Design and construction of applications for an interactive spherical display

S.W. Duineveld

July 19, 2013

Supervisor: R.G. Belleman
Signed: R.G. Belleman
Abstract

This research studies the development of an application optimised for spherical multi-touch displays. In particular, the development of an open-source prototype of such an application that can serve as a reference for future implementations is analysed. In order to do so, the requirements that any spherical multi-touch application should fulfil are drawn up, along with additional requirements that might be imposed on a spherical multi-touch application in certain use cases. In relation to these requirements, the choices that need to be made during the design of a spherical multi-touch application are discussed and design decisions are made to maximise the relevance of the prototype developed in this research. Thereafter, the implementation of a spherical multi-touch application is analysed using the prototype application as a starting point. Finally, a number of experiments are carried out to verify the compliance of the prototype application with the requirements.
## Contents

1 Introduction .................................................. 7
   1.1 Previous work .............................................. 7
   1.2 Contribution .............................................. 8

2 Design considerations ........................................ 11
   2.1 Mandatory requirements ................................... 11
   2.2 Additional requirements ................................... 11
   2.3 Application type ........................................... 12
      2.3.1 Marble ................................................... 12
   2.4 Extending the application to a spherical multi-touch display ......... 13
   2.5 Handling input ............................................. 13
   2.6 Generating output ......................................... 14

3 Implementation ............................................... 15
   3.1 Application structure ...................................... 15
   3.2 Coordinate systems ......................................... 16
      3.2.1 Normalised Mercator coordinates ...................... 16
      3.2.2 Spherical coordinates .................................. 16
      3.2.3 Three-dimensional Cartesian coordinates .............. 16
      3.2.4 Latitude and longitude ................................ 16
      3.2.5 Screen coordinates .................................... 17
   3.3 Model ....................................................... 17
      3.3.1 Place within the application ......................... 17
      3.3.2 Preparing view data ................................... 17
      3.3.3 Allowing control of the model ......................... 17
   3.4 Controller .................................................. 18
      3.4.1 Dragging ............................................... 18
      3.4.2 Rotating ............................................... 19
      3.4.3 Zooming ............................................... 20
      3.4.4 Menu interaction ....................................... 23
   3.5 View ......................................................... 24
      3.5.1 Basic shader functionality ............................ 24
      3.5.2 Rotation ............................................... 25
      3.5.3 Zooming ............................................... 25
      3.5.4 Menu .................................................. 25

4 Results ......................................................... 27
   4.1 Intuitiveness ............................................... 27
   4.2 Responsiveness ............................................. 28
   4.3 Suitability for an academic and/or educational use case ................ 28

5 Conclusion ..................................................... 31
   5.1 Future work ............................................... 31
CHAPTER 1

Introduction

Spherical displays offer a unique way of presenting data. Apart from being able to present inherently spherical data (e.g. a map of the planet) more accurately than a flat display, a spherical display offers its users the possibility of moving around it to view the displayed data from different perspectives. Multiple users standing on different sides of the display can use it collaboratively.

An interesting extension of spherical displays is to make them touch-sensitive. This way, users can work directly on the sphere without the need for additional input devices. Simple gestures can be used to move data around the screen, zoom into it and share it between different users.

The University of Amsterdam (UvA) has expressed its interest in the acquisition of a spherical multi-touch display, since it can be used as a research tool but also as an eye-catching demonstration object when placed in a public area of a university building. Under the supervision of the Scientific Visualization and Virtual Reality (SVVR) research group, this display and the required software will be built as a joint effort of five Bachelor students of Computer Science at the UvA.

One of the challenges in creating this spherical multi-touch setup is that software developed with a conventional screen in mind cannot be used on it. At the same time, very little research has been done on the design and implementation of software specifically designed for spherical multi-touch displays. This research project contributes to this area through the development and analysis of an open-source prototype application that — in terms of both input- and output-related aspects — is optimised for spherical multi-touch displays. This prototype application will then be available for use on the UvA’s display as it is, but above all also be a reference for future spherical multi-touch display applications.

1.1 Previous work

Flat multi-touch-based interactive displays have existed for some time, but their uptake was limited until Han [1] presented a low-cost multi-touch sensing technique based on frustrated total internal reflection (FTIR). Total internal reflection (TIR) is a phenomenon where a light beam that enters a material under an angle greater than the critical angle (defined by the material’s refraction index) is reflected back into the material by its boundaries and therefore does not escape the material. Placing an object (e.g. a finger) on the material frustrates TIR, causing the light to escape the material at that point. The material becomes an input device when it is combined with a camera that captures the escaping light and image processing software that converts the acquired image into multi-touch gestures.

At the UvA, research into multi-touch displays was described by Muller [2] and included the design and construction of a camera-based multi-touch device, the design and implementation of a gestural interaction library, the implementation of test case applications and the performance evaluation of multi-touch-enabled tasks.

Interactive spherical displays were introduced by Ushida et al. [3]. An image was projected into a full sphere and the physical rotation of this sphere was captured using track ball sensors to allow for interaction. Eight optical switches around the sphere, triggered by placing a hand above them, provided additional interaction possibilities.
Spherical displays and multi-touch functionality were first combined by Benko et al. [4]. This article describes *Sphere*, a spherical multi-touch display developed by Microsoft Research in cooperation with the University of Toronto. The output pipeline of the *Sphere* setup consisted of a video projector that, through a fish-eye lens, projected an image into a hollow sphere made of a diffuse material. Rear-diffused illumination was used for touch detection, employing infrared light so as not to interfere with the picture on the sphere. A cold mirror\(^1\) allowed an infrared camera to receive light from the opening in the sphere also used by the video projector. The setup is illustrated in figure 1.1.

The hardware setup of the display built for the UvA differs from *Sphere* in that it does not consist of a full sphere, but of a hemisphere. The surface of a hemisphere can be covered by a video projector without an additional lens and the larger size of the bottom opening allows it to be used by the projector and the camera simultaneously without the requirement of a cold mirror.

---

\[^1\]A mirror that reflects visible light but lets infrared light pass through it.

---

**Figure 1.1:** The *Sphere* setup as illustrated in [4].

Spherical displays are available commercially from Global Imagination [5], ARC Science [6] and Globe 4D [7]. Only Globe 4D’s models support direct touch interaction; to control the other implementations a separate input device such as a mouse, trackball or touchscreen is required.

1.2 Contribution

While Benko et al. [4] describes a number of sample applications built for Microsoft’s *Sphere*, the focus of the article is mainly on user experience aspects. Technical details of the implementation of applications for a spherical multi-touch display are not offered, nor is source code available.

The commercial parties mentioned in the previous section also provide different programs that can be used on their spheres. This software is, however, closed-source and not available free of charge.

The existence of a prototype that can be used as a reference facilitates the creation of open-source applications for spherical multi-touch displays. When open-source applications are available, the use of spherical multi-touch displays becomes attainable for non-commercial institutions, for example in the academic world.

This research examines the implementation of an open-source prototype of an application optimised for spherical multi-touch displays. Questions that need to be answered include:

- What are the characteristics of applications suitable for use on a spherical multi-touch display?
- How can the creation of a spherical multi-touch application prototype be leveraged into a generic reference for future applications?
• What requirements must be borne in mind when developing a spherical multi-touch application?
• What modifications are necessary in an application in order to handle spherical multi-touch input?
• What modifications are necessary in an application in order to render onto a spherical display?

