Abstract

Binaural spatialisation techniques can create extremely convincing audio environments, but their widespread use has been hindered by the need to personalise reproduction to each listener's physical head-shape. Researchers in the Intelligent Systems group within the department have developed a means to explore how changes in the shape of the head affect its acoustic properties, including a method aimed at determining which parts of the head have the greatest effect on the way we hear.

The interpretation of the results of this work would be aided by the display of the data in a visual rather than a mathematical or statistical form. This report documents the development of a set of tools to manipulate and ultimately visualise the data resulting from the group's simulations, to allow conclusions to be more easily drawn about the results and about the simulation methods themselves.

Acknowledgments

First and foremost thanks to my first and second supervisors, Tony Tew and Adar Pelah, for constructive guidance throughout the project. Thanks also to members of the media research group, particularly Jonathan Thorpe and Carl Hetherington for patiently explaining their existing work and supplying me with test data, and Seiichiro Shoji for allowing me to participate in a series of binaural spatial audio listening tests.

Gratitude is also due to the authors of the supporting software used in the project - the many authors of Blender (`http://www.blender.org`), especially Chris Want for his examples demonstrating integration with VTK.

Contents

Chapter 1

Introduction and context

1.1 The human spatial hearing system

Like most mammals, humans have two ears on opposite sides of the head. As well as providing redundancy and sufficient reception of sound from all around us, the position of our ears enables our hearing system to determine the direction from which a sound arrives. There are three main methods, or auditory cues, by which we perceive directionality:

- Interaural Intensity Difference, or IID - if a sound source is positioned to one side of our head, the ear on that side will pick up more of the sound than will the ear on the opposite side. This difference in intensity between ears is more apparent for higher frequencies, since below a certain frequency sound is not appreciably absorbed or deflected by the head. For a normally-sized human, IID is most important above about 637 Hz[4, p 102].

- Interaural Time Difference, or ITD - if a sound arrives from one side, it will reach one ear before the other. The brain appears to use the phase difference between ears to help determine direction, which means that above a certain frequency this cue becomes ambiguous since the phase shift exceeds 180 degrees, or one full cycle. Thus IID is most effective below about 742 Hz [4, p 101].

- Pinnae effects - the fleshy exterior part of our ears (the pinnae) is ridged in complex ways, and sound reflecting from it into the ear will be changed depending

on which direction it came from. The effect of this filtered and slightly delayed reflection, when combined with the original sound by the brain, can be used to determine directionality. The effects are thought to be most important at higher frequencies. Unlike the other two cues, the effect of the pinnae can give us information about the vertical height of sound sources as well as the horizontal angle. Since head and pinnae shape differ between individuals, the exact effect on the sound received by the eardrum will be different for different people. The brain adapts to these differences, but a change in the physical shape of the head can upset this cue until the hearing system can learn how to interpret the changed effect.

The effects of all these cues can be combined into a single measurement of how the head and ears affect incoming sound of any frequency and from any direction, the Head-Related Transfer Function (HRTF). The HRTF defines an amplitude and phase response for each direction from which sound can arrive. The same information can be expressed in a time-domain form as the Head-Related Impulse Response (HRIR).

1.2 Reproduction of spatial audio

In the earliest days of audio recording technology, cost prohibited the use of more than one microphone or speaker, thus limiting the reproduction of an auditory scene to a single point source. Later experiments in the 1930s included the use of up to eighty microphones and speakers to reproduce the experience of listening to a live orchestra, with sound coming from a variety of directions. It was at this time that the term "stereophonic" was introduced, meaning literally "solid sound". The cost of early home reproduction systems meant that "stereo" became analogous with a two-speaker system, which systems remained the norm until the recent commercialisation of "surround sound" six-speaker systems driven by the home cinema market.

These systems can provide some of the auditory cues described above but they generally do not provide a fully immersive experience and are expensive compared to a standard two-channel system, requiring extra speakers and amplification, and a more complex storage medium. Another reproduction system with potential to overcome these disadvantages is binaural stereo. Instead of attempting to reproduce the envi-

ronment of the recording in the listening room by approximating spatially positioned sound sources with multiple distinct speakers, binaural stereo merely reproduces the signals which would arrive at the eardrums of someone actually in the recorded environment. Thus it requires only two channels of information, only two microphones to record, and only two speakers to replay. So far the use of true binaural stereo has been limited by the following factors:

- It must be reproduced using headphones rather than speakers, since each channel must be presented to only one ear. Speakers inevitably introduce some crosstalk as sound travels around the head and is reflected from surrounding surfaces.
- It requires recordings to be made with two microphones positioned inside the ear canals of a head, either human or artificial, which limits the quality of the microphones. This head must also be positioned in a suitable listening position in the audio environment to be recorded, since the reproduction will site the listener in that same place.
- The effect works best when the head used for recording is physically of a similar shape to that of the listener, since as stated above our hearing system is adapted to the shape of our own individual head and pinnae.

A better knowledge of the listener's HRTF could be used to mitigate these disadvantages:

- A recording made with more conventional multi-track microphone technique could be spatialised in post-production by applying the HRTF to each instrument, creating the illusion that each is coming from a different direction. This could include the creation of moving virtual sources, or sound coming from positions that were not physically possible during the recording.
- In the future it may be possible to replay binaural recordings through loudspeakers by eliminating crosstalk between ears, using an inverse HRTF to remove the effect of sounds reflecting around the head.

1.3 HRTF measurement

Currently an individual's HRTF can only be determined accurately by direct acoustic measurement. This involves placing small microphones inside the ear canal of the subject, and using a large array of speakers surrounding the listener to record the effects of the head and pinnae on test signals coming from various directions. The resulting recordings are then processed to determine a transfer function for each direction, and combined into a full HRTF. This procedure is expensive, requiring a great number of loudspeakers ideally situated in an anechoic chamber, and arduous for the subject, who must sit in the same position for a lengthy period without moving, having had cement poured down their ears to block the ear canals and hold microphones in place.

A more economically feasible method of acquiring an HRTF would be one that relies on measuring only the physical shape of the head and ears, and infers the acoustic transfer function. Ideally such a method could be automated, and could even require no more than photographs of the head to extract measurements of important features of the shape. Research into the development of such a method is a focus of the Intelligent Systems group at York, and some of their progress is described in the next chapter.

1.4 Commercial relevance

A cheap and reliable method of determining an individual's HRTF would allow much more widespread use of binaural stereo reproduction. This has an obvious market in the entertainment industry, but the technology could also be applied to situations where headphones are already in use such as air traffic control where different sources of auditory information could be better separated by spacialisation. It could also add a new dimension to the emerging field of sonification for scientific data interpretation.

Chapter 2

Existing work

2.1 Sensitivity analysis of the head and pinnae at York

This chapter describes the processes developed by members of a research group in the Department which will create the data that this project is designed to visualise. The research is centered around a method to describe the shape of the human head in a parameterised form which is amenable to both alteration and computational acoustic simulation. The primary goal is to discover which parts of the head and pinnae have significant effect on the HRTF, by repeatedly simulating the effect of many small modifications to the shape.

2.2 Head geometry capture and processing

The first stage of the process is the acquisition of a single base model of the head and ears. An artificial mannequin known as KEMAR, an established standard in psychoacoustic research, is used. The main part of the head is scanned using a FastScan laser scanner, which produces data in the form of a point cloud which is triangulated to form a mesh. Since lasers cannot accurately scan non-convex objects the ear geometry is acquired separately, using Computerised Tomography (CT) scanning producing volume data, which is also triangulated. The two meshes are combined, and a hemisphere added to seal the hole formed where the neck would normally attached to the head.

2.3 Parameterisation using the EFT

The scheme developed to parameterise the spatial model uses the Elliptic Fourier Transform (EFT), a technique allowing conversion of head and pinnae meshes to a frequency domain representation in a similar manner to the way the Fourier Transform can be used to represent one-dimensional functions. The method is detailed in full in [3], but as an overview it consists of the following steps:

- The source mesh is sliced by a series of planes, rotated about the ear-to-ear axis as shown in figure 2.1. The intersection of the mesh with these planes generates a series of closed contours in two dimensions, as shown in figure 2.2.
- A Fourier transform of each individual contour is then taken, by expressing each as a two-dimensional parametric function of a parameter which sweeps along the contour, as shown in figure 2.3.
- The resulting frequency domain spectra for each slice then form the input to a second Fourier transform, which outputs a frequency domain representation of the changes between successive slices, moving around the head. These Fourier coefficients form a set of parameters describing the original mesh.

2.4 Perturbation and reconstruction

The set of parameters resulting from the EFT can then be used to create slightly deformed versions of the template mesh. One or more of the final Fourier coefficients are modified, and the EFT process is reversed to generate a new set of 2D contours forming a sliced version of a new head shape. A polygonal mesh is then reconstructed from this slice set, a process which is described in [3, p 121]

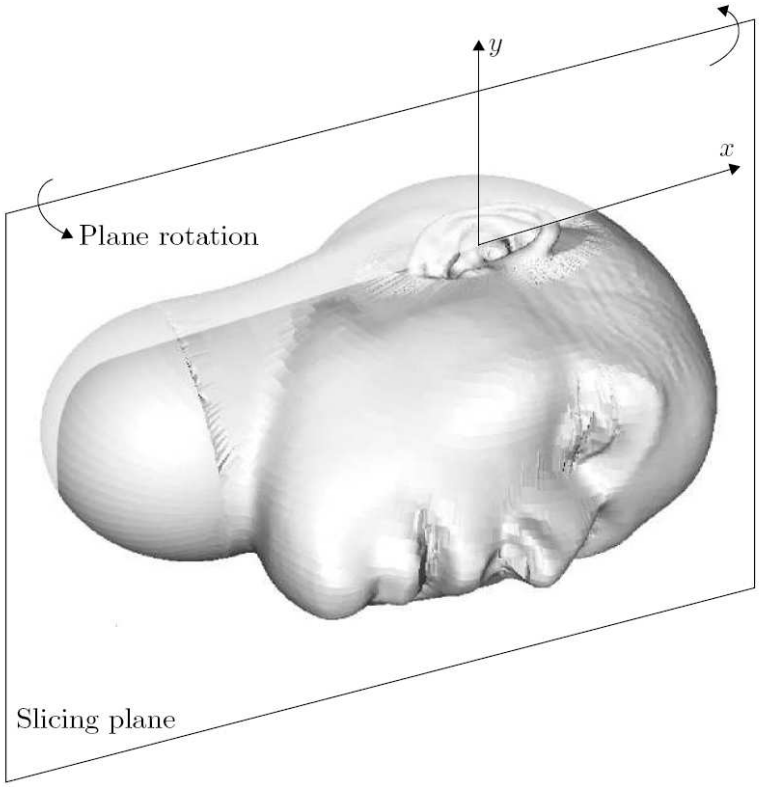

Figure 2.1: Slicing of a head mesh for EFT parameterisation, taken from [3]

