 5, 6, 7, 8]. Volume data is stored in a three dimensional texture which allows for trilinear interpolation between samples. Modern GPUs provide fast 3D texture access and filters [9].

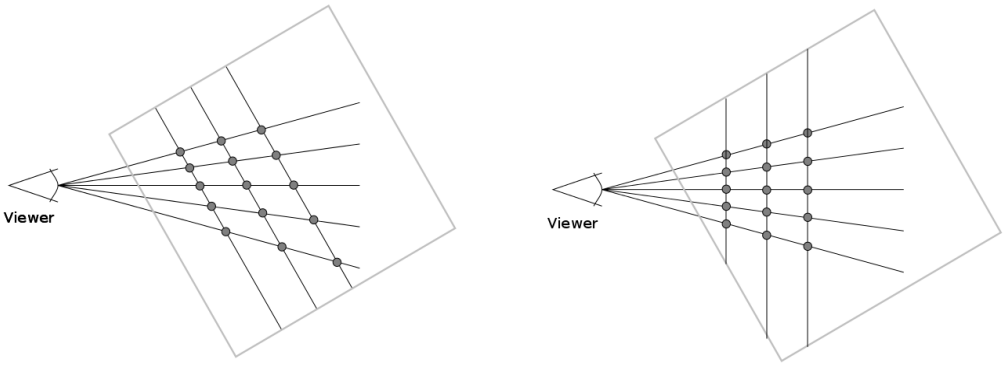

Figure 2.1: The left image shows a 2D visualisation of object aligned slices being taken through square proxy geometry. As the camera rotates about the volume, the slices become parallel to the viewing direction. The right image shows view aligned slices being taken through the same volume. Image from [5].

Proxy geometry (e.g. a box) is used to represent the volume's bounds. Texture coordinates are assigned to the geometry's vertices to create a mapping between the proxy and the volume data. There are two main ways in which a volume can be sliced: viewport aligned slices parallel to the view plane can be taken through the volume's proxy geometry, or object aligned slices can be taken through the proxy geometry. Each approach gives a number of polygons to render. When taking viewport aligned slices, view aligned planes must be computed and clipped to the proxy geometry based on the geometry's current transform. Object aligned slices are easier to compute, but give poor visual results when not looking perpendicular to the slicing direction. To alleviate this issue, multiple sets of object aligned slices may be taken and the set which provides the best rendering for the current viewpoint chosen. In the case of rectangular proxy geometry, a set of slices can be taken aligned with each of its six faces. It is possible to use a set of 2D textures with object space slicing instead of a single 3D texture but

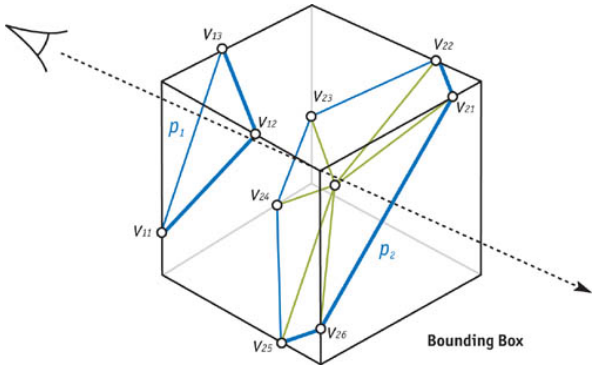

Figure 2.2: A box used as proxy geometry with two camera aligned slices p_1, and p_2. All rendered volumetric data must be contained within the box's bounds. Image from [8].

this prevents taking advantage of hardware trilinear interpolation.

The texture coordinates for a slicing polygon are computed based on where the generated slice intersects the proxy geometry. Each slice takes its interpolated colour and any other required rendering information from one or more 3D textures. The volume is then rendered by drawing each slice from back to front to get correct blending results.

Typically, simple proxy geometry would be used, as many plane intersections must be performed against it to generate the slices for rendering. In interactive applications, slices and polygons would have to be generated each time the volume is rendered since slice polygons are generated based on both the camera and volume transforms.

Major performance problems occur when the volume data required to render an image is too large to fit into texture memory. In such a case, the volume would have to be split into smaller sub-volumes which can fit into memory, and each sub-volume rendered individually. Such an approach incurs expensive memory transfer to the GPU and requires padding each chunk to get correct interpolation results across chunk seams [7].

2.1.2 Ray Casting

Rendering of volumetric data using ray casting involves casting rays from the camera, stepping along each ray into the volume data, and evaluating a rendering equation at

each sample point [7]. Each ray can be evaluated independently, making ray casting an embarrassingly parallel rendering algorithm. Ray casting is much more flexible than slice based approaches, and can be efficiently implemented on modern GPUs using shader programs. The algorithmic complexity for ray casting is based more on the quality of the final image produced (image based) rather than the data set used (object based) [7]. Empty space can be skipped on a per ray basis and each ray can be terminated early when it reaches a predefined opacity or hits an occluder.

The data structure the rays are cast into can be stored on the GPU in a compact and efficient representation. Crassin et al. [3] used an octree where each leaf contained a small 3D grid of voxels. This provided a good representation for scenes with large areas of contiguous volume data or empty space but is not ideal when voxels are sparsely distributed. Laine et al [2] focused on rendering extremely detailed scenes where the view frustum could potentially include several gigabytes of volume data. A sparse octree where each leaf contained a single voxel was used to provide quick traversal and empty space skipping. It is possible to stream small chunks of voxel data into memory as required, and dynamically update the rendering data structures on the fly. Hierarchical tree representations, such as sparse octrees, allow level of detail rendering and coarser approximations to be rendered while fetching voxels from memory [3].

2.1.3 Empty Space Removal

Volume data is likely to contain a non-trivial amount of empty space. This can reduce the performance of some rendering algorithms which treat all space within a volume equally, such as the slice based algorithm described in section 2.1.1. The idea behind empty space removal is to improve performance by breaking up a volume into multiple smaller volumes to reduce the overall amount of empty space [10]. Depending on the rendering algorithm used, the ratio of space removed to the the number of additional volumes generated must be controlled to get optimal efficiency since an additional volume could produce overhead greater than the removed empty space.

The algorithm described by Vidal [10] works well in the general case, but due to it only performing local optimisation, it cannot remove the space inside an object (e.g. a hollow cube). Using global optimisation, at the cost of additional computational complexity, the internal empty space could be removed.

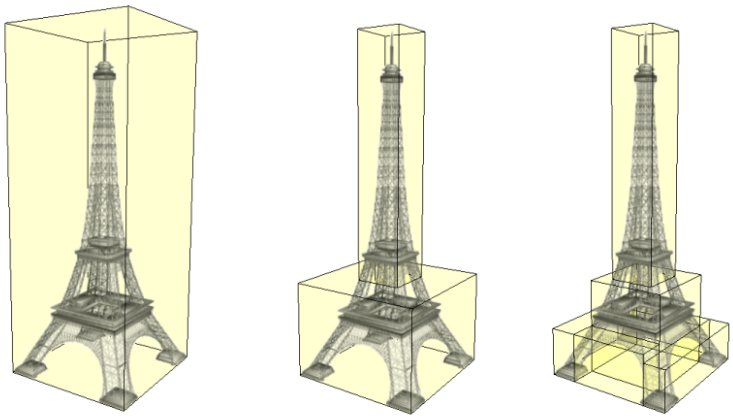

Figure 2.3: A volumetric data set divided up into multiple sub-volumes to remove areas of empty space. Approximately 75% of the original volume's space is removed and 7 volumes generated. Rendering time is approximately 3 times faster. Image and results by [10].

2.2 Impostors

Impostors are simplified representations of an original object and are used when rendering the object in full detail is not necessary (e.g. due to being far from the viewer), or when a detailed representation cannot be rendered fast enough [11]. View-independent impostors use the same data regardless of the view while view-dependent impostors are only valid for a subset of possible views. While view-dependent impostors might provide quicker rendering than view-independent alternatives, either multiple impostors must be precomputed and loaded/stored in memory or generated on the fly when the view changes.

2.2.1 Billboards

Billboards are used to efficiently render distant objects by rendering a small number of textured quadrilaterals instead of a more detailed polygon mesh. In the most simple form it consists of a single textured quad. Billboards can be aligned to always face the camera and is typically done when representing an object using a single quad. An

object may have multiple single view-dependent billboards to represent it from a set of different view directions [12, 11]. While billboards are typically extremely fast to render they are only suitable for rendering small, extremely distant objects since they do not provide an accurate representation of an object, lack suitable parallax effects, and lack accurate depth information.

2.2.2 Billboard Clouds

A simplified representation of a mesh is created using a set of view-independent billboards [13]. The billboards are created by computing a set of planes that represent a mesh's geometric shape to within a defined error bound, and are textured projecting mesh triangles onto the billboard cloud. The number of billboards is significantly smaller than the number of triangles in the original mesh but each billboard must be uniquely textured. Such a representation has many artifacts when viewed up close and is intended to represent distant objects while maintaining an approximate geometric shape, parallax, and dept information.

