Examensarbete
LITH-ITN-MT-EX--06/037--SE

Visualizing Radar Signatures

Tobias Forslöw

2006-06-07

Linköpings universitet
TEKNISKA HÖGSKOLAN

Department of Science and Technology
Linköpings Universitet
SE-601 74 Norrköping, Sweden

Institutionen för teknik och naturvetenskap
Linköpings Universitet
601 74 Norrköping

LITH-ITN-MT-EX--06/037--SE

Visualizing Radar Signatures

Examensarbete utfört i Medieteknik
vid Linköpings Tekniska Högskola, Campus
Norrköping

Tobias Forslöw

Handledare Christer Larsson
Examinator Björn Kruse

Norrköping 2006-06-07

Avdelning, Institution
Division, Department

Institutionen för teknik och naturvetenskap

Department of Science and Technology

Datum
Date

2006-06-07

Språk
Language

☐ Svenska/Swedish
☒ Engelska/English

☐ _____

Rapporttyp
Report category

Examensarbete
☐ B-uppsats
☐ C-uppsats
☒ D-uppsats

☐ _____

ISBN

ISRN **LITH-ITN-MT-EX--06/037--SE**

Serietitel och serienummer
Title of series, numbering

ISSN

URL för elektronisk version

Titel
Title

Visualizing Radar Signatures

Författare
Author

Tobias Forslöw

Sammanfattning
Abstract

It is important for the military to know as much as possible about how easily detected their vehicles are. One way among many used to detect vehicles is the use of radar sensors. The radar reflecting characteristics of military vehicles are therefor often rigorously tested. With measurements and simulations it is possible to calculate likely detection distances to a vehicle from different angles. This process often produces very large data sets that are hard to analyze.

This thesis discusses and implements a method for visualizing the detection distance data set and also discusses a lot of related issues with a focus on computer graphics.

The main concept is called spherical displacement and the idea is to visualize the detection distances as a surface with the imagined vehicle in the center point. Detection is likely inside the surface but not on the outside. This concept is the next step from the colored sphere where the colors represent the detection distance which was previously used.

The thesis project resulted in a visualization tool that uses the new concept and can handle large data sets. The spherical displacement concept is more intuitive and shows detail better than the colored sphere visualization.

Nyckelord
Keyword

visualization, scientific visualization, radar, radar signatures

Abstract

It is important for the military to know as much as possible about how easily detected their vehicles are. One way among many used to detect vehicles is the use of radar sensors. The radar reflecting characteristics of military vehicles are therefor often rigorously tested. With measurements and simulations it is possible to calculate likely detection distances to a vehicle from different angles. This process often produces very large data sets that are hard to analyze.

This thesis discusses and implements a method for visualizing the detection distance data set and also discusses a lot of related issues with a focus on computer graphics.

The main concept is called spherical displacement and the idea is to visualize the detection distances as a surface with the imagined vehicle in the center point. Detection is likely inside the surface but not on the outside. This concept is the next step from the colored sphere where the colors represent the detection distance which was previously used.

The thesis project resulted in a visualization tool that uses the new concept and can handle large data sets. The spherical displacement concept is more intuitive and shows detail better than the colored sphere visualization.

Preface

For me coming into this project with a background in media technology rather than physics has been very interesting. Radar was a new subject to me so the first thing I did was to read a few books on the subject. I suspect this will often be the case when working with scientific visualization and that makes it all the more interesting.

This thesis project was carried out at SAAB Communication in Linköping during the spring of 2006. It was advertised on the Internet during the fall of 2005 and I'd like to thank David Berg for pointing me to it, without his help I would never have found it. I also want to thank my supervisor Christer Larsson at SAAB Communication for his help and support and many interesting discussions. Finally I want to thank my opponent Erik Carlson and my examiner Björn Kruse for their patience and support.

Table of contents

Table of figures

Table of tables

1 Introduction

This chapter discusses the background and purpose of the thesis project and also contains the requirements and considerations implied by SAAB Communication.

1.1 Background

Information has always been a central part of warfare and this is even more so today. The key problem for military forces is to gather as much information as possible while keeping the enemy guessing. One of the most important means of gathering information today is through the use of radar, therefore the research in this area is intense. Since the 1950s researchers have been working on the problem of hiding vehicles from detection using so called stealth technology. The most commonly known stealth vehicles are probably the American B-2 bomber and F-117A fighter aircrafts.

To make it easier to create and improve on stealth vehicles the radar reflections can be measured and analyzed to find weaknesses in the design. One approach is to use Inverse Synthetic Aperture Radar (ISAR) to create a set of color coded images that show the magnitude of the reflections for several different angles. These images can be projected onto a model of the vehicle itself to illuminate the parts that cause the most reflections.

The ISAR data also contains the information to make it possible to calculate the likely detection distances for a vehicle from all possible angles or at least an approximation of them.

1.2 Purpose

Military platforms usually have specifications involving radar cross section (RCS). The RCS is a measurement of the amount of the radar signal that is reflected back to the radar from the target. The specifications are often broken up into different angular sectors for the platform. These sectors correspond to different types of radar threats. For each sector a maximum RCS value is typically specified. These values will vary greatly across a threat sector and it is often very hard to meet the specifications for all sectors. It is also hard to show what difference a change in the design will provide.

This project aims to create a tool able to visualize the overall radar reflectance properties of a platform as well as compare different measurements or simulations. The tool should be highly interactive and able to highlight the different threat sectors to help showing that requirements have been met. Such a tool may also be useful to educate military personnel about the radar signature of their vehicle.

1.3 Further requirements and considerations

The project was carried out at SAAB Communication who had already done some work in the area. A tool called BaseTool that could calculate detection distances and show them as colored spheres had already been developed but SAAB felt more work could be done. The idea of using spherical displacement was presented to them and they decided that it was worth exploring. They also wanted to see if it was possible to optimize the preprocessor module that produced the detection distances. Other things on the wish list was the ability to produce high quality visualizations for marketing purposes and a way of visualizing RCS specifications.

There were also prerequisites on how the application should be designed and what tools to use. The previous version had been created using open source tools and one requirement was that the new

version continued to use only such tools. The existing version of the application used the script language python. To make it easy to maintain and update the application SAAB wanted the new version to use python as well since the language was known by key individuals within the project.

One of the tools used in the first version was the Visualization Toolkit (VTK), an open source library for scientific visualization that could be controlled from python. The company wanted to continue to use VTK if it was possible.

2 Theory

This chapter contains the background theory needed to understand the rest of the report. It is divided in three parts, radar, computer graphics and interpolation.

2.1 Radar

The acronym *radar* stands for “radio detection and ranging”. Radar was initially developed to replace visual target detection for several reasons. Radio waves suffer much less attenuation through the atmosphere than light waves, and signals in the lower frequency ranges actually propagate over the visible horizon. This makes it possible to detect targets long before they are visible optically. Radars also work well at night when there is little or no ambient light to illuminate the target. [Knott et al, 1993]

The basic principle of a radar is to emit electromagnetic energy and listen to the echo to determine something about the surrounding environment. Since the speed of electromagnetic waves is known it is easy to find the distance to the target by clocking the time between transmission and reception. To find the location of the target the azimuth and elevation angles must also be determined. This can usually be done by measuring the direction of the antenna since most radar antennas are a lot more sensitive in one direction. [Knott et al, 1993]

It is also possible to determine the velocity of the target because of the Doppler effect.

2.1.1 Radar Cross Section

Radar Cross Section (RCS) is a measurement of the amount of energy reflected by the object back towards the radar antenna. It is defined as the effective area of reflection and is uniform in all incident angles only for a perfect sphere but for all practical objects it varies with the azimuth and elevation angles to the radar and therefore shows how detectable the object is to radar from different points of view. The RCS also varies with the frequency of the incoming radar signal.

RCS can be used to calculate detection distances for different radar threats. The detection distances are much more intuitive than SAR images and can thus, if visualized in a good way, be used as information to fighter pilots and other military personnel without a need to understand the underlying physics.

2.1.2 Inverse Synthetic Aperture Radar

Synthetic Aperture Radar (SAR) and Inverse Synthetic Aperture Radar (ISAR) are two commonly used radar modes that can be used for mapping. Any time two objects have relative rotation to one another, a SAR map can be formed in the plane of rotation. [Lynch, 2004]

In ISAR the object that is mapped is rotated and the radar is stationary rather than the other way around. This is a method used under controlled forms for measurements of radar reflection properties of aircraft and other vehicles.

2.1.3 Stealth

A stealth vehicle is often thought upon as a vehicle that is nearly invisible to radar, but making a vehicle invisible to radar is called radar signature control and is just a part of the much larger concept of stealth.

Stealth is not a single concept but an assemblage of techniques, which makes a system harder to

find and attack. Included in this concept is controlling emissions like heat, sound, exhaust fumes, communications, radar, et cetera and also everything that requires external illumination of some kind, like magnetic and gravitational anomalies, reflection of sunlight, sound, radar and laser illumination and also reflection of the ever present ambient radar waves from unrelated sources (often called splash track). [Lynch, 2004]

2.1.3 Detection distances

To calculate the detection distances for a platform more information than the RCS is needed. As the RCS is a measurement of the reflected energy it is possible to calculate how much energy will be reflected to the radar sensor if the amount of energy that is sent out is known. The next consideration that has to be made is how much reflected energy is needed for the radar sensor to be likely to detect the object. The frequency of the radar sensor is also needed since the RCS varies with the frequency of the incoming signal. If the RCS of the object, the output power and frequency of the radar sensor and the reflected energy needed to detect the object are all known it is possible to calculate the likely maximum detection distances.

The magnitude of the response for different angles is proportional to the power of the incoming radar signal but as the RCS varies with the frequency it should be noted that the detection distances are very much dependent on the radar sensor and calculations need to be made for different kinds of sensors to get a good picture of the radar reflectance properties of the platform.

2.2 Computer graphics

"The purpose of a graphics pipeline is to take a description of a scene in three-dimensional space and map it into a two-dimensional projection on the view surface – the monitor screen." [Watt, 2000] Figure 1 shows the different spaces that the vertices are transformed between to achieve the rendering and the different operations that are carried out in those spaces.

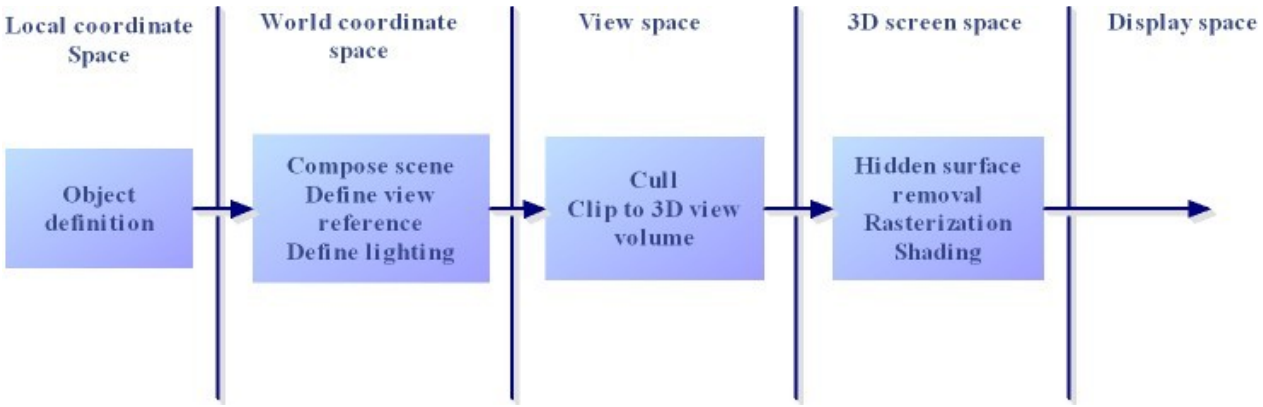


Figure 1: Graphics pipeline [Watt, 2000]

2.2.1 Local coordinate space

The vertices of a polygon mesh objects are usually stored in their own coordinate system. This makes it possible to, for example, set the origin in the middle of the object and align the axles with the object. It also helps when creating local animations such as the walk of a 3D model of a human.

The face and vertex normals are usually also stored in the local coordinate system. [Watt, 2000]

2.2.2 World coordinate space

This is a global coordinate space into which all objects are transformed so that their spatial relationship can be defined. The light sources of the scene are also specified in the world coordinate space. Animations of things moving in relative of each other are also done in world coordinate space.

2.2.3 View space

To be able to render the scene some viewing parameters has to be set. The minimum definitions needed are a view point, a view direction, a view coordinate system, a view plane and a view volume. The view point is also called the camera point or eye point and describes the viewer's position in the world space and is usually defined together with a view direction. The view plane is setup normal to the viewing direction and can be seen as the screen onto which the scene later will be projected. The view coordinate system is defined by the viewing direction and the view plane.

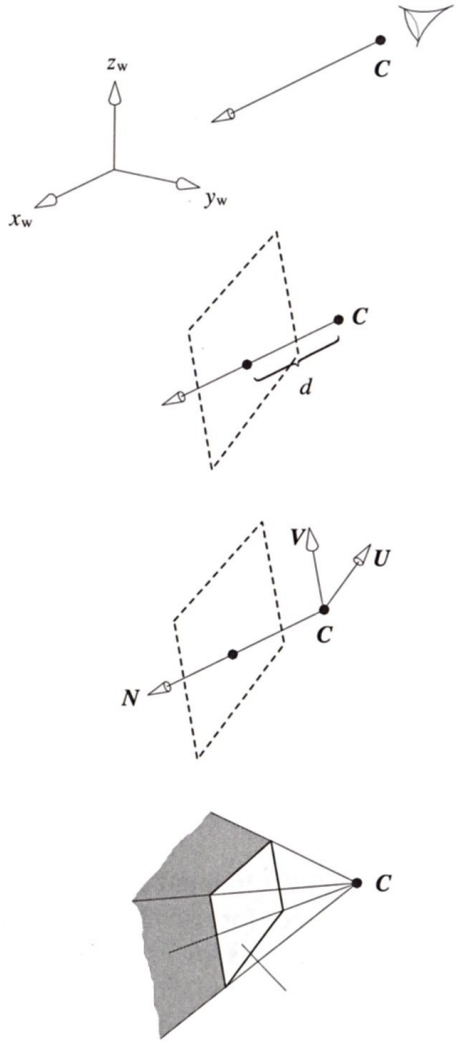


Figure 2: Viewing parameters [Watt, 2000]

Figure 2 shows the different viewing parameters. The first image shows a view point C with a

view direction. The next image shows a view plane that is normal to the view direction. The second image from the bottom shows the view coordinate system that is defined by the view direction and view plane and finally the bottom image shows the view volume as the shaded area created by the frustum of the view point, the view plane and usually also a far clipping plane. [Watt, 2000]

Another operation that is carried out in view space is culling or back-face elimination. It compares the orientation of polygons with the view point and removes those polygons that cannot be seen. This is a very simple test which will remove on average half the polygons in the scene.

2.2.4 3D screen space

There are two common projections used to go from 3D to 2D: Parallel projection and perspective projection. Parallel projection is used when the relative size of objects needs to be preserved. Perspective projection is much more common since it provides a feeling of depth in the scene. In perspective projection objects that are far away appear smaller than objects that are close. Figure 3 shows the difference between perspective and parallel projection.

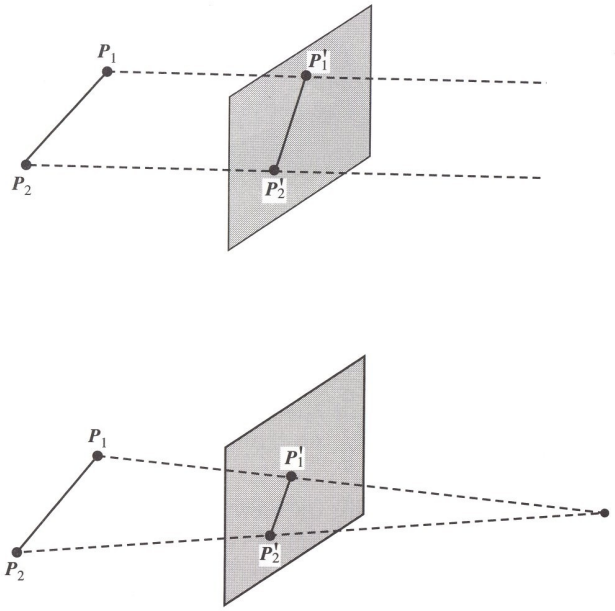


Figure 3: Parallel and perspective projections [Watt, 2000]

While it would be possible to go directly from view space to display space there are a few operations that benefit from an intermediate state. This is the 3D screen space. When going from view space to 3D screen space points are transformed so that a parallel projection would give a perspective projected scene. Figure 4 shows this transformation on a box.