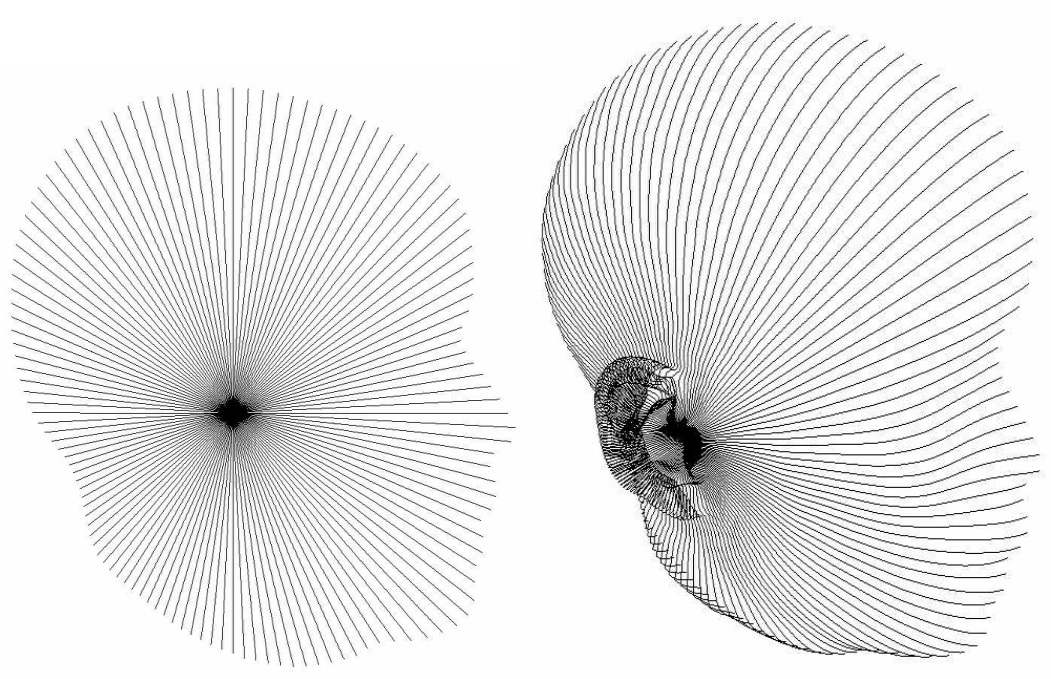

Figure 2.2: Contours generated by slicing, shown from the side of the head on the left, and from an angle on the right (showing a portion of the head only, for clarity)

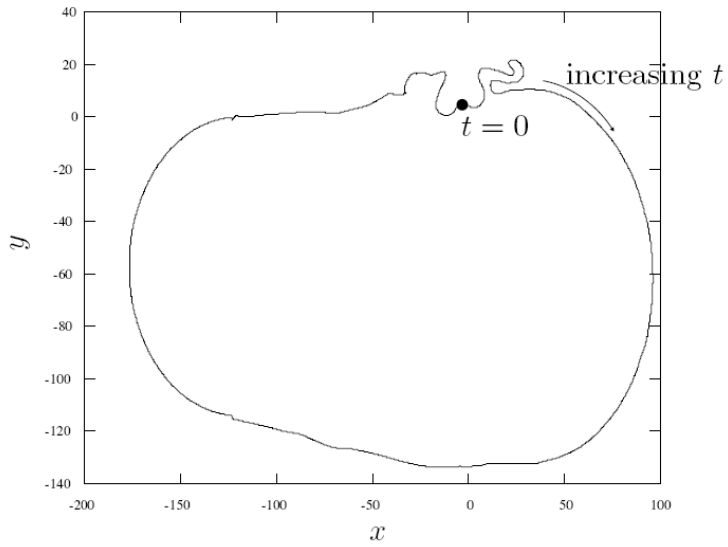

Figure 2.3: A contour forming a single slice of a head mesh (with one ear only), taken from [3]

2.5 Simulation

The effects of these small changes to the template mesh are measured by performing acoustic simulation using a commercial package, PAFEC, which uses the Boundary Element Method (BEM) to compute the impulse response of a system comprising a sound source at the eardrum, the head and pinnae mesh, and a series of receptors around the head. Thus an HRIR and HRTF can be found for the new, changed, head shape.

This simulation is extremely computationally expensive, so performing simulations for every possible combination of changes to the EFT parameters is not feasible. A technique known as Differential Pressure Synthesis (DPS) is used to enable much faster computation of the results of multiple changes. Using the principal of superposition and the fact that the basis functions of the EFT are orthogonal, DPS allows the effect of small changes in many individual parameters to be combined to calculate the result of changes to all of the parameters simultaneously. Once the effect of a change to each individual parameter has been calculated using BEM, a DPS database can be constructed allowing the effect of any arbitrary change to be calculated, merely by querying this database.

2.6 Results

To extract data about how sensitive each part of the head is, the DPS database could be queried for the effects of changes to spatially distinct regions of the head. However, working out which EFT parameter changes are necessary to perform an arbitrary head-shape-space change is not easy. A simpler solution which provides a result for the whole head at once is to explore the entire parameter space, inspecting the results of each combination of parameter changes, and for parameter changes which cause a large change in HRTF, reconstructing the slices for that parameter set. Accumulating all such reconstructed slice sets into a single error function, combinations of changes to head-shape which caused a large effect will tend to add, while those which did not will tend to cancel out. Thus the accumulated result will be a set of slices with large changes where changes to the head caused significant changes to the HRTF.

This single error-function slice set, along with the template slice set which produced

it, are the data on which the rest of this project operates.

2.7 Specification

The results of the sensitivity analysis outlined above are not necessarily easy to understand by inspecting the raw slice contours. Displaying the data as an entire head showing both the original geometry and the error information would allow easier interpretation. The aim of this project is to create the tools required to display this data. A number of requirements can be given:

- The solution should display the data in an intuitive form which is easy to understand visually
- It should be possible to display the entire data set at one time rather than only one spatial portion or only a subset of the total information, to allow interpretation of the results without unnecessary manipulation of the display
- The tools should be easy to use for researchers working on the project, who are familiar with the data being displayed but not necessarily the graphical visualization tools and processes used to display it
- The display should allow easy and fast navigation to different parts of the data, and should allow view of all parts of the head including parts which might be obscured such as the inside folds of the pinnae
- The implementation should be developed such that it can be used on standard computer equipment already available in the department
- The solution should be able to be modified and extended if nessecary by future developers to suit their changing needs.

Chapter 3

Slice set editing

3.1 Requirements

While the format of the data that will be taken as the input to the visualisation process is well defined, the experimental data itself was not available during the course of this project since the research was still ongoing. Thus it was important that test data sets could be generated to test the solution.

Since the error functions and template slice set data share a simple file format (see Appendix B), it would have been possible to generate simple test data by writing the files by hand or using simple mathematical functions to write files. However since slice files were available for the template heads, it was decided that a tool to modify these existing slice sets would be more useful, since it would be closer to the final usage conditions.

I chose not to attempt to create a geometry editing tool completely from scratch, since I was aware from background research into visualisation that other groups had used the open source 3D package Blender (`http://www.blender.org`) as part of visualisation pipelines to inspect 3D geometry. In the case of [8], integration with the Visualization Toolkit (VTK, `http://www.vtk.org`) is demonstrated, using Python scripts within Blender to create a native Blender mesh object from an external source. This allows editing of the mesh with Blender's extensive toolset, and export of the mesh back to another format. The bulk of this part of the project involved writing import and export scripts which run inside Blender, to handle conversion to and from the slice

set file format.

3.2 Implementation

Blender is scriptable using the language Python (http://www.python.org), and includes its own Python interpreter. It exposes many of its features to Python scripts via an API consisting of several custom objects representing Blender's internal data structures. Most useful for the import and export scripts were:

- The ability to create new mesh objects and link them into the current 3D scene
- The methods of these mesh objects allowing addition of new vertices
- The ability to add a list of pairs of vertices representing edges to a mesh object
- Access to mesh vertex groups, allowing lists of vertices to be associated with each other
- Various utility functions for constructing and transforming vectors in 3D space
- Functions allowing use of standard Blender interface elements such as file open and save dialogs and progress bars

In addition to the custom Blender objects, standard Python objects were used for file input and output, regular expression parsing and floating-point string formatting.

The algorithm used for the import script is as approximately as follows:

```
Prompt the user for an input file path
For each line in the input file:
    Read the coordinates of the vertex
    Rotate the coordinates into the plane of the slice
    Add the vertex to a mesh object
For each slice:
    For each vertex in the slice bar the last:
        Add an edge between this slice and the next
```

```
For the last vertex in the slice:
    Add an edge to the first vertex in the slice
```

The slice membership of each vertex is calculated simply by knowledge of the number of points in each slice and the number of slices in the set, which are stated in the header of the input file. Each vertex must be rotated to match the orientation of the slice it belongs to, since in the slice set file the coordinates of the vertices are given in the XY plane only. The rotation is about the Y axis, which runs between the ears, and is the same as the rotation of the plane used to slice the original head mesh (see figure 2.1).

The vertices belonging to each slice are also added into a vertex group named simply "slice n" where n is the index of the slice, which allows easy selection of individual slices from the Blender interface. Figure 3.1 shows a 3D view of an imported slice set on the left, with slice 16 of 64 selected, and a 2D view of that slice on the right.

The imported data can be edited with all the normal 3D modeling tools. An example is shown in Appendix A.1.3.

The script to export back to a slice set file follows the same procedure in reverse: each vertex is rotated back into the XY plane before being written to the file. The operation of the two scripts can be tested by importing a slice file and exporting it straight back out to another file, then visually comparing the contents of the two files. The coordinates sometimes differed but never more than in the sixth or seventh significant figure, possibly due to small inaccuracies in the rotation to and from the vertex's true 3D position.