2.2.3 3-View Impostors

3-View impostors are represented by three views of the object stored as textures, which are used to construct a volume to intersect view rays against [14]. Unlike billboard clouds, the memory usage per impostor is bounded. The created impostors are view independent, but are only useful for rendering objects at a distance. Since the rendering performance is based on the number of pixels an impostor covers, performance can get worse than using the object's true geometry for close views. Each rendered pixel is more expensive to generate compared to using textured polygons as the pixel's view ray must be intersected with the 3-view geometry. A major limitation of the technique is the inability to represent complex objects such as trees. 3-View impostors greatly improve rendering performance but lack a comparison against other impostor techniques that provide a similar level of detail.

2.2.4 Volumetric Billboards

Volumetric billboards are image based representations of three dimensional objects [1]. The aim of the technique is to render complex and translucent objects. The technique can be used for object level of detail, volumetric texturing (e.g. rendering fur over the surface of a mesh), and rendering of volumetric models (e.g. clouds, trees, fur) which would require a large number of polygons to represent. Such data can be produced, for example, by voxelising a polygonal representation of an object. This approach solves some visual problems commonly associated with billboards: incorrect parallax effects, popping on level of detail transition (e.g. when transitioning from a mesh to a billboard representation of a model), translucency, and correct depth interaction with the rest of the scene.

The volume data is stored in a 3D texture and can be efficiently rendered on modern GPUs. Hardware MIP-mapping reduces aliasing and ensures smooth transitions as view distance changes. Due to the 3D representation and rendering of the billboards, correct parallax effects and depth interaction with other scene objects, both volumetric and polygonal, is achieved. As a volumetric object is stored in a 3D texture, memory consumption for this technique is quite high and may significantly limit how applicable volumetric billboards are in practise.

A volumetric billboard is represented by one or more cells. A cell is a prism and can be arbitrarily placed in a scene. Prisms were chosen as they can be efficiently sliced for rendering and can be used to apply volumetric texturing to a triangular mesh. The triangles on the surface of a mesh can be extruded to create prisms that cover the surface. Typically, billboards of objects are boxes which can be made by using two cells.

2.3 Mesh Voxelisation

Mesh voxelisation is the process of converting a polygon mesh into a volumetric representation. This process is necessary when importing polygonal assets to be used in voxel rendering. Surface voxelisation generates voxels that follow the surface of a mesh, while solid voxelisation generates voxels that are inside a mesh [15]. Techniques exist to perform GPU accelerated voxelisation of meshes in a relative short time. For meshes

typically found in interactive applications, voxelisation could be performed when a scene is loaded or possibly on a per frame basis, depending on the number of triangles, mesh properties, and desired volume size [16].

GPU based voxelisation algorithms usually involve creating a bounding box (the volume) around a mesh, and projecting and rasterising the mesh's triangles to a view plane. An orthographic projection maps the rendered triangle to a plane of the volume's box. The rasterised triangle's pixel coordinates and depth information can be examined to determine which voxel in the 3D bounding volume it maps to. This process is illustrated in figure 2.4. Depending on the algorithm, constraints may be applied to the input mesh such as it being water tight and having no internal geometry, to gain performance advantages.

Zhang et al. [17] presented a GPU based conservative voxelisation algorithm (all voxels the mesh intersects are correctly recognised), but this approach is slower than other less accurate hardware accelerated algorithms. Schwarz et al. [15] perform fast GPU accelerated surface and solid conservative voxelisation without using the hardware rasteriser. Their custom rasteriser runs at speeds comparable to the hardware's built in rasteriser while providing more flexibility.

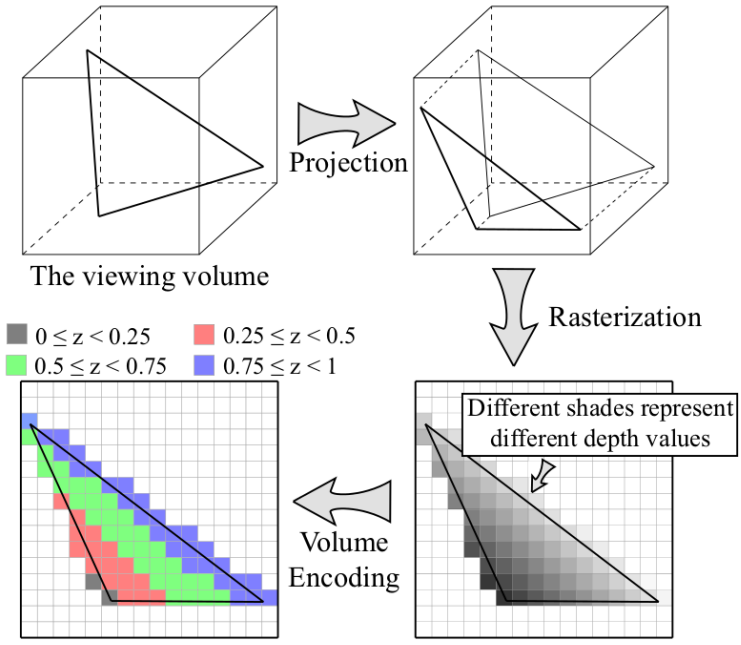

Figure 2.4: Pipeline for performing mesh voxelisation on the GPU. An orthographic projection is used to project a triangle to a plane on the volume's bounding box. The triangle is rasterised and the voxels it passes through obtained from its pixel and depth information. Image by [17].

2.4 Level of Detail

Level of detail (LOD) is a general term that refers to reducing the complexity and the work done by various stages in the rendering pipeline. This leads to faster rendering by doing less work at the expense of a worse final image. By reducing the level of detail on certain objects, more can be rendered in a scene. Which LOD to use for an object is chosen by a LOD algorithm. Common approaches include taking into account distance from the viewer, number of pixels the object will project to in the final image, or the object's importance. Level of detail can also be used to ensure a scene gets rendered within a certain time limit; object complexity can be automatically reduced until the desired frame rate is achieved. Geometric level of detail involves using a coarser representation of an object's geometry. LOD doesn't just apply to objects in

the scene, but anything that can have its complexity changed such as post processing image space techniques, final image resolution, shadows, etc. There are two main categories of LOD algorithms: discrete and continuous [18].

In a discrete LOD algorithm there exists a finite number of LOD representations for an object. Simply changing an object's LOD between frames can cause visual popping. In the case of geometric objects, blending can be used to help smooth the transition [19]. A linear blend between two LODs over a short period of time or any other metric is used but requires rendering the two LOD objects to blend between. In practise this is not a huge issue as only a small number of objects will be transitioning at any time. Another used technique, called alpha LOD, is to only have one representation of an object and to fade it out based on a metric [18]. The objects slowly gets more transparent and is not immediately obvious to the user. Eventually the object will be fully transparent and the object does not have to be rendered any more. The downside to this technique is that the fully detailed object has to be rendered until it has completely faded out and the user may notice the object turning transparent. The first problem can be avoided by switching between simplified representations of the object, perhaps also using LOD blending, but may introduce popping.

In continuous LOD algorithms new LODs for an object are generated as required. For geometry, geomorphing is one possible algorithm that uses a number of discrete object representations with the connectivity between their vertices maintained allowing vertices to be interpolated between LODs [20]. This avoids popping but requires more computation than the discrete alternative and may result in the object noticeably morphing. Subdivision surfaces involve having a low resolution control mesh and recursively dividing it according to some rule. Recent GPUs have programmable tessellation hardware. Tessellation is the process of splitting a polygon into multiple polygons. Displacement mapping is a technique that uses a texture map to define the height a point should be displaced from the surface or a mesh. By combining GPU tessellation and displacement mapping, meshes with high geometric detail can be produced by sending a low resolution mesh to the GPU, having it dynamically tessellate it based on a LOD metric, and using displacement mapping to create more detailed geometry [9]. By combining GPU tessellation and displacement mapping, meshes with high geometric detail can be produced by using a low resolution mesh. This process involves getting the GPU to dynamically tessellate the low resolution mesh based on a

LOD metric, and using displacement mapping to perturb the extra triangle's vertices to create more detailed geometry [9]. Certain fractal and procedural generation techniques fall into this category as they can be re-evaluated as required to produce the exact detail required.

2.4.1 Polygon Mesh Level of Detail

Many real time applications use multiple versions of a model that are chosen to be rendered at run time based on a level of detail algorithm, such as distance to the viewer. Since the simplified model contains fewer vertices and (possibly) smaller texture maps, it can generally be rendered faster than the original mesh. These models can be created manually but there are also automatic methods that aim to simplify meshes.

2.4.2 Automatic Mesh Simplification

The goal of mesh simplification is to provide a mesh with fewer polygons than the true geometry while trying to preserve the overall geometric appearance as much as possible. This particular research area has been studied in depth and many suitable methods currently exist [21]. Decimation methods involve eliminating vertices, edges, and triangles based on given criteria. The approach described by Schroeder [22] removes vertices based on distance or angle metrics and fills in any holes produces. Vertex clustering based approaches replace groups of vertices with a single one but does not preserve topology or details. Wu et al. [23] added global feature preservation to quadratic error metric based simplification.