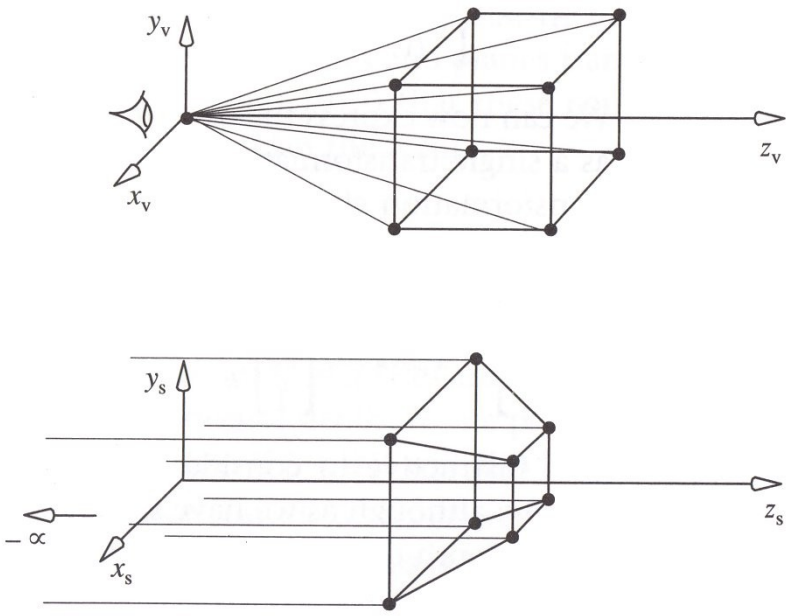


Figure 4: Transformation of a box into 3D screen space [Watt, 2000]

Clipping is the process of discarding all polygons outside the view volume as well as clipping the polygons that intersect the view volume. One of the neat things with 3D screen space is that the sides of the view volume becomes parallel, the view volume in fact becomes a box rather than a pyramid, and this box is also aligned with the coordinate system. Rather than comparing coordinates with plane equations you now simply have boundaries in x, y and z to compare against, which is much more efficient. [Watt, 2000]

In the culling process we removed polygons that were located on the back side of objects but we still need to remove polygons that are obscured by other objects. This is called hidden surface removal. There are several different approaches to hidden surface removal but the most commonly used is the Z-buffer algorithm. The Z-buffer algorithm was invented by Edwin Catmull in 1974. [Catmull, 1974]. The Z-buffer algorithm works in 3D screen space for much the same reasons as clipping. In 3D screen space the camera can be said to be infinitely far away, making lines from the camera towards the scene parallel. This means that if a point (x,y) on a polygon has a higher z value than the same point on another polygon then the first point is hidden. The Z-buffer algorithm basically uses this fact by creating a buffer containing the smallest z value for each pixel.

Figure 5 shows the difference between culling and hidden surface removal where step *a* is culling and *b* is hidden surface removal.

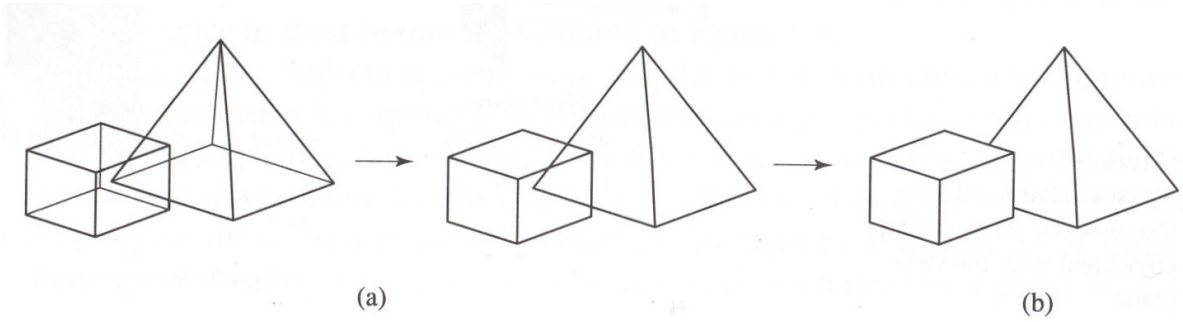


Figure 5: Culling and hidden surface removal [Watt, 2000]

What now remains is the process of creating pixels out of the 3D dimensional content. This is done by rasterization and shading. The rasterization is the process that actually creates pixels from the edges and polygons and the shading process is what colors the pixels. A programmer often has little interest in controlling the rasterization process but shading is one of the most interesting aspects of 3D computer graphics.

2.2.5 Local reflection models

A local reflection model enables the calculation of the reflected light intensity from a point on the surface of an object. [Watt, 2000] The most common model was developed in 1973 by Bui Tuong Phong and published in 1975 in Communications of the ACM. [Phong, 1975]

This model estimates the intensity of the light reflected from a point depending on the normal of the surface and the position of the light source and viewing directions. It is called a local reflection model since it only considers direct illumination.

This model considers the reflected light to consist of three components.

$$I = I_a + I_d + I_s$$

Where I is the reflected light, I_a is an ambient light component, I_d is the diffuse component and I_s is an imperfect specular reflection component.

The ambient component is added to compensate for the lack of indirect illumination in the model. A surface that does not have a clear view to a light source would be completely black without this component since the only light that normally would reach it is light that is reflected off other surfaces. This component is usually just a constant.

The diffuse component is assumed to be perfect diffuse reflection which means that it reflects light equally in all directions. This component is modeled as follows:

$$I_d = I_i \cos(\theta)$$

Where I_i is the intensity of the incident light and θ is the angle between the surface normal and the incident light. In vector notation this becomes:

$$I_d = I_i(\vec{L} \cdot \vec{N})$$

$\vec{L}$ is the light vector, a vector from the light source to the point of interest. $\vec{N}$ is the surface normal at the point of interest.

Finally, the specular component is what is commonly known as a highlight, practically an image of the light source reflected in the surface. This reflection is only seen if the view point is near the

mirror direction of the light source. This specular component is modeled by:

$$Is = Ii * \cos^n(\Omega)$$

Where Ω is the angle between the viewing direction and the mirror direction and n is an index that simulates the degree of imperfection of a surface. The surface becomes a perfect mirror as n reaches infinity. In vector notation this is:

$$I_s = I_i(\vec{R} \cdot \vec{V})^n$$

$\vec{R}$ is the mirror direction and $\vec{V}$ is the viewing direction. Computing $\vec{R}$ is unnecessarily expensive. Instead a halfway vector $\vec{H}$ is usually calculated. $\vec{H}$ is the unit normal to a hypothetical surface that is oriented in a direction halfway between the light direction and the viewing vector such that:

$$\vec{H} = (\vec{L} + \vec{V})/2$$

Using this halfway vector the specular intensity can be calculated as follows:

$$I_s = I_i(\vec{N} \cdot \vec{H})^n$$

Using all this together gives an expression for the light reflected from a point. [Watt, 2000]

$$I = I_a k_a + I_i(k_d(\vec{L} \cdot \vec{N}) + k_s(\vec{N} \cdot \vec{H})^n)$$

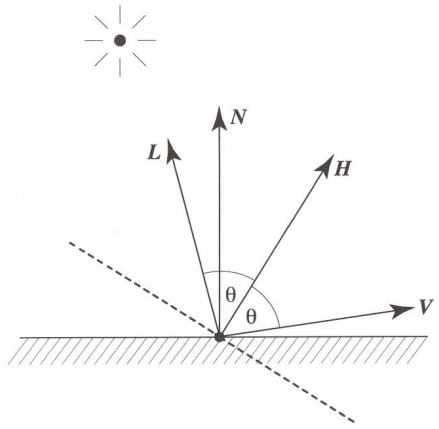


Figure 6: Local reflection model [Watt, 2000]

Figure 6 shows the vectors and angles used in the local reflection model.

2.2.6 Gouraud shading

In Gouraud shading [Gouraud, 1971] the light intensity is calculated in each vertex using the local reflection model method described in the previous section. These intensities are then interpolated across the polygon using bilinear interpolation. To achieve different values at each vertex special normals called vertex normals are used. See section 2.2.8 for details on vertex normals.

The big problem with this method is its inability to correctly handle highlights. Since the intensity values are only calculated in the vertices the algorithm will miss highlights that should have appeared somewhere on the polygon. [Watt, 2000]

2.2.7 Phong shading

A polygon mesh is often an approximation of a smooth surface and as such the normal of the surface should vary across the polygons. Phong shading [Phong, 1975] is a way of dealing with this problem by interpolating vertex normals across a polygon rather than just interpolating the intensity. This solves the problems with the highlights and makes it possible to get a smooth look to objects that in reality are made up of polygons. Using Phong shading is considerably more expensive in terms of computations, but in modern hardware it is not very likely to become the bottleneck since most hardware is built to use this method. [Watt, 2000]

2.2.8 Generating vertex normals

Phong and Gouraud shading use vertex normals to estimate the curvature of the surface. These normals are usually calculated as the average of the normals of all the faces that the vertex belongs to.

$$\vec{N}_A = \vec{N}_1 + \vec{N}_2 + \vec{N}_3 + \vec{N}_4$$

Simply averaging the normals of the adjacent faces sometimes create problems since all faces are weighted equally. This method was introduced in 1971 be Henri Gouraud. [Gouraud, 1971] Better results can be achieved through different weighting schemes. One idea is to weight the face normals by the angle under which a face is incident to a vertex. Figure 7 shows the concept, the normal a is weighted by the angle α when added to the vertex normal for the center vertex. This algorithm was proposed by Thürmer and Wütrich in 1998. [Thürmer et al, 1998] Another idea first proposed by Max in 1999 [Max, 1999] is to weight each face normal with the area of the face.

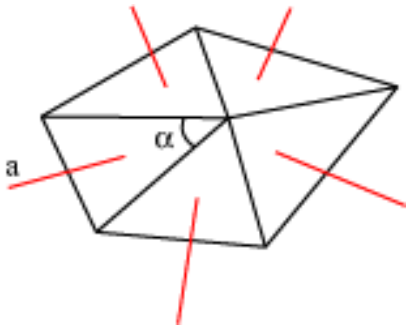


Figure 7: Weighting by angle

Both the weight by area and the weight by angle methods works well for smooth objects but when it comes to objects with edges there will be ugly artefacts.

A problem when generating vertex normals is that not all objects in reality are smooth. In particular it is hard to preserve edges when using Phong or Gouraud shading. For an edge to look good it needs two normals, one for each side of the edge and the normal models does not take this into account.

When modeling new meshes the designer is usually given the option of making an edge hard. In practice this means duplicating the vertices on the edge and using one vertex normal per vertex. This makes it possible for the surface to have two vertex normals at the same position which can account for the discontinuity of the edge.

If the vertex normals are generated we need a way of detecting edges and then splitting the vertices accordingly. One way of accomplishing this is the edge preserving method proposed by Nate Robins [Robins, 1997]. It compares the angles between adjacent faces and splits the shared vertices

if the angle is larger than some given crease angle. The vertices in these points will use the face normal instead giving a sharp edge. The method is not perfect. It will still miss some edges and split some vertices that are not on edges. But it gives a better result for objects with edges than a purely smoothing algorithm would.

2.2.9 Vertex Shaders

A vertex shader is a small compiled program that is executed once per vertex and frame. Such a program has to be loaded onto the graphics card itself and has a limited number of inputs and outputs. The vertex shader is a replacement for the transformation and lighting parts of a standard graphics pipeline and is thus executed in world coordinate space. The usual inputs to a vertex shader are position, the vertex normal, some often user specified uniform variables that are the same for all vertices and vertex attributes which are usually arrays of data with one value per vertex.

A vertex shader is supposed to output the transformed homogeneous vertex position. It can also pass other variables to the fragment shader. All the output variables will be interpolated across the polygon which means that the fragment shader will be executed a lot more than the vertex shader.

Vertex shaders are often used to manipulate the mesh or the normals in some way, but also to do preparation calculations for the fragment shader.

2.2.10 Fragment Shaders

Fragment shaders are executed in a later stage in the rendering pipeline. After passing the vertex shader vertices go through culling, clipping, hidden surface removal until they reach the fragment shader.

It is the job of the fragment shader to calculate the color of a fragment. It gets input from uniform variables just like the vertex shader and it also gets varying attributes from the vertex shader. While the vertex shader is required to output a position the fragment shader is allowed to discard the fragment. If not it should output a color and an alpha value. Using the colors of the different fragment together with the alpha values and the z-buffer pixels are created by the fixed pipeline in the next step after the fragment shader is executed.

2.3 Interpolation

When dealing with discrete data it is quite common to find that you need the value in between two samples. Interpolation is the art of estimating that value. This chapter lists and describes some of the most common interpolation techniques. They are described for one dimension but can easily be applied for two or more dimensions as well.

2.3.1 Nearest neighbor

This method is quite simple, as the name implies you just pick the nearest neighbor. If two samples are equally close either one will do. This method is very fast but not very accurate.

2.3.2 Linear interpolation

In linear interpolation the value to be estimated is a blend of the two closest points. Each of the points are weighed with how close they are to the new point. The concept can be understood by examining figure 8. Given the points (x_0, y_0) and (x_1, y_1) we need to find the value y for any given x .

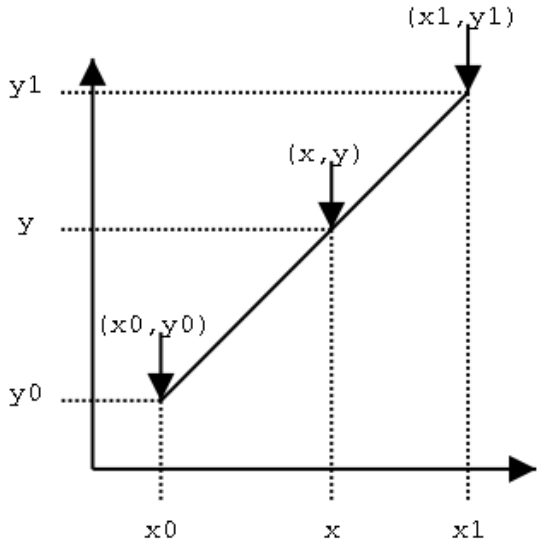


Figure 8: Linear interpolation [Wikipedia]

The following equation describes the relationship.

$$\frac{(y-y_0)}{(y_1-y_0)}=\frac{(x-x_0)}{(x_1-x_0)}$$

We introduce the parameter α to describe as the interpolation coefficient. It describes the proportion of the distance from x_0 to x_1 you have traversed when you encounter x .

$$\alpha=\frac{(x-x_0)}{(x_1-x_0)}$$

Since x is known we can calculate α and insert it into the first equation to get the expression of y shown in the equation below. [Wikipedia]

$$y=(1-\alpha)y_0+\alpha y_1$$

Linear interpolation is slower than nearest neighbor interpolation but still quite fast and sufficiently accurate for most implementations.

2.3.3 Cubic spline interpolation

Spline interpolation is a more complex interpolation method which uses piecewise polynomials also known as splines. There are a few different methods in this family of interpolation algorithms. In the case of cubic spline interpolation cubic polynomials are used. This method often produces very good results but is a lot more complex than linear interpolation.

3 The Method

This chapter describes how the different problems of the project where attacked and solved and the tools that were used.

3.1 Spherical Displacement

The detection distance data was previously presented as a colored sphere, or sectors of a sphere. The key idea of the project is very simple: Displace each value from a center point in the normal direction of the sphere with a distance proportional to that value instead of just showing it as a colored sphere. The intention of this was to make it easier to detect small variations in the data.