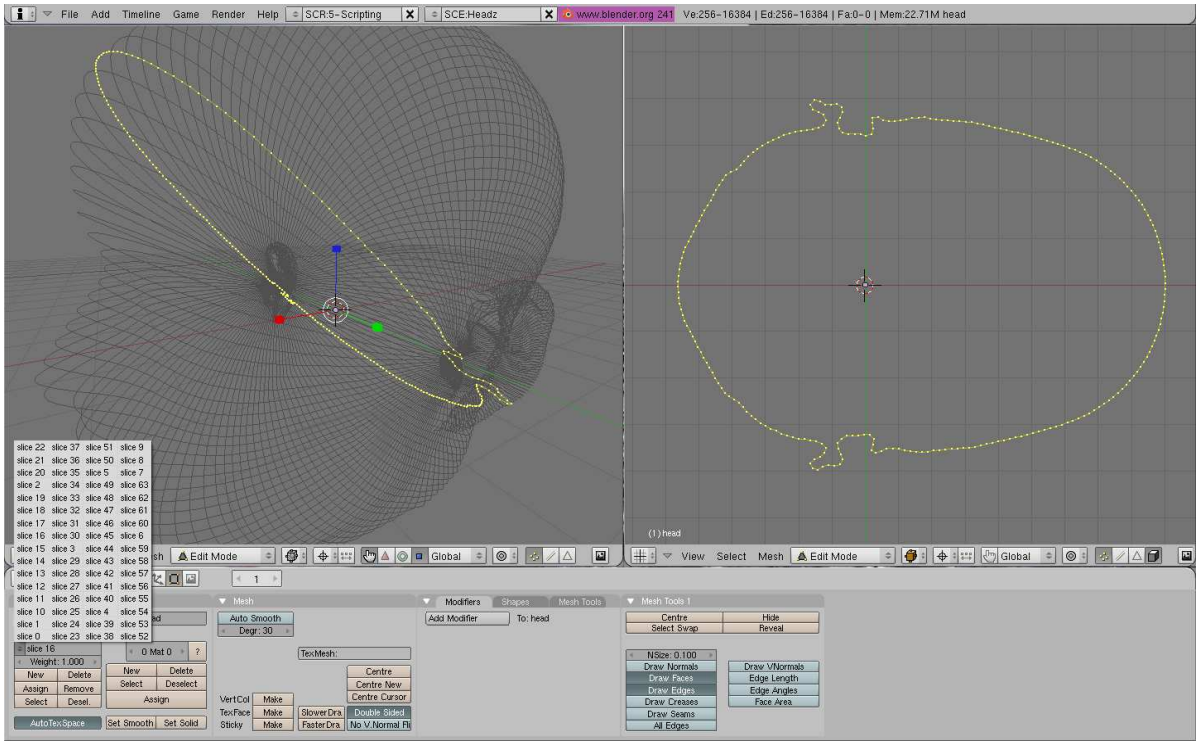

Figure 3.1: An imported slice set in Blender

Chapter 4

Decimation

4.1 Design

During development of the slice set editing, it was noted that interactivity in Blender became very slow when certain operations were performed, particularly when using Proportional Vertex Editing, which involves moving a large number of vertices at a time. The example slice sets provided by the research group contained 64 slices with 2048 points per slice for a total of 131,072 points. While this amount of detail is necessary for exact simulation results, for this project the reduced interactivity hindered the rapid creation of test data sets, and the precision was not deemed necessary during the development phase. A decimation tool was produced which reduces the number of points in each slice.

4.2 Implementation

The algorithm used is very simple: simply read points from one slice file and write every n^{th} point to an output slice file. The information about the number of points per slice in the output file must also be updated correctly.

4.3 Limitations

Simply discarding the majority of the information contained in the input slice file will have consequences for the accuracy of the shape contained in the output file. This form of decimation is somewhat analogous to image or audio resampling using a filter kernel consisting of a unit impulse, which produces poor-quality results. Normally a more sophisticated filter kernel such as a box (averaging) or Gaussian would be used, and the same holds true decimation of slice contours. Although as stated above the crude algorithm used is sufficient for the purposes of this project, a more complex algorithm could preserve more information. The problem has been extensively studied for the three-dimensional case of polygonal mesh simplification in the graphics literature, for example an overview is given in [7].

Chapter 5

Slice arithmetic

5.1 Design

The most intuitive way to create error-function slice sets for use as test data is to import a template set into Blender, deform it and define a new slice set as the difference between the template and deformed sets. The tool to perform this is described in this chapter.

5.2 Implementation

All that is required is the ability to perform simple arithmetic on the coordinates of the points in slice sets. The calculations can be performed vector-wise, that is, separately on the X and Y components of each coordinate. This is implemented in a simple loop that reads input lines from two input files, performs the operation and writes the resulting line to an output file.

Both addition and subtraction are available. Subtraction is most useful for the case above, creating an error slice set from a template and a deformed set. Addition can be used to check the result by performing the reverse operation, and to see the effect of error function slice sets on other template shapes.

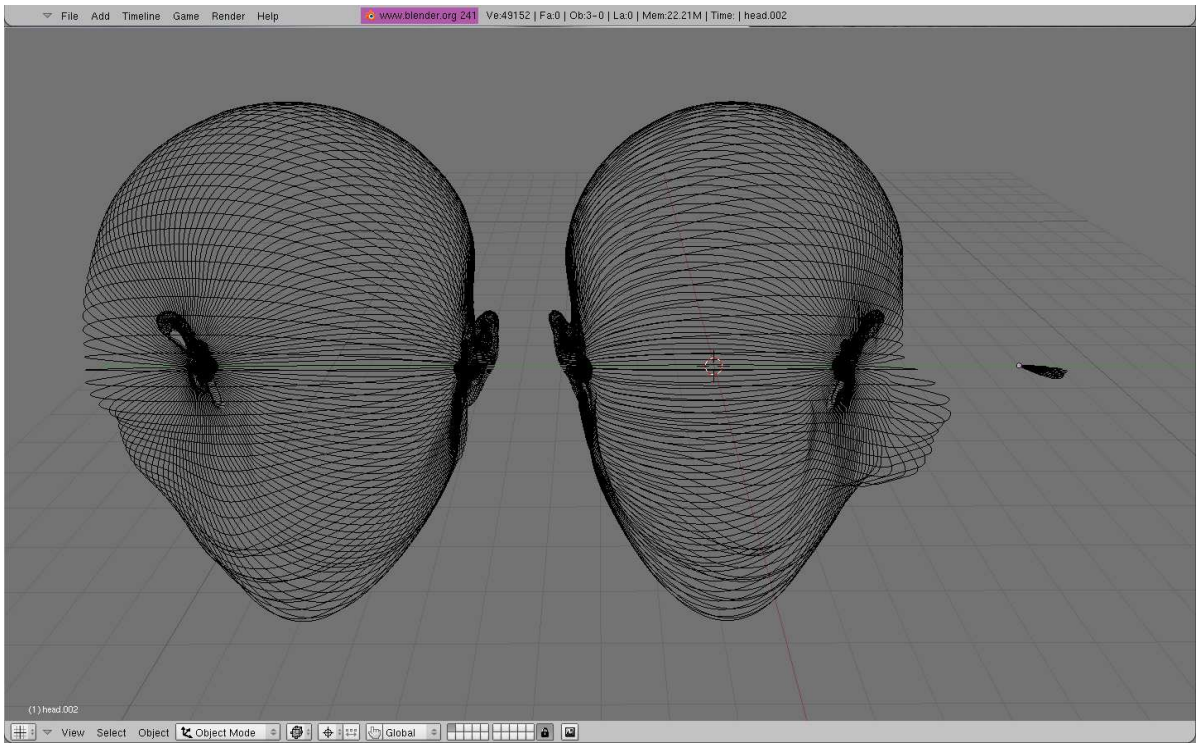

Figure 5.1: Slice arithmetic example, showing from left a template slice set, a deformed version with a greatly enlarged cheekbone, and the difference between the two

5.3 Testing

The input and output files can be compared in a text editor to ensure that the arithmetic is taking place correctly. Visually, the process can be tested by deforming a template slice set in Blender, exporting the result, performing the subtraction and importing the result. The error slice set thus created can also be added onto other template shapes. For example, figure 5.1 shows a template, a deformed version, and the resulting error function. The appearance of the error function slice set is correct: the majority of the points lie at the origin, indicating no difference. Figure 5.2 demonstrates slice set addition.

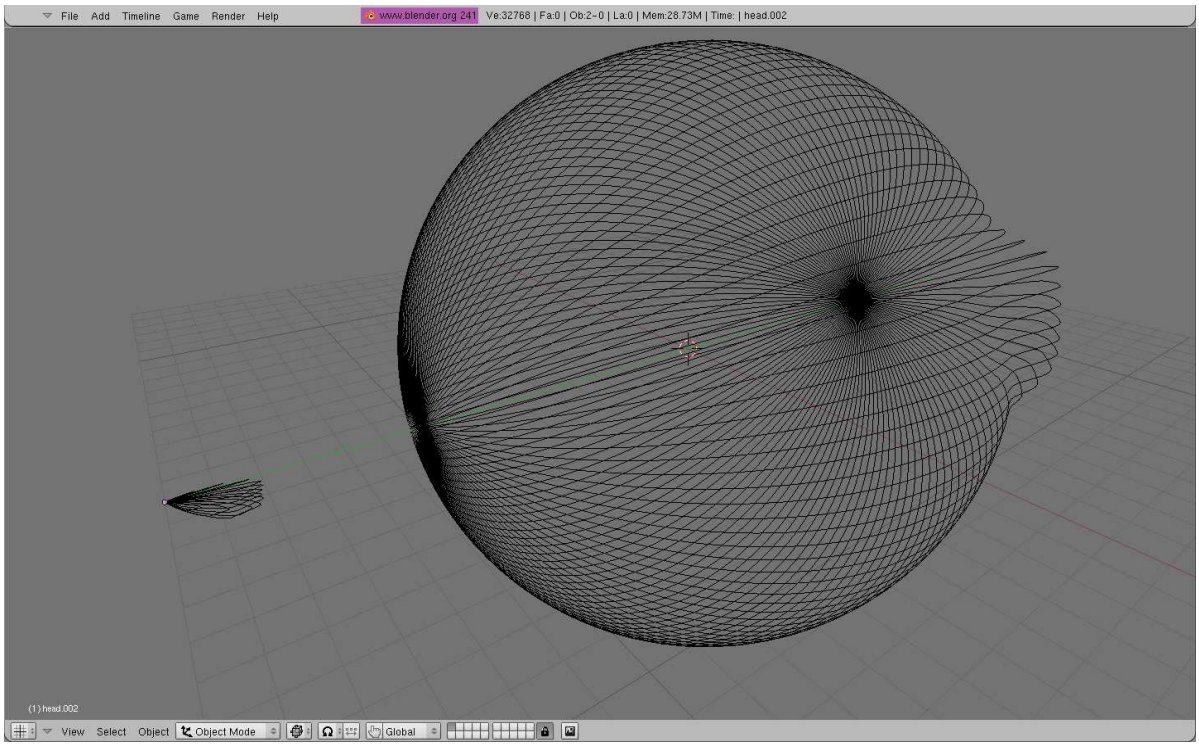

Figure 5.2: Example of slice addition: cheekbone error set from figure 5.1 added onto to a sphere

Chapter 6

Scalarisation

6.1 The problem

Intuitively, the sensitivity of each point on the head should be represented by a scalar value. Either a point has little or no effect on the HRTF, or it has a larger effect describable as a single quantity. However, the data represented in the error function slice set which is the output of the sensitivity analysis method described in 2.6 consists of a two-dimensional vector for each point on the head. In order to visualise this data, a scalar value is required. The obvious way to resolve this is to use the magnitude of the vector, however this does not necessarily produce correct results as figure 6.1 shows. In the upper example, the deformation is mainly radially outwards, and the lengths of the vectors joining corresponding points give a good measure of how much the shape has changed at each point. However in the lower example, the vectors are of similar lengths, but the shape change is clearly much less. In general this problem will occur whenever points move along the contour rather than "inwards" and "outwards". The relationship between EFT parameters and spatial changes is quite complex, and it is entirely possible that deformations like this will happen.