Research has also been done for commonly used special case meshes. Real time optimally adapting meshes were designed for real time mesh simplification in terrain rendering [24, 25, 26]

GPU tessellation has been used to generate a more detailed mesh from a simpler one and can be used when the original mesh doesn't contain enough detail in the final image. Details can be added to a mesh using displacement mapping, and view-dependent geometry created on the fly as the mesh is rendered [9].

Chapter 3

Implementation

This chapter describes the implementation of the volume billboards system. The system is capable of loading and rendering polygonal models and volumetric billboards as described by Decaudin and Neyret, and incorporates a simple distance based level of detail metric. The process of creating a practical implementation using current graphics API specifications is detailed, something missing from previous works. Decaudin and Neyret define an optimal slicing rate but some extra considerations that must be taken into account when rendering multiple volumes with different voxel sizes are discussed. A separate tool was created to generate volume data from polygon models and is based on previous works.

3.1 Technologies Used

The language chosen was C++ and the OpenGL API is used for rendering. GLSL is the OpenGL shading language which allows user code to be executed at certain stages in the GPU's pipeline. Various other utility libraries were chosen and used to accomplish functionality unrelated to the main system.

The Simple and Fast Multimedia Library (SFML) provides an abstraction layer on top of a number of operating systems for tasks common to interactive applications. SFML was used for window and OpenGL context creation, receiving keyboard input, performance timers, and text rendering.

The Configurable Math Library is a mathematics library intended to be used in

games and graphics applications. It provides a fast templated implementation of mathematical operations and structures commonly required by such applications. The library was used in the system for vector and matrix operations as well as some useful utility operations, e.g. clamping variables to within a legal range.

The Boost project provides a large set of libraries for use in C++ programs. Specifically Boost was used for file system access, command line argument processing, smart pointers, and string manipulation.

Figure 3.1: A scene from the volume billboards system. All objects are rendered using volumes.

3.2 Scene Objects

An object in the system represents a single visual entity in a scene and it is capable of being rendered using either polygonal models or volume billboards. Internally, an object stores a transformation matrix, a list of meshes, a list of volumes, and a LOD metric. The LOD algorithm chooses a single volume or mesh from an object's lists for rendering. The system uses a LOD metric that is based on the object's distance from the viewpoint. A LOD base class is defined and allows specific metrics to be implemented by inheriting the base class and implementing the required virtual functions.

All scene objects are loaded when the application is started. Objects are defined in external YAML files, a data format designed to be human readable, allowing the

properties of scene objects and level of detail algorithms to be changed easily. A C++ YAML parsing library, yaml-cpp, is used to read the object files.

Figure 3.2: A scene with a volumetric object (left car) and a polygonal object (right car).

3.3 Model Loading and Rendering

Common model file formats such as Collada, Autodesk's 3ds Max, and Wavefront object files have large non-trivial specifications and can be complicated to parse. The Open Asset Import Library is an open source C and C++ library for reading various model formats into its own internal representation an application can more easily use. Only a subset of the data typically associated with a model is required by the system. Specifically vertex positions, normals, texture coordinates, and texture data are used for rendering.

A mesh represents a set of triangles that can be drawn on screen and must store a list of vertices, a list of indices, and a single material. A vertex defines a single point in 3D space and some additional data associated with that point. In addition to a point, a vertex stores a normal vector used for lighting, and texture coordinates. Each element in the indices list is an index into the vertex list. Three indices represent a single triangle which implies the index list must always be a multiple of three. A mesh can be rendered by iterating over the list of indices three indices at a time, using the three indices to index into the list of vertices, and using the vertices to draw a triangle. Index and vertex lists are used since a single vertex can be shared by many triangles. A material defines the surface appearance of the mesh's polygons by storing colour and

texture information. Textures used by materials are cached and reused so the same data is not loaded and stored multiple times.

Since a mesh can only have one material and a model may use multiple materials, a model is represented using a list of meshes. Once a model has been loaded its data is never modified by the system allowing a model to be cached and reused by multiple scene objects. A scene object's transform is used to uniquely position, orient and scale a model before it is rendered.

3.4 Volume Rendering Pipeline

A volume billboard is a 3D texture bounded by a box created using two prisms. A volume stores a 3D texture id, a transform to position it in the scene, and a bounding box for slicing optimisation. All volumes are assumed to be 2×2 cubes with the transform holding a scale that defines their actual size in the scene. By assuming a fixed base size the same vertex data can be used by every volume.

The stages in the system's rendering pipeline can be divided up into two sections. The first deals with loading and initialising volumes which happens at application start up, though this could potentially also happen intermittently as an application is running to load and unload volumes as required. The second section deals with what must be done per frame to render volumes and polygon meshes together. The initialisation stages are as follows:

1. Buffers are created for rendering volume prisms. Each volume uses the same vertex and index buffers along with a unique transform to place it in the world. The buffers need to be created only once and do not need to be modified during rendering.

2. When a volume is required its data is loaded from a file produced by the mesh voxelisation tool. First, the volume's dimensions are read and from this information the number of mipmap levels and the number of colours in each mipmap can be determined. Colour data is then read into a temporary buffer on the CPU. When a colour is read its red, green, and blue components are multiplied by its alpha component (referred to as premultiplied alpha) to simplify the blend

equation and avoid artifacts when the volume is being magnified [1] and the GPU interpolates between colour samples[27, 28, 29]. The colour data is then uploaded to the GPU as single byte red, green, blue, and alpha values, meaning each voxel requires 4 bytes of memory when uncompressed. The volume's bounding box is then set and its transform is created using the scale read from the volume file. Volume data is cached and can be shared between multiple scene objects.

Once the rendering buffers are initialised and one or more volumes have been created the data can then be used for rendering a scene in the following way:

1. All polygonal models should be rendered first. This fills the depth buffer so volumes can be correctly clipped against rendered polygonal data on a per pixel basis. When rendering translucent polygons, the depth buffer should not be updated since any intersecting volume should be visible through the polygon.

2. All volumes to be rendered in the frame are gathered. This step may involve any world space culling the application decides to implement. The current system does not perform any volume culling and assumes all volumes are visible.

3. Data required for the slicing optimisations discussed in section 3.9 are computed.

4. Each slice is drawn by iterating over the list of volumes, and issuing a draw call using the prism vertex and index buffers. The geometry shader can then read a volume's two prisms as GPU primitives to intersect the current slicing plane with. Before a draw call is issued, a volume specific GPU state must be set. The volume's 3D texture must be bound and the volume's transform matrix must be set correctly so the volume appears at the correct size. The actual transform set should be a complete modelview matrix, including the volume's transform and the object that positions it in the scene. For each slice, the GPU must be told about the camera space z-coordinate of the current slicing plane. Volumes are sliced back to front to ensure correct blending. When using premultiplied alpha the blend equation for correct back to front rendering of translucent objects is:

$$colour_{out} = 1 \times colour_{source} + (1 - alpha_{source}) \times colour_{destination}$$

[1] Texture magnification occurs when individual pixels in a texture map to multiple pixels on screen. Similarly, texture minification occurs when multiple pixels in a texture map to one pixel on screen.

where $colour_{out}$ is the final computed colour, $colour_{source}$ is the colour sampled from a volume's 3D texture, and $colour_{destination}$ is the current framebuffer colour at the current pixel.

5. The vertex shader transforms the prism's vertex positions into world space.

6. The geometry shader then performs the prism/plane intersection and outputs a polygon if an intersection is found. The process is described in more detail in 3.5.

7. The pixel shader samples the currently active 3D texture and outputs the source colour the graphics library will use with the specified blending equation.

3.5 GPU Prism-Plane Intersection

Prism/plane intersection is performed using a GPU's geometry shader. A geometry shader receives a single primitive as input and can output zero or more primitives. The input and output primitive types do not have to be the same but the number of primitives output is limited by a particular implementation.

The intersection algorithm used is described in [1] and [30] and has been modified to run using version 3.30 of GLSL. While the prism/plane intersection algorithm is detailed, no information is given about how the rest of it should operate within a complete volume billboard renderer. This section details the issues that had to be solved when integrating the algorithm into the volume billboards system. A code listing of the GLSL shaders created can be found in appendix A.

To perform an intersection test the geometry shader must have access to a volume's prism, transform matrix, projection matrix, and the current slice distance from the camera. The GPU primitive with the maximum number of vertices is the triangle with adjacency information, requiring six vertices, and allows processing a prism in the geometry shader. Using this primitive twice, all 12 vertices of a volume's two prisms can be sent to the GPU at once using index and vertex buffers. The same buffers with a unique transformation matrix are used for each volume.

Prism vertices are required to be in camera space for the intersection test. This transform is done in the vertex shader which gets executed before the geometry shader.

Since vertices output by the intersection algorithm are in camera space they must then be transformed into screen space using the projection matrix before being emitted from the geometry shader.

Once an intersection between a plane and the edges of a prism have been found, texture coordinates for mapping the volume's texture to the rendered polygon must be computed. This can be done using linear interpolation with texture coordinates assigned to the vertices of each intersecting edge found. The amount to interpolate by is computed by finding how far along an edge a slice intersects.

$$t = (zslice - p0.z)/(p1.z - p0.z)$$

In the above equation $zslice$ is the z coordinate of the current slicing plane, $p0$ and $p1$ are the vertices of an edge the plane intersects, and t is the amount to interpolate by. The interpolation amount will always be in the range $0 \leq t \leq 1$ since interpolation is only done along edges that have an intersection with the slicing plane. All coordinates are in camera space with the z-axis pointing in front of the camera.