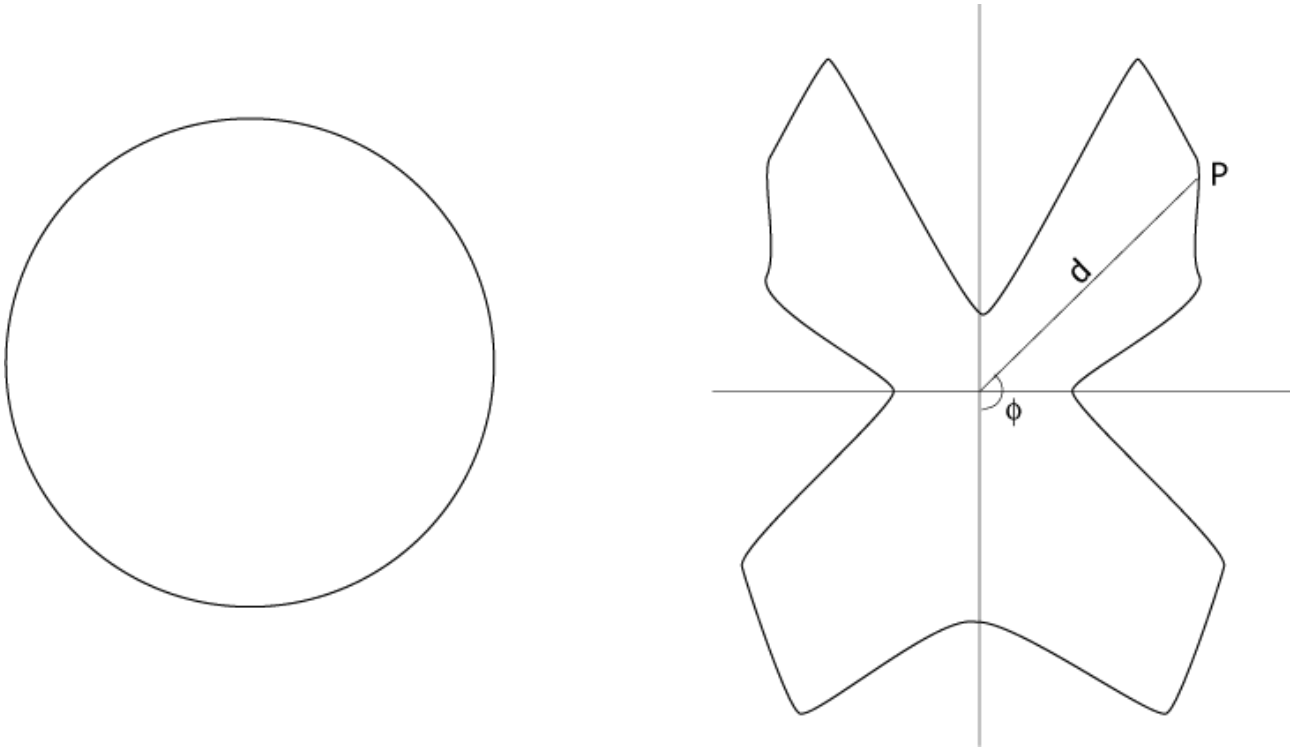


Figure 9: Spherical displacement in 2D

Since the values represent distances in some way, spherical displacement was also considered to be a more intuitive way of visualizing the data. Using spherical displacement will result in a visualization of the actual detection volume. To understand the concept, imagine a vehicle inside a transparent object. The distance from the vehicle to any point on the surface of the transparent object describes the detection distance for that angle. Figure 9 shows the concept in 2D. Each point P is displaced a distance d in the normal direction of a sphere or a circle in the 2D case. The distance d is the detection distance and the angle ϕ describes from what direction the vehicle has this detection distance. In this 2D example the vehicle is seen from above with its nose pointing upward. This is usually the case for the 3D visualizations as well but an elevation angle is also introduced to give a full sphere of angles.

3.2 Scaling

Since the range of the data is so high it is often hard to see small variations. One way to counter this was the spherical displacement. A method to further increase the visibility of small variations was

the use of different scaling options. Before displacing the values they can be scaled, for instance, using logarithmic scaling. The use of a logarithmic scale presents a problem in the visualization. Values between zero and one suddenly becomes negative and while this would be possible to visualize if values are displaced from a flat surface it becomes completely confusing if values are displaced spherically from a point. The negative values then end up on the opposite side of the sphere. One solution to this is to add the inverse minimum value to all values. This pushes everything outwards and creates a shape that is less confusing. It may be misleading that the center point of the visualization is not zero but an arbitrary negative number. The method also falls apart if the RCS is exactly zero in some point since that would displace everything infinitely.

Another solution is to take the fourth root of the value. This is, somewhat simplified, what the preprocessor module already does to convert RCS values to detection distances. The detection distances also depend on other variables than the RCS, such as the sensor and dampening, but is essentially proportional to the fourth root of the RCS. This means that “fourth root scaling” is rather useful for visualization purposes. This method doesn't have the problem with negative values and is more intuitive.

3.3 Visualizing Specification Data

One of the key problems for the military industry is to show that they have met the specifications for the project. This is already hard enough since the measurements are very expensive and the values are often simulated. The specifications are usually in the form of different maximum values allowed for certain sectors. The idea here is to visualize these specifications much like a normal dataset, with spherical displacement. If the specification data is shown like this but with transparency and no color information it is possible to show a specification set and the corresponding detection distance or RCS data simultaneously. Where the detection distances show outside the specification set the detection distance doesn't meet the requirements. This would be very much easier, faster and more intuitive than searching the data sets for too high values by hand.

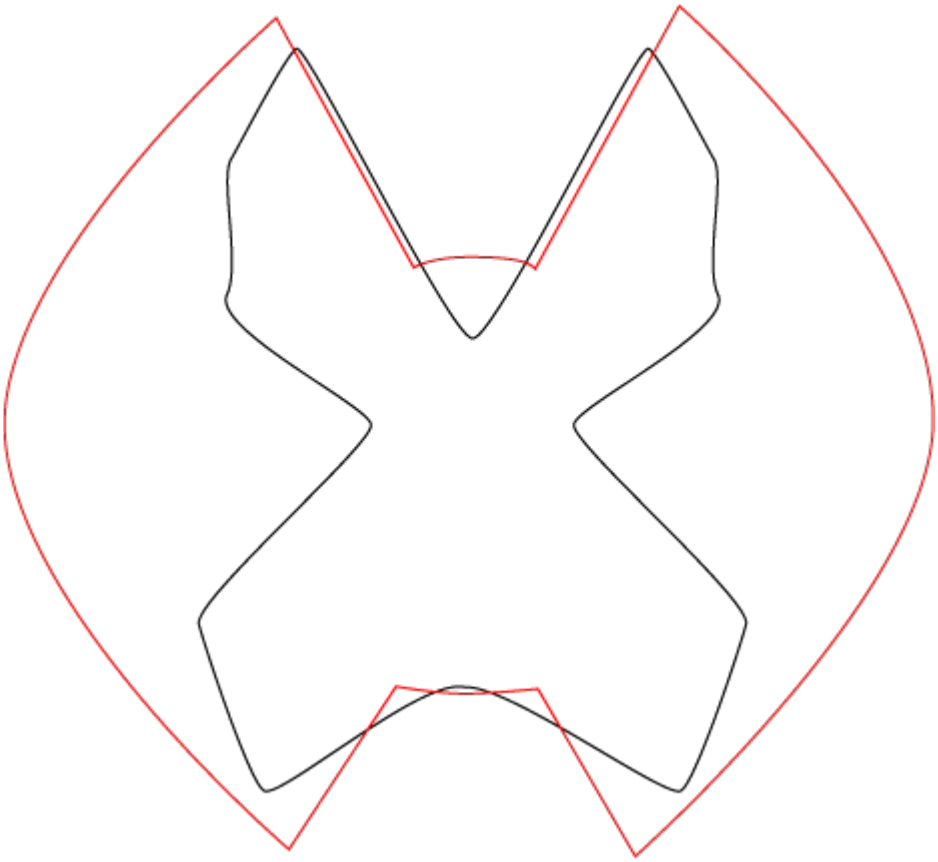


Figure 10: 2D specification data

Figure 10 shows how it could look in 2D. The black shape is the detection distance set and the red set is the specifications. Wherever the black set is outside the red the requirements have not been met.

Currently the specifications are usually rather simple. A sector is indicated with start and end angles in azimuth and elevation and then an RCS value is specified for that sector. Often different RCS values are specified for different frequency bands. In reality such specifications can usually not be met. When creating stealth vehicles the main tool is shaping the vehicle so that the radar reflection is concentrated in a few directions. While there are other methods of reducing the radar signature of a vehicle, such as radar absorbing materials (RAMs) the major contribution comes from shaping. Therefore a good stealth vehicle will have heavy spikes in RCS in certain directions. The application is meant to help the creators of the vehicle to prove to the client that the spikes cover a small sector and located in directions where sensors are unlikely to be located.

In order to do this it was decided to add modifications to the simple flat sectors mostly used. These modifications are meant to give greater weight to the center of the sectors and less to the corners.

The first idea was to create elliptical sectors. These would be useful when examining one sector at the time. Section 4.4.6 describe the implementation of the elliptical sectors. At a later stage in the project it was clear that this feature was unlikely to be used since it is just as easy to look at the whole data set at once rather than one sector at the time.

To still be able to support weighting by distance to the center of the sector a few options were added when creating these sectors in the application. These options allow the user to let the RCS or detection distance value decrease with the distance from the center.

The options that were implemented are shown below in table 1.

Sector Type	Value
Flat	min
Linear	$min+(max-min)\frac{(\lvert x\rvert+\lvert y\rvert)}{2}$
Elliptical	$min+(max-min)\frac{(\lvert x^p\rvert+\lvert y^p\rvert)}{2}$
Gaussian	$min+(max-min)e^{(\frac{1}{2}((x/0.4)^p+(y/0.4)^p))}$

Table 1: Specification sector modifiers

Where min is the minimum value of the sector, usually the original value. The maximum value is max and describes the highest value that the sector will take. The sector will take maximum value in the corners and the minimum value in the center. The p value controls how fast the values decreases. A high p value makes most of the sector use the minimum value and only the corners use the maximum while a lower value lets the maximum value control more of the sector. A normal value of p would be 3 or 4 for the elliptically decreasing sectors. The Gaussian sector creates a normal distribution curve over the sector. It is not possible to assert that the Gaussian method creates a sector with the maximum values in the corners and the minimum value in the center. The Gaussian sector should also have the standard deviation as an extra control parameter but it has been set to 0.4 for simplicity.

3.4 Generate high quality visualization

One of the key problems in the project was to generate good looking high quality visualizations of the data. Scientific visualizations often have a rather dull and functional look to them but since one of the intended uses for the application was producing images for marketing it was important to produce good looking results. To achieve this a decision was made to use smooth shading with the Phong model described in section 2.2.7. Using smooth shading also allowed for the use of good looking specular effects. For smooth shading to produce sufficiently good results high quality vertex normals had to be calculated. Since both CAD models and the generated meshes contain a lot of hard edges it was decided to use the smooth normal generation with edge preservation algorithm created by Nate Robins [Robins, 1997]. For an even better result this was coupled with the weighting by angle algorithm proposed by Thürmer and Wütrich in 1998. [Thürmer, 1998] These algorithms are described in section 2.2.8.

3.5 Tools, languages and maintainability

As described in section 1.3 SAAB wanted the application to be easily maintained. To make it easier to change the user interface and add new features it should be separated from the core of the program. A script language controlling the application and the user interface would be helpful for future programmers and would also add some structure to the application. Creating a user interface in a script language is also helpful since it does not require the programmer to compile the program every time a new feature is tested. The user interface and part of the rendering controls was programmed in the script language Python. It was chosen for several reasons. It is an open source project which is in line with the requirements of the project, it has a multitude of open source modules available and since it was used in the previous version of the application there were people able to give support on the language available at SAAB but the major reason was that SAAB

explicitly asked that the project continued to use python.

Several of the python add-in modules were also used in the project. For instance wxPython, which is a wrapper for wxWidgets that provides access to the platform specific user interface widgets for Windows, Apple and Linux platforms. Another module that was used is PyOpenGL, which provides access to most of the OpenGL functionality within Python. PIL or Python Imaging Library was used for saving captured screen shots as image files. The Numeric module was used for its excellent mathematical operations on matrices, which was especially useful in the preprocessor module of the application. PyTables was used as an interface to the HDF5 hierarchical file format that was used in the input files as well as the intermediate files.

The base functionality and rendering facilities of the application created for this thesis was built in C++, using OpenGL as the interface to the graphics hardware. The Visualization Toolkit (VTK) was used in the early stages of the project but was eventually replaced by the C++/OpenGL module to add more control and better performance to the application. While VTK is a great tool for quickly creating scientific visualizations some of the specific features that was needed in this project were not available in VTK.

The OpenGL Extension Wrangler Library (GLEW) was used in the C++ module to simplify access to a few of the newer OpenGL extensions. All of the tools and languages used are Open Source and more or less platform independent. Even though the application was designed with Windows in mind it could easily be ported to another platform.

4 Implementation

This chapter goes into detail on the application design and visualization solution that was the result of the project. It contains program code comments and a discussion about the rendering techniques that was chosen.

4.1 Application Design

The basic idea of the application design was to do all the performance sensitive calculations and rendering in C++ and OpenGL and use a highly flexible script language and a good widgets library for the user interface and application control. This allowed for using advanced rendering techniques to speed up the application. Using a script language also makes it easier to change the application later without a need to understand C++ or OpenGL.

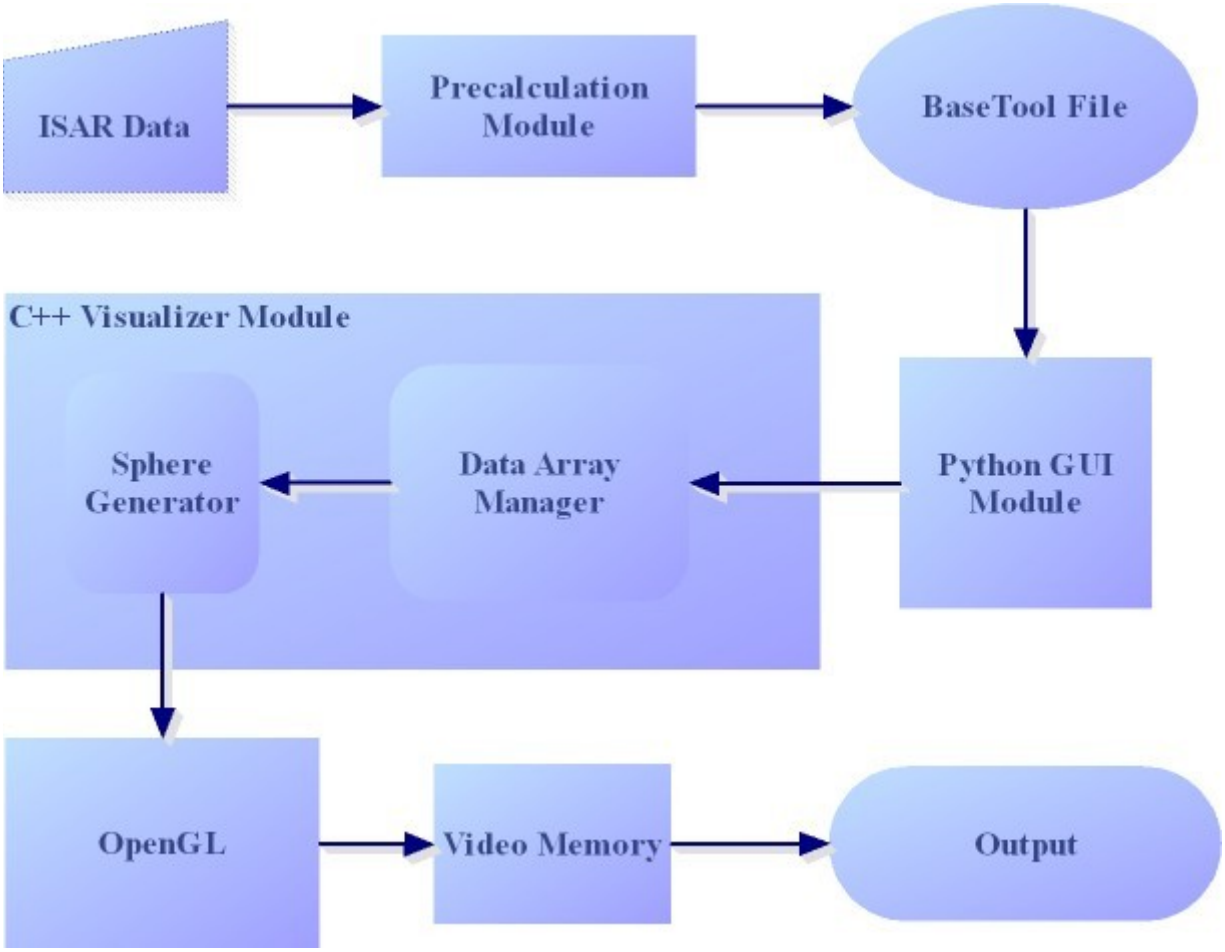


Figure 11: Data flow

Figure 11 shows the data flow for the actual test data in the application. The ISAR data is read from an input file and processed by the preprocessor module. This input data contains values per angle and frequency with the phase information intact meaning that each angle and frequency is represented by a complex number. Before detection distance values can be estimated the data radar cross section values must be calculated for all available angles. This is done by averaging the frequency data and calculating the magnitude of the complex numbers. After this processing we have one value per angle which is the RCS.