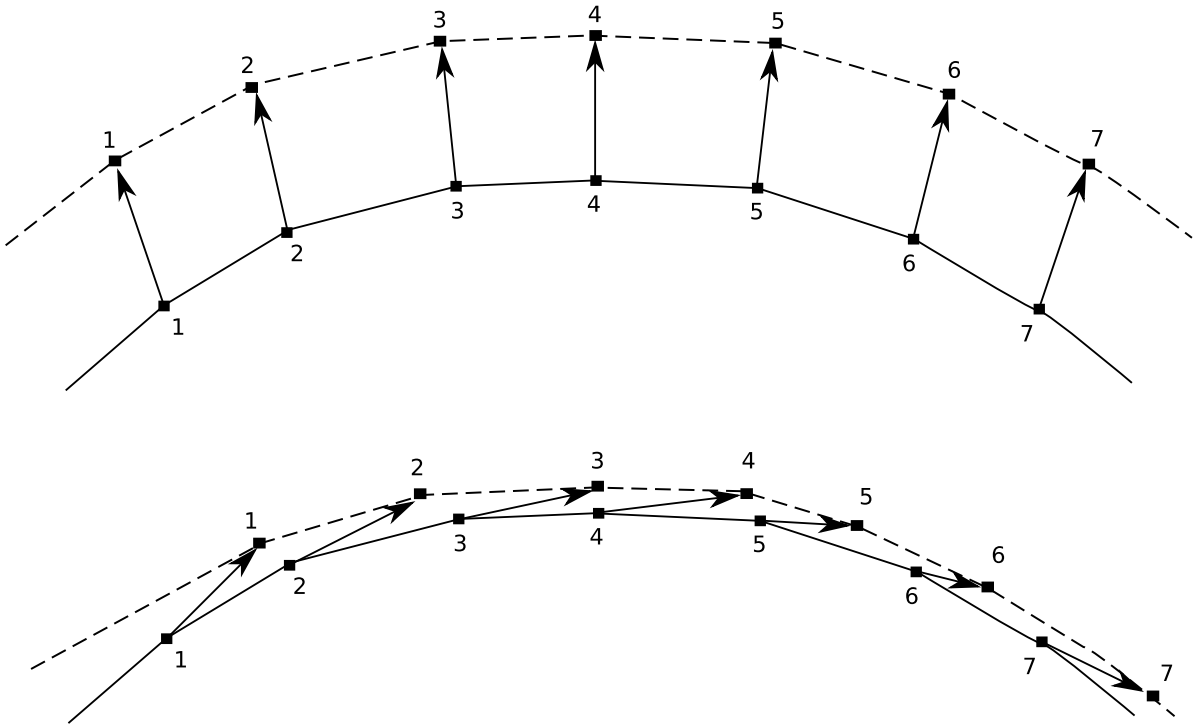

Figure 6.1: Top, a case in which the magnitude of each vector is a good measure of shape difference. Bottom, a case in which it is not.

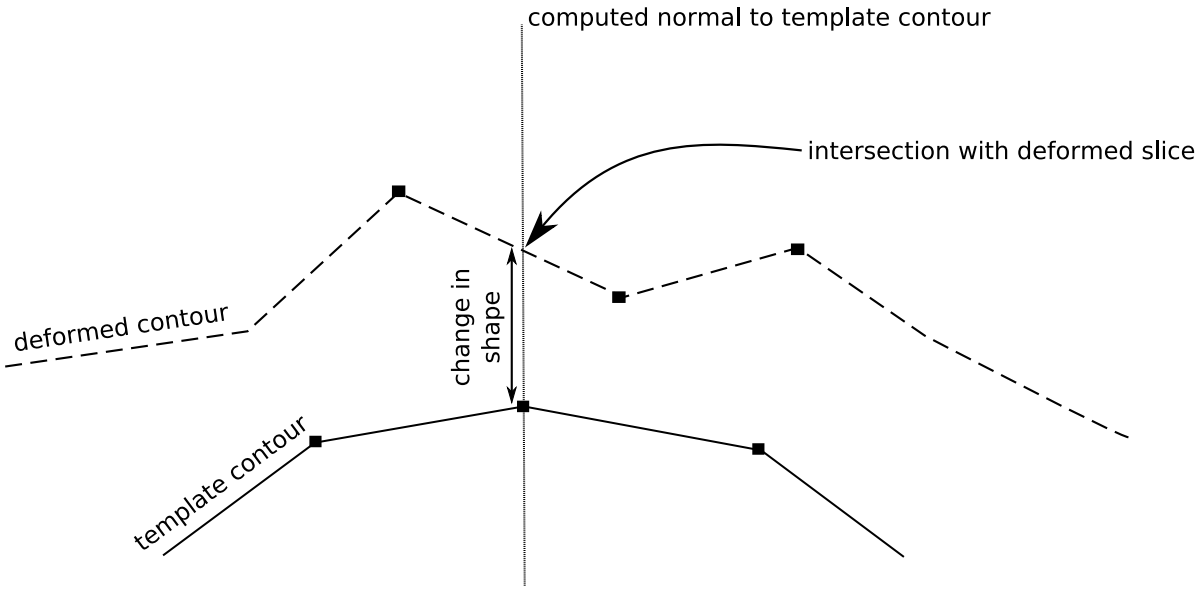

Figure 6.2: Measuring change in shape along a vertex normal

6.2 Design

A solution to this problem might be to use vectors from each template point to the nearest deformed point instead of using vectors between corresponding points (as supplied in an error function slice file). For most of the points in the lower example of figure 6.1 this would give a better result but it can be seen from the case of point 6 in that diagram that where there are no deformed points near the template point this does not work well - measuring from template point 6 to either deformed points 5 or 6 would give an inflated measure of how much the shape has actually been deformed.

A better solution is to interpolate new points on the deformed contour. The method chosen uses a simple linear interpolation, which effectively joins pairs of points on the contour with straight lines. To calculate the distance from a point on the template shape to this line, a normal is constructed perpendicular to the template shape at the point in question, and the intersection of this normal with the straight line elements of the deformed shape is found (figure 6.2). The distance to this point of intersection is a better measure of the magnitude of the shape change at that point.

6.2.1 Vertex normal computation

The above scalarisation method requires a normal to the template contour to be generated at each vertex, even though when using straight-line interpolation each vertex is a discontinuity and the derivative of the contour is not defined at that point. Many other algorithms in graphics require vertex normals, and there are various methods for determining them. A survey of methods used for vertices of three-dimensional polygonal meshes [5] concludes that a method weighting the normals of adjacent faces by the angles they form at the vertex in question performs the best for a variety of mesh types. The conversion of such a technique to two dimensions in not obvious, but a simpler algorithm, Mean Weighted Average (MWA), can be used which simply averages the normals of the two adjacent lines. This is also the fastest algorithm in the survey. In two dimensions this algorithm is equivalent to finding the bisector of the angle formed by the two lines. It is used to find the normal for each vertex in the error scalarisation tool.

6.2.2 Ambiguities in the error function

The accumulation of deformed slice sets which takes place as the last stage of the sensitivity analysis described in 2.6 has potential to introduce ambiguities into the error function slice set. The absolute magnitude of each vector in this slice set has little meaning: it is the relative differences between parts of the head that are important. If the absolute magnitude of some of the vectors becomes too large, the shape might become distorted as shown in figure 6.3.

This distortion can creates ambiguities when intersecting the normal of the template slice with the deformed version: the normal would intersect in multiple places, and the closest intersection would not necessarily be the correct one. This possible problem is resolved by scaling down the error function before the scalarisation process. The calculation of the scale factor is as follows:

```
For a particular template point:

    Find the closest deformed point
    If the closest deformed point is not
    the correspondingly numbered point:
```

```
Scale all error vectors down by a factor of A
```

This process is repeated until all ambiguities are resolved, possibly scaling all error values down several times. The scale factor A is given by:

$$\frac{distance\,to\,nearest\,deformed\,point}{distance\,to\,correspondingly\,numbered\,point} \times 0.9$$

The factor 0.9 is present to ensure that the values are scaled down far enough that the correspondingly numbered point becomes the closest point on the deformed contour, rather than just equally as close as the former nearest deformed point.

As well as removing the ambiguity shown in figure 6.3, this conform step allows the intersection algorithm to be optimised somewhat, since the normal to a point will now always intersect either the previous or the next edge in the deformed slice. Without the conform step intersection would have to be performed against more than just these two lines.

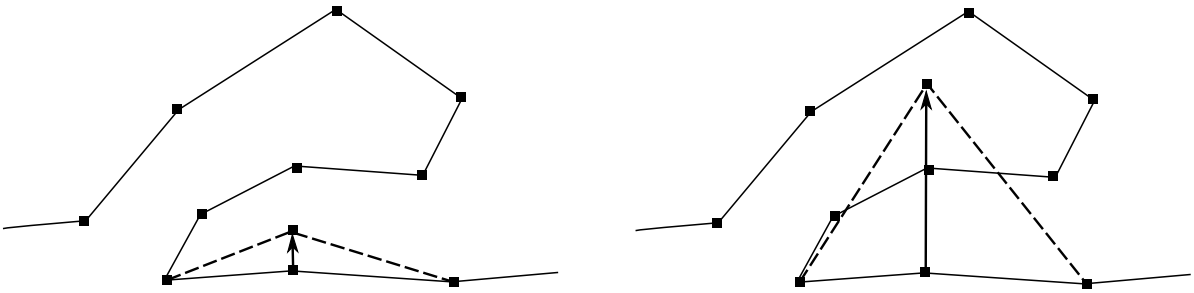

Figure 6.3: Left, a small deformation of one point. Right, the same deformation scaled up, showing how it can corrupt the slice contour.

6.3 Output

The output of the error scalarisation is a scalar value associated with each point on the template mesh, which gives a measure of how much the deformed slice set has deviated from the template at that point.

Chapter 7

Visualisation

7.1 Displaying the information

There are four dimensions of information to be displayed: the three spatial dimensions containing the geometry of the head model, and one dimension of sensitivity information. It is logical to display the three dimensional head model with the sensitivity overlaid on it somehow. There are several methods of doing this, including:

- A "hedgehog" plot with a series of lines rendered pointing outward from the surface, with the length of the line modulated by the sensitivity value at that point
- Changing the geometry in some way influenced by the sensitivity values, for example rendering the template plus error function as a deformed head
- Changing the surface properties of the head in some way driven by the sensitivity value, for example making less sensitive areas more transparent, or changing colour, shininess or texture.