The output polygon from an intersection test contains between three and five vertices which the original algorithm rendered by emitting up to three triangles from the shader. GLSL only allows points, line strips, and triangle strips to be output from the geoemtry shader. The implementation outputs a triangle strip primitive with up to five vertices; three for the initial triangle and up to two extra points to draw up to the maximum of three required triangles. The resulting polygon from a prism/plane intersection is always convex so triangle strips can be used without any special considerations to the number of vertices required. The problem with emitting a triangle strip is that the intersection test gives vertices in a counter clockwise winding order. This is shown in figure 3.3 where each vertex of the polygon is numbered in the order it is given by the algorithm. To create a valid triangle strip, the vertices must be emitted in the following order: 1, 2, 0, 3, 4.

3.6 CPU Prism-Plane intersection

A CPU version of the prism/plane intersection algorithm was also implemented. This allows the volume billboards system to run on older graphics hardware that supports

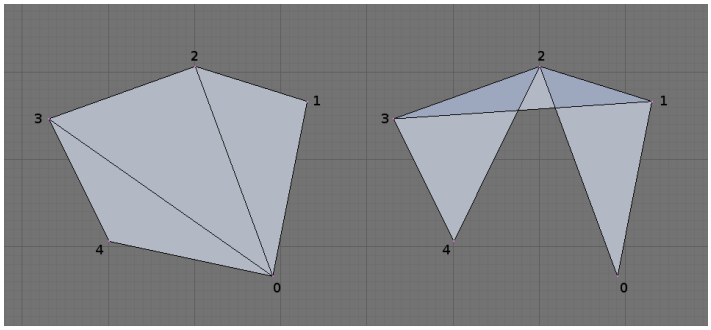

Figure 3.3: The left part of the image shows the winding order of a polygon found by the prism/plan intersection algorithm. The right image shows the error when rendering a triangle strip by using the vertices in order. To produce the correct polygon as a triangle strip vertices 1,2,0 must be submitted first to create the initial triangle, then vertices 3 and 4 are submitted to create the remaining two triangles.

3D textures but lacks a geometry shader. The same concepts described in the previous section are used to integrate it into the main system. In immediate mode OpenGL an application submits vertex information to the graphics library when it draws an object, rather than creating buffers in GPU memory. The polygons created by the intersection algorithm are rasterised using immediate mode. All matrix and vector operations required by the algorithm are done using the optimized configurable math library. Results on the performance of this version compared to the GPU implementation can be found in section 4.5.

3.7 GPU Volume Compression

Graphics cards offer built in S3TC texture compression with on the fly decompression allowing more texture data to be stored in GPU memory and can help increase cache performance[18]. The disadvantage of the S3TC algorithms is that the compression scheme is lossy but in practise this loss in quality is surpassed by the memory reduction and performance gained. There are five variants of the compression algorithm available on recent hardware: DXT1, DXT2, DXT3, DXT4, and DXT5. For volume billboards DXT5 is the ideal choice as it supports a compressed and interpolated alpha channel. The other variants can also store alpha but DXT1 only stores a 1 bit alpha which

cannot be used for translucency. DXT2 and 3 store a 4 bit alpha component per pixel. One of the properties of S3TC algorithm is that the compression ratio is constant meaning the exact number of bytes any texture will require can be determined. RGBA colour data using DXT5 compression offers a 4:1 compression ratio.

The S3TC algorithm works by dividing the image into blocks of 4×4 pixels. Compression and decompression of each block is handled independently of any others. The colour components of the original texture are compressed to 64 bits of information. Two 16 bit RGB color values are computed based on the original colours in the block. During decompression two more colours are created by interpolating the stored 16 bit colours. The interpolation is linear and the generated colours are evenly spaced in the interval $[colour0, colour1]$. Each of the 16 pixels in a block stores a 2 bit number that corresponds to one of the four colours. Alpha is stored in an additional 64 bits by storing two 8 bit alpha values and a 3 bit index per pixel. Unlike the colour lookups, the index is used to choose a predefined function that operates on the stored alpha values and best approximates the original value.

3.8 Slicing Rate

The slicing rate is defined by the distance between each successive slice taken through all rendered volumes. The rate is not constant and should be adjusted based on the current camera projection and volume mipmap levels. Ideally one slice should be taken per slab of voxels in world space. Decaudin and Neyret suggest the slicing plane is stepped along the camera's z-axis in steps the size of one world voxel and suggest the slicing rate could be reduced as a way to increase performance. Mipmapping is taken into account to ensure the volumes aren't needlessly oversampled. This approach works well when the voxels in all the currently visible volumes are the same size, but since volumes can be transformed arbitrarily, each can potentially have a different world voxel size. Therefore the slicing rate must be more carefully chosen based on all visible volumes and will result in either oversampling or undersampling volumes.

When translucency is involved, different sampling rates will produce a different number of slices per volume. If the alpha contribution is not scaled, when blended together, more slices will result in a more opaque object, while fewer slices will result in a more translucent object. For views where the volume is being magnified, volumes

representing opaque objects with thin surfaces might need the sample rate increased to avoid slicing interpolation artifacts. When minifying volumes, the hardware interpolates between mipmaps to sample a value that corresponds to roughly one pixel in the frame buffer. Since the sample rate is dynamically adjusted based on this, potential oversampling is only possible when volumes are magnified. The alpha values for each volume mipmap are correct, and the slicing rate is ideal (approximately one slice per stack of voxels) therefore alpha compensation does not need to be done under minification.

Aside from varying alpha values, oversampling does not produce any artifacts in the final image, while undersampling produces noticeable slicing artifacts as can be seen in figure 3.4. Ideally, the sampling rate should be at least the size of the smallest world space voxel if undersampling is to be avoided.

Figure 3.4: A volume sliced at different sampling rates. The left volume is undersampled by taking slices twice the ideal distance apart causing noticeable slicing artifacts to appear. The middle volume is sliced at the ideal rate with the slicing planes one voxel apart. The right volume is oversampled by slicing at twice the ideal rate producing a slightly smoother image. Oversampling makes translucent parts of the volume appear more opaque as the alpha component of the voxels has not been scaled to compensate for the increased number of slices.

3.9 Slicing Optimisation

The slicing plane is stepped perpendicular to the camera and intersected with all volumes. If a volume is completely in front or behind the current slicing plane the prism/-plane test finishes and does not render a polygon. There are two easy optimisations to consider: skipping the slicing plane over space with no volumes, and only testing the plane against a subset of volumes it could potentially intersect. Decaudin and Neyret

suggest partitioning visible volumes into a number of slabs, however, our implementation takes a different approach based on the idea that volumes will only be used as distant representations of objects.

Before any slicing takes place, all volumes to be rendered are traversed and the prism vertices that are closest and furthest from the camera are found and stored per volume. Volume slicing is started at the furthest point and ends at the closest point found. This skips all empty space between the camera and the nearest volume, and all space after the furthest volume. Empty space between volumes is not skipped over at once but the intersection test will not be run for any volumes when the slicing plane is in such a region. Before running the prism/plane intersection algorithm on a volume, a check is done to make sure the plane's z coordinate is within the volume's minimum and maximum distance from the camera. If it is not, a slice cannot intersect the volume and the intersection test can be skipped completely. Further optimizations were not considered as it is assumed that volumes are being used to render distant objects and that they all lie in roughly the same region.

3.10 Volume Generation

A separate tool was developed to create volume data from polygonal meshes. GPU accelerated voxelisation tools work by rendering slices through a mesh using rasterisation and reading back the frame buffer pixels which represent voxel colours [17, 15]. This process is introduced and described briefly in section 2.3. The voxelisation tool generates volumes as follows:

- A polygonal model is loaded and scaled to fit in a 1×1 cube. This simplifies the camera positioning, slice size and increment distance, and pixel to voxel calculations.
- An orthographic camera is placed in front of each of the cube's six faces (see figure 3.5).
- The framebuffer resolution is set to the desired volume resolution so that one pixel represents one voxel.

- The near and far clipping planes are set to be the distance of one voxel apart, and are stepped along the camera's axis in single voxel increments. The clipping planes ensure one slab of voxels are rendered at a time, effectively taking slices through the model.

- Each time the planes are incremented the model is rendered filling the frame buffer with colour information. During rendering backface culling should be disabled otherwise only half the required colour information will be gatherd.

- The framebuffer is read back and the colour information stored.

The above process is repeated for each of the cube's six faces giving six 3D arrays of colour information. The arrays are then combined by averaging the colour values in each to give the final volume information. Mipmaps are generated in using the same process except the volume resolution is divided by two each time. Figure 3.5 illustrates how the near and far planes are stepped along the model.

The implementation currently only handles volumes that are a power of two and have the same width, height, and depth. A scale factor is stored along with the volume's dimensions and colour information so the volume's proxy geometry can be scaled to match the size of the original polygonal model when rendered.