This thesis continues with chapter 2, which discusses the requirements of a spherical multi-touch application and their consequences for the design of a prototype application. Chapter 3 then uses the prototype as a starting point to discuss the implementation of a spherical multi-touch application. In chapter 4, a number of experiments are carried out to verify that the prototype meets the requirements specified in chapter 2. Finally, chapter 5 concludes the research.
CHAPTER 2

Design considerations

This chapter specifies the requirements that any spherical multi-touch application needs to fulfil and lists a number of additional requirements that might be imposed on a spherical multi-touch application in certain use cases. To maximise the relevance of the prototype developed in this research, both categories of requirements will be decisive in its design process. The decisions made during this process are discussed in the second part of this chapter.

2.1 Mandatory requirements

Two requirements spherical multi-touch applications need to fulfil are relevant to any interactive application:

**The user can interact with the application in an intuitive manner.** In the case of a spherical multi-touch application, intuitiveness is strongly related to the degree to which the implemented multi-touch gestures match the user’s expectations. Especially if a spherical multi-touch display is used as a demonstration object in a public area, users should be able to interact with it without prior instructions.

**The responsiveness of the application is very high.** Because a spherical multi-touch display lets the user manipulate the displayed data with his hands, it has the potential of providing a more realistic experience than a regular screen. However, this sense of reality is lost when a noticeable delay occurs between the gestures the user makes and the resulting action on the display.

Additionally, spherical multi-touch applications need to fulfil a specific requirement:

**The application is suitable for use on a spherical multi-touch display.** A spherical display should fit the application just as well or better than a flat display as far as the nature of the data the application handles is concerned. Also, all interaction taking place with the application must be implementable in multi-touch gestures.

2.2 Additional requirements

In addition to the above requirements, it is not inconceivable that developers and commissioning parties of future spherical multi-touch applications will impose one or more of the following additional requirements on their applications:

**Existing open-source software is used whenever possible.** If open-source implementations exist for certain parts of the application, there is no need to redevelop them. By basing parts of the application on existing software, the required time investment is reduced and existing knowledge can be reused.

**The application can be used on a large number of systems.** The cost of the setup is reduced if the setup’s projector can be connected to an average desktop or laptop computer. Therefore, a requirement might be that the application does not require specific or high-end hardware and is compatible with multiple operating systems.
The application can be tested without being connected to a hardware setup. Because testing the application might not always be possible on the actual display hardware, a requirement might be that it is possible to simulate the application’s in- and output on a regular desktop or laptop computer without additional hardware.

The kind of application is suitable for an academic and/or educational use case. Schools and academic institutions are an important target group for spherical multi-touch displays. Therefore, future applications built for these displays might be required to fit an educational and/or research use case.

2.3 Application type

An informed decision needs to be made regarding the type of application used as a test case. Most importantly, it needs to be suitable for use on a spherical multi-touch display, but the additional requirements of being suitable for an academic and/or educational use case and reusing existing open-source technology when possible are also relevant at this point. These three requirements are fulfilled at once by basing the prototype on an existing open-source application that has both educational value and handles data of spherical nature. Marble is such an application.

2.3.1 Marble

Marble [8] is a virtual globe application developed by the KDE project. Unlike Google Earth, Marble is an open-source application available under the GNU LGPL licence [9]. It is developed for regular screens, but as a virtual globe application it has a clear potential of being used on a spherical display.

Marble supports spherical, equirectangular and Mercator map projections and map themes such as road map, physical map and satellite view (see figure 2.1). This core package can be extended by custom map themes and plug-ins. Currently, a list of around 30 plug-ins is available, for example providing routing functionality, displaying live positions of the sun, stars and satellites, providing live weather information and displaying Wikipedia articles for places on the map [10]. This large feature set makes Marble a valuable application in an educational context.

Because Marble is available for the Linux, Microsoft Windows and Mac OS X operating systems [11] and is designed to be lightweight [12], it also satisfies the additional requirement of not making high demands on the system it runs on.

Figure 2.1: A screenshot of the free software program Marble, displaying satellite imagery in Globe mode.
2.4 Extending the application to a spherical multi-touch display

To create a prototype application based on Marble, adaptations of its functionality are required to provide for the differences between the in- and output mechanisms that Marble was built for and the ones used within a spherical multi-touch display setup. These adaptations could be realised by changing or adding to Marble’s source code in all the areas where differences exist. However, this will not result in a prototype that clearly expresses the differences between a regular application and an application optimised for spherical multi-touch displays, as it will be difficult to reconstruct the exact changes it introduces from Marble’s large source tree. An alternative is offered by the fact that Marble can be used as a library. This allows for the creation of a new, separate application that uses Marble’s functionality when appropriate (without the need of duplicating it) and at the same time can have functionality of its own in case Marble lacks what is needed. The latter can then be achieved without changing the source code of Marble itself. Because of its advantages, the alternative approach is used for the prototype.

The fact that Marble is written in the C++ programming language [13] means that a prototype extending it needs to be written in C++ as well. Because C++, in contrast to programming languages such as Java and Python, has standards that differ per operating system [14], fulfilling the additional requirement of operating system flexibility is not trivial. For the prototype, it was decided that at least one common compiler should be supported on both Windows and Linux.

The last requirement that is relevant when exchanging a regular display for a spherical multi-touch display is that of intuitive interaction. If the display shows a representation of an object that also exists in reality, interaction with it is considered intuitive if it corresponds to the user’s experience with handling such objects in the real world. In the setting of a virtual globe application, for example, the user should be able to use his experience with handling real globes to know how to interact with the application.

For interactions to which this principle does not apply (e.g. displaying a menu), the user should be able to rely on any experience he has with flat multi-touch displays, such as those in smartphones.

2.5 Handling input

The open standard for the transmission of events generated by multi-touch hardware is the TUIO protocol [15][16]. The image processing software that converts the data from the infrared camera to touch events in the UvA’s setup is an example of software using this open standard, as it uses TUIO to send the events to the applications running on the spherical multi-touch display. By using the TUIO protocol, the prototype built in this research becomes relevant for all future applications built for a device employing this standard. Furthermore, using TUIO satisfies one of the additional requirements, as ready-to-use client and simulation libraries are available from the TUIO project for a substantial number of programming languages, among which C++ [17]. This means that existing open-source software can be used for the reception of touch data sent by the image processor.

In order to maximise the possibility of using existing TUIO libraries, the prototype application will expect to receive touch events in a Mercator-projected form instead of other systems such as three-dimensional Cartesian coordinates. The reason for this decision is that, while the TUIO specification also defines a profile for three-dimensional coordinates [15], the libraries provided by the TUIO project only support the transmission of two-dimensional coordinates. Since all touch events on the display will occur at the same radial distance, their positions can be mapped to a Mercator projection without losing information. These two-dimensional Mercator coordinates can then be sent using the default TUIO profile, which avoids the use of the 3D profile that is not supported by the existing libraries. An additional advantage of using coordinates in Mercator-projected form is that existing TUIO simulators can be used too, since all coordinates on the sphere will be mapped to a square two-dimensional area. This, for the input side, satisfies the additional requirement of being able to use a spherical multi-touch application without a hardware setup present.
2.6 Generating output

The image data a spherical multi-touch application sends to the setup’s video projector needs to be of a round shape and needs to compensate for the fact that projecting an image onto a sphere leads to a different pixel density at different locations. This preparation of the image and the manipulation of the objects on the screen using multi-touch gestures require a large number of graphical operations. Therefore, to ensure the requirement of responsiveness is met, it is worthwhile for a spherical multi-touch application to reduce its CPU load by carrying out as many graphical operations on the GPU as possible. This is often referred to as hardware acceleration.