The method chosen was modulation of surface colour, and more specifically hue. Changing the geometry of the head removes the ability to look at the data for specific features, and the other methods could also make the geometry harder to perceive. The advantage of using hue is that the head can be shaded as if illuminated by a light source, which greatly enhances the ability to distinguish relief without needing to examine an

object from different viewpoints. Hue is not changed by shading, which affects only the value, or lightness, of the surface[1].

The colour encoding chosen was a range from blue for zero sensitivity through green, yellow and orange to red for the highest sensitivity. This loosely tallies with our ideas of "cold" and "hot" colours.

7.2 Interaction

It is important that the user be able examine every part of the head model, including those areas in the pinnae which are generally occluded from direct view. Thus the rotation system used must be intuitive and able to orient the model in any desired way. A survey of mouse-based rotation techniques, [1], indicates that quantitatively a technique known as the Two-axis Valuator is the most efficient; however the testing method on which this is based uses an object which has no particular "up" direction. For navigation around an object such as the head, which the user will be familiar with and expect to be in an upright position, it seems acceptable to use a slightly simpler method, the Two-axis Valuator with Constant Up-Vector. This is also the method used by Blender. The method is fairly simple: mouse motions in a horizontal direction rotate the view about the global "up" direction, in this case the neck-to-crown axis of the head. Mouse movements in a vertical direction rotate the view about the current "left" vector, which is intuitive as the motion is like tilting a physical object toward and away from the eye.

The implementation also allows zooming using the keyboard, at the same time as rotation with the mouse. This allows faster orientation than using a secondary mouse button to zoom, which requires interrupting the rotation to switch buttons.

[1]Strictly, the human vision system does not perceive all hues as a constant luminance, so surface colour is not totally independent of changes due to lighting. The simple colourmap used here could likely be improved by techniques such as those used in [2].

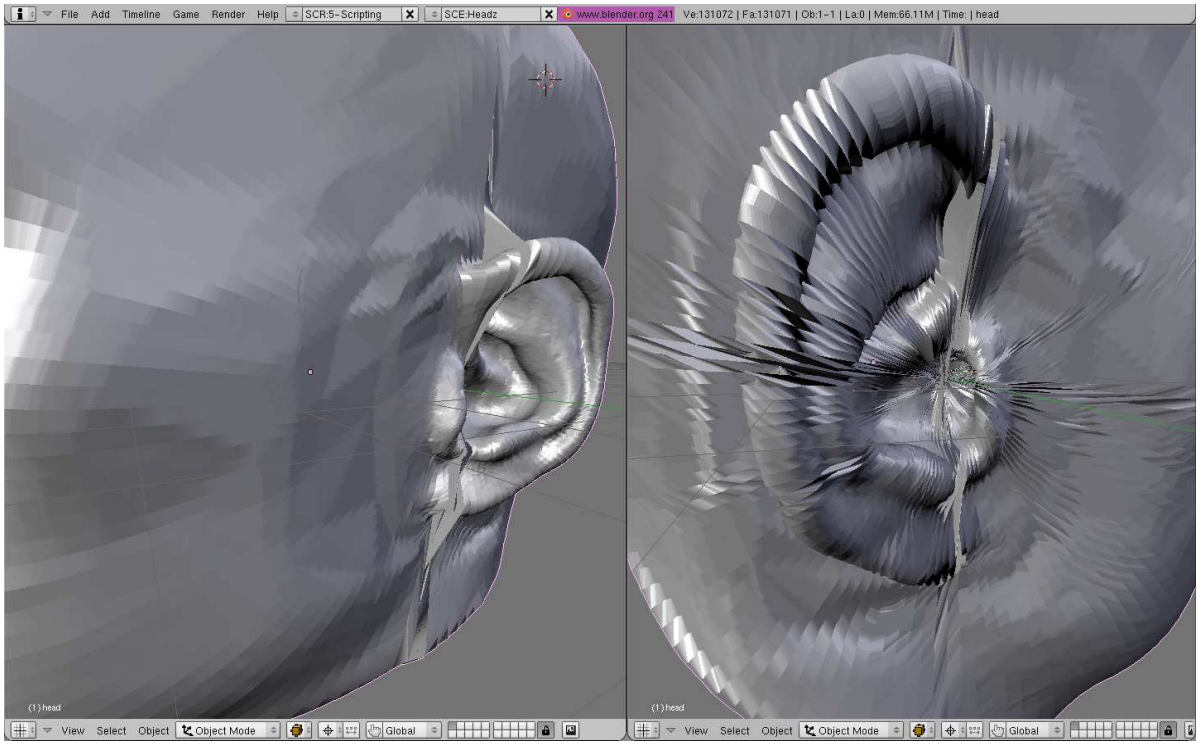

Figure 7.1: Crude meshing showing relatively good areas on left and artifacts on right.

7.3 Meshing slice sets

The vast majority of three dimensional rendering is performed with triangles as primitives [6, p. 1]. During the development of the slice editing tools described in Chapter 4 it was possible to experiment with a simple method of reconstructing a triangle-based mesh from the sliced representation of the head. It was implemented as an extension to the slice set import script, and created quadrilateral faces from the vertices in the slice contours by naively joining correspondingly-numbered points across slices. For some areas of the head this worked fairly well, but for others the were serious artifacts. Figure 7.1 shows the results of this meshing, with the right hand view showing the worse side of the head. These artifacts are due to the fact that points are not always distributed in the same way between neighbouring slices, so correspondingly-numbered points are often in different positions from one slice to the next.

Similar problems were encountered by the research group during the development of the simulation procedure and are documented in [3, p 121]. In fact, overcoming these problems is a serious challenge which is still being worked on.

Whilst using Blender to inspect and modify slice sets it was also noted that the shape of the head was being communicated well despite the rendering taking place only from a set of points, with no mesh information. Thus, instead of attempting a visualisation method based on polygonal meshes, it was decided to investigate techniques for rendering the slice set data directly.

7.4 Point based rendering

While most current graphics software and hardware is optimised around working with polygonal meshes, recently the use of points as a graphics primitive has become more common. This has been motivated in part by the need to visualise point-based data sets, such as those captured from laser scanners, and partly by the high speeds possible with point-based rendering techniques, which can allow visualisation of very large data sets. A variety of rendering methods of varying complexity are described in [9]. The next section describes the implementation of the simplest of these methods and steps taken towards the implementation of the more complex ones.

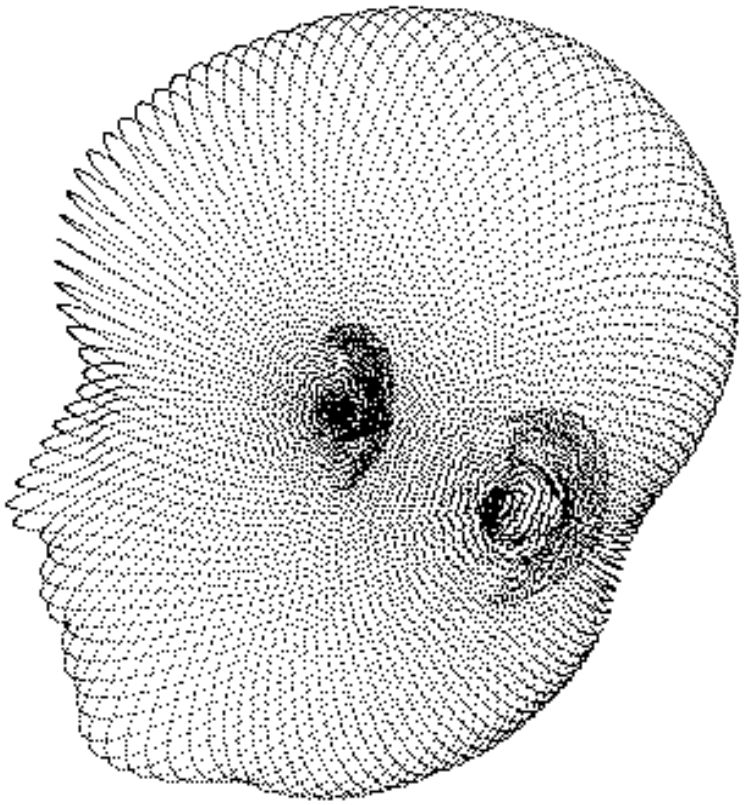

Figure 7.2: Rendering as single-pixel points

7.5 Implementation

7.5.1 Screen-aligned discs

The simplest point based rendering technique is to simply render each point in the dataset as a single pixel on the screen. This was simple to implement using OpenGL's standard point primitive. This level of rendering was mostly useful to test the operation of the interaction methods and to check that the dataset was being read correctly. The results can be seen in figure 7.2.

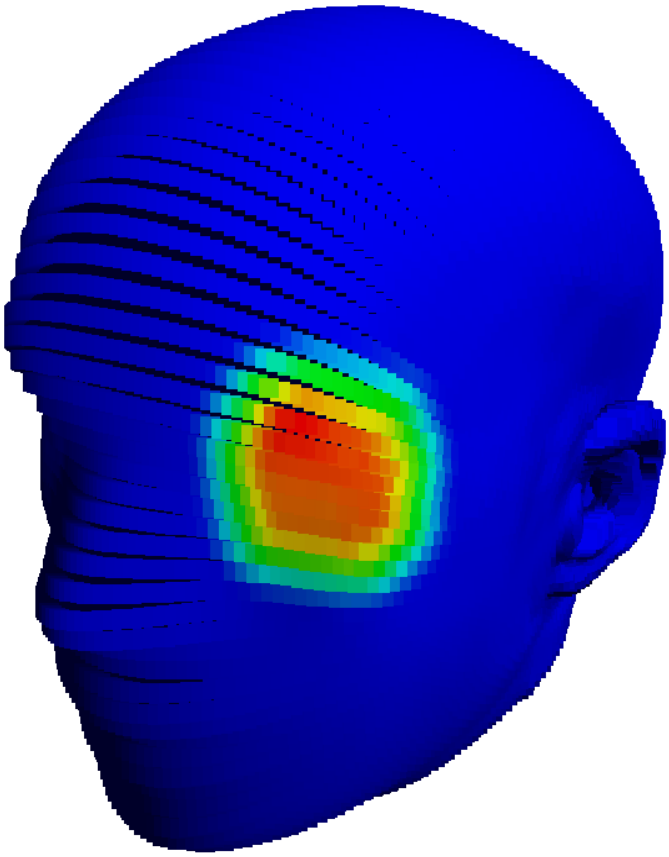

Figure 7.3: Rendering as squares with lighting

Using OpenGL's native point rendering it is also possible to render points at a constant size in screen-space, and with one of two shapes: square or soft-edged disc. Figure 7.3 shows square points, including lighting and colouring of a simple error function on the surface, representing the same enlarged cheekbone shown in figure 5.1. This rendering shows some undesirable artifacts:

- Obvious holes between slices in the forehead
- Uneven outline
- Loss of fine detail in the pinnae
- Quantisation of colouring on the cheekbone

These artifacts are addressed in section 7.5.3.