Volumes are written to disk in a binary file format. The data written by the voxeliser tool and read by the application is not compressed. Section 4.6.2 discusses compression in relation to disk storage. The first three entries in the file are 4 byte integer values representing volume's width, depth, and height. The next entry is a 4 byte float representing the volume's scale factor. The remainder of the file contains colour information for each mipmap level. Each colour is stored using four 4 byte floats representing red, green, blue, and alpha intensities respectively. Intensity values are normalised to be between 0 and 1. Mipmaps are stored sequentially from largest to smallest.

3.11 Errors With Flat Opaque Surfaces

When voxelising a polygonal model, an infinitely thin plane maps to one slice of voxels. When viewing this part of a volume, to get the final colour at a particular sample point

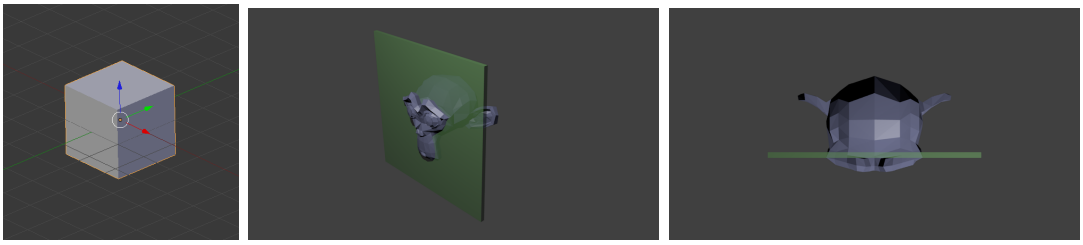

Figure 3.5: When a model is voxelised it is scaled to fit inside a 1×1 cube and an orthographic camera renders slices through it along each of the six cardinal directions as illustrated in the left image. The middle and right images visualise the near a far clipping planes, which are spaced one voxel apart, and get stepped through the model in single voxel increments. The parts of the polygons between the current clipping planes get rasterised into the framebuffer while others are discarded.

on a slice the graphics card will interpolate between the opaque voxels the surface is mapped to and the surrounding transparent voxels. As the camera moves the position of a slice intersecting the opaque voxels may change slightly causing the 3D texture's sample position to also change leading to different interpolation results. This appears as a shimmering effect which can be seen as the camera moves. Figure 3.6 shows the dark colour value from transparent pixels getting mixed in with the white voxels on a surface.

Figure 3.6: Shimmering artifacts caused by moving slicing planes and bilinear texture interpolation. Note the dark diagonal lines appearing along white roof of the car. When the camera is moved, new slicing plane orientation causes the dark lines to appear in a different location.

Chapter 4

Evaluation

The question of whether it is useful to consider using volume billboards in games is answered by using the implemented system to analyse performance and memory consumption. The performance of volume and polygon rendering is compared to determine when volumes might be used instead of polygonal impostors. From this information we can derive approximately how many triangles a volume is equal to in terms of performance. Memory usage during rendering is analysed and contrasted against what meshes require.

4.1 Test Setup

The hardware used to perform the tests is:

- AMD Athlon 64 6400+ @ 3.2GHz.
- 6GB RAM.
- NVIDIA 8800GTS with 320MB GDDR3 RAM (96 stream processors).
- Linux (64 bit) using NVIDIA's Linux driver.

4.2 Volume and Mesh Performance

The performance of rendering a mesh and a volume was measured to determine when volumes should be used in place of meshes in terms of rendering speed. An object

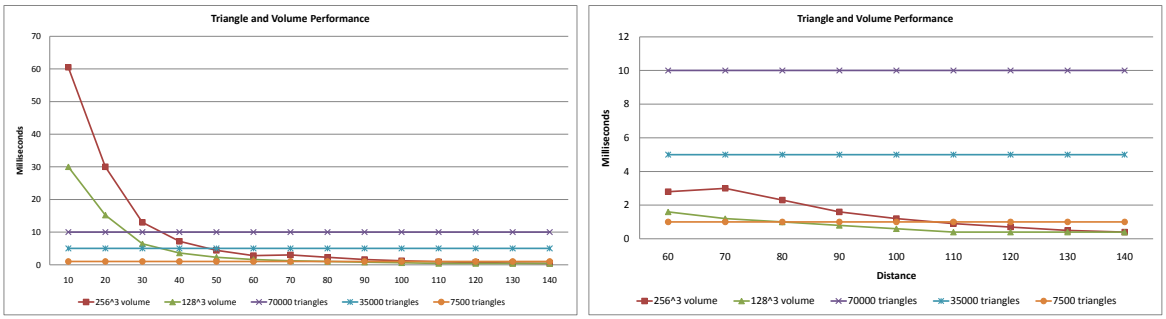

Figure 4.1: The left chart shows rendering time for a meshes and volumes. The right chart shows a more detailed result of the rendering times between a distance of 60 and 140 units from the camera. Volumes outperform the 7500 triangle mesh at a distance of 80 and 105, when projecting to approximately 225^2 and 170^2 screen pixels.

was placed in the scene and rendering time was recorded as the camera moved away from the object. 128^3 and 256^3 volumes and triangular meshes consisting of 7500, 35000, and 70000 triangles were used. Mesh performance is constant over distance indicating its rendering speed is geometry bound. Volume performance is six times worse than polygons up close when all screen pixels were filled by the volume. Figure 4.2 shows the number of pixels a volume projects to over distance at a screen resolution of 1680×1050. From figure 4.1 it can be seen that volume performance scales inversely to the number of screen space pixels a volume projects to.

128^3 volumes outperform a triangle mesh with 7500 vertices at a distance of 80 from the viewpoint, where the volume projects to 225×225 screen pixels. 256^3 volumes only become faster than the mesh at a distance of 105, a projection of 170×170 screen pixels. This is caused by the larger volume having more voxels and therefore requiring more slices to be taken when rendered. At a distance of about 135 the performance of both volumes merges as the projected screen space approaches 128^2. At this distance, the size of the projection means the second mipmap level will be chosen for the larger volume leading to it being treated as the 128^2 volume is.

Since rendering can quickly become fillrate [1] limited, LOD algorithms that switch between meshes and volume billboards should take into account the number of pixels in screen space a volume billboard will end up projecting to. If the calculation was

[1]The number of pixels a graphics card can compute in a given time frame.

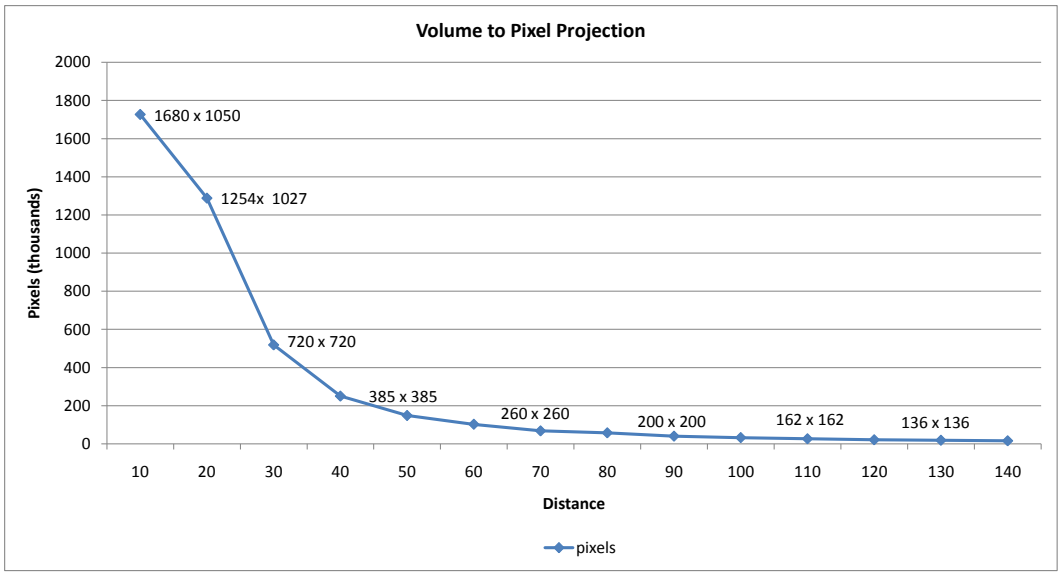

Figure 4.2: The number of pixels a volume projects to based on distance to the camera. Each volume used was a 1×1 cube in world space. The screen resolution was 1680×1050.

based on distance alone then the resolution the application is running at will directly affect performance as the number of pixels rendered will scale with the resolution. Combined with the fillrate requirements of back to front rendering this will cause fillrate limitations to slow down the application when it runs at higher resolutions than its distance based LOD algorithm was designed for.

By observing the rendering time of a volume when one voxel maps to one pixel, an approximate triangular equivalent can be found. From figure 4.1 and 4.2 a 128^3 volume achieves this at a distance of 140. At this distance, the time it takes to render the volume could be used to render approximately 2500 triangles. Similarly, a 256^3 volume has an equivalent voxel to pixel mapping at a distance of 70, making it the performance equivalent of approximately 21000 triangles.