The prototype built in this research demonstrates the implementation of hardware acceleration in spherical multi-touch applications. In order to do so, the amount of graphical processing Marble carries out is minimised by only using Marble’s software-based (i.e. CPU-based) rendering [18] to generate a two-dimensional texture in Mercator projection. The conversion of this texture to the format projected onto the sphere is then implemented as a so-called shader. This is a type of program code that is loaded into the graphics card, where it takes up a place in the graphics pipeline [19].

A major graphics framework that allows the use of shaders is OpenGL [20]. The use of OpenGL fits into the additional requirements as it is open-source software and is the de facto standard for open-source applications using hardware-accelerated graphics [21]. The latter once again increases the value of the prototype as a reference for future open-source applications for spherical multi-touch displays, because if they use any hardware-accelerated graphics, these will most likely be implemented in OpenGL as well.

In light of the additional requirement of using existing software when possible, the shader used for the prototype application will be based on a proof-of-concept developed by Mustafa Karaalioğlu that maps an image file in Mercator projection to the projection required for the spherical display.

A final consideration regarding the output section is the version of the OpenGL Shading Language (GLSL) used for the shader. As GLSL relies on the functionality of the graphics card’s hardware, only recent, high-end graphics cards support the latest GLSL versions [19]. Therefore, which GLSL version is used for the shader is a tradeoff between the required functionality and the range of hardware that should be supported. While Mustafa Karaalioğlu’s proof-of-concept shader was developed in GLSL 3.30, the required functionality is present starting from GLSL version 1.20. Because the latter version is supported by a much larger number of graphics cards, a downgrade of the proof-of-concept shader will be performed and GLSL 1.20 will be used for all additions to it.
CHAPTER 3

Implementation

Following on from the previous chapter concerning the decisions relevant during the design of a spherical multi-touch application, this chapter covers the actual implementation of such an application. The chapter is split up into five sections. Section 3.1 describes the structure of a modular spherical multi-touch application, section 3.2 explains the various coordinate systems relevant to spherical multi-touch applications and the remaining sections describe the different modules a typical application consists of in detail.

3.1 Application structure

A suitable structure for a spherical multi-touch application is the model-view-controller paradigm [22]. In an application built according to this paradigm, a model module contains the logic, functions and data of the application, a view module displays the data the model generates in a specified way and a controller module passes instructions to the model according to the user input. In a spherical multi-touch setting, the view and controller modules need to be aware of the spherical nature of the display and the multi-touch input method, but this is not necessarily true for the model. The model can in fact be (the backend of) a regular application developed for flat screens if the view and controller act as a wrapper handling the different in- and output method.

The latter is the approach used for the prototype application built in this research: the model module contains the functionality provided by Marble and is neither aware of the spherical nature of the display nor of the multi-touch input method. The controller module receives the multi-touch gestures and converts them to commands sent to Marble. The view module uses the data from Marble to send instructions to the shader, which generates the rendering on the display. The structure of the prototype application is illustrated in figure 3.1.

Figure 3.1: The structure of the prototype application.
3.2 Coordinate systems

A developer of a spherical multi-touch application may be faced with a total number of five coordinate systems that can be used to indicate points on a sphere:

3.2.1 Normalised Mercator coordinates

Normalised Mercator coordinates are a two-dimensional coordinate system where x and y run from 0 to 1. The point (0, 0) is in the upper left corner of area described. In the prototype application, positions on the texture and the positions of multi-touch gestures are expressed in this system.

3.2.2 Spherical coordinates

Spherical coordinates describe positions on a sphere using azimuth and polar angles. Azimuth is the angle between the point and the prime meridian and, from west to east, runs from $-\pi$ to $\pi$. Polar is the angle between the point and the equator and, from south to north, runs from $-\frac{1}{2}\pi$ to $\frac{1}{2}\pi$.

The spherical coordinate system uses the notion that the radius is constant to represent points on the sphere in two coordinates. However, although both systems have two degrees of freedom, an important difference to Mercator coordinates is that spherical coordinates still refer to the actual points on the sphere in three-dimensional space instead of to a two-dimensional projection of them.

The relationship between spherical and normalised Mercator coordinates is as follows:

\[
\begin{align*}
\text{azimuth} &= 2\pi \left( x - \frac{1}{2} \right) \\
\text{polar} &= \arcsin \left( \tanh \left( 2\pi \left( \frac{1}{2} - y \right) \right) \right)
\end{align*}
\]

\[
\begin{align*}
x &= \frac{\text{azimuth}}{2\pi} + \frac{1}{2} \\
y &= \frac{1}{2} - \frac{\log \left( \frac{1 + \sin(\text{polar})}{\cos(\text{polar})} \right)}{2\pi}
\end{align*}
\]

In the prototype application, positions on the sphere are expressed in spherical coordinates.

3.2.3 Three-dimensional Cartesian coordinates

A rotation matrix cannot be applied to spherical coordinates; therefore, they need to be converted to and from three-dimensional Cartesian coordinates when a rotation needs to be performed.

The relationship between spherical and three-dimensional Cartesian coordinates is as follows:

\[
\begin{align*}
x &= \cos(\text{polar}) \cos(\text{azimuth}) \\
y &= \cos(\text{polar}) \sin(\text{azimuth}) \\
z &= \sin(\text{polar})
\end{align*}
\]

Apart from these three major systems, two minor variants might be encountered if an existing application used as a model expects input in this form:

3.2.4 Latitude and longitude

Latitude and longitude describe the same angles as azimuth and polar, but where the latter are normally expressed in radians, latitude and longitude normally use degrees. The prototype uses latitude and longitude angles when Marble functions expect input in this form.
3.2.5 Screen coordinates

If a position on the texture needs to be expressed relative to the screen or window it is displayed in, or if the location of a multi-touch gesture needs to be expressed relative to the touchable area, the prototype application converts normalised Mercator coordinates to screen coordinates by multiplying them by the size of the respective screen, window or area.

3.3 Model

The model module of a spherical multi-touch application is neither concerned with the application’s input nor with its output. Therefore, the model contains relatively little functionality that is specific to spherical multi-touch displays. The differences that do exist between the model of a spherical multi-touch application and a regular application are explained in this section using the prototype application as an illustration. Because the model of the prototype is an extension of Marble — an existing application developed for flat displays — it exactly contains these differences.

3.3.1 Place within the application

Because the model module is located between the controller and view modules in the flow of the application (see figure 3.1), it plays a central role within the application. The prototype application uses this fact by making the model the leading module of the application: the model contains all data that is shared between the modules, determines when the application is terminated and, in that case, is responsible for shutting down the other modules and clearing the memory space used.

In general, if the model of a spherical multi-touch application is an extension of an existing application, the required additions fall into two categories:

3.3.2 Preparing view data

An existing application acting as a model within a spherical multi-touch application might need to supply data to the view module that is different from what it would display on a flat screen.

In the case of the prototype application, the view module needs to be supplied with a texture that contains both the map data and the image data for a menu that can be used to select a map theme. Within Marble, the map data is displayed in a widget contained in a larger graphical user interface (GUI), but Marble’s modular structure offers a way to copy only the relevant items (map tile images, text overlays, etc.) to the texture: the model opens a window that only contains said map widget and configures it such that it displays the entire earth in Mercator projection. It then suffices to save this widget’s contents to the texture (effectively creating a screenshot).

Because projecting onto a sphere inevitably means that the resulting pixel density is variable, the texture should be of the highest possible quality to ensure satisfactory results even in the areas with a low pixel density. Therefore, the prototype’s model sets the quality of the map widget to the highest mode available. Further configuration of the widget is performed to remove elements from it that should not be displayed on the sphere, such as the compass, scale bar and overview map Marble normally displays on its maps.