Soft circular points are more desirable since they better match the spread of energy each point represents. Rendering "splats", as point-based primitives are referred to in the literature, can be thought of as a resampling process. As in any other such process, the energy of each discrete point should spread over the surround space to some degree. In the case of surface splatting, the amount of spread is harder to determine than with a medium such as digital audio where there is a rigidly defined sampling grid.

Attempting to render with soft circular points produces different artifacts, as shown in figure 7.4. These artifacts are examined in the next section.

7.5.2 Two-pass depth buffering

There is a fundamental conflict between the ways in which depth-buffering and alpha-blending are performed in current graphics hardware. Alpha blending is a method for blending two primitives together. First, one primitive is drawn, then the other is drawn over the top, with parts slightly transparent. The result is a merging of the two. Depth buffering is a means to display only the front-most objects in a scene. It works by deciding, every time a pixel is about to be drawn, whether it falls in front or behind what is currently being displayed. If it would fall behind, it is not drawn.

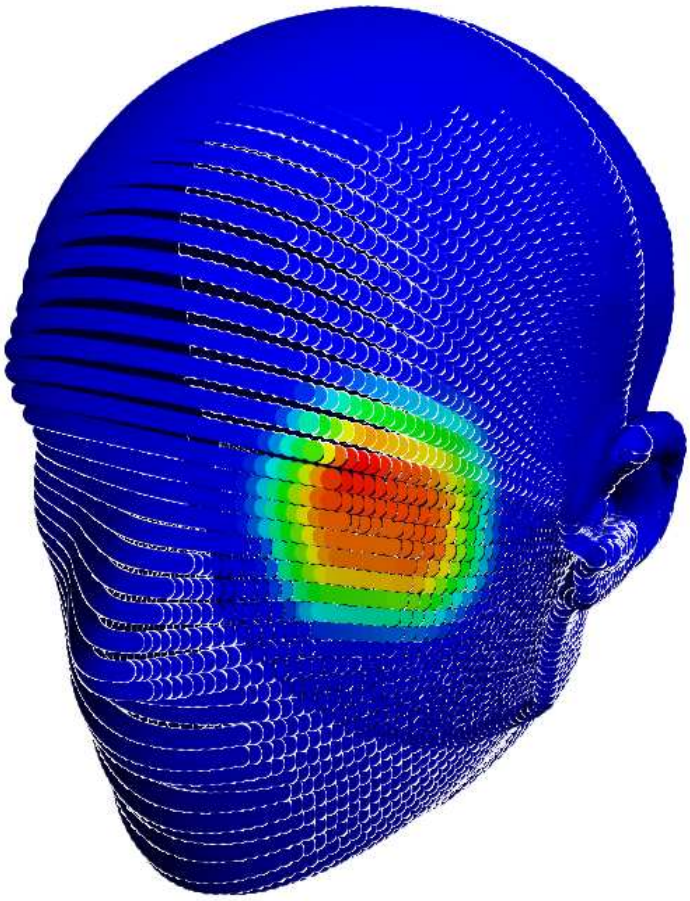

Figure 7.4: Attempting to render soft-edge points, showing z-buffering artifacts

In order for the soft circular points of figure 7.4 to blend into each other, they must be drawn using alpha blending. However, in order for only the front surface of the head to be drawn, and not the side which faces away from the viewer, the head must be drawn with depth-buffering enabled. Since this causes only the front-most parts of objects to be drawn, in places where two circular points should overlap, only one of them is seen. The other would fall behind the current display, and so is discarded by the depth buffering algorithm. This results in the bright outlines around the discs.

A technique to address this problem, ε-z buffering, is mentioned in [9]. The method is to render the scene in two passes: first a pass which determines the depth of the front-most objects, and secondly a pass which draws objects only when they are close to this depth. Thus, overlapping discs forming the front of the head are all drawn, but discs much further away forming the back of the head are not.

Unfortunately because of the behaviour of the OpenGL point primitives this technique cannot easily be used in conjunction with simple point drawing. However it can be applied to the more sophisticated rendering method described in the next section, and it is implemented in the visualisation application.

7.5.3 Splat sizing and orientation

The artifacts in figure 7.3 have two causes:

- The squares are drawn as a constant size, no matter what the spacing of the points in the data set is, leading to holes where they are too small and a loss of detail where they are too large, and also the quantisation of colouring
- The squares are drawn facing the camera, no matter what the orientation of the surface at that point is, causing the outline of the shape to show the outline of the squares it is built from.

The remedy is to render points which are dynamically sized according to how much space on the surface of the head they need to fill, and to render them aligned along a tangent to the surface such that they wrap around the shape. In order to implement this, higher-level primitives than simple OpenGL points must be used. The implementation in this project uses quadrilateral polygons, textured with a Gaussian radial gradient to give the impression of soft-edged discs.

Sizing these primitives requires a measure of the area taken up by a particular point in the data set. In the current implementation, this is calculated simply from the distance to the ear-to-ear axis, since the radial slicing of the head about the Y axis means that points on the outer edges are spaced more widely in the XZ plane.

Aligning these primitives with the surface requires a 3D normal vector at each point. These are calculated as for the error scalarisation program in chapter 6, with the third dimension provided by rotating them to face radially outward from the ear-to-ear axis, such that they lie in the planes originally used to slice the head mesh.

The results of implementing this rendering method can be seen in figures 7.5 and 7.6.

7.5.4 Towards EWA splatting

Figures 7.5 and 7.6 improve on the artifacts produced by the simpler rendering methods. The outline of the sphere is perfectly smooth, and it has holes only towards one side, near the axis of slicing. The quantisation of the colouration is still present but is not as sharp as before. The fine detail in the pinnae is not obscured although there are additional artifacts which make this hard to see. The artifacts present in this rendering method are two:

- Issues with the ε-z buffering leading to corruption around the edges of the head model
- Holes near the axis of slicing due to the distribution of the points there.

Time prohibited further investigation of these problems but the literature does contain one further enhancement to the splatting process: using ellipses instead of discs as the primitives. This is known as EWA Splatting and is presented in [10]. It seems that this might help solve problems around the slicing axis: in this area, points are closely spaced circumferentially about the axis but fairly loosely radially, so using a longer, thinner shape could leave fewer holes.

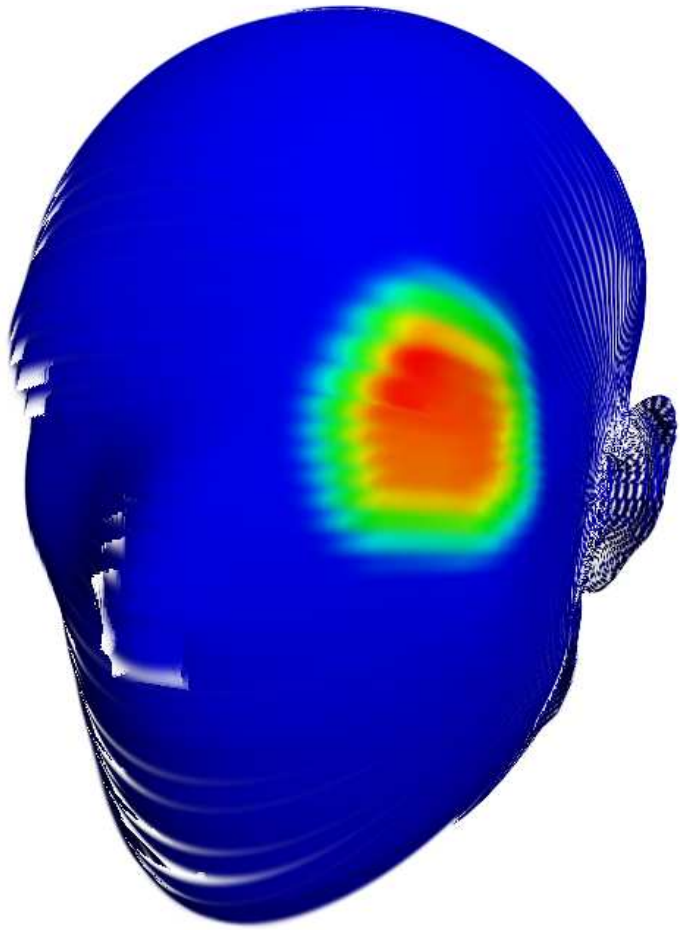

Figure 7.5: Rendering a head with sized, aligned, Gaussian splats

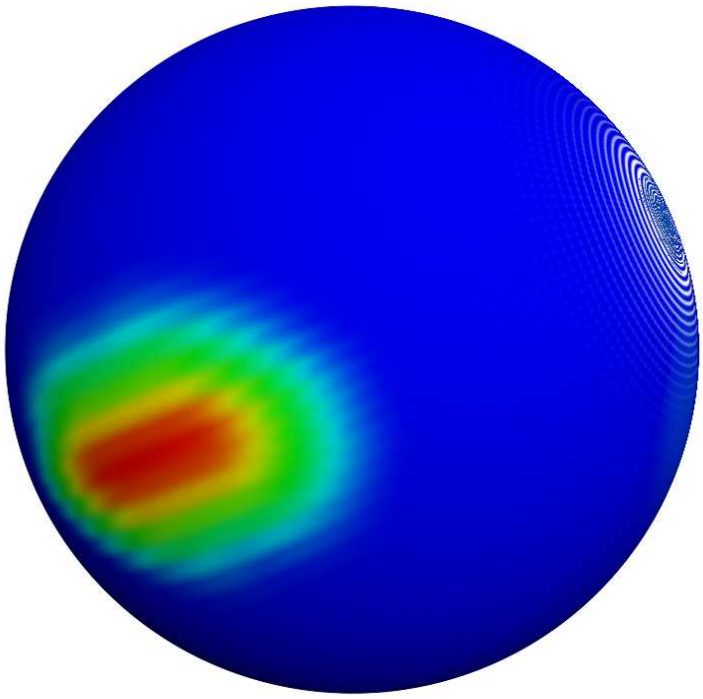

Figure 7.6: Rendering a sphere with sized, aligned, Gaussian splats

Chapter 8

Conclusion and extensions

This project has developed a series of tools to assist with the interpretation of data about the sensitivity of the acoustics of the human head to changes in head and pinnae shape. Three of these tools, the means to edit slice sets in Blender, the slice set decimator, and the slice arithmetic program, have mainly been useful in the development and testing of the others, the error scalariser and the visualisation application itself. These last two provide some insight into how the data produced by the research group could be presented in a visual form, and with some further work they may be suitable for use in the analysis of that data.