The data is read from the intermediate BaseTool file via pyTables to an array in python. Python then sends it into the data array manager class in the C++ module. The user can now choose what parts of the data to create a mesh from, the resolution of the mesh and what kind of mesh to create. The data array manager resamples the data into the new resolution and sends it to the sphere generator which creates the mesh. Using vertex buffer objects the sphere generator saves the mesh onto the graphics card via OpenGL and during rendering the data is read directly from the video memory.

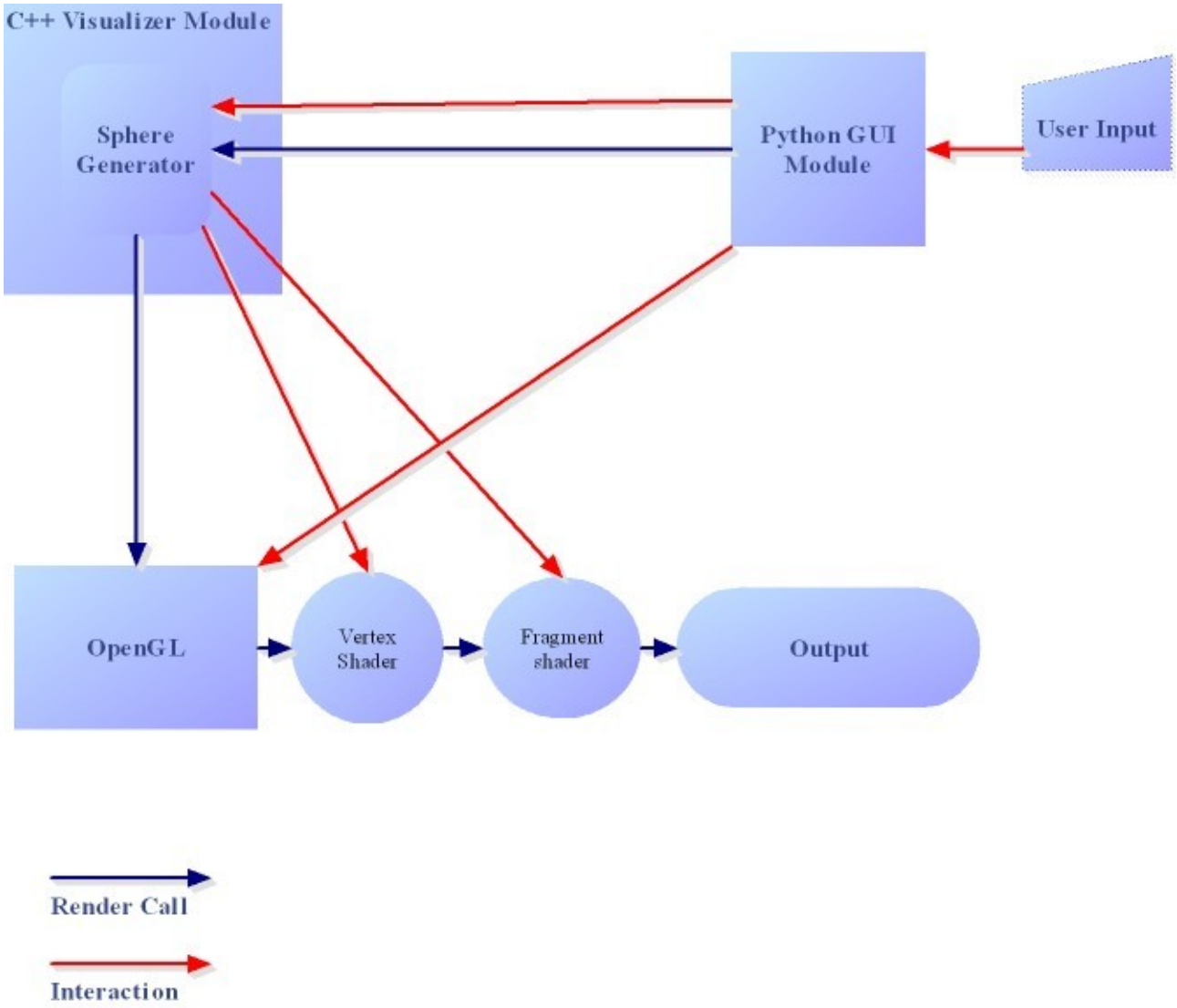


Figure 12: Rendering

Figure 12 shows the information flow in the rendering process. The rendering is controlled by the python module. Most of the instructions to OpenGL such as navigation are sent directly from the python module via pyOpenGL. The call to render the mesh itself is sent via the sphere generator class in the C++ module. The sphere generator asks OpenGL to read from the video memory where the data was previously stored. OpenGL then sends the data through the vertex and fragment shaders. Interactive parameters to the shaders are also sent via the sphere generator. These parameters control the transparency, color, specularity and scale of the mesh.

When designing the user interface the main objectives were to make it flexible and easy to use. These are often two conflicting interests since flexibility often also creates complexity. The script language python and widget library wxPython was used to make the creation of the user interface

easier and faster.

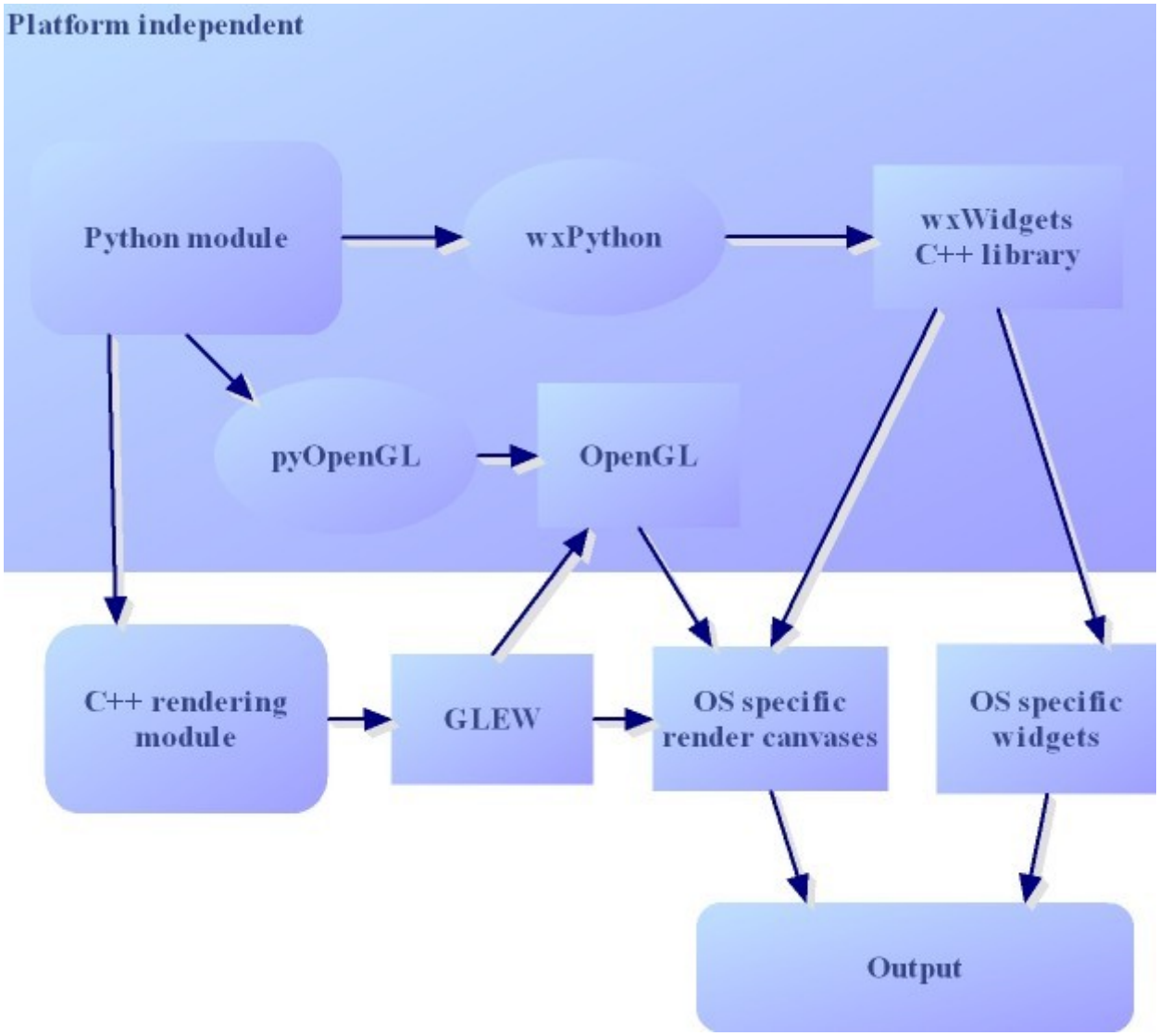


Figure 13: User interface and tool interaction

Figure 13 shows how the different tools used in the user interface interact as well as how the OpenGL calls are handled. The normal interface widgets such as buttons, sliders, menus, windows, dialogs, et cetera are created through wxPython. The python module of the application contains wxPython instructions that describes how the application should look. These instructions are sent through wxPython, which is a thin wrapper around the much more complex wxWidgets C++ library. WxWidgets contains functionality to ask the OS for its specific widgets to use. This gives the application the look of a Windows XP application.

To be able to show OpenGL content you need to create an OpenGL canvas to render on. In most labs and projects on university level this is done with the OpenGL Utility Toolkit (GLUT). While GLUT is simple to use it does not provide much control over the canvas it creates. The canvas is coupled with a window so to use this the user interface would either have to be in a separate window or programmed in OpenGL as well. None of these were an option in this project but there are other ways to create an OpenGL canvas. WxWidgets and also wxPython has limited support for this. Again what wxWidgets actually does is utilize the OS specific functions to create such a

canvas. In the case of Windows it provides a simplified interface to Windows OpenGL (WGL).

A nice feature of OpenGL is that once a canvas has been created all OpenGL calls will be directed to that canvas, or if there are several canvases they will be directed to the currently active canvas. The application uses 2-3 canvases, all created by wxWidgets. The main render function is located in the python module and is executed whenever wxWidgets is idle, which means that the controls remain responsive. The render function uses a mix of pyOpenGL calls and calls to the C++ rendering module. PyOpenGL is a thin wrapper for OpenGL much like wxPython is a wrapper for wxWidgets. Some of the graphic calls uses extensions to OpenGL. These are made available by the OpenGL Extension Wrangler Library (GLEW). For simplicity all OpenGL calls from the C++ module also go through GLEW. Before making a call to a function in the C++ module all that needs to be done is to set the correct canvas in python since the python module controls the canvases.

As shown in figure 13 most of the application is actually platform independent. To make it possible to run the application on another platform than Windows it would be necessary to change and recompile the C++ module and replace the current GLEW version with the correct version for the new platform.

4.2 Rendering Techniques

The data sets visualized by the BaseTool applications are quite large and complex. Because of this the objects that are rendered tend to be complex as well, often containing several million triangles. The graphic hardware of today can handle these triangles, but only if the right rendering techniques are used and the most common bottlenecks are avoided. To achieve the high performance that the application currently produces the rendering engine was reworked several times.

The most significant increase in performance came from the use of vertex buffer objects (VBO). The vertex buffer objects used to be an extension of OpenGL but are part of the core language since version 2.0. They allow saving vertex and index arrays in video memory rather than system memory. Instead of copying the data from system memory to video memory every frame it can simply be read from the video memory directly. Since the graphics card reads data a lot faster from the video memory than the system memory this is a very helpful tool, especially for large meshes.

The drawback of using vertex buffer objects is that the data cannot easily be changed dynamically since it is locked in place in the video memory but that was not a problem in the case of BaseTool.

Vertex and fragment shaders were used to allow some dynamic control of the objects and to gain full control of the rendering pipeline. Through the shaders the user can control transparency, scale, color and specularity of the objects.

Vertex normal calculation was implemented to allow specular shading. The algorithm that was used also allows for some degree of edge preservation and was based on work by Nate Robins [Robins, 1997]. The reason for using such a complex method is that the generated meshes often contain a lot of edges. Other methods were tried but didn't produce acceptable results. Another reason was that the CAD models did not contain vertex normals. Usually hard edges are indicated by hand when converting from CAD format to a mesh but that wasn't an option in this project. The idea behind the algorithm is described more in detail in section 2.2.8

Multiple render windows with shared context was implemented to improve performance and save video memory. This means that rather than creating several copies of a mesh or other render data the information is shared between the different windows or canvases. Limited support for this was available in wxPython and was made to work with GLEW with a few modifications.

4.4 Program Comments

4.4.1 Optimizations

One of the problems with the existing software was that the module that calculated the detection distances was very slow. Even though there was a significant amount of calculations to be done the module was still unreasonably slow. One of the first things that was done in the project was to analyze this module to find weaknesses in the design. The main problem was found to be that the module was written almost entirely in core python. The module does computations on large matrices and a lot of loops like the following were found.

```
if project.Background=='Free space':
    for ix in range(el):
        for jx in range(az):
            Detdist[ix,jx]=50.0*0.76*sqrt(sqrt(RCS[ix,jx]))
```

Since python is not a compiled language it does not take advantage of the multitude of optimizations available in modern compilers. To speed these loops up numerical python's matrix operations were used reducing the code to the following statement.

```
if self.background=='Free space':
    Detdist=20.0*sqrt(sqrt(RCS))
```

This is in fact also a loop but it is executed in numerical python's precompiled C++ module, which does take advantage of loop optimizations as well as other compiler optimizations. Several similar loops were replaced. The old preprocessor module also carried out a lot of rather slow and heavy graphical computations such as sphere generation, Delaunay triangulation and normal generation. Some of these were done in python and were slow because of that, others like Delaunay triangulation was done in VTK C++ modules but are still heavy. Sphere mesh and normal generation were moved to the new C++ visualization and some other features were removed altogether since they were no longer needed with the new implementation.