The image data for the menu is generated separately: a list of available map themes is requested from Marble and the preview image each of the themes provides is drawn onto the texture as a menu icon.

3.3.3 Allowing control of the model

The other main aspect of using an existing application as a basis for a spherical multi-touch application’s model, is that the controller module needs to be able to change the model’s state in all ways required.
In the prototype application, Marble itself provides functions the controller module can use to change the map position and zooming level. The model adds functionality to allow the controller to change the map theme with one function call when a menu icon is clicked.

Functionality is also added to automate updating the texture every time the controller changes the map widget’s state. In order to do this, a system called signals and slots is used that is offered by the Qt framework [23] Marble is developed in. The principle behind this system is that a function emits a signal every time a certain event occurs. Special functions called slots can be connected to a signal, which means they are called automatically when this signal is emitted by a function.

The receipt of new map tiles from online sources causes Marble to emit the `downloadComplete` signal. Changes in the zoom level and — although the name of the signal might not suggest this — the centre of the map cause the `distanceChanged` signal to be emitted. By connecting both of these to the function (now slot) that updates the texture, the latter is automatically updated every time the state of the model changes, without the need for the controller to explicitly request this.

A different aspect of allowing the controller to map multi-touch gestures to changes in the state of the model, is the fact that an application used as a basis for the model usually provides its own input handling. Conflicts between the two input methods may arise. In the prototype application, this problem is solved by disabling the default mouse and keyboard input handling provided by Marble, as it is not used in the spherical multi-touch application.

### 3.4 Controller

The controller of a spherical multi-touch application is concerned with interpreting multi-touch events and changing the state of the application’s model module based on this interpretation. The controller of the prototype application strictly limits itself to this task, as a separate TUIO client\(^1\) handles the actual receipt of TUIO events. This receipt is a black box to the controller. The TUIO client runs in a separate thread, in which it is constantly listening to port 3333 to receive TUIO events. Whenever such an event occurs, it notifies the controller.

If a new finger appears on the display, the TUIO client warns the controller that a cursor has been added. The controller adds a pointer to the cursor to an internal list and calls its own `update` function to interpret the list of currently active cursors. If the position of a finger that was already present changes, the TUIO client updates the cursor data in place, therefore the controller’s internal list of pointers is still valid and does not need to be changed. It does, however, need to be reinterpreted by the `update` function. Finally, if a finger is removed from the display, the controller first calls the `update` function once more to process the position of the cursor just before it was removed. Then, it removes the cursor pointer from the internal list.

#### 3.4.1 Dragging

If the `update` function is called while one cursor is active and the sphere is not zoomed in, the gesture is classified as dragging. Dragging refers to the gesture where one finger is used to move an element to a different position. On a flat display, dragging leads to a translation of this element, but on a spherical display the element rotates around the centre of the sphere (see figure 3.2).

To calculate the exact rotation, the current and previous position of the finger are first converted from normalised Mercator coordinates to three-dimensional Cartesian form using spherical coordinates as an intermediate step. Since the positions in three-dimensional Cartesian form are in fact vectors with the centre of the sphere as a starting point, the axis around which the finger has rotated between the two positions is the vector perpendicular to the two position vectors. It is found by calculating the cross product of the position vectors:

\(^1\)Part of the TUIO project’s C++ reference implementation [17], but somewhat adapted to eliminate compiler warnings.
The perpendicular vector is normalised by dividing each of its coordinates by the vector’s magnitude (length):

\[
\text{mag} = \sqrt{\text{perp}_1^2 + \text{perp}_2^2 + \text{perp}_3^2}
\]

\[
\text{perp} = \left(\frac{\text{perp}_1}{\text{mag}}, \frac{\text{perp}_2}{\text{mag}}, \frac{\text{perp}_3}{\text{mag}}\right)
\]

The angle of rotation is found using the dot product of the two position vectors. The dot product of two vectors is the cosine of the angle between them, therefore the arccosine of the dot product produces the angle:

\[
\text{angle} = \arccos(\text{prev} \cdot \text{cur}) = \arccos(\text{prev}_1\text{cur}_1 + \text{prev}_2\text{cur}_2 + \text{prev}_3\text{cur}_3)
\]

The actual rotation based on the axis and the angle is performed by a separate \texttt{rotate} function, because the same functionality is used by the rotating gesture discussed in the next subsection. The \texttt{rotate} function composes a rotation matrix based on axis \((u \ v \ w)^\top\) and angle \(\theta\) using the following template:

\[
\begin{vmatrix}
\cos(\theta) + u^2(1 - \cos(\theta)) & uv(1 - \cos(\theta)) - w\sin(\theta) & uw(1 - \cos(\theta)) + v\sin(\theta) \\
vu(1 - \cos(\theta)) + w\sin(\theta) & \cos(\theta) + v^2(1 - \cos(\theta)) & vu(1 - \cos(\theta)) - u\sin(\theta) \\
wv(1 - \cos(\theta)) - v\sin(\theta) & vw(1 - \cos(\theta)) + u\sin(\theta) & \cos(\theta) + w^2(1 - \cos(\theta))
\end{vmatrix}
\]

Because the new rotation should not replace the previous rotation of the sphere but build upon it, the generated rotation matrix is multiplied with the previous matrix. For \(3 \times 3\) matrices \(A\) and \(B\), matrix multiplication is defined as:

\[
(AB)_{ij} = \sum_{k=1}^{3} A_{ik} B_{kj}
\]

In the prototype application, the result of this multiplication is made available to the view module, which uses the matrix to correctly rotate the texture.

### 3.4.2 Rotating

If the \texttt{update} function is called while two cursors are active, it carries out calculations to detect the “classical” rotating gesture. This is the two-finger gesture that rotates an element around a point on a
flat display. On a spherical display, the element is rotated around an axis (see figure 3.3).

On a flat display, a line is drawn through the starting position of both fingers and another line through the current position of the fingers to find the rotation point at the intersection of the lines. A similar technique can be used on a sphere, where the rotation axis passes through the points where the great circle\(^2\) through the starting positions and the great circle through the current positions intersect.

All vectors corresponding to points on one great circle are perpendicular to one vector, the normal vector of the great circle. The points where the great circles intersect are on both circles and thus their vectors are perpendicular to the normal vectors of both great circles. Therefore, to find the rotation axis, first the normal vectors of the two great circles are calculated from the two points known on each of them. For this, the previously described cross product is used. Then, the cross product of the two normal vectors is calculated to find a vector that is perpendicular to both of them. This is the rotation axis.

The angle of rotation is the angle between the two great circles’ normal vectors, which, as described previously, is the arccosine of the dot product of the two vectors.

Once the rotation axis and angle have been determined, the remaining steps carried out by the prototype application correspond to the final steps of the dragging gesture: the rotate function generates the appropriate rotation matrix and makes it available to the output module.

### 3.4.3 Zooming

Zooming is a two-finger gesture where changing the distance between the fingers changes the zoom level of the element underneath them (see figure 3.4).

Zooming the image of a flat display can simply be achieved by changing the scale of this image. On a spherical display, however, zooming is more complicated than zooming the texture using the method used on flat displays. Firstly, this is because the left side of the sphere’s texture should seamlessly fit to the right side of the texture. Increasing the scale implies that part of the image can no longer be displayed and therefore the seamless fit will be lost. Secondly, a zoomed-in version of the texture would still be mapped to the sphere as if it were in Mercator projection, which it is not. The entire top and bottom areas of the texture would each be contracted to one point, which is correct for the entire texture but not for smaller parts of it.