There are many possible enhancements and extensions to the work presented here:

- Clearly further refinement of the rendering of the data set is necessary to display the data in a clear and obvious manner
- The visualisation application could be extended to allow quantitative measurements to be extracted from the display, for example by allowing a point on the surface to be chosen and the data associated with it displayed - both position and sensitivity value
- The interface of the visualisation could be improved to allow the user to explore the data in the frequency domain as well. DPS queries could be run searching for areas sensitive to particular frequency areas in the HRTF, and the results could be interpreted by sweeping through these areas in real-time with the head

showing how areas of sensitivity move as frequency changes, effectively adding another dimension to the data

- The visualisation could be adapted to the display of similar data. One interesting application might be to investigate how uniformly the DPS database covers all possible surface deformations, by accumulating an error function of all possible parameter changes. If the database contains results for changes to all parts of the head and pinnae, the result should be a fairly uniform distribution of values over the whole head. Thus the visualisation could be used to highlight parts of the shape itself that are more sensitive to changes in the EFT parameters
- The rendering algorithm could be adapted to output higher-resolution still images for use in presentations and publications, as well as interactive data analysis.

Bibliography

[1] Bade, R., Ritter, F., Preim, B. *Usability Comparison of Mouse-based Interaction Techniques for Predictable 3d Rotation*, Smart Graphics 2005: 138-150

[2] Bergman, L. D et al. *A Rule-based Tool for Assisting Colormap Selection*, IBM Research, available at `http://www.research.ibm.com/dx/proceedings/pravda/`

[3] Hetherington, Carl T. *HRTF estimation by shape parameterization of the human head and pinnae*, PhD thesis, University of York, 2004

[4] Howard, David M and Angus, James. *Acoustics and Psychoacoustics,* Focal Press, 2002

[5] Jin, S., Lewis, R., West, D. *A comparison of algorithms for vertex normal computation,* The Visual Computer Vol. 21 No. 1 pp. 71-82

[6] Kobbelt, L. and Botsch, M. *A survey of point-based techniques in computer graphics,* Computers & Graphics, Vol. 28 No. 6 pp. 801-814

[7] Luebke, David P. *A Developer's Survey of Polygonal Simplification Algorithms*, IEEE Computer Graphics and Applications May/June 2001 pp24-35

[8] Research Support Group, University of Alberta. *Visualization: VTK and Blender*, from `http://www.ualberta.ca/AICT/RESEARCH/Vis/VTKBlender/index.html`

[9] Sainz, M., Pajarola, R., Lario, R. *Points Reloaded: Point-Based Rendering Revisited*, Eurographics Symposium on Point-Based Graphics 2004

[10] Zwicker, M., Pfister, H., van Baar, J., Gross, M. *EWA Splatting*, IEEE Transactions on Visualization and Computer Graphics Vol. 8 No. 3 pp. 223-238

Appendix A

User instructions

A.1 Slice editing with Blender

A.1.1 Introduction

Space prohibits even a basic introduction to Blender in its entirety, but the basic steps required to inspect and modify a slice set are detailed here. More information about the program can be found in the user manual at `http://mediawiki.blender.org`.

The slice import and export Python scripts have been written and tested against Blender 2.41 but should work with later versions. Earlier versions may not function correctly or at all due to changes in Blender's Python API. A functioning installation of Python is also required.

Opening slices.blend results in a window similar to figure A.1. The screen is divided into four main sections: a large 3D view on the left, an empty buttons window at the bottom and two text editors on the right. The upper text editor contains the slice import script, and the lower the export script. To run either of these scripts, select `Run Python Script` from the `File` menu below the appropriate text editor window.

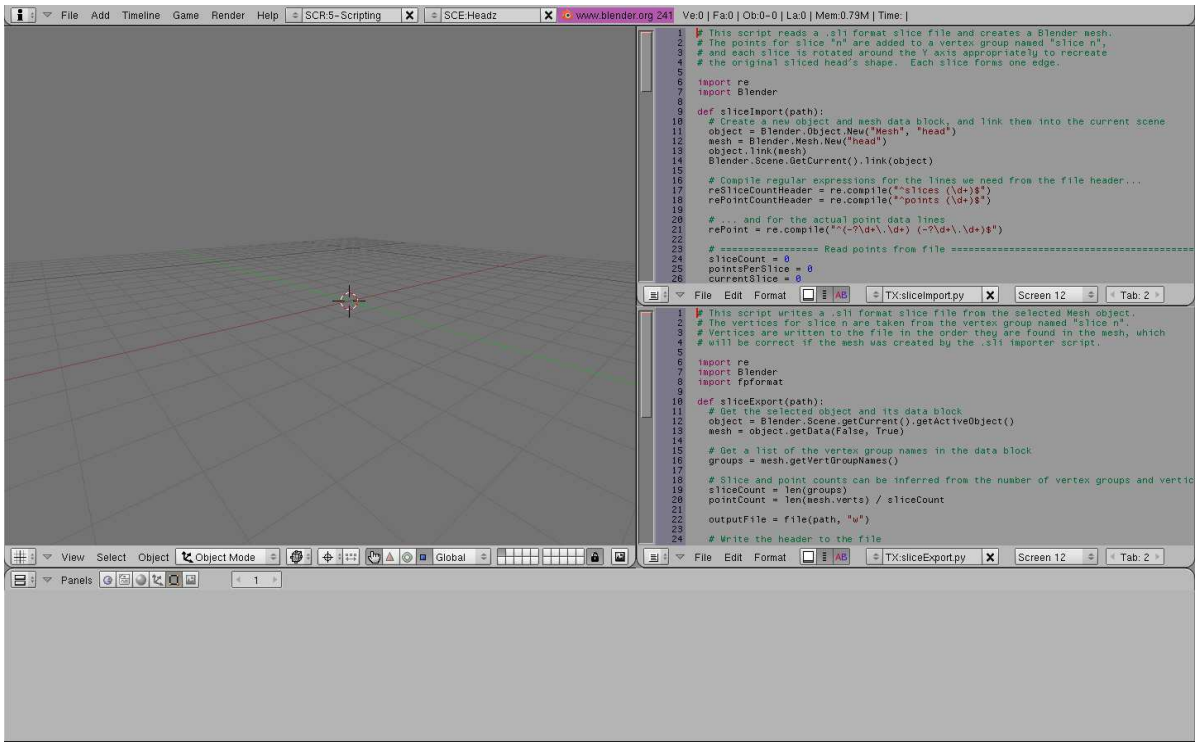

Figure A.1: Initial view of slices.blend

A.1.2 Importing a slice set

Running the import script in the upper text editor window results in a file browser used to select a .sli file for import. Once a file has been chosen and `Import Slices` clicked, the slices are imported as a new Blender mesh object, which appears in the 3D view. It can be inspected using standard Blender view navigation controls. Of principal interest are orbiting the camera by dragging with the middle mouse button over the 3D view, panning the camera by shift-dragging with the middle mouse button, and zooming with the - and + keys on the numeric keypad. The radial slicing structure can be seen most easily in an orthographic view: select `View/Front`, then `View/Orthographic` from the menus below the 3D view.

A.1.3 Editing points

The vertices making up each slice can be viewed by selecting the head object (by right clicking it), and choosing `Edit Mode` from the mode popup below the 3D view.

Depending on the resolution of the particular slice set, it may be necessary to zoom in considerably to be able to distinguish individual points. The vertices can be edited with all the standard Blender tools. Figure A.2 shows an area around the tip of the nose being moved outward away from the centre of the head, scaling along the radial direction only in order to keep the slices parallel. This was achieved by setting the `Pivot` popup below the 3D view to `3D Cursor`, and the `Proportional` popup to `On`. Scaling is then performed by selecting a vertex by right-clicking, then pressing `s` and moving the mouse, and confirming the alteration by left-clicking. If modifications are made that move vertices out of the plane of their slice, the slice set may still be exported but the errant points will be replaced with their projections onto the plane of their slice.

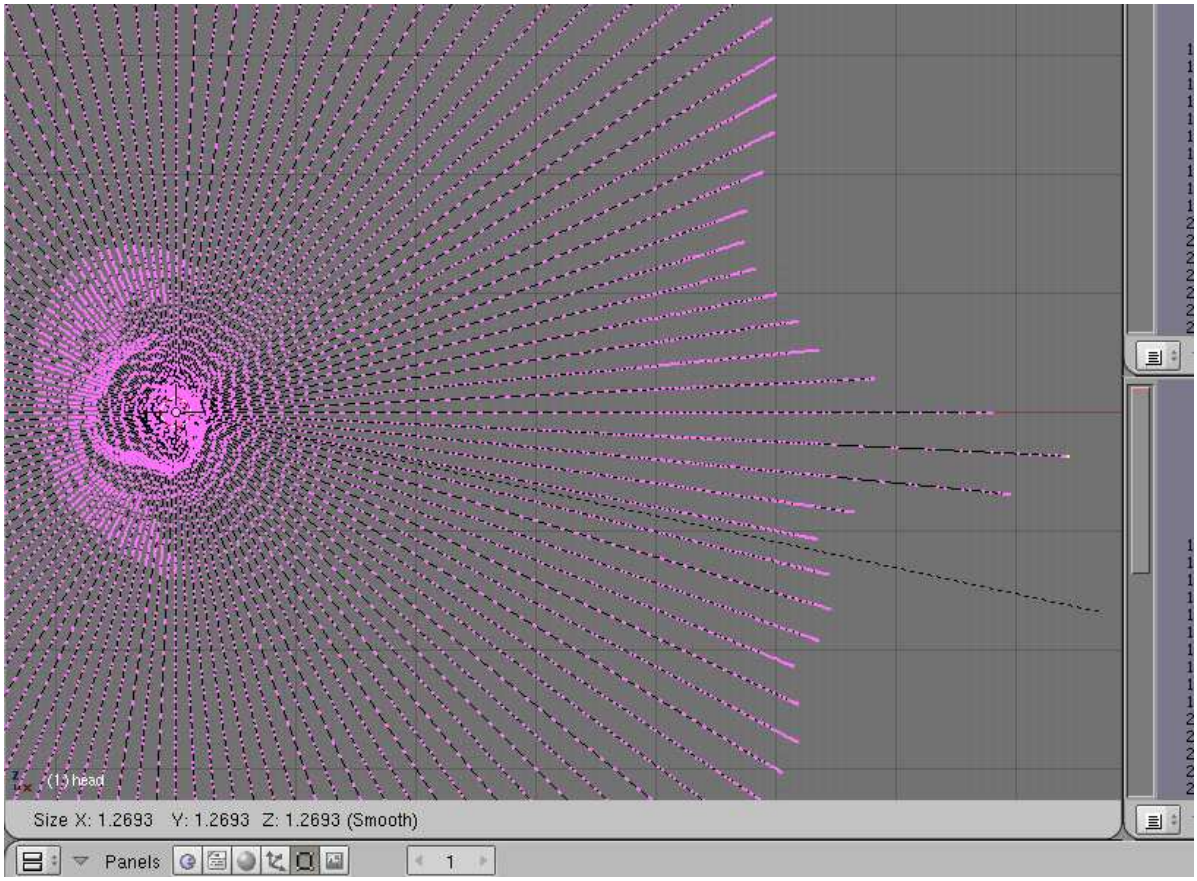

Figure A.2: Altering slice vertices in Blender

A.1.4 Exporting slices

A modified data set may be exported to a .sli file by running the script in the lower text editor window. The object should be selected in the 3D view, and care must be taken to switch to Object Mode using the Mode popup below the 3D view before running the script. Again a file browser will appear, prompting for a path to which the slice set will be written when `Export Slices` is clicked.