4.4.2 Interpolation

The ability to interpolate freely across the data was important to allow the user to change the resolution in phi and theta. In this way it is possible to control the complexity of the calculations. The method used for the interpolation was simple bilinear interpolation. A function that returns an interpolated value for any given phi and theta was created.

```
// Takes values in spherical coordinates and returns interpolated values
float CDataArray::GetInterpolatedValue(float theta, float phi)
{
      // calculate decimal indices
      float x = theta/m_pAllData->fThetaStep - m_pAllData->nThetaOffset;
      float y = phi/m_pAllData->fPhiStep - m_pAllData->nPhiOffset;

      // Calculate indices
      int i             = (int)x;
      int j             = (int)y;

      // Get the offset
      float dx          = fabsf(x - i);
      float dy          = fabsf(y - j);
```

```
	// recalculate indices for angles outside the boundaries of the array
	// really only works for phi
	int phiRes=(int)m_pAllData->nPhiRes; // if not done % will break
	if(j<0)
		j		= phiRes - abs(j%(phiRes-1));
	else
		j		= j%(phiRes-1);

	// clamp in case the interpolated value lies outside data boundaries
	if(i>m_pAllData->nThetaRes-1)
		return 0.0f;
	if(j>m_pAllData->nPhiRes-1)
		return 0.0f;
	if(i==m_pAllData->nThetaRes-1)
		return m_pAllData->ppData[i  ][j  ];
	if(j==m_pAllData->nPhiRes-1)
		return m_pAllData->ppData[i  ][j  ];

	// Bilinear interpolation
	return
		m_pAllData->ppData[i  ][j  ]*(1-dx)*(1-dy)	+
		m_pAllData->ppData[i+1][j  ]*(  dx)*(1-dy)	+
		m_pAllData->ppData[i  ][j+1]*(1-dx)*(  dy)	+
		m_pAllData->ppData[i+1][j+1]*(  dx)*(  dy);
}
```

4.4.3 Sphere Generator

All the meshes created by the program use the sphere generator class. It creates a sphere with arbitrary resolution, displaces the vertices according to the input data and triangulates the mesh. The following function adds a new vertex with correct displacement depending on the scaling setting provided by the application.

```
// Adds a vertex to the vertex array, also adds a vertex normal since they are
// used to calculate the position of the vertex anyway. If displacement by
// scalar is used the normals will need to be recalculated since the topology
// of the object will change.
void CSphere::AddVertex(float phi,float theta, Vec3f vOrigin, UINT nVertex)
{
	float fCurrPhi	=	phi*M_PI/180;
	float fCurrTheta	=	theta*M_PI/180;

	Vec3f normal;

	// Normals
	// x is the horizontal on the screen (left to right)
	// y is the vertical on the screen (bottom to top)
	// z points out of the screen (negative values far away, positive close)
	normal.x = sinf(fCurrPhi)*cosf(fCurrTheta);
	normal.y = sinf(fCurrTheta);
	normal.z = cosf(fCurrPhi)*cosf(fCurrTheta);
	m_pMesh->m_vVertexNormals.push_back(normal);

	Vec3f vert;
	if(m_bDisplaceByScalars)
	{
		// Vertices
		if(m_bLogScale)
		{
			float scale = m_pMesh->m_vVertexScalars[nVertex] =
```

```
                        logf(m_pMesh->m_vVertexScalars[nVertex])-logf(m_fRMin);

                  vert.x=vOrigin[0]+normal.x*scale;
                  vert.y=vOrigin[1]+normal.y*scale;
                  vert.z=vOrigin[2]+normal.z*scale;
            }
            else if(m_bFourthRootScale)
            {
                  float scale = m_pMesh->m_vVertexScalars[nVertex] =
                        powf(m_pMesh->m_vVertexScalars[nVertex],0.25f);

                  vert.x=vOrigin[0]+normal.x*scale;
                  vert.y=vOrigin[1]+normal.y*scale;
                  vert.z=vOrigin[2]+normal.z*scale;
            }
            else
            {
                  vert.x=vOrigin[0]+normal.x*m_pMesh->m_vVertexScalars[nVertex];
                  vert.y=vOrigin[1]+normal.y*m_pMesh->m_vVertexScalars[nVertex];
                  vert.z=vOrigin[2]+normal.z*m_pMesh->m_vVertexScalars[nVertex];
            }
      }
      else
      {
            vert.x=vOrigin[0]+normal.x;
            vert.y=vOrigin[1]+normal.y;
            vert.z=vOrigin[2]+normal.z;
      }

      m_pMesh->m_vVertices.push_back(vert);
}
```

Triangulating a sphere is very simple if you can afford not to have a closed surface. The vertex normal calculations in this application allows the surface to be open without any visible effects. Therefor the triangulation can be solved in a simple double loop.

```
      // Four corners of a quad
      UINT p1,p2,p3,p4;

      // Triangulation, counter clockwise ordering
      for(i=0;i<m_nThetaRes-1;i++)
      {
            for(UINT j=0;j<m_nPhiRes-1;j++)
            {
                  p1=i*m_nPhiRes+j;       // Bottom left
                  p2=p1+1;                // Bottom right
                  p3=p1+m_nPhiRes;        // Top left
                  p4=p3+1;                // Top right

                  m_pMesh->m_vIndices.push_back(p1);
                  m_pMesh->m_vIndices.push_back(p4);
                  m_pMesh->m_vIndices.push_back(p3);

                  m_pMesh->m_vIndices.push_back(p1);
                  m_pMesh->m_vIndices.push_back(p2);
                  m_pMesh->m_vIndices.push_back(p4);
            }
      }
```

4.4.4 Vertex Normals with Edge Preservation

The use of edge preservation is motivated in section 4.2 and the idea behind it is described in section 2.5.8. in The following code shows (somewhat simplified) how vertices can be split using this algorithm. Before this code is executed face normals and angles are calculated. The vector vVertexInformation contains links to all faces that the vertex belongs to as well as the corresponding angle. The algorithm calculates the dot product between normals of the first two faces in the list. If the value is larger than the cosine of the crease angle it splits the vertex. Otherwise the normals are added and weighted by their corresponding angles. When an edge is found a new vertex is created for all the following vertices. This may split an unnecessarily large amount of vertices but simplifies the algorithm somewhat.

```
for(i=0; i<m_nVertices;i++)
{
	if(!vVertexInformation[i].bIsUsed)
		continue;
	vector<UINT> faces   = vVertexInformation[i].vFaces;
	vector<float> angles = vVertexInformation[i].vAngles;
	Vec3f normal(0.0f,0.0f,0.0f);
	normal[0]+=vFaceNormals[faces[0]][0];
	normal[1]+=vFaceNormals[faces[0]][1];
	normal[2]+=vFaceNormals[faces[0]][2];
	bool bSplit = false;
	for(UINT j=1;j<faces.size();j++)
	{
		UINT face   = faces[j];
		NormalizeVec3(normal);
		float dot = Vec3fDot(vFaceNormals[face],normal);
		if(!bSplit && dot < cosf(m_fCreaseAngle))
			bSplit=true;
		if(bSplit)
		{
			UINT v1 = m_vIndices[face*3];
			UINT v2 = m_vIndices[face*3+1];
			UINT v3 = m_vIndices[face*3+2];
			UINT id=0;
			UINT vert;
			if(i==v1)
			{
				id=face*3;
				vert=v1;
			}
			else if(i==v2)
			{
				id=face*3+1;
				vert=v2;
			}
			else if(i==v3)
			{
				id=face*3+2;
				vert=v3;
			}
			if(id!=0)
			{
				m_vIndices[id]=nVertices;
				m_vVertexNormals.push_back(vFaceNormals[face]);
				m_vVertices.push_back(m_vVertices[vert]);
				nVertices++;
			}
```

```
            }
            else
            {
                  normal[0]+=vFaceNormals[face][0]*angles[j];
                  normal[1]+=vFaceNormals[face][1]*angles[j];
                  normal[2]+=vFaceNormals[face][2]*angles[j];
            }
      }
      NormalizeVec3(normal);
      m_vVertexNormals[i]=normal;
}
```

4.4.5 Specification Sectors

One new feature in the application was the ability to show specification sectors. The following code sample shows how the different types of sectors discussed in 3.2 was implemented.

```
for(int i=iStart;i<=iEnd;i++)
{
      for(int j=jStart;j<=jEnd;j++)
      {
            x = float((i-iStart)*2 - iRange)/float(iRange);
            y = float((j-jStart)*2 - jRange)/float(jRange);
            switch(mode){
            case SECTOR_MODE_FLAT:
                  value = fMin;
                  break;
            case SECTOR_MODE_LINEAR_THETA:
                  value = fMin+(fMax-fMin)*fabsf(x);
                  break;
            case SECTOR_MODE_LINEAR_PHI:
                  value = fMin+(fMax-fMin)*fabsf(y);
                  break;
            case SECTOR_MODE_LINEAR_BOTH:
                  value = fMin+(fMax-fMin)*((fabsf(x)+fabsf(y))/2.0f);
                  break;
            case SECTOR_MODE_SPHERICAL_THETA:
                  value = fMin+(fMax-fMin)*fabsf(powf(x,xPow));
                  break;
            case SECTOR_MODE_SPHERICAL_PHI:
                  value = fMin+(fMax-fMin)*fabsf(powf(y,yPow));
                  break;
            case SECTOR_MODE_SPHERICAL_BOTH:
                  value = fMin+(fMax - fMin)*((fabsf(powf(x,xPow)) +
                        fabsf(powf(y,yPow)))/2.0f);
                  break;
            case SECTOR_MODE_GAUSSIAN_PHI:
                  value = fMax - (fMax-fMin)* expf(
                        -0.5f*(fabsf(powf(y/yStd,yPow))));
                  break;
            case SECTOR_MODE_GAUSSIAN_THETA:
                  value = fMax - (fMax-fMin)*expf(
                         -0.5f*(fabsf(powf(x/xStd,xPow))));
                  break;
            case SECTOR_MODE_GAUSSIAN_BOTH:
                  value = fMax-(fMax-fMin)*expf(-.5f*(fabsf(powf(x/xStd,xPow)) +
                        fabsf(powf(y/yStd,yPow))));
                  break;
            }
      }
```

```
}
```

4.4.6 Elliptical Sectors

When analyzing a sector of the sphere data it can sometimes be interesting to disregard the corners of the sector. To help with this the application can create sectors of elliptical shape. These are created by sampling the data in a grid, ignoring points that lie outside the imagined ellipse, moving certain points to the edge of the ellipse and interpolating needs correct values for the new points. The following code sample finds the new position for points that are near the edge of the ellipse. It is based on the marching cubes algorithm [Lorensen and Cline, 1987] but is ported to the 2D case (marching squares) and somewhat simplified since ellipses are convex objects.

```
// Grid sampling of an ellipse. Moves points that are outside the edge but
// close to it to the perimiter itself. Does nothing with values that are
// inside the ellipse or way outside it.
// To find out if (x,y) is outside the ellipse but near the edge the function
// checks if the neighbours (x+xStep,y), (x-xStep,y), (x,y+yStep) and
// (x,y-yStep) are inside the ellipse.
bool CDataArray::FindNewPosition(float &x, float &y, float xStep, float yStep,
float a, float b)
{
      // If the coordinates are inside the ellipse - use them as they are
      if(IsInsideEllipse(x,y,a,b))
            return true;

      bool bLeftInside  = IsInsideEllipse(x-xStep,y,a,b);
      bool bRightInside = IsInsideEllipse(x+xStep,y,a,b);
      bool bAboveInside = IsInsideEllipse(x,y+yStep,a,b);
      bool bBelowInside = IsInsideEllipse(x,y-yStep,a,b);


      // squareIndex defines which of the four neighbours are inside the ellipse
      // basically sets the 4 bits with lowest significance
      // 0010 = 2 => bRightInside is true
      // 1001 = 9 => bLeftInside and bBelowInside are true
      // etc
      int squareIndex=0;
      if (bLeftInside   )     squareIndex |= 1;
      if (bRightInside) squareIndex |= 2;
      if (bAboveInside) squareIndex |= 4;
      if (bBelowInside) squareIndex |= 8;

      // Check if any neighbours are inside the ellipse
      // if so, use a point on the ellipse
      // uses the simple fact that
      //                x=a*cos(theta)
      //                y=b*sin(theta)
      // defines the ellipse
      switch(squareIndex)
      {
      case 0: // All neighbours outside
            return false;
      case 1:     // Left
            x     =     a*cosf(asinf(y/b));
            return true;
      case 2: // Right
            x     =   -a*cosf(asinf(y/b));
            return true;
```

```
	case 4: // Above
		y	=	-b*sinf(acosf(x/a));
		return true;
	case 8: // Below
		y	=	b*sinf(acosf(x/a));
		return true;
	case 5: // Left and above
		x	=	a*cosf(asinf(y/b));
		return true;
	case 6:	// Right and above
		x	=	-a*cosf(asinf(y/b));
		return true;
	case 9: // Left and below
		x	=	a*cosf(asinf(y/b));
		return true;
	case 10: // Right and below
		x	=	-a*cosf(asinf(y/b));
		return true;
	default: // The other cases only apply to concave objects and are ignored
		return false;
	}
}
```

4.4.7 Rendering with Vertex Buffer Objects in OpenGL

The vertex buffer objects technique allows the application to save data in the video memory. This provides a major speed increase as discussed in section 4.3. The following code sample shows how the data is saved to the video memory.

```
// Initialize the mesh for rendering, if vertex arrays are used, stores
// vertex arrays in video memory for fast access in rendering
void CSphere::RenderInit()
{

	// Store vertex array in video card memory
	glGenBuffers( 1, &m_nVBOVertexID );
	glBindBuffer( GL_ARRAY_BUFFER, m_nVBOVertexID );
	glBufferData( GL_ARRAY_BUFFER, m_pMesh->m_nVertices*3*sizeof(float),
			*m_pMesh->m_vVertices.begin(), GL_STATIC_DRAW );

	// Store vertex normal array in video card memory
	glGenBuffers( 1, &m_nVBONormalID );
	glBindBuffer( GL_ARRAY_BUFFER, m_nVBONormalID );
	glBufferData( GL_ARRAY_BUFFER, m_pMesh->m_nVertices*3*sizeof(float),
			*m_pMesh->m_vVertexNormals.begin(), GL_STATIC_DRAW );

	// Store vertex attribute array in video card memory
	glGenBuffers( 1, &m_nVBOAttribID );
	glBindBuffer( GL_ARRAY_BUFFER, m_nVBOAttribID );
	glBufferData( GL_ARRAY_BUFFER, m_pMesh->m_nVertices*sizeof(float),
			&m_pMesh->m_vVertexScalars[0], GL_STATIC_DRAW );

	// Store index array in video card memory
	glGenBuffers( 1, &m_nVBOIndexID );
	glBindBuffer( GL_ELEMENT_ARRAY_BUFFER, m_nVBOIndexID );
	glBufferData( GL_ELEMENT_ARRAY_BUFFER, m_pMesh->m_nIndices*sizeof(UINT),
			&m_pMesh->m_vIndices[0], GL_STATIC_DRAW );
}
```

To access the data during the actual rendering pass the following code can be used.

```
// This render method uses vertex arrays that have previously been uploaded
// to the video memory. Probably the fastest method available.
void CSphere::Render(){

      // Set vertex array from video memory
      glBindBuffer( GL_ARRAY_BUFFER, m_nVBOVertexID );
      glVertexPointer( 3, GL_FLOAT, 0, (char *) NULL );
      // Normal array
      glBindBuffer( GL_ARRAY_BUFFER, m_nVBONormalID );
      glNormalPointer( GL_FLOAT, 0, (char *) NULL );
      // Vertex attribute array
      glBindBuffer( GL_ARRAY_BUFFER, m_nVBOAttribID );
      glVertexAttribPointer(m_nAttribLocationID,1,GL_FLOAT,0,0,(char *) NULL );
      // Set index array from memory and draw
      glBindBuffer( GL_ELEMENT_ARRAY_BUFFER, m_nVBOIndexID );
      glDrawElements(GL_TRIANGLES, m_pMesh->m_nIndices, GL_UNSIGNED_INT,
                     (char *) NULL);
}
```

4.4.8 The vertex shader

The use of shaders allow for better control over the rendering pipeline. How they work and where they fit into the graphics pipeline is described in section 2.2.9. In this project the vertex shader is used to calculate the specular component of the light intensity. Some of the CAD models used in the project are does not have uniform orientation of the face normals, or rather, they have inconsistent ordering of the vertices in the triangles. This causes some triangles to show as zero intensity unless dealt with. The solution was to treat the object as a two-sided object. If the dot product between the light vector and the normal is negative, simply invert the normal. The vertex shader calculates the intensity according to the local reflection model described in section 2.5. Dynamic scaling is also done by the vertex shader.

```
    // Get the position
    vec4 pos            = gl_Vertex;

    // Dynamic scaling
    pos.x         = pos.x * fScale;
    pos.y         = pos.y * fScale;
    pos.z         = pos.z * fScale;

    // Pass the scalar vertex attribute to the fragment shader
    scalarFrag    = scalarVert;

    // Calculate transformed position, light vector and normal
    vec3 P        = vec3(gl_ModelViewMatrix * pos);
    vec3 L        = normalize((gl_LightSource[0].position.xyz)-P);
    vec3 N        = normalize(gl_NormalMatrix * gl_Normal);

    // Two-sided object, if dot product between normal and
    // light vector is negative, invert normal
    if(dot(N,L)<0.0)
    {
      N=-N;
    }
```

```
// Calculate eye vector and halfway vector
vec3 E        = normalize(-P)
vec3 H        = normalize(E+L);

// Calculate ambient, diffuse and specular intensities
// and pass them to the fragment shader
Iamb          = 0.0;
Idiff         = max(dot(N,L),0.0);
Ispec         = pow(max(dot(N,H),0.0),50.0);

// Save transformed position
gl_Position   = gl_ModelViewProjectionMatrix * pos;
```

4.4.9 The fragment shader

In this project it is the job of the fragment shader to map scalar values into colors. It does this by interpolating linearly between two out of three colors. A rate value between 0 and 1 is calculated from the scalar depending on the the minimum and maximum values of the scale. Rate values between 0 and 0.5 will result in a mix of the minimum color and the mean color and values between 0.5 and 1 gives a mix of the mean color and the maximum color. It also uses the intensity values calculated in the vertex shader and sets the alpha value dynamically from user input.

```
vec3 color;

float scalar = scalarFrag;
float rate = (scalar-scalarMin)/(scalarMax-scalarMin);
rate = clamp(rate,0.0,1.0);
if(rate<=0.5)
{
  rate=rate*2.0;
  color = mix(minColor,meanColor,rate);
}
else
{
  rate=(rate-0.5)*2.0;
  color = mix(meanColor,maxColor,rate);
}

gl_FragColor = vec4(color*(Idiff+Iamb)+Ispec*specColor,fAlpha);
```

5 Results

In this chapter the results of many different aspects of the project are presented. It contains images from the resulting application and comparisons with the old implementation. It also contains the results of the optimization tests.