In the prototype application created in this project, the problem that a zoomed-in image cannot seamlessly cover a sphere is avoided by the fact that the display used only consists of one hemisphere. On

\(^{2}\)A circle that spans the sphere and shares its circumference, radius and centre.
an actually spherical display, a solution could also be to only display the zoomed-in image on one hemisphere. The projection problem is solved by displaying zoomed-in versions of the map in a custom projection instead of a Mercator projection. The equations describing this projection are:

\[
x = \frac{\text{hypotenuse} \cdot \cos(\text{azimuth}) + 1}{2}
\]

\[
y = \frac{1 - \text{hypotenuse} \cdot \sin(\text{azimuth})}{2}
\]

where \( \text{hypotenuse} = 1 - \frac{2}{\pi} \text{polar} \)

In words, this custom projection cuts the largest possible circle\(^3\) from the texture and distributes it evenly across the sphere. The top and bottom areas of the texture are no longer contracted to one point, but the inevitable result is that the corners of the texture are not visible on the sphere.

Because a user intending to carry out a zooming gesture might also slightly rotate his fingers by accident and, conversely, a user intending a rotation might also accidentally change the distance between his fingers, strictly separating the two gestures in the \texttt{update} function is not possible. Instead, \texttt{update} interprets all two-cursor gestures as both a rotating and a zooming gesture. This dual interpretation will not lead to noticeable side-effects if the changes in zoom level while rotating and the changes in rotation while zooming are minimal. At the same time, it allows for intended combination of rotating and zooming into one gesture.

In the same way the position of the rotation point or axis does not change during a rotation, zooming has a focus point that remains in the same position. If a gesture combines zooming and rotating, the rotation point is also the focus point of the zoom. If the gesture, however, contains no rotation at all, there is also no rotation point, which means the focus point of the zoom needs to be found using a different technique. In this case, the position of the focus point on the line between the two fingers corresponds to the ratio in which the two fingers contribute to the zoom. For example, if the right finger is kept in place and the left finger is moved, 100% of the zoom is caused by the left finger. This means the focus point is at 100% of the line from the left finger, i.e. it coincides with the position of the right finger. If the left finger is moved twice as much as the right finger, the focus point is at two-thirds of the line between the left and right finger.

The zoom factor is determined by dividing the angle between the two fingers’ current positions by the angle between the two fingers’ starting positions. Because the projection used when zoomed in uses the texture to cover one half of the sphere while the regular texture covers the entire sphere, a zoom factor smaller than 2 would actually zoom out the image. Therefore, zoom factor 2 is used as a threshold:

\(^3\)A circle having a diameter equal to the width and height of the square texture.
the zoom level exceeds 2, the zoom projection is used, if it is smaller than 2 it is interpreted as “no
zoom”. An additional advantage of using a threshold is that a minimal zoom introduced as a side effect
of a rotation gesture does not unintentionally open the zoom view but is discarded instead.

Switching to zoom view

When switching from global to zoom view, the rotation of the sphere must be taken into account.
Firstly, the central point of the hemisphere on which the zoomed image will be projected must show
the same point of the texture before and after the change of view. On the hemispherical display used
for the prototype application, the point with spherical coordinates (0, 0) is centred when the sphere
is not rotated. The coordinates of the currently centred point can therefore be calculated by applying
the inverse of the current rotation matrix to the spherical coordinates (0, 0). The inverse of a three-
dimensional matrix is defined as:

\[
\frac{1}{\det} \cdot \begin{vmatrix}
(m_{22}m_{33} - m_{23}m_{32}) & -(m_{12}m_{33} - m_{13}m_{32}) & (m_{12}m_{23} - m_{13}m_{22}) \\
-(m_{21}m_{33} - m_{23}m_{31}) & (m_{11}m_{33} - m_{13}m_{31}) & -(m_{11}m_{23} - m_{13}m_{21}) \\
(m_{21}m_{32} - m_{22}m_{31}) & -(m_{11}m_{32} - m_{12}m_{31}) & (m_{11}m_{22} - m_{12}m_{21})
\end{vmatrix}
\]

where

\[\det = m_{11}(m_{22}m_{33} - m_{23}m_{32}) - m_{12}(m_{33}m_{21} - m_{23}m_{31}) + m_{13}(m_{21}m_{32} - m_{22}m_{31})\]

By setting the centre of the texture used for the zoom view to the centre point found using the inverse
matrix, two of the three rotation angles of the original sphere are preserved.

Secondly, the rotation of the image around the centre point must be preserved. This is the third rotation
angle of the original sphere and is what remains if the current rotation matrix is multiplied by matrices
that rotate the centre point to spherical coordinates (0, 0):

\[
\begin{pmatrix}
\cos(\text{azimuth}) & -\sin(\text{azimuth}) & 0 \\
\sin(\text{azimuth}) & \cos(\text{azimuth}) & 0 \\
0 & 0 & 1
\end{pmatrix}
\begin{pmatrix}
\cos(-\text{polar}) & 0 & \sin(-\text{polar}) \\
0 & 1 & 0 \\
-\sin(-\text{polar}) & 0 & \cos(-\text{polar})
\end{pmatrix}
\]

From the resulting matrix, the third angle can be extracted as follows:

\[
\text{rot} = \arctan\left(\frac{\text{matrix}_{32}}{\text{matrix}_{33}}\right)
\]

At this point, the prototype application’s controller makes the zoomed-in and correctly centred texture
and the third rotation angle available to the view module and notifies it of the fact that the projection
needs to be switched to zoom mode.

Changing zoom level

When the view mode has been switched from global to zoom, the texture needs to be set to the zoom
level that corresponds to the calculated zoom factor. Changing only the zoom level, however, means the
point at the centre of the texture is the point that stays in the same position. Instead, this should be
the previously calculated focus point. In the prototype application, this is corrected by converting the
coordinates of the focus point from three-dimensional Cartesian coordinates to screen coordinates and
asking Marble to which latitude and longitude these screen coordinates map in the current zoom level
and in the zoom level after the calculated zoom factor is applied. The focus point remains in the same
position if at the same time the zoom level is changed, the centre point of the texture is set to:

\[
\text{centre}_{\text{new}} = \text{centre}_{\text{old}} + \text{focus}_{\text{old}} - \text{focus}_{\text{new}}
\]

A problem that might surface during this process, is that zooming out with a focus point close to the
top or bottom edge of the complete map causes the appearance of black bars on the texture as part of
the displayed area falls outside the map. To prevent this, the prototype application contains a function
that ensures the centre of the texture is always located far enough from the top and bottom edges to
have image data available for the entire texture.
Dragging while zoomed in

While for the user of the display the dragging gesture should not work differently when zoom view is active, it needs to be processed differently by the controller because it needs to result in a translation of the texture instead of a rotation of the sphere.

To calculate the translation, the prototype’s controller first converts the coordinates of the previous and current position of the cursor from normalised Mercator coordinates to spherical coordinates. Because the translation of the texture should be relative to the rotation around its centre, the two sets of spherical coordinates are corrected for this rotation. In the custom zoom projection, this does not require a rotation matrix, but can be achieved by adding the rotation angle to the azimuth values. Using the equations for the custom zoom projection, the two sets of spherical coordinates are then converted to x and y coordinates on the texture. Marble’s functionality is used to map the two sets of x and y coordinates to latitude-longitude positions on the map. Using these, the new centre of the texture is calculated:

\[
\text{centre}_\text{new} = \text{centre}_\text{old} + \text{cursor}_\text{prev} - \text{cursor}_\text{cur}
\]

The previously mentioned function preventing the texture centre point from reaching the top and bottom edges of the map is also applied when dragging the map.