A.2 Decimation

The slice decimator is run from the command line, taking three arguments: a decimation factor n, the path to an input file, and the path to an output file. The decimation factor should be an integer which divides exactly into the number of points in each slice. Every nth point from the input file is written to the output file, all other points being ignored.

For example, to decimate the slice set he6_sym2a.sli by a factor of 8 and store the result in the file he6_sym2a_deci8.sli, one would execute the command:

```
slicedecimate 8 he6_sym2a.sli he6_sym2a_deci8.sli
```

If the input file had 2048 points per slice, the output would have 256 points per slice. The number of slices would remain unchanged.

A.3 Slice arithmetic

`Slicearithmetic` is a command-line program which can either add two slice sets together, or subtract one slice set from another. It takes four arguments, the first of which defines the operation to be performed. Supply a `+` character for addition, or a `-` character for subtraction. The remaining three arguments are the paths to the two input files, and the path to the output file. For subtraction, the order of the input files is important. For example,

```
slicearithmetic - apple.sli pear.sli banana.sli
```

...would subtract the file apple.sli from the file pear.sli, writing the result to banana.sli. The order is the same as the order in which the operation would normally be spoken aloud - "subtract one slice set from another slice set and write the output to this file".

For addition, the order of the two input files is not important - the commands

```
slicearithmetic + cabbage.sli lettuce.sli broccoli.sli
```

and

```
slicearithmetic + lettuce.sli cabbage.sli broccoli.sli
```

are equivalent, both performing addition of the slice sets `lettuce.sli` and `cabbage.sli` and storing the result in `brocolli.sli`. The arithmetic is performed on a point-by-point basis, simply adding or subtracting the 2D vectors forming the coordinates of each point.

A.4 Error scalarisation

The conversion of error data from vector to scalar form is performed by `errorscalarise`, which operates on two input files: the template (undeformed) slice set and the error (delta) slice set. The output is stored in a .sli3 file containing both the geometry of the template slice set and a per-point scalar error value. The process is launched from the command line with a command such as the following:

```
errorscalarise template.sli error.sli output.sli3
```

The program will first conform the error function to remove any ambiguities. This may take anywhere from a second to several minutes, depending on the resolution of the template slice set and the magnitude of the changes described by the error slice set. Status messages will be printed during this process. Once it is finished, the graphical display will appear (figure A.3)

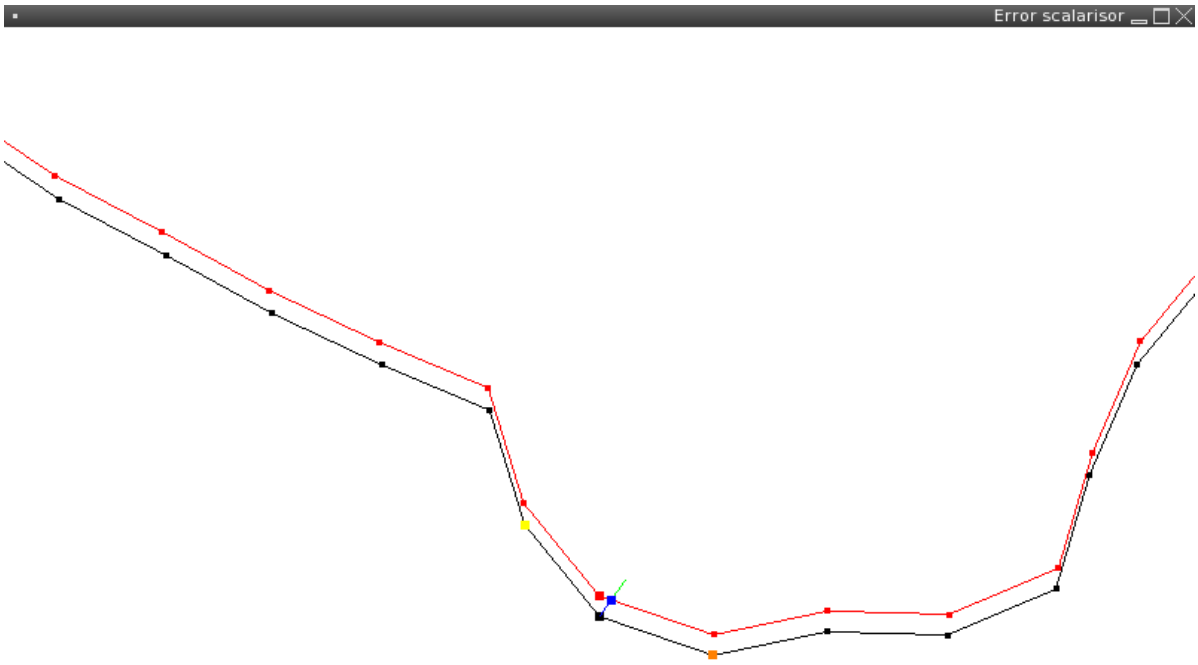

Figure A.3: The error scalarisation display

This display shows a single slice at a time, focusing on a single point of that slice and drawing the construction lines used to determine the distance from that point to the deformed slice (formed by adding the error slice onto the template slice). This display was primarily useful during the design of the algorithm, but it may be useful for inspection of the relationship between the template and deformed contours. It should be noted that the absolute magnitude of the error slice set is not preserved; the conform process uniformly scales the error vectors down such that there are no ambiguities.

The template slice is drawn in black, and the deformed slice in red. The current point is highlighted on both with a larger square. The previous and next points are highlighted on the template slice in yellow and orange respectively. A computed normal to the template slice is drawn in green, and the intersection of this normal with the deformed slice is highlighted in blue. The display can be zoomed in and out with the + and - keys.

The user can step through the slices one point at a time using the `n` key to move forward and the `b` key to move backward, but normally the algorithm should be run at full speed without display by pressing `space`. Normally the process will rapidly reach the end of the dataset, displaying a message in the command-line window prompting the user to press `q` to write the output file and exit. If the process finds an error it will stop at that point and display the corresponding point. An error is likely to indicate a problem with the conform or scalarisation algorithms, although an anomaly in the data set such as two exactly co-incidental points may be possible. There are no features for correcting the error, but hopefully the problem can be analysed by inspecting the display.

A.5 Visualisation

The `visualise` program takes just one command-line argument, the path to a .sli3 file to display. It launches a graphical display of the dataset.

The model can be navigated by dragging with the mouse to orbit the camera, right-dragging to pan the camera, and pressing `+` and `-` to zoom in and out.

The rendering method can be changed on-the-fly with the following keys:

`1` - Render a single point for each vertex, coloured according to the error value

`2` - Render a single point for each vertex, with no colouring, and render every 8th vertex normal as a green line

`3` - Render points as constant-size, screen-aligned aliased circular splats, coloured according to the error value

`4` - Attempt to render points as variable size, surface-aligned, perspective correct anti-aliased disks, coloured according to the error value.

A.6 Overall workflow

The decimation and slice arithmetic tools were principally useful for test data generation during development - the decimator to increase interactivity within Blender and the visualisation, and the slice arithmetic program for producing error delta slice sets from a base slice set and a deformed version edited with Blender. For display of results

from the simulation process, the two steps which need to be performed are scalarisation, and visualisation. Assuming a template slice set named `template.sli` and an error function slice set output from simulation named `error.sli`, the two commands to be issued would be:

```
errorscalarise template.sli error.sli result.sli3
visualise result.sli3
```

Appendix B

File format specifications

B.1 Slice files

The slice sets accepted and generated by the programs described in this report use the .sli file format devised within the research group at York. The format is a simple ASCII text file, with Unix-style line endings. The file header consists of three lines:

```
slice_set
slices 64
points 2048
```

These are a "magic" sequence to identify the file type, the number of slices in the file, and the numbers of points in each slice respectively. They are followed by the point data, presented as ASCII-encoded floating point Cartesian X/Y coordinates of the points forming the contour of each slice, beginning with slice zero:

```
slice 0
0.00000000000000000000 64.70647913543892600000
0.32377325143353658000 64.63572477921538000000
0.64752872519084592000 64.56488906342572400000
0.97449943263103200000 64.53132096456312200000
1.29841304261249450000 64.59964278367390000000
...
```

The slice number line appears at the start of each slice. There is no file footer other than the final line break. The number of decimal places after the point varies between the example slice sets provided by the research group, so the programs presented here accept any reasonable precision.

Files generated by the slice arithmetic process which represent differences between two slice sets are stored in the same format since each delta value is an X/Y vector. For the results of the simulation process which will feed the visualisation, again the same format can be used, with each vertex containing the X and Y components of the accumulated error at that point.

B.2 Combined slice and error files

The error scalarisation process outputs a file format containing both the geometry of the head and the per-point error value to be visualised. The files are very similar to the above geometry-only slice sets, with only the addition of a third floating-point value to each vertex data line, representing the scalar error value associated with that point. The first header line is changed to "slice3_set", and the file extension to ".sli3", to indicate this. The rest of the header is the same as above:

```
slice3_set
slices 64
points 2048
slice 0
0.0000000000 196.1210021973 0.5234000000
0.6078772545 196.1210021973 0.0001234500
1.2157545090 196.1210021973 0.4563234000
1.8236317635 196.1210021973 0.0023450000
2.4315090179 196.1210021973 0.0064240000
3.0393862724 196.1210021973 0.0014534000
...
```

B.3 Storing slice data in Blender scene files

Slice set data can also be stored in standard Blender scene files. If necessary several slice sets can be stored in a single file, as separate Blender mesh objects. The objects generated by the slice import script can be saved as any other Blender object with no loss of information such as slice count or the per-point slice membership attribute. As described in 3.2 on page 16, the contour for each slice is stored as a single edge, with all vertices for a particular slice a member of a vertex group named "slice n" where n is the zero-based integer slice number.

Since the .blend file format also allows storage of arbitrary text data, the original .sli files can be imported and saved as part of the scene, along with the import and export Python scripts, allowing several sets of data to be stored along with the tools to operate on them in a single portable, compressed file for easy transport between systems.