5.1 Spherical displacement

One of the major improvements in the new version of BaseTool is the spherical displacement. With spherical displacement the actual detection volume can be shown which is considerably more intuitive for the end user than the colored sphere approach. The concept of spherical displacement is explained in section 3.1. Figure 14 shows the colored sphere approach of the old version while figure 15 shows the new spherical displacement. The old method of visualization is still available in the new version.

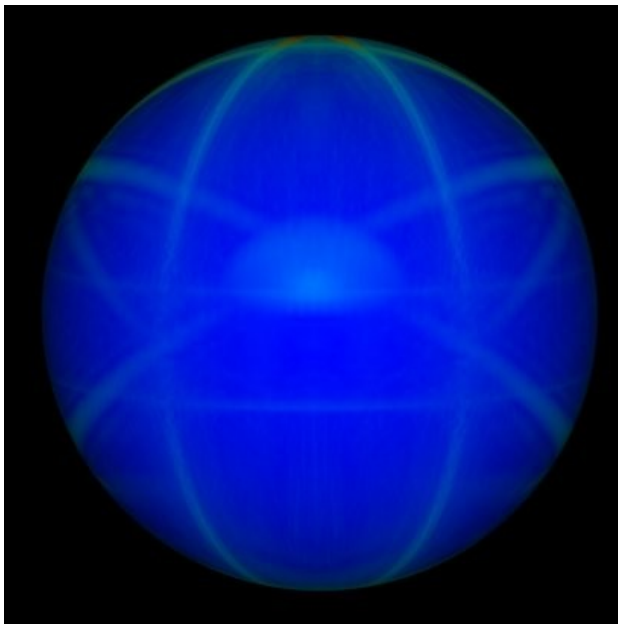


Figure 14: Colored sphere

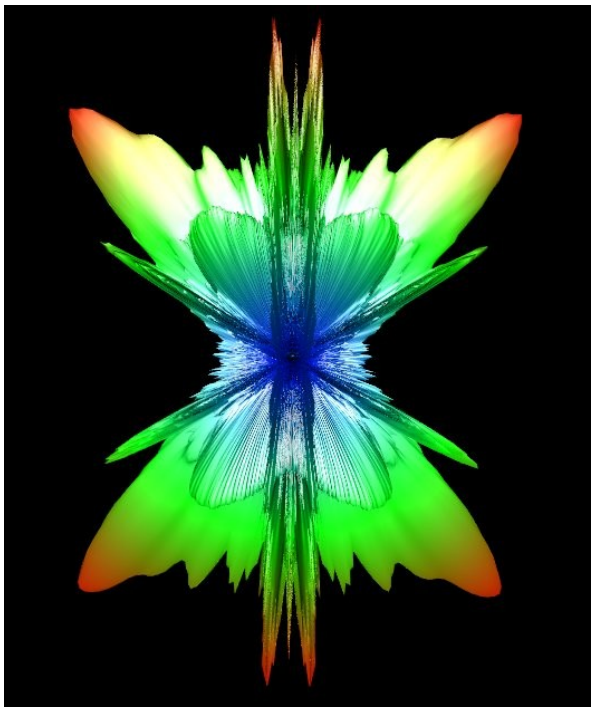


Figure 15: Spherical displacement

Both images show the same data but the spherically displaced data conveys it in a better way. It shows both detail and the general appearance at the same time and it is a lot easier to get an idea of the magnitude of the variations in the data. The color information is still available in the new version but it shows the same thing as the distances to the center. By looking at the colors and the minimum and maximum values of the object one can see what values the colors represent.

5.3 Specularity

The use of specular effects is mostly eye candy but also has some advantages when analyzing the data. While good looking visualization is a motive in itself the specular effects also help in enhancing details on the spherically displaced meshes. Figure 16 shows a spherically displaced detection distance mesh with specularity effects off and figure 17 shows the same mesh with

specularity on. Geometric detail is more visible in figure 17 while color information is somewhat destroyed.

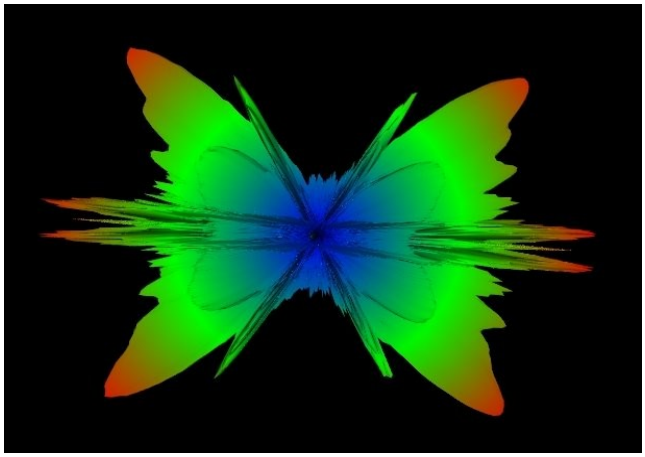


Figure 16: No specular effects

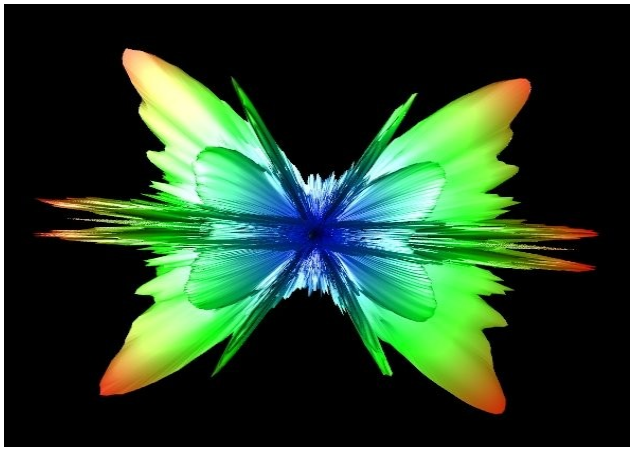


Figure 17: Full specular effects

Using specular effects on a colored sphere is not very useful. Since the light source is located in the same direction as the camera it only causes annoying glare coloring the sphere white in the position where the user is most likely to be looking. Figure 18 shows the specular glare corrupting the color information on a colored sphere.

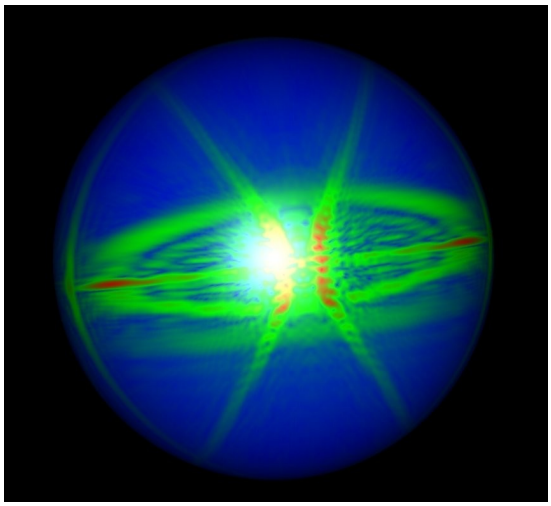


Figure 18: Specular effects on a colored sphere

Another example of an object that should use no or low specularity is the specification sets. These are often spherical or at least partly spherical and are usually used with transparency. It's easier to see through the object if the specularity is low. Because of this problem the user can control the specularity of all object by setting a value between 0 and 100 with a slider.

5.4 Downsampling data

The data sets that this application is meant for are very diverse. Some contain huge amounts of data and even though the application could handle showing several million triangles on the test computer there is a possibility that the application will be run on a less advanced machine. Being able to

change the resolution of the data before visualization can help ease the load on the hardware. Figure 19 shows the same data set with different resolutions. The top image contains 518 400 triangles and the bottom image contains 129 600. The difference is barely visible in these images but is a lot easier to detect in the application itself. Naturally some of the small details are lost but the main features seem to remain untouched. Increasing the resolution above that of the original data is also possible but probably has little use. The interpolation method used is bilinear interpolation which is described in theory in section 2.3 and the implementation in section 4.4.2. It is possible that a more advanced interpolation method such as bi-cubic spline interpolation would give better results but as the interpolation is a heavily used feature in the application this would have a big impact on the performance.

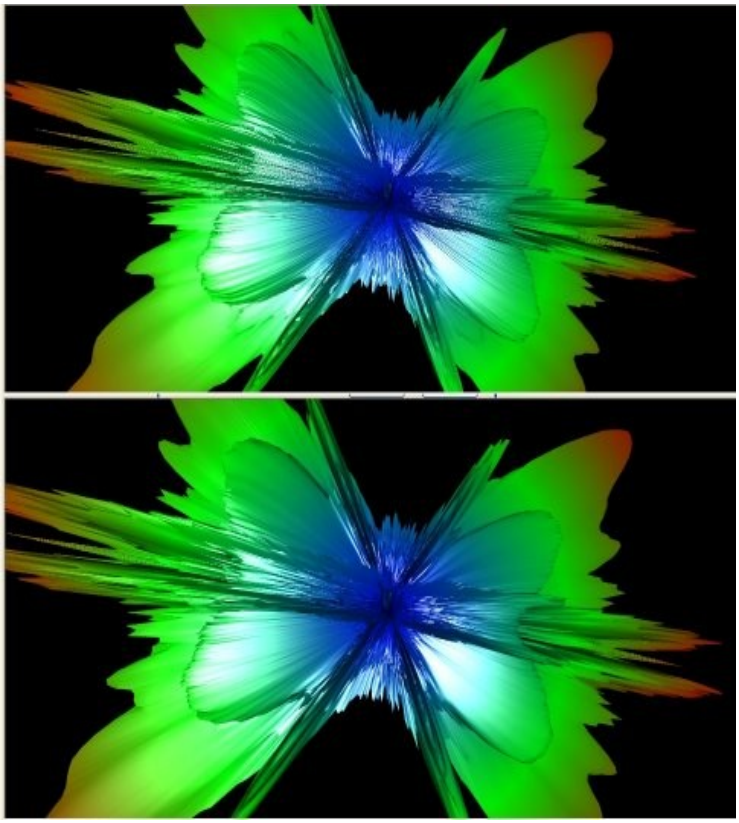


Figure 19: Downsampling comparison

5.5 Visualizing specification data

One of the major new features in this version of BaseTool is the ability to show specification data. The process of creating these specification data sets is described in the user manual in Appendix A. After such a set is created it is treated mostly like a normal data set. The main difference is that the default settings for color, transparency and specularity are different for specification sets. Figure 19 shows a fictional specification set with the default appearance settings and containing a detection distance set. It is very easy to see where the detection distances intersect with the specification set and thus fail to meet the specifications.

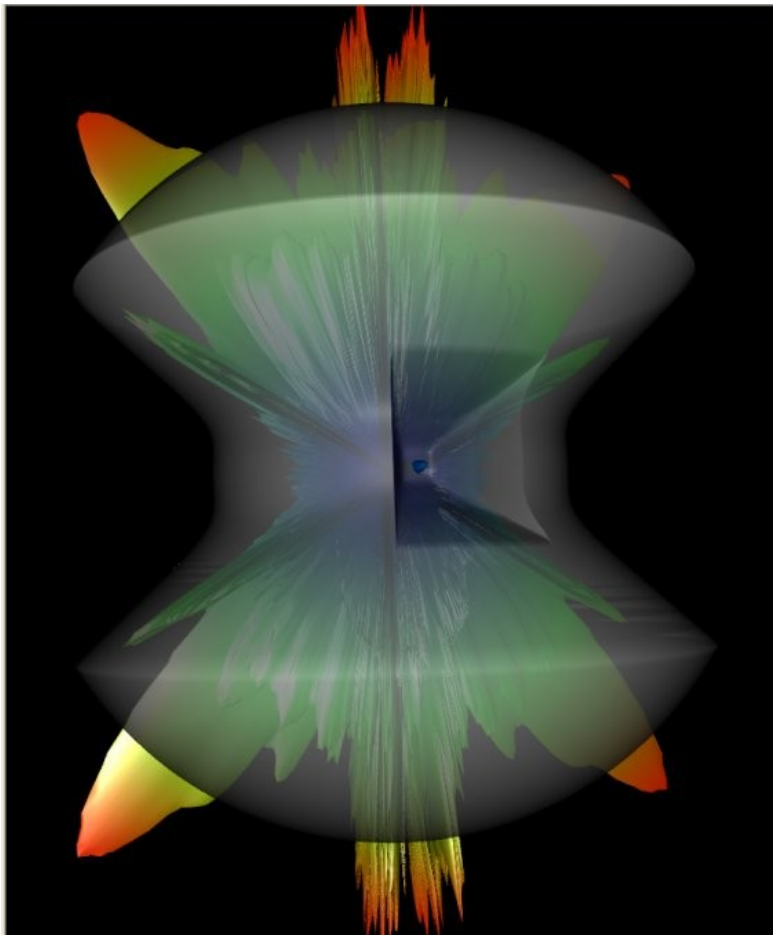


Figure 20: Specification sets

The specification set in figure 20 uses elliptically diminishing sectors as described in section 3.4. Most of the different sector settings are easy to use and produce good results but the Gaussian method is rather complex and doesn't always produce good results.

5.6 Optimizations

The optimizations described in section 4.4.1 were tested with different data sets to find the performance gain. The time to run the preprocessor module and get show the result on screen was tested and the times for the preprocessor and preparations to show the data was also noted. These tests were carried out on a 1.7 GHz Pentium 4 with 1 GB of RAM.

Test	Calculation time	Loading time	Total time
GR2 full, old implementation	1 hr 40 min 11 sec	45 sec	1 hr 40 min 56 sec
GR2 small, old implementation	1 min 58 sec	5 sec	2 min 3 sec
Thin metal plate, old implementation	39 sec	4 sec	44 sec
GR2 full, new implementation	22 sec	8 sec	30 sec
GR2 small, new implementation	< 1 sec	< 1 sec	~1 sec
Thin metal plate, new implementation	< 1 sec	< 1 sec	< 1 sec

Table 2: Optimization tests

Table 2 shows execution times for three different data sets with the old and new implementation. *GR2 full* is the data set shown on most of the images in the report. *GR2 small* is a narrow band of the full GR2 set and is about 50 times smaller. The *thin metal plate* set is a very small set which should be executed nearly instantly and therefore is good for testing how much overhead the two implementations have. It should be noted that the two implementations do not perform the exact same tasks. In particular the old implementation does Delaunay triangulation which isn't needed in the new version and the new version does a lot more sophisticated vertex normal calculations. For large data sets the new implementation is about 200 times faster than the old one in total. It is also about 5-6 times faster in just loading the precalculated data and showing it on screen even though it does significantly more advanced computations in this step.