Rotating while zoomed in

Because the zoom view only supports rotation around the centre of the hemisphere, rotation gestures also need to be interpreted differently.

The prototype’s controller does this by simply interpreting all rotation gestures carried out in zoom view as rotations around the centre, but a more precise approach based on the two-dimensional rotation gesture could be considered.

Closing zoom view

When the view is zoomed out until the threshold between zoom view and global view is crossed again, any translations and rotations carried out while zooming must be converted to the appropriate rotation of the global view.

In the prototype application, this is realised by creating a new rotation matrix for the global view based on the last centre point and centre rotation of the zoom view:

\[
\begin{bmatrix}
1 & 0 & 0 \\
0 & \cos(\text{rot}) & -\sin(\text{rot}) \\
0 & \sin(\text{rot}) & \cos(\text{rot})
\end{bmatrix}
\cdot
\begin{bmatrix}
\cos(\text{polar}) & 0 & \sin(\text{polar}) \\
0 & 1 & 0 \\
-\sin(\text{polar}) & 0 & \cos(\text{polar})
\end{bmatrix}
\cdot
\begin{bmatrix}
\cos(-\text{azim}) & -\sin(-\text{azim}) & 0 \\
\sin(-\text{azim}) & \cos(-\text{azim}) & 0 \\
0 & 0 & 1
\end{bmatrix}
\]

3.4.4 Menu interaction

The gestures that are used for interaction with a menu fall in a different category than dragging, rotating and zooming, as they do not correspond to interactions possible with real objects. Therefore, more attention should be paid to the menu interaction gestures, to ensure that they, too, are intuitive.

Opening menu

Placing a finger on an item for a period longer than a typical tap is a gesture smartphones often associate with opening a menu for that item. Therefore, it can be argued that the gesture is also a suitable choice for opening a menu on a spherical multi-touch display.

In the prototype application built in this research, the update function considers all one-finger gestures where the cursor position remains virtually unchanged for 1.5 seconds as a gesture meant to open the menu. The actual position of the finger is not relevant.

\[^4\text{These spherical coordinates express the positions of the finger in relation to the sphere, they are not related to the coordinates of the point of the map being touched.}\]
Interacting with menu

Once the menu of a spherical multi-touch application is opened, the natural gesture to select one of the buttons is tapping. Tapping outside the menu area is an intuitive gesture to close it again, provided the application’s menu does not cover the entire surface of the sphere.

The prototype application’s menu is a ring around the edge of the hemispherical display from which a map theme can be selected. To process the interaction with this menu, the prototype’s controller analyses all one-finger gestures where the cursor exists for less than 1 second. The menu covers the area between polar angles $\frac{5}{12}\pi$ (75°) and $\frac{1}{2}\pi$ (90°). Therefore, the controller converts the position of the tap from normalised Mercator coordinates to spherical coordinates to determine whether it should close the menu or select a different theme. Taps that are closer to the top of the hemisphere than $\frac{5}{12}\pi$ radians close the menu, while for taps between $\frac{5}{12}\pi$ and $\frac{1}{2}\pi$ radians the azimuth angle is divided by the total number of buttons to determine which of them was clicked.

3.5 View

The view module of a spherical multi-touch application is responsible for converting the data in the model to the image data that is fed to the projector. The latter needs to be transformed in such a manner that it exactly covers the surface of the sphere.

The prototype application built in this research demonstrates a view module that outsources this transformation to an OpenGL shader. The actual view module as displayed in figure 3.1 therefore does not contain any of this mathematical logic, but is only concerned with compiling and starting the shader and supplying it with the required data. This causes the view module to be very generic, which means it could be reused in any spherical multi-touch application that wishes to use an OpenGL shader.

The prototype’s shader is based on a proof-of-concept developed by Mustafa Karaalioğlu, which demonstrates the mapping of a two-dimensional texture to an image that can be projected onto a spherical multi-touch display. This research extends this proof-of-concept to allow for rotation of the sphere, zooming and displaying a menu.

3.5.1 Basic shader functionality

The type of shader that can be used to transform a two-dimensional texture to an image that can be projected onto a spherical multi-touch display is a so-called fragment shader. Fragment shaders are executed once for every pixel in the resulting image, of which they calculate the value based on their input data, but not based on the values of surrounding pixels.

The main way in which Mustafa Karaalioğlu’s fragment shader calculates the pixel values of the image remains unchanged in the final version of the fragment shader used in the prototype application: first, the coordinates of the pixel of which the value is currently calculated are scaled to fall between -1 and 1. Then, the radius of the pixel is calculated, that is the distance of the pixel to the centre of the resulting image located at pixel (0, 0). Pixels with a radius equal to or larger than 1 fall outside the sphere and should therefore remain black. For pixels with a radius smaller than 1, the positions they represent on the sphere are calculated:

\[
\text{azimuth} = \begin{cases} 
-\frac{1}{2}\pi & \text{if } x = 0 \text{ and } y < 0 \\
\frac{1}{2}\pi & \text{if } x = 0 \text{ and } y \geq 0 \\
\arctan(y/x) & \text{if } x \neq 0
\end{cases}
\]

\[
polar = \arccos(\text{radius})
\]

When no rotation or zooming is performed, the shader uses the equations described in section 3.2 to find the position on the texture that corresponds to these spherical coordinates. It then sets the pixel in the resulting image to the colour found in the texture at this position.
3.5.2 Rotation

In this research, the shader is extended to rotate the image on the sphere based on a rotation matrix composed by the controller module. It does this by adding steps to the process described in the previous subsection: before the spherical coordinates are looked up in the texture, they are converted to three-dimensional Cartesian coordinates, multiplied by the rotation matrix and converted back to spherical coordinates.

Additionally, the prototype’s shader differs from Mustafa Karaalioglu’s version in that it rotates the image on the sphere by $\frac{1}{2}\pi$ radians by default. The reason for this is that on a hemispherical display, the point to which the entire top edge of the texture is contracted would otherwise be in the most prominent position on the display. Rotating the image by $\frac{1}{2}\pi$ radians places the centre of the texture in that position.

3.5.3 Zooming

Another extension to the shader is the addition of support for the custom projection used when zooming into the sphere’s texture. When the prototype’s shader is instructed by the controller module to use the zoom view, the spherical coordinates are not looked up in the texture using the equations described in section 3.2, but using the equations of the custom projection described in subsection 3.4.3. In order to also apply the rotation around the centre of the texture — another variable received from the controller — this angle is subtracted from the azimuth angle before the latter is converted to texture coordinates. The zoom projection already places the centre of the texture at the centre of the display, therefore no additional rotation by $\frac{1}{2}\pi$ radians is performed when zoom view is active.

3.5.4 Menu

Finally, the shader is extended to support a texture that does not only contain the image data that must be displayed on the sphere at all times, but also the image data for a menu that the shader should only display if it is instructed so by the controller.

The menu image data is located in the top 7.5% of the texture, but when enabled the menu appears just above the edges of the hemispherical display, i.e. just above the equator of the corresponding full sphere. Therefore, if the menu is enabled and the polar coordinate of the point the shader processes falls between $\frac{5}{12}\pi$ and $\frac{7}{12}\pi$ radians before any rotation or zooming is applied, the shader samples the texture at the normal x coordinate the point’s azimuth maps to, but at a y coordinate translated to the menu area of the texture. If the polar coordinate of the point falls outside the menu range, it displays the same data as otherwise, but slightly darkened to emphasise the menu.