4.3 Volumes on Screen

The number of volumes that could be rendered on screen at once was measured by randomly placing 128^3 volumes in front of the camera. Each volume projected to between

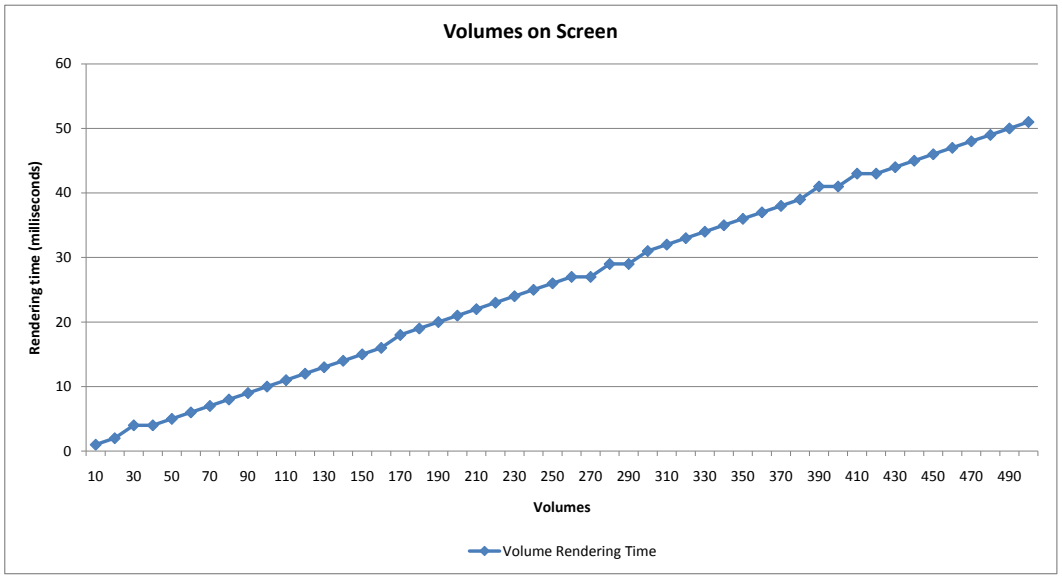

Figure 4.3: Rendering performance for a number of distant volumes.

50×50 and 150×150 screen pixels causing volumes to be both magnified and minified. Our results show that performance scales linearly based on the number of volumes rendered. At a target frame rate of 30 (33 milliseconds per frame) approximately 300 volumes can be drawn.

4.4 Texture Switching Cost

Since one slice must be intersected with multiple volumes before the next slice is taken, it is necessary to change some per volume GPU state for each prism/plane intersection. This requires the application to issue one draw call per volume for each slice, giving a total of *number of volumes* $\times$ *number of slices* draw calls to render all volumes. Each volume may be using a different texture meaning the GPU's texture state must be set between each call. Constant texture switching is not always necessary. In the case of volume texturing (e.g. fur rendering), many volumes use the same texture and therefore require no switching. When rendering polygonal meshes, triangles using the same texture are grouped together in batches. Due to the fact that volume billboards are being used to represent distinct objects, each volume will likely require different

texture data.

The performance of using the same texture for a number of volumes versus switching textures between each volume was measured. The camera was positioned so no pixels were drawn allowing the application and geometry shader performance to be measured without the results being hidden by fillrate costs. The results in figure 4.4 show that texture switching reduces performance by an average of 51% compared to using the same texture.

It would be beneficial if as few textures as possible were used to reduce the switching frequency. Sorting volumes within a slice by texture can help reduce the number of switches required if the slice intersects multiple volumes using the same texture. If all volumes used the same texture, the texture switching overhead would be eliminated. A single virtual texture could be used to store all volumes in a single texture [31]. Depending on the number and sizes of volumes required, creating a virtual texture may waste some memory due to empty texture blocks. If all volume textures are the same size, then the virtual texture can be divided into equally sized sub textures which can be updated individually. Different size textures would cause some complications when updating the atlas. Frequent swapping of volumes of different sizes may cause holes of empty space to appear in a similar fashion memory fragmentation which may require a form of periodic memory compaction and updating the texture coordinates of all volumes as appropriate.

4.5 CPU vs GPU Implementation

The CPU implementation outperformed the GPU for our test scene. Figure 4.5 shows the performance of both implementations measured by rendering an increasing number of volumes. The CPU implementation was an average of 2.64 times faster.

The performance of just volume slicing was measured by positioning the camera so that no slice polygons were rasterised. From figure 4.6 the GPU is consistently faster than the CPU leading to the conclusion that the performance difference must lie somewhere else in the pipeline. While the exact cause of the performance difference hasn't been determined, a potential candidate is that the OpenGL driver may be batching polygons when submitted using immediate mode while the geometry shader may cause the pipeline to stall or be flushed each time it's run.

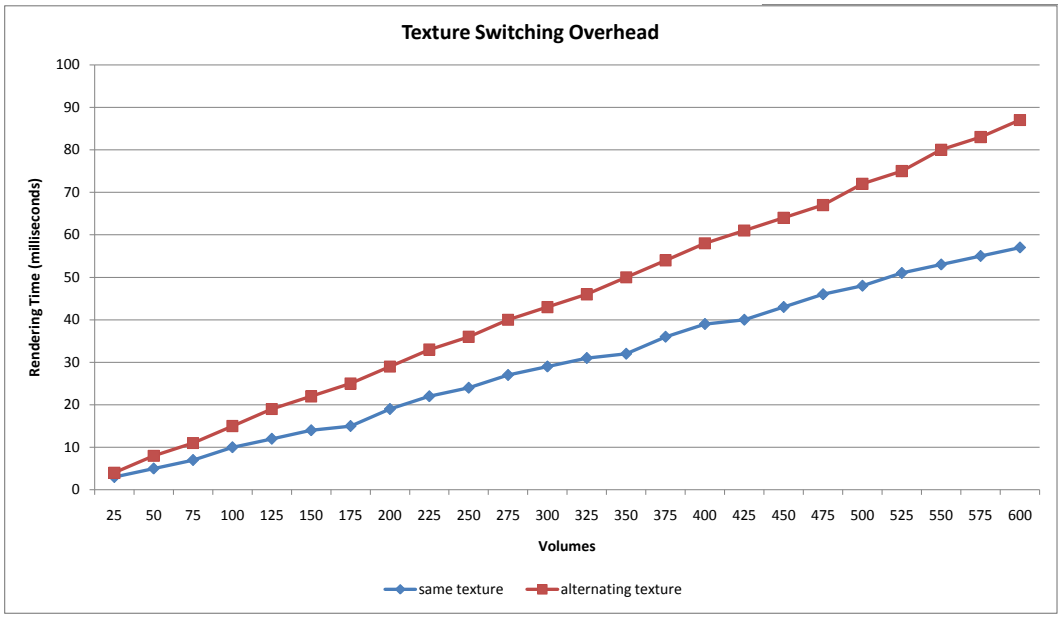

Figure 4.4: Milliseconds per frame to render all volumes using the same texture against different textures. GPU texture state must frequently be changed when volumes use different textures resulting in a 50% drop in performance.

4.6 Volume Memory Requirements

Memory is discussed in two categories: GPU memory usage when rendering volumes and disk space required to store the volume data.

4.6.1 GPU Storage

Volume memory requirements are predictable and based on the size of the 3D texture required to represent the volume rather than the complexity of the data. A 128^3 uncompressed RGBA (8 bits per component) texture would require approximately 9.15 megabytes of memory for the texture and all its mipmaps. Using DXT5 compression the memory requirement is reduced to approximately 2.3 megabytes. A 256^3 compressed volume would require around 18.25 megabytes of memory, reducing the number of possible volume billboards in memory by a factor of eight. Moving up to the next power, a 512^3 would require nearly 150 megabytes when compressed. On current hardware, it would seem reasonable to use many 128^3 or a few 256^3 volumes. A GPU

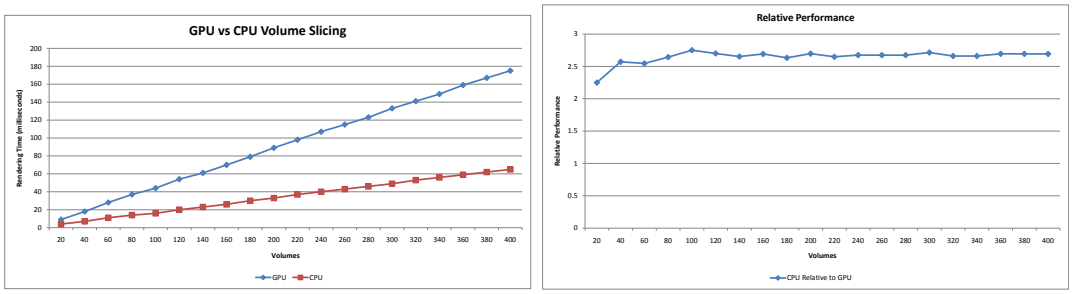

Figure 4.5: Performance difference between the CPU and GPU version of the volume slicing algorithm. The left graph shows rendering time in milliseconds for a number of volumes. The right graph shows the speed gain of the CPU version.