5.7 Showing elliptical sectors

The idea behind using elliptical sectors and the implementation is discussed in section 4.4.6. In the end the usefulness of this feature in the context of radar visualizations could be questioned. But it was implemented and is still a part of the application. As can be seen in figure 21 the edge vertices are located on the edge of the ellipse itself making the shape look good. A few of the triangles break away from the general pattern. Interpolating the vertex positions in one more dimension would probably solve that problem.

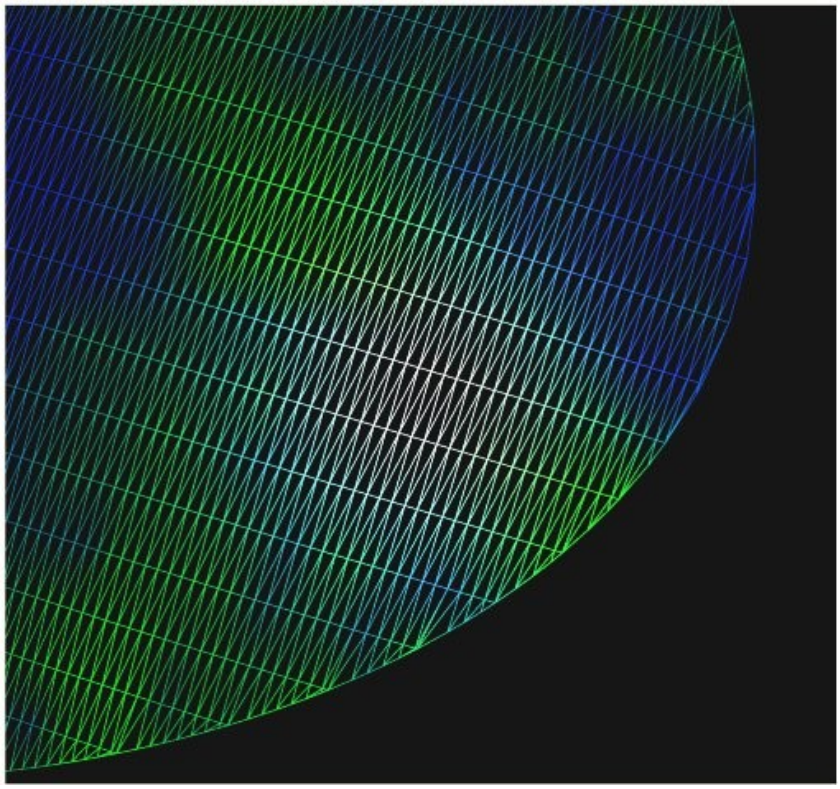


Figure 21: Elliptical sector in wireframe mode

5.8 User interface

As figure 22 shows the old version of the application was rather simple. Being simple it was also easy to use. The user just had to open a file and the CAD model and sphere was shown on screen. A problem with the old user interface was that much of the screen was wasted by showing the CAD on half the screen. A nice detail was the color bar on the right showing how to interpret the colors.

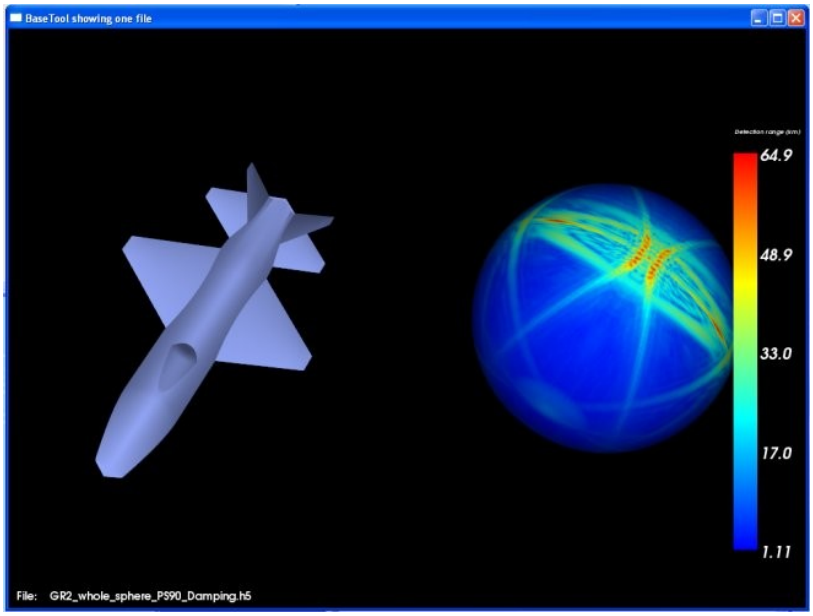


Figure 22: Visualization window of the old version

The few options that existed were accessed on a separate panel shown in figure 23. This approach was simple and sufficient for the features that the old version had.

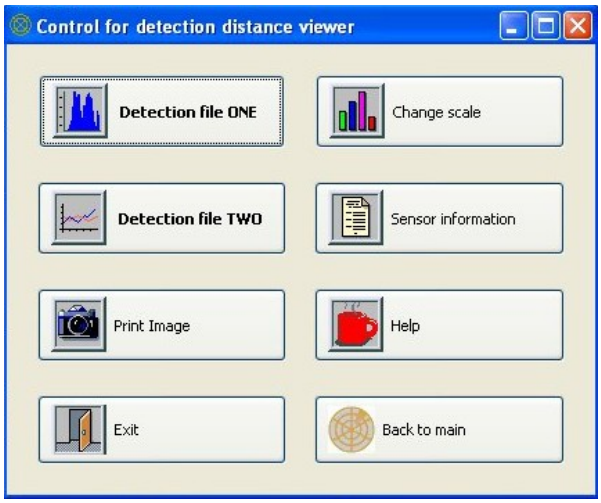


Figure 23: Menu of the old version

In the new version the CAD model is shown in the upper left corner instead, leaving much more room for the actual data visualization. All the new interactive options such as transparency, specularity, scale and color are located beneath the CAD model on the left side. The process of creating a mesh out of a data set is handled in a separate dialog. The new version contains a lot of new options and features but since there are default options for almost everything it is very easy to get started. The BaseTool user manual in Appendix A shows more details on how the program works and contains more screen shots. Figure 24 shows the main interface of the new application.

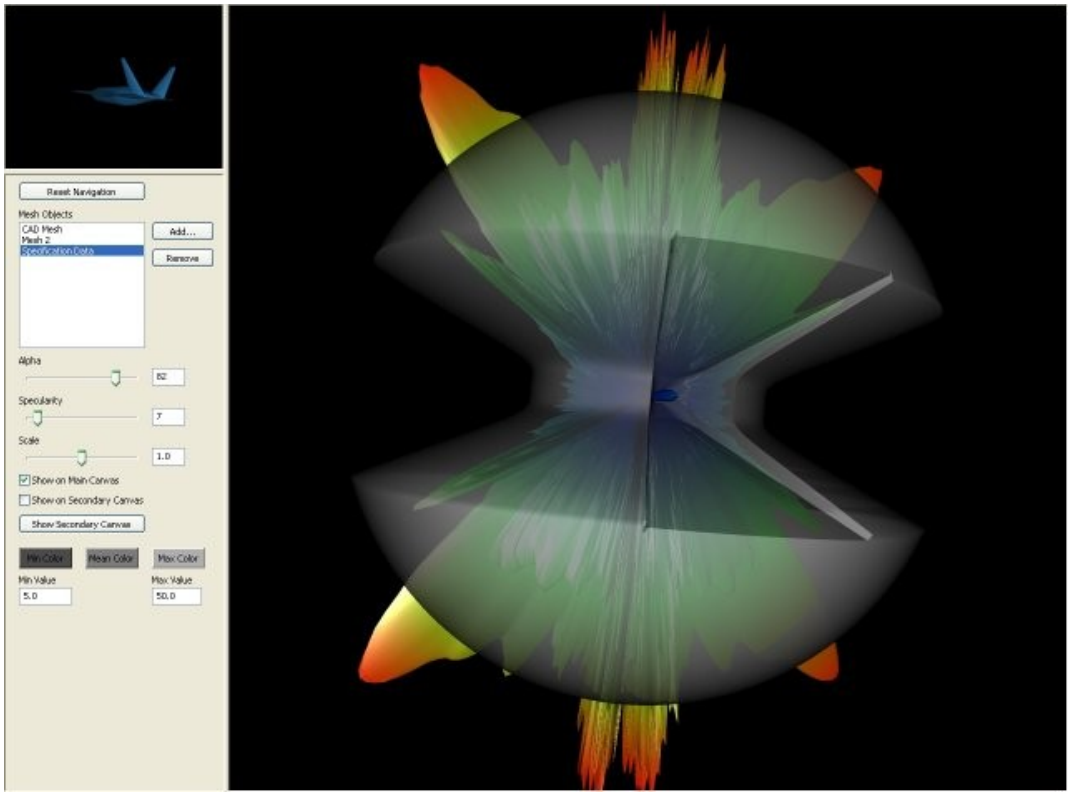


Figure 24: User interface of the new version

5.8 User manual

A simple user manual was created to help the end users getting started with the application. The manual contains step by step instructions for how to go from an input data set, via the preprocessor module, and finally displaying the visualization. Apart from the “getting started”-guide there is also more detailed information about the options and settings in the application. The manual is enclosed as appendix A.

6 Discussion and conclusions

This chapter discusses the results presented in chapter 5 and draws conclusions from these results. It also contains recommendations for future work and improvements.

6.1 Discussion

In creating an application such as BaseTool the usual approach is to ask the end users what they want from the program and then try to implement as much as possible of those wishes. In the BaseTool project the input from the end users was mixed. In terms of general characteristics the requirements were well defined. The end users wanted a good looking application that was fast, easy to work with and easily maintainable. When it came to actual features there was simply not enough input. This is a common scenario in software engineering. The application designer asks the end users what they want and the end users reply by asking what the application designer can do.

In this project it meant that some of the features that were asked for and implemented are of limited value for the user. With better interaction with the end users they could probably have been avoided.

How and if the application will be used remains to see. In its current form it is very much a showcase of different techniques to visualize radar signatures. The most likely uses are demonstrations and marketing rather than scientific analysis. It probably works well as a tool to aid in discussions about test data and simulated data.

6.2 Conclusions

The main idea with the project was to explore the idea of using spherical displacement rather than just colored spheres. The spherical displacement was implemented together with several other features resulting in the BaseTool application. The admittedly few that has tested the application claim that the spherical displacement is indeed more intuitive and easier to work with in comparison with the colored spheres. More testing would be required to properly assert this.

SAAB also wanted an optimization of the preprocessor code and a general improvement of loading times. Section 5.6 clearly shows a significant improvement with loading times up to 200 times faster in the new version of the application.

Visualization of specification data was also implemented. As seen in section 5.5 it is easy to find the points where the specifications are not met. In reality it is very likely that mostly the flat sectors will be used because of their simplicity. But the other methods could be used to prove a point if the application is used as a demonstration tool during a session discussing the specifications. The Gaussian sector is unlikely to be used at all since it is very complex and not all control parameters are available in the application. It was left in the application mostly for reference.

Achieving a high quality visual appearance was one of the aims of the project. The new implementation is undoubtedly better looking than the old and some of the images generated by the application could probably be used for marketing purposes.

Very little testing has been carried out on the user interface but comments have been positive from those who have tried to use the program and they have had no problem getting results with it. More testing is required before any further conclusion can be drawn about the user interface.

6.3 Future work and improvements

Time was a major limiting factor in the project. There are a lot of features that would add value to the application. The ability to save a visualization with all the objects and settings as some sort of project would save a lot of time for the users. Adding color bars and angle information would help the user understand and analyze the visualization. Color bars would act as a legend explaining the color information instead of the currently showed min and max values that are not very intuitive. Angle information in this case would be the ability to select a point on a surface and get a reading on what angle that point represents. Selecting a point would also make it possible to show the exact distance value in that point. Such information is crucial when using BaseTool in conjunction with software that analyze other radar data, such as ISAR maps. The screen capture function in the application should be improved so that it can produce images of arbitrary resolution. Such images could be used in marketing. Some or all of these features will be implemented during the summer.

Further testing of the user interface and the application features themselves should be carried out. Adjusting the user interface with the help of such tests should be fairly easy thanks to the use of python and wxWidgets.

Even though the bilinear interpolation currently implemented in the application seems to be sufficiently accurate it would be interesting to try using bi-cubic spline interpolation or some other more advanced interpolation technique instead.

References

[Knott et al, 1993] Eugene F. Knott, John F. Schaeffer, Michael T. Tuley. *Radar Cross Section*, Artech House, Inc., Norwood, MA, USA, 1993. ISBN 0-89006-618-3.

[Lynch, 2004] David Lynch, Jr. *Introduction to RF Stealth*, SciTech Publishing Inc. Raleigh, NC, USA, 2004. ISBN 1-891121-21-9

[Watt, 2000] Alan Watt, *3D Computer Graphics, Third Edition*, Pearson Education Limited, Essex, England, 2000. ISBN 0-201-39855-9

[Phong, 1975] Bui Tuong Phong, *Illumination for Computer Generated Images*, Communications of the ACM, Vol 18(6):311-317, June 1975.

[Gouraud, 1971] H. Gouraud, *Continuous shading of curved surfaces*, IEEE Transactions on Computers, 20(6):623–628, 1971.

[Robins, 1997] Nate Robins, *Smooth Normal Generation with Preservation of Edges*, http://www.xmission.com/~nate/smooth.html, Scientific Visualization Project at University of Utah

[Catmull, 1974] Edwin Catmull, *A Subdivision Algorithm for Computer Display of Curved Surfaces*, Ph.D. Thesis, Report UTEC-CSc-74-133, Computer Science Department, University of Utah, Salt Lake City, UT, 1974

[Thürmer et al, 1998] Grit Thürmer, Charles A. Wüthrich, *Computing Vertex Normals from Polygonal Facets*, Journal of Graphics Tools, 3(1), 1998, pps. 43-46.

[Max, 1999] Nelson Max, *Weights for Computing Vertex Normals from Facet Normals*, Journal of Graphics Tools, 4(2) 1999, pps. 1-6.

[Lorensen and Cline, 1987] William E. Lorensen, Harvey E. Cline, *Marching Cubes: A High Resolution 3D Surface Construction Algorithm*, Computer Graphics (Proceedings of SIGGRAPH '87), Vol. 21, No. 4, pp. 163-169.

[Wikipedia] *Linear interpolation,* http://en.wikipedia.org/wiki/Linear_interpolationBaseTool User Manual

Appendix A

Introduction

BaseTool is an application that offers the user an easy way to visually analyze and demonstrate magnitude of RCS or detection distance data.

Features

- Show RCS data as a colored sphere or a displaced sphere
- Show the corresponding CAD-models
- Rotate and zoom data freely
- Create and show specification sets and compare with RCS data
- Control transparency, scale, specularity and color of all objects
- 1-2 visualization windows that can both show several objects at once
- Small CAD-window to show the CAD-model at all times
- Save screen shots to .jpg, .bmp or .png files
- Resize all windows
- Fast loading times and high performance 3D visualization

Getting Started

Start by opening an ISAR data file and create a .base file with the precalculation functions. This will remove phase information and calculate an averaged RCS data set from the frequency data. Next load the .base file, you'll notice the CAD-model showing up in the upper left window (the CAD-window). Now press Add Mesh in the options panel. Depending on how you did the precalculation part one or more data sets will show up in the left list. Pick one and edit the settings or just click Create Mesh to start with the default settings. The dialog will close and your mesh will show up in the list in the Options Panel. Select the mesh from the list, this will allow you to control settings for that mesh.

To show the mesh in the Main Canvas, check the box for Show on Main Canvas. Alpha controls the transparency of the object, scale controls the size in relation to other objects (while zoom, scales everything in all windows) and specularity controls the direct light reflections. The object is color coded with a mix of three colors, the default is Red at maximum distance from the center, Blue at minimum distance and Green in the middle. These values can be changed per object. To show two screens press the Split Window button. This enables the secondary canvas and allows you to compare different data sets.