A consequence of using the top 7.5% of the texture for the menu image data, is that the normalised Mercator coordinates used in the rest of the shader are no longer correct. They are all multiplied by 0.925 in the vertical direction to refer to the same point of the regular texture area.
CHAPTER 4

Results

For most mandatory and additional requirements formulated in chapter 2, the measures taken to fulfil them in the prototype application have been discussed in the sections on the design decisions and the implementation:

- the suitability for a spherical display is guaranteed by the fact that the application handles data of spherical nature and the main interaction it supports can be replaced by multi-touch gestures;
- the use of existing open source software when possible is realised by using Marble, the TUIO project’s reference implementation for C++ and OpenGL;
- the possibility of using the application on a large number of systems is guaranteed by making it compatible with at least one compiler on both Windows and Linux and by using a downgraded GLSL version that more graphics cards support;
- the requirement of being able to use the application without a hardware setup present is fulfilled by supporting simulator software.

However, there are three requirements for which more research is required to verify they are fulfilled. The research carried out to verify the intuitiveness of the application, its responsiveness and its suitability for an academic and/or educational use case is discussed in this chapter.

4.1 Intuitiveness

The requirement of intuitiveness has been tested by means of a user survey. Four users not familiar with the project were invited to use the spherical multi-touch display\(^1\) running this research’s prototype application. They were only informed of the fact that they could use touch gestures to control the display, but not of the functionality of the application and not of the gestures the application supports.

The first part of the survey asked the users to explore the interaction with the display without a particular assignment. The goal was to see to what extent their expectations of the interaction with the sphere and the implemented gestures matched. The second part of the survey consisted of specific assignments: the users were shown animations of particular features of the prototype application and were asked to replicate the behaviour shown using touch gestures.

The results of this survey can be summarised as follows:\(^2\)

**Dragging:** All four users discovered the dragging gesture during the first part of the survey, but three of them also expected the possibility of using an entire hand for this gesture instead of one finger. During the second part of the survey, all users repeated the two assignments containing a dragging gesture correctly.

**Rotating:** One user discovered the two-finger rotation gesture during the first part of the survey, a second user achieved the same rotation of the sphere by moving one finger along the edge of the display. In the second part of the survey, all users managed to rotate the sphere in the direction the example showed, but the one user again used one finger to achieve this before also trying the two-finger gesture.

\(^1\)In fact, a simulated version of the display, as the display hardware was not available at the time of writing.

\(^2\)The complete results are available in the appendix of this thesis.
**Zooming**: Three users discovered the zooming gesture during the first part of the survey and therefore also applied it correctly in the second part. The fourth user stated he did not expect a spherical display to support zooming, but did apply the correct gesture in the second part of the survey.

**Menu interaction**: None of the users expected the existence of a menu in the first part of the survey and none of them succeeded in opening it in one attempt in the second part. Three users successfully opened the menu during a second attempt, after first trying to open it using a tap, a double tap and a two-finger gesture quasi pulling the menu out at the display’s edges, respectively. Once opened, the menu was correctly handled by these users: they succeeded in selecting a theme and closing the menu again. The fourth user did not succeed in opening the menu.

**Other gestures**: Apart from the gestures the application supports, all users expected the possibility of tapping on points on the map to receive more information about them. One user also expected the application to support a flick-like gesture that would spin the sphere with momentum.

All in all, the results of the survey are considered satisfying, as all gestures were discovered by at least three out of four users in one or two attempts. The touch-and-hold gesture opening the menu proved to be the most difficult gesture to guess correctly without receiving instructions. However, there was little consistency in the gestures the users tried to use first, so using a different gesture to open the menu would not have led to considerably better results. More likely, the cause of the problem is that opening the menu is the only gesture that does not represent a natural interaction with the object shown on the display. Therefore, the conclusion drawn from the user survey is that the prototype can be considered intuitive and that the requirement is fulfilled.

### 4.2 Responsiveness

The requirement of very high responsiveness has been tested using time measurements. The time between the creation of the touch events and the completion of the resulting change on the display should not exceed 100 milliseconds, as this is considered to be the limit for the user to feel “that the system is reacting instantaneously” and to have a sense of “directly manipulating objects in the UI” [24].

The average length of this timespan is displayed in table 4.1 for the six types of gestures the application distinguishes.

<table>
<thead>
<tr>
<th>Gesture</th>
<th>Average response time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Drag</td>
<td>7.956</td>
</tr>
<tr>
<td>Rotate</td>
<td>8.276</td>
</tr>
<tr>
<td>Zoom</td>
<td>10.596</td>
</tr>
<tr>
<td>Zoomed drag</td>
<td>10.488</td>
</tr>
<tr>
<td>Zoomed rotate</td>
<td>9.932</td>
</tr>
<tr>
<td>Open/close menu</td>
<td>6.155</td>
</tr>
</tbody>
</table>

While the results show that the prototype application’s response time is longer if the processing of a gesture requires the rerendering of the texture (which is the case for zooming and for dragging while zoomed in), it is well within the limit of 100 milliseconds for all gesture types. The time measurements therefore prove the application meets the requirement of high responsiveness.

### 4.3 Suitability for an academic and/or educational use case

To confirm the suitability of the prototype application for an academic and/or educational use case, a case study has been performed into the application’s suitability as a demonstration tool for academic research carried out at the UvA. Specifically, this case study involved using the prototype application to demonstrate the research carried out by the UvA-BiTS project.

BiTS is the UvA’s “flexible, state of the art Bird Tracking System”, which “includes a solar powered,
light weight GPS tag with rechargeable batteries, a tri-axial accelerometer, two way data-communication to a ground station network, automated data processing and visualization in the Virtual Lab. Researchers from multiple organizations are working with this system to study migration, navigation, foraging strategies on land and at sea” [25].

Currently, UvA-BiTS has a database containing over 10 million records aggregated during the tracking of birds. To analyse and draw conclusions from such large amounts of data, the use of visualisation is essential. For this, the UvA-BiTS project uses the above-mentioned Virtual Lab, a web service that mainly uses Google Earth to visualise the data points, either by using Google Earth as a plug-in within the web site or by offering KMZ files³ as a download.

While the Virtual Lab software is already a very valuable tool for researchers analysing the data gathered by UvA-BiTS, there is room for improvement mainly in terms of demonstrating the data to an audience. This has a couple of reasons: firstly, the Virtual Lab allows the researchers to demonstrate the data to the audience in a one-to-many fashion, but it does not allow those interested to discover the research interactively by themselves. Secondly, the current way of presenting always concerns a two-dimensional projection of the bird movements, which is inevitably accompanied by distortions of the Earth’s surface. It does not provide the audience with a good impression of the speed and scale at which the bird movements take place.

The prototype application developed in this research can provide a solution to these issues. As Marble supports the Keyhole Markup Language, the prototype application can read and display the files UvA-BiTS’s Virtual Lab service generates. The spherical multi-touch display running the prototype application can then be placed in a public area of the university building or at a symposium or conference to allow for accessible and interactive exploration of a selection of bird movement data sets by guests. Because the data is displayed on an actual sphere, there is no longer the problem of map distortions and the scale of the bird movements in relation to the earth is instantly visible. Moreover, the possibility of choosing from different map types allows the user to relate the bird movements to other data. For example, satellite imagery can be used to analyse the area a bird resides and weather maps can be used to relate bird migration to changes in temperature.

To allow for an even better demonstration experience, the prototype application can be extended into an application dedicated to the demonstration of the UvA-BiTS research.