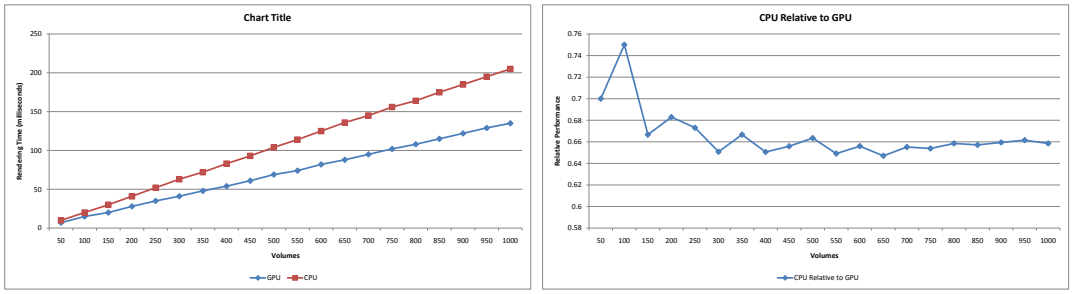

Figure 4.6: Performance difference between the CPU and GPU version of the volume slicing algorithm when no sliced polygons are rasterised. The left graph shows the rendering time in milliseconds for a number of volumes. The right graph shows the speed of the CPU implementation relative the to GPU. This indicates that for performing the prism/plane intersections the CPU implementation is approximately 0.65 times slower.

with 1GB of video memory, as is common on current consumer hardware, could store approximately 445 volumes.

Figure 4.7 compares volume and mesh memory requirements. It is assumed mesh vertices are uncompressed and all textures are compressed at a 4:1 ratio as is common using the S3TC algorithms. A 128^3 volume uses as much memory as 100,000 vertices with position and normal data. Meshes are typically paired with a number of texture maps. In addition to a colour texture, normal maps and specular maps are often used to achieve detailed shading computations. However, the next power of two sized volume, 256^3, requires as much memory as mesh with 100,000 vertices and four 2048^2 texture maps.

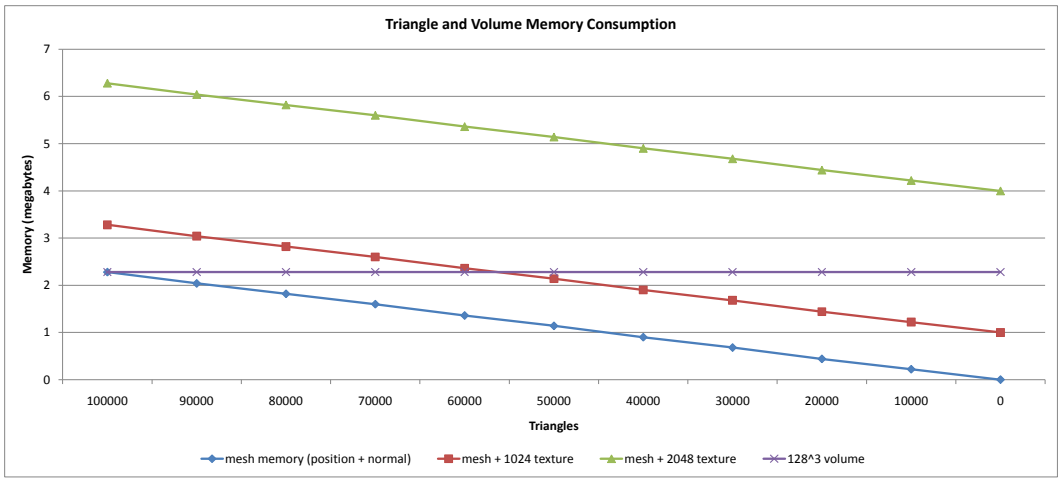

Figure 4.7: GPU memory required for storing volumes and meshes. Memory required for a volume is based on the size of the volume data while the memory for a mesh varies depending on the number of vertices and texture maps it requires.

4.6.2 Disk Storage

The system uses volumes with colour information stored in an uncompressed form. Each colour consists of 4 floating point values between 0 and 1 representing the red, green, blue, and alpha intensities. While such a representation was useful for the test system, in practise, major spacing savings can be made on disk storage. Volume data could be compressed using a GPU compression algorithm discussed in section 4.6.1 and loaded directly into graphics memory. However, this will only achieve a 4:1 compression ratio and cause more data to be read from disk compared to a better compression scheme. It may be quicker to read smaller files from slow disks, or optical media in the case of games consoles, and decompress the data on the CPU.

Better space savings can be obtained by compressing the texture using a standard image compression algorithm. Table 4.1 shows the uncompressed and compressed sizes for a number of test volumes. Volumes were compressed using the lossless PNG compression algorithm. A volume and its mipmaps were split up into 2D slices and each slice saved separately. At run time, the images must be decompressed and assembled back into a 3D texture. It is assumed that a volume only stores colour information in 32bit RGBA format as is commonly used for game art. A high compression ratio is

Volume	Size	Uncompressed	PNG	% Reduction	Model Size (zip)
fir tree	128^3	9362.2KB	158.6KB	1.7%	1466.0KB
tree	128^3	9362.2KB	189.7KB	2.0%	1754.5KB
police car	128^3	9362.2KB	175.5KB	1.9%	398.7KB
van	128^3	9362.2KB	286.0KB	3.0%	4713.1KB
fir tree	256^3	74898.3KB	1045.3KB	1.4%	1466.0KB
tree	256^3	74898.3KB	1086.3KB	1.4%	1754.5KB
police car	256^3	74898.3KB	1106.5KB	1.4%	398.7KB
van	256^3	74898.3KB	1525.9KB	2.0%	4713.1KB

Table 4.1: Disk space required for a set of test volumes. The uncompressed size is compared against PNG compression showing between 97% and 98.6% space savings for our test data. For comparison the size of the source model data is show using zip compression.

achieved reducing storage to 1.5% to 3.0% of the original uncompressed colour data. This can be attributed to the fact that many volume slices consist of large areas of space containing the same colour. Depending on the nature of the data, compressed sizes may vary. The size of the original model data was compressed using DEFLATE (zip). It can be seen that for 128^3 volumes storage requirements are less than the models in all cases. Volumes can be stored using a relatively small amount of space even for large volumes and as such disk space storage is unlikely to be an issue for modern media.

Chapter 5

Conclusions

We have shown that volume billboards can outperform polygonal meshes with performance based on the number of pixels a volume will project to. The number of triangles in the mesh a volume is intended to replace should be taken into account.

Since performance scales inversely to the number of pixels rendered, it is only possible for volumes to outperform meshes when they map to relatively few pixels. Therefore, volumes can be used as an alternative to meshes but only for very distant representations. Volume billboards could be used in addition to other LOD techniques, with volumes used for lowest levels. A level of detail metric used with volumes should take into account the screen resolution and the number of pixels a volume will project to in order to avoid fillrate limitations which causes the framerate to drop at high resolutions. During rendering, volumes did not use less memory than polygonal meshes, with even a small 128^3 volume requiring as much memory as a textured mesh with 50,000 vertices. Disk storage space can be reduced using PNG compression. While compression results vary, storage space required was relatively small and comparable to the original model data.

The volume billboards system implemented highlighted some implementation details missing from the academic literature and considerations to take into account when attempting to integrate volume billboard rendering into a polygonal pipeline. Care must be taken when choosing the slicing rate for multiple volumes to ensure undersampling doesn't occur and oversampled volumes don't become overly opaque. The slicing algorithm presented in previous works can be implemented in a current shading

language with only some minor modifications.

5.1 Future Work

There are still areas which need to be researched to help determine how to use volume billboards in interactive applications. While it is practical to use volumes in certain situations perceptual tests should be performed to help uncover ideal values for various metrics. We have seen when volumes can outperform triangle meshes but the question of quality between each type of impostor was not tested. It would be useful to perform a study of how volumetric impostors compare against traditional polygonal impostors in terms of perceived image quality and to investigate which types of objects benefit most from volumetric representations.

Transitions between volume mipmap levels is done using hardware bilinear interpolation but transitioning polygonal objects to volumes is something that has not been discussed. It would be useful to evaluate the application of current LOD blending techniques to volumes to ensure smooth level of detail.

There was a noticeable performance difference between the CPU and GPU implementations of the slicing algorithm. The exact cause of the difference in performance has not been determined. Modern GPUs provide new facilities to speed up rendering such as building lists of GPU commands from multiple threads[32]. Another possible improvement might be to reduce the required state changes when performing slicing on the GPU. Texture switching could be reduced or eliminated using virtual textures. Such techniques could be investigated to help create a more efficient volume rendering pipeline.