To create a specification set, click Add Mesh, then Add Specification Set. Set the base value to the maximum value the set will have. After creating the set, select it from the list and click Add Specification Sector. You can add several sectors to one set. When finished just click Create Mesh as with a normal mesh.

Menus

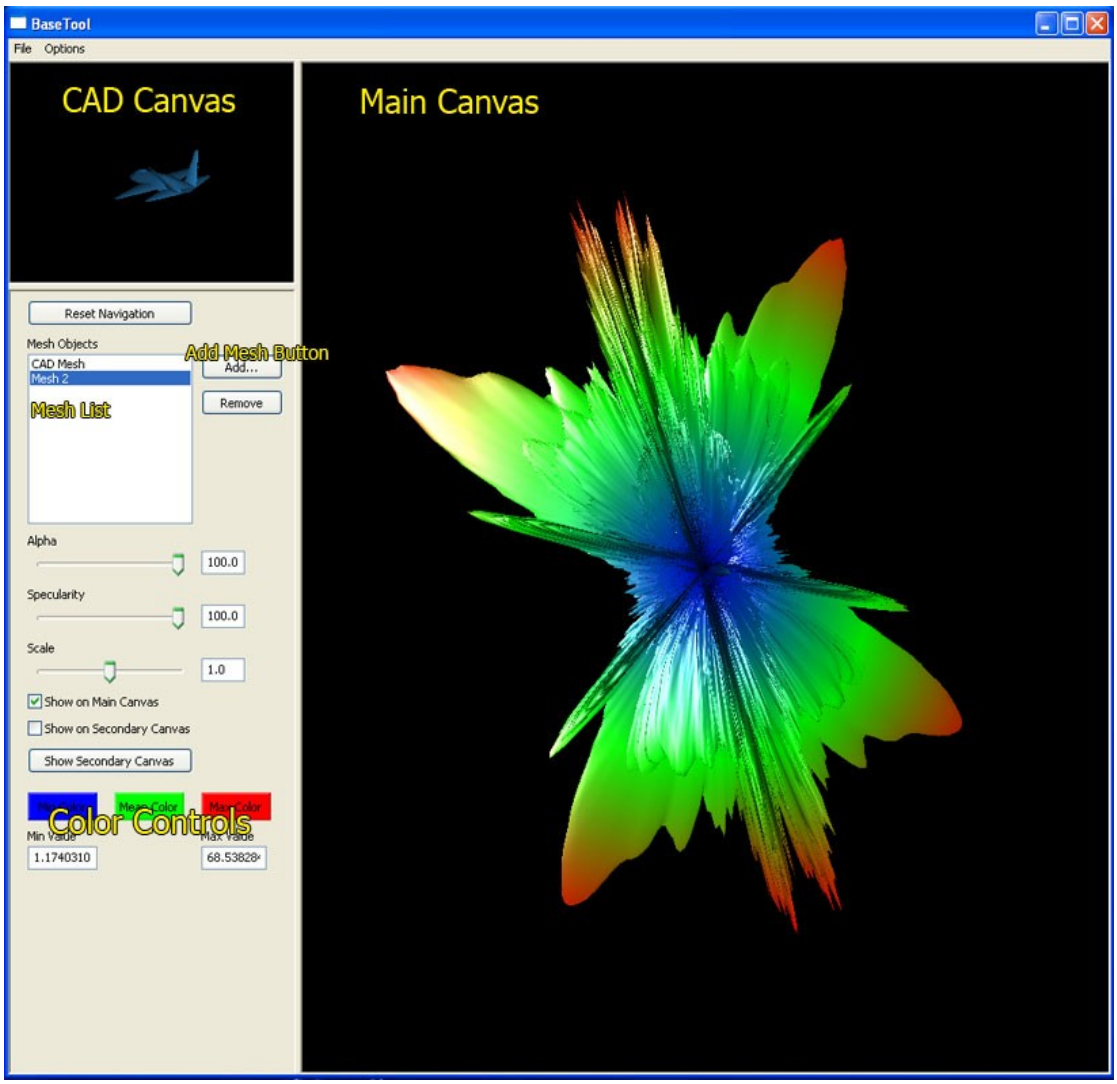
The menus of the application contain the following items.

- *File - Create BaseTool file from ISAR data.* This allows you to create a .base file for further use in BaseTool. The ISAR data needs to be in hdf5 format.
- *File - Open BaseTool file.* Opens a .base file and adds its data to the project.
- *File - Capture Image.* Creates a screen shot of the application and saves it as a jpeg, png or bmp image file.
- *File – Exit.* Exits the application.
- *Options – Wireframe.* Toggles wireframe rendering mode on and off.

Options Panel

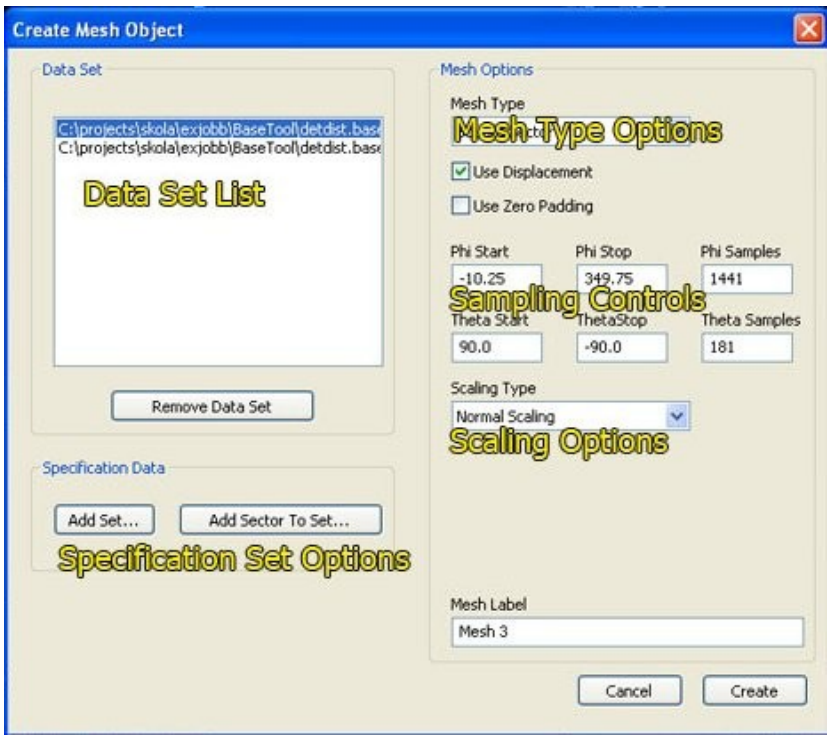
The *Options Panel* is located on the left hand side of the application and contains options for *Mesh Objects*. The *Mesh Objects List* shows all mesh objects. You can select a mesh object by clicking it in the *Mesh Objects List*. Options for the mesh will then activate in the *Options Panel.* The *Alpha Slider* controls the transparency of the current mesh. The *Specularity Slider* controls the shininess of the current mesh. The *Scale Slider* controls the relative scale of the current mesh. Check the *Main Canvas Checkbox* to show the mesh on the *Main Canvas. Check* the *Secondary Canvas Checkbox* to show the mesh on the *Secondary Canvas.* Click the *Show/Hide Secondary Canvas Button* to show or hide the *Secondary Canvas*.

The *Reset Navigation Button* resets all rotation and zooming in the canvases but leaves the mesh options intact. To change the color of a mesh click one of the color buttons. For CAD objects they will all change the color of the object. For other meshes the *Min Color* represents the color for the smallest detection distance or RCS values and the Mean and Max Colors represents the mean and max values in the same way. The colors for values in between are interpolated linearly. The *Min and Max Value Text boxes* show the smallest and highest detection distance or RCS values for the currently selected mesh. Changing these values will change the coloring of the object but not the actual geometry. For example, setting the *Min Value* to something larger than the first value will make a larger portion of the mesh use the *Min Color*.



Create Mesh Object Dialog

The *Create Mesh Object Dialog* is accessed with the *Add...* button on the *Options Panel.* It contains a *Data Set List* which shows all the data sets currently loaded. To create a mesh, first select a data set from the list, set all the options or use the default options, set a name for the mesh with the *Mesh Label Text box*, and then click the *Create Mesh Button.*

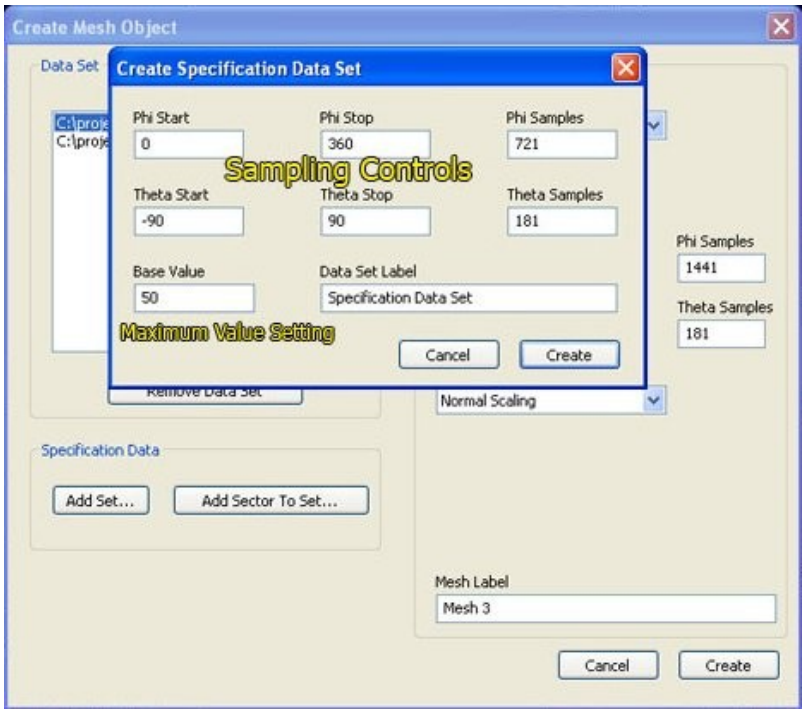


The *Mesh Type Choice Box* lets you chose between *Square Sector* and *Ellipse Sector*. The square sector cuts a square sector from phi start to phi stop and theta start to theta stop out of the data set. The ellipse sector does the same but with an elliptical cut, discarding data in the corners. Unchecking the *Use Displacement Check Box* will put the values for all angles at the same distance from the origin, forming a perfect sphere for a full data set. Checking it will displace each value proportional to itself in the direction of the normal, forming an object that is easier to analyze. The *Use Zero Padding Option* is useful if the data set contains a very limited number of theta angles. It will add two theta angels, one above and one below, with only zeros. This makes the object visible even if it's just one angle thick. Only use this option with data sets that are not full or if you limit the theta angles shown. It does nothing for phi angles since data sets are assumed to contain full range in phi for all theta angles that are present.

Phi Start, Phi Stop, Theta Start and Theta Stop control what angles are shown. *Phi Samples and Theta Samples* control the number of samples in phi and theta. If the number of samples are changed bilinear interpolation will be used to determine the values in the new sample points. The *Scaling Type Choice Box* lets you use normal, logarithmic or fourth root scaling. For logarithmic scaling log10 of the value is shown, the smallest negative logarithm is shown as zero and all the other values are translated outwards which makes it hard to see what the values actually mean in terms of RCS or detection distance. Fourth root scaling simply shows the fourth root of each value. Taking the fourth root of RCS values roughly gives values that are proportional to the detection distance. Using logarithmic or fourth root scaling for detection distance values probably doesn't make much sense.

Specification Set Dialog

To compare RCS or detection distance data to specifications you can manually create a data set with the *Add Set Button* in the *Create Mesh Object Dialog*. The S*ampling Controls* are similar to those of creating a new *Mesh Object*, except here you create a new data set instead. The *Base Value* is the maximum value that the specification set will take in any point. Later on you can modify the set by adding *Specification Sectors* but for overlapping sectors the smallest value will always be picked which is why the *Base Value* will never be exceeded for the set.

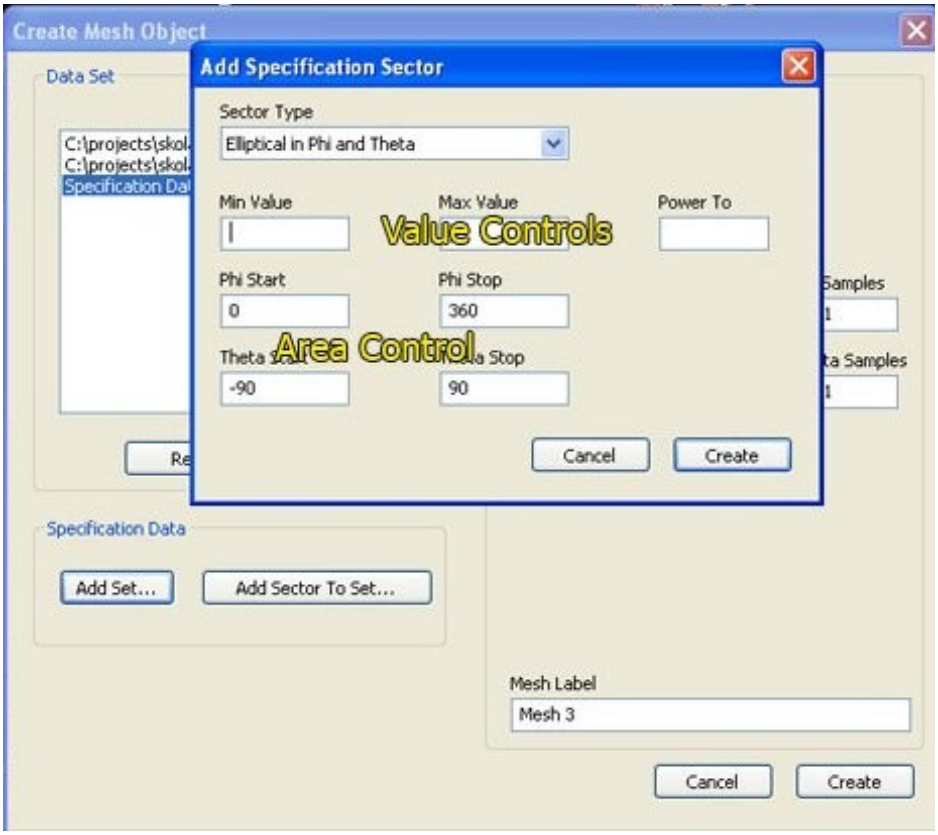


Specification Sector Dialog

To add more information to the specification set, select it from the *Data Set List* and click the *Add Sector To Set Button* in the *Create Mesh Object Dialog*. You can chose from several different types of sectors to add. The sector type controls how the value varies over the sector according to the following table.

Sector Type	Value
Flat	min
Linear	min + (max-min)*(|x| + |y|)/2
Elliptical	min + (max-min)*(|x^p| + |y^p|)/2
Gaussian	min + (max-min)*e^(1/2 * ((x/0.4)^p + (y/0.4)^p))

In practice this means that the *Flat Sector* mode gives a constant value across the sector. The *Linear* mode gives linearly diminishing values. For the *Elliptical* mode values will be close to the min value for most of the sector but near the max value in the corners. The p or P*ower To* parameter controls how fast the values diminish, a higher *Power To* means most of the sector will be close to the min value. The *Gaussian* mode is mostly for testing purposes and should not be used for any real projects. For each of the modes, except the *Flat* mode, you can chose to let the values diminish in phi only, theta only or both angles. The data set created like this can then be used in making a new *Mesh Object*.



Precalculation Dialog

The *Precalculation Dialog* is accessed via the *File Menu* item *Create BaseTool file from ISAR data*. To create a .base file, first select an input file with the *Browse Button*. Choose the wanted *Sensor* and *Background* options. The *Data Redundancy* option controls how the redundant data is handled. There are usually more than 360 degrees of data in the phi orientation so you can either average the redundant data or simply remove it. The *Data Type* controls the output data type, you can get raw magnitude of RCS data, detection distances or both. If you only want RCS data you don't need to care about the *Sensor* and *Background* options. The *Frequency Band* options are not yet implemented and cannot be changed.