The extensions that can be considered in this case are mainly related to increasing the amount of information displayed about the bird movements. By default, Marble’s KML support does not reach further than displaying the route the bird travelled (figure 4.1a): the location of the measurement points is not clearly indicated. The first possible extension is therefore to clearly indicate these locations using marker icons (figure 4.1b). Furthermore, extensions are conceivable that change the appearance of the route and markers to convey more information about the flight of the bird: the size of the markers can be made variable to indicate the height of the bird (figure 4.1c) and the time of day on which the bird travelled a certain route section can be expressed by varying the colour of the route (figure 4.1d).

Figure 4.1: From left to right, the amount of information about the bird’s movement increases.

³A compressed form of the Keyhole Markup Language, an XML-based open data format for geographical data.
A second possible extension of the prototype application is the addition of an automatic display mode. In this mode, the application automatically rotates the sphere from the first to the last measurement point. If the rotation speed is made to be proportional to the actual speed of the bird at the currently centred measurement point, the rotation provides the display’s user with a very clear impression of the speed the bird travels at. As has been described earlier, this is one of the aspects the current visualisation solution lacks. Apart from rotating automatically, the application can zoom in the view automatically if the bird does not cover great distances but remains in a small area. A secondary advantage of an automatic display mode is the fact that the spherical multi-touch display shows movement at all times and is therefore more likely to attract the attention of a potential user passing by.

Altogether, the case study leads to the conclusion that the prototype application succeeds in displaying a large number of aspects of the data gathered by the UvA-BiTS project on the spherical multi-touch display, although small additions to the application might be required. Because the prototype application allows the presentation of the UvA-BiTS data in a more accessible, interactive and clarifying way than the current solution provided by Google Earth on a regular computer, it has not only met the requirement of being suitable for an academic use case, but has actually opened up new possibilities within another area of research.
CHAPTER 5

Conclusion

This research studied the development of an application optimised for spherical multi-touch displays. In particular, the development of an open-source prototype of such an application that can serve as a reference for future implementations was analysed.

In order to do so, three requirements were drawn up that any spherical multi-touch application should fulfil, along with four additional requirements that might be imposed on a spherical multi-touch application in certain use cases. In relation to these requirements, the choices that need to be made during the design of a spherical multi-touch application were discussed and design decisions were made to maximise the relevance of the prototype developed in this research.

Thereafter, the implementation of a spherical multi-touch application was analysed. The model-view-controller paradigm was explained to be a suitable structure for a spherical multi-touch application. The various coordinate systems a developer of a spherical multi-touch application may be faced with were analysed and finally the implementation of an application’s model, view and controller modules was analysed in reference to the prototype application.

It was demonstrated that the differences between the model of a spherical multi-touch application and a regular application can be divided into two general categories: preparing the data the view requires and allowing the controller to control the model. For the controller model, the logic and equations required to interpret dragging, rotating, zooming and menu interaction gestures were explained. Finally, the outsourcing to an OpenGL shader of the conversion from a two-dimensional image to the projection onto the sphere was discussed for the view module.

As three of the requirements imposed on spherical multi-touch applications were not explicitly fulfilled during the design and implementation of the prototype application, a more elaborate analysis of the prototype was performed to ensure these three requirements, too, had been fulfilled. The intuitiveness of the application was verified by means of a user survey, the responsiveness of the application using time measurements and finally, the suitability for an academic and/or educational use case by detailing a possible use case of the prototype application.

All in all, this research has succeeded in the analysis of the development process of a spherical multi-touch application, while at the same time producing a prototype application that meets all requirements of a spherical multi-touch application and can therefore serve as a reference for future implementations.

5.1 Future work

The user survey carried out to test the intuitiveness of the prototype application revealed that a number of gestures users expect to be able to use on a spherical multi-touch display have not been developed in this research. Mainly, research should be carried out into the recognition of gestures that are based on the entire hand instead of separate fingers and the possibility of rotating the sphere with momentum. The prototype application in particular could be made more useful by adding the possibility of tapping on a point on the map to receive more information.

A second point that requires more research is the projection used when zooming in. In this research, zooming in causes a “hard” transition to a projection based on the flat Mercator map. If a zoom
projection is created that uses a spherical version of the map as a basis and only adapts its curvature to project it onto the display; zooming in becomes seamless and the dragging and rotating gestures no longer need to work differently when zoomed in.


Image acknowledgement

Figure 1.1: taken from [4] (figure 3).
Figure 2.1: taken from http://tinyurl.com/fig2-1.
Figure 3.1: drawing by the author, partially based on http://tinyurl.com/fig3-1a and http://tinyurl.com/fig3-1b.
Figure 3.2: drawing by the author, based on [4] (figure 8) and partially on http://tinyurl.com/fig3-2a and http://tinyurl.com/fig3-2b.
Figure 3.3 and 3.4: drawings by the author, based on figure 3.2.
Figure 4.1: drawing by the author.
User survey results

<table>
<thead>
<tr>
<th>Gestures tried before receiving instructions</th>
<th>Cédric</th>
<th>Wouter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lat drag</td>
<td>Correct one-finger gesture.</td>
<td>Correct one-finger gesture.</td>
</tr>
<tr>
<td>Lon drag</td>
<td>Correct one-finger gesture.</td>
<td>Correct one-finger gesture.</td>
</tr>
<tr>
<td>Rotate</td>
<td>First moves one finger along the edge of the display, which has the same effect, then tries the regular two-finger gesture.</td>
<td>Correct two-finger gesture.</td>
</tr>
<tr>
<td>Zoom in</td>
<td>Correct two-finger gesture.</td>
<td>Correct two-finger gesture.</td>
</tr>
<tr>
<td>Zoom out</td>
<td>Correct two-finger gesture.</td>
<td>Unsuccessfully tries tapping on different points on the sphere.</td>
</tr>
<tr>
<td>Menu</td>
<td>Because the animation shows the menu appearing at the edges of the display: tries to drag the menu out of the edges by placing two fingers on different points on the edge and moving them in the direction of the display’s centre. This is unsuccessful as the sphere interprets this as a zoom out gesture while the sphere is already fully zoomed out. Second attempt: correct holding gesture. Correctly selects a theme from the menu and correctly closes the menu by tapping outside of it.</td>
<td>Second attempt: correct holding gesture. Correctly selects a theme from the menu and correctly closes the menu by tapping outside of it.</td>
</tr>
</tbody>
</table>

The user survey results are continued on the next page.
<table>
<thead>
<tr>
<th>Gestures tried before receiving instructions</th>
<th>Jelle</th>
<th>Jasper</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lat drag</td>
<td>Correct one-finger gesture.</td>
<td>Correct one-finger gesture.</td>
</tr>
<tr>
<td>Lon drag</td>
<td>Correct one-finger gesture.</td>
<td>Correct one-finger gesture.</td>
</tr>
<tr>
<td>Rotate</td>
<td>Correct two-finger gesture.</td>
<td>Correct two-finger gesture.</td>
</tr>
<tr>
<td>Zoom in</td>
<td>Correct two-finger gesture.</td>
<td>Correct two-finger gesture.</td>
</tr>
<tr>
<td>Zoom out</td>
<td>Correct two-finger gesture.</td>
<td>Correct two-finger gesture.</td>
</tr>
<tr>
<td>Menu</td>
<td>Unsuccessfully tries double tapping. Second attempt: correct holding gesture. Correctly selects a theme from the menu and correctly closes the menu by tapping outside of it.</td>
<td>Unsuccessfully tries double tapping. Does not succeed in opening the menu.</td>
</tr>
</tbody>
</table>