Appendix A

Appendix

A.1 Vertex Program

```
#version 330

in vec3 in_pos;
in vec3 in_tex;

out vec3 tex_coords;

void main()
{
    gl_Position = vec4(in_pos, 1.0);
    tex_coords = in_tex;
}
```

A.2 Fragment Program

```
#version 330

uniform sampler3D tex;

in vec3 in_pos;
in vec3 tex_coord_fp;
layout(location = 0) out vec4 out_color;

void main()
{
    out_color = texture(tex, tex_coord_fp);
}
```

A.3 Geometry Program

```
#version 330

layout(triangles_adjacency) in;
layout(triangle_strip, max_vertices=5) out;

uniform mat4 modelview;
uniform mat4 projection;
uniform float z_slice;

in vec3 tex_coords[];

out vec3 tex_coord_fp;

vec4 pts[6];
vec4 outv[5];
vec3 outt[5];
int indices[5] = int[](1, 2, 0, 3, 4);
int num = 0;


void intersect(int k0, int k1)
{
    float t = (z_slice - pts[k0].z) / (pts[k1].z - pts[k0].z);
    outv[num] = projection * mix(pts[k0], pts[k1], t);
    outt[num] = mix(tex_coords[k0], tex_coords[k1], t);
    num++;
}

void main()
{
    /*
     * Transform points to camera space.
     */
    for(int i = 0; i < 6; i++)
    {
        pts[i] = modelview * gl_in[i].gl_Position;
    }

    float z0 = pts[0].z;
    float z1 = pts[1].z;
    float z2 = pts[2].z;
    float z3 = pts[3].z;
    float z4 = pts[4].z;
    float z5 = pts[5].z;

    float s[3];
    s[0] = (z_slice-z0) / (z3-z0);
    s[1] = (z_slice-z1) / (z4-z1);
```

```
49      s[2] = (z_slice-z2) / (z5-z2);
50
51      float s0 = (z_slice-z0) / (z3-z0);
52      float s1 = (z_slice-z1) / (z4-z1);
53      float s2 = (z_slice-z2) / (z5-z2);
54
55      if((s[0] < 0.0 && s[1] < 0.0 && s[2] < 0.0) || (s[0] > 1.0 && s[1] > 1.0 && s[2] > 1.0))
56          return;
57
58      for(int k = 0; k <= 2; k++)
59      {
60          int kr = (k+1) % 3, kl = (k+2) % 3;
61
62          if(0.0 <= s[k] && s[k] <= 1.0)
63          {
64              intersect(k, k+3);
65          }
66          else if(s[k] < 0.0)
67          {
68              if(s[kl] >= 0.0) intersect(k, kl);
69              if(s[kr] >= 0.0) intersect(k, kr);
70          }
71          else
72          {
73              if(s[kl] <= 1.0) intersect(k+3, kl+3);
74              if(s[kr] <= 1.0) intersect(k+3, kr+3);
75          }
76      }
77
78      for(int i = 0; i < num; i++)
79      {
80          gl_Position = outv[indices[i]];
81          tex_coord_fp = outt[indices[i]];
82          EmitVertex();
83      }
84      EndPrimitive();
85  }
```

Bibliography

[1] P. Decaudin and F. Neyret, "Volumetric billboards," *Computer Graphics Forum*, vol. 28, no. 8, pp. 2079–2089, 2009.

[2] S. Laine and T. Karras, "Efficient sparse voxel octrees – analysis, extensions, and implementation," NVIDIA Technical Report NVR-2010-001, NVIDIA Corporation, Feb. 2010.

[3] C. Crassin, F. Neyret, S. Lefebvre, and E. Eisemann, "Gigavoxels : Ray-guided streaming for efficient and detailed voxel rendering," in *ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (I3D)*, (Boston, MA, Etats-Unis), ACM, ACM Press, feb 2009.

[4] R. A. Drebin, L. Carpenter, and P. Hanrahan, "Volume rendering," in *Proceedings of the 15th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '88, (New York, NY, USA), pp. 65–74, ACM, 1988.

[5] T. J. Cullip and U. Neumann, "Accelerating volume reconstruction with 3d texture hardware," tech. rep., Chapel Hill, NC, USA, 1994.

[6] B. Cabral, N. Cam, and J. Foran, "Accelerated volume rendering and tomographic reconstruction using texture mapping hardware," in *Proceedings of the 1994 symposium on Volume visualization*, VVS '94, (New York, NY, USA), pp. 91–98, ACM, 1994.

[7] M. Hadwiger, J. M. Kniss, C. Rezk-salama, D. Weiskopf, and K. Engel, *Real-time Volume Graphics*. Natick, MA, USA: A. K. Peters, Ltd., 2006.

[8] R. Fernando, *GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics*. Pearson Higher Education, 2004.

[9] M. Pharr and R. Fernando, *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation (Gpu Gems)*. Addison-Wesley Professional, 2005.

[10] V. Vidal, X. Mei, and P. Decaudin, "Simple empty-space removal for interactive volume rendering," *J. Graphics Tools*, vol. 13, no. 2, pp. 21–36, 2008.

[11] P. W. C. Maciel and P. Shirley, "Visual navigation of large environments using textured clusters," in *Proceedings of the 1995 symposium on Interactive 3D graphics*, I3D '95, (New York, NY, USA), pp. 95–ff., ACM, 1995.

[12] S. Gernot, "Image-based object representation by layered impostors," in *Proceedings of the ACM symposium on Virtual reality software and technology*, VRST '98, (New York, NY, USA), pp. 99–104, ACM, 1998.

[13] X. Décoret, F. Sillion, F. Durand, and J. Dorsey, "Billboard clouds," Tech. Rep. RR-4485, INRIA, Grenoble, June 2002.

[14] A. Hardy and J. Venter, "3-view impostors," in *Proceedings of the 7th International Conference on Computer Graphics, Virtual Reality, Visualisation and Interaction in Africa*, AFRIGRAPH '10, (New York, NY, USA), pp. 129–138, ACM, 2010.

[15] M. Schwarz and H.-P. Seidel, "Fast parallel surface and solid voxelization on gpus," *ACM Trans. Graph.*, vol. 29, pp. 179:1–179:10, December 2010.

[16] E. Eisemann and X. Décoret, "Single-pass gpu solid voxelization for real-time applications," in *Proceedings of graphics interface 2008*, GI '08, (Toronto, Ont., Canada, Canada), pp. 73–80, Canadian Information Processing Society, 2008.

[17] L. Zhang, W. Chen, D. S. Ebert, and Q. Peng, "Conservative voxelization," *Vis. Comput.*, vol. 23, pp. 783–792, August 2007.

[18] T. Akenine-Möller, E. Haines, and N. Hoffman, *Real-Time Rendering 3rd Edition*. Natick, MA, USA: A. K. Peters, Ltd., 2008.

[19] D. Luebke, B. Watson, J. D. Cohen, M. Reddy, and A. Varshney, *Level of Detail for 3D Graphics*. New York, NY, USA: Elsevier Science Inc., 2002.

[20] H. Hoppe, "Progressive meshes," in *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, SIGGRAPH '96, (New York, NY, USA), pp. 99–108, ACM, 1996.

[21] P. Cignoni, C. Montani, and R. Scopigno, "A comparison of mesh simplification algorithms," *Computers & Graphics*, vol. 22, pp. 37–54, 1997.

[22] W. J. Schroeder, J. A. Zarge, and W. E. Lorensen, "Decimation of triangle meshes," *SIGGRAPH Comput. Graph.*, vol. 26, pp. 65–70, July 1992.

[23] Y. Wu, Y. He, and H. Cai, "Qem-based mesh simplification with global geometry features preserved," in *Proceedings of the 2nd international conference on Computer graphics and interactive techniques in Australasia and South East Asia*, GRAPHITE '04, (New York, NY, USA), pp. 50–57, ACM, 2004.

[24] M. Duchaineau, M. Wolinsky, D. E. Sigeti, M. C. Miller, C. Aldrich, and M. B. Mineev-Weinstein, "Roaming terrain: real-time optimally adapting meshes," in *Proceedings of the 8th conference on Visualization '97*, VIS '97, (Los Alamitos, CA, USA), pp. 81–88, IEEE Computer Society Press, 1997.

[25] M. Hesse and M. L. Gavrilova, "An efficient algorithm for real-time 3d terrain walkthrough," in *Proceedings of the 2003 international conference on Computational science and its applications: PartIII*, ICCSA'03, (Berlin, Heidelberg), pp. 751–761, Springer-Verlag, 2003.

[26] M. White, "Real-time optimally adapting meshes: terrain visualization in games," *Int. J. Comput. Games Technol.*, vol. 2008, pp. 12:1–12:7, January 2008.

[27] T. Porter and T. Duff, "Compositing digital images," *SIGGRAPH Comput. Graph.*, vol. 18, pp. 253–259, January 1984.

[28] A. R. Smith, "Image compositing fundamentals," tech. rep., 1995.

[29] A. R. Smith, "Alpha and the history of digital compositing," in *Microsoft Technical Memo 7*, 1995.

[30] H. P. A. Lensch, K. Daubert, and H.-P. Seidel, "Interactive semi-transparent volumetric textures," in *VMV*, pp. 505–512, 2002.

[31] M. Mittring and C. GmbH, "Advanced virtual texture topics," in *ACM SIGGRAPH 2008 classes*, SIGGRAPH '08, (New York, NY, USA), pp. 23–51, ACM, 2008.

[32] Microsoft, "Introduction to multithreaded rendering in direct3d 11." `http://msdn.microsoft.com/en-us/library/ff476891(v=VS.85).aspx`, August 2011. Retrieved August 27th 2011.