WebMGA 3.0

Refinement of an Interactive Viewer for Coarse-Grained Liquid Crystal Models

Joe Down[1]

MEng Computer Science

Supervisor: Guido Germano

Submission Date: 26th April 2024

[1]**Disclaimer:** This report is submitted as part requirement for the MEng in Computer Science at UCL. It is substantially the result of my own work except where explicitly indicated in the text. The report may be freely copied and distributed provided the source is explicitly acknowledged.

Abstract

TODO

Contents

Chapter 1

Introduction

Chapter 2

Context

2.1 Molecular Simulation Visualisation

The ability to visualise outputs from molecular simulations, particularly in the domain of liquid crystals, is important for understanding and communicating findings. QMGA[1] is a tool which can be used to generate 3D graphical representations of molecular configurations. Despite being unmaintained since 2009[2], it remains in active use to this day, having been used within the last year in publications by notable authors such as Ramírez González and Cinacchi [3] and Mazzilli, Satoh, and Saielli [4]. While another visualisation tool exists within the liquid crystal domain, LCview[5, 6], it produces plots of director and/or potential fields, rather than showing the structure of large multi-molecule system.

Since QMGA has not been updated in so long, it continues to depend on the severely dated Qt 3 framework (Qt 4 was released in 2005, 2 years before QMGA was released) requiring the installation of unmaintained and difficult to acquire libraries (e.g. the Debian Linux distribution removed all Qt 3 libraries in 2012[7]). Additionally, since the program is distributed as source code, it must be manually built by the user which is not trivial for inexperienced users. This is complicated further by the fact that modern C compilers will fail without certain modifications to the source code CITE THIS!!!. All of these problems make installation on a modern system a significant barrier to usage.

WebMGA is a project begun by Battistini [8] in 2021 which aims to address this accessibility issue whilst replicating the functionality of QMGA. It was continued in 2023 by CITE YUE. It's written in JavaScript using the React framework with the three.js library for 3D rendering. This addresses the accessibility issues of QMGA since it can be easily accessed using just a web browser. While WebMGA contains full functionality for rendering most molecule configurations from QMGA, it still has performance and functionality limitations, as well as some bugs. WebMGA 3.0 aims to address a majority of these issues.

2.2 Liquid Crystal Modelling

Chapter 3

Requirements and Analysis

TODO

Chapter 4

Design and Implementation

4.1 Director

4.1.1 Defintion

TODO

4.1.2 Colour From Director

TODO

4.2 Axes

4.2.1 WebMGA 2.0 Implementation

In WebMGA 2.0, the 3D axes are displayed as shown in Figures 4.1a and 4.1b, and controlled through the user interface as shown in Figure 4.1d (visibility and colour toggles).

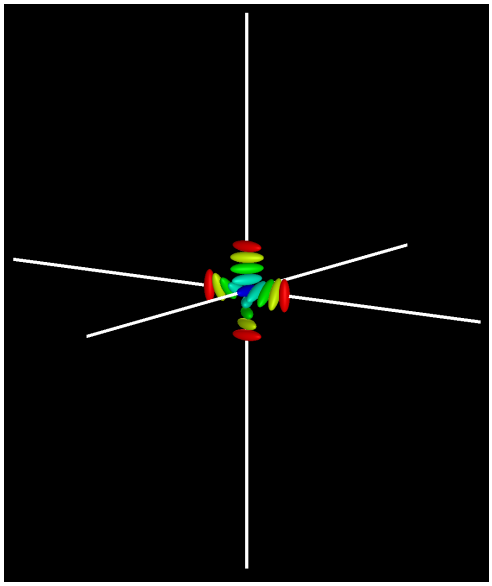
Axes take the form of three lines of fixed lengths in the x, y, and z directions. Each line's midpoint is the world coordinate $(0, 0, 0)$, where all axes meet. Axes extend in both positive and negative directions. They are not shown by default and, when first enabled, are uncoloured. When coloured, the x axis is red, the y axis is green, and the z axis is blue.

Visibility is toggled using the "Show" button and colour is toggled with the "Multi-Colour" button. A question mark icon is next to the "Multi-Colour" which shows a tooltip when hovered specifying the axis colour scheme.

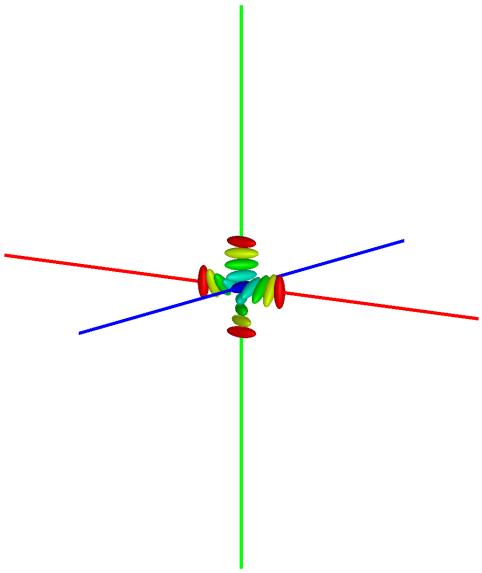
4.2.2 WebMGA 2.0 Bugs

When the axes are toggled to visible for the first time, if the "Multi-Colour" toggle has not been interacted with first, the axes will be uncoloured, despite the "Multi-Colour" toggle being enabled by default. This is shown in Figure 4.2. To enable colour for the first time, the "Multi-Colour" toggle must be disabled and then re-enabled.

When the environment is set to light mode (white background) with coloured axes disabled, the axes become difficult to view since they retain a white colour as default, blending into the background as shown in Figure 4.1c.

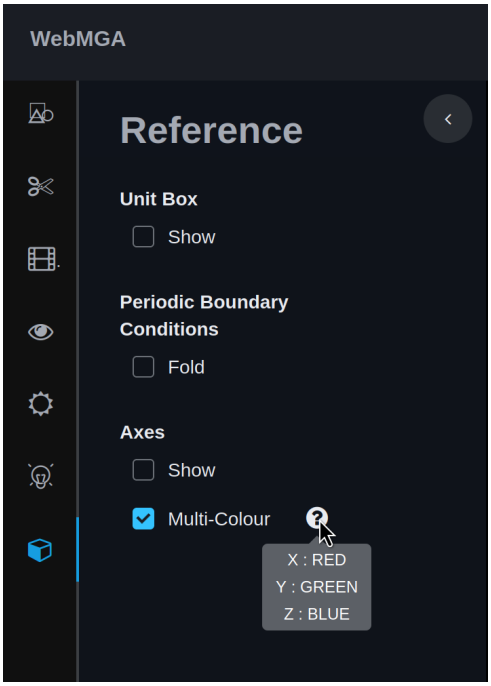


(a) Colour disabled



(b) Colour enabled

(c) Colour disabled (light background)



(d) GUI controls

Figure 4.1: Axes in WebMGA 2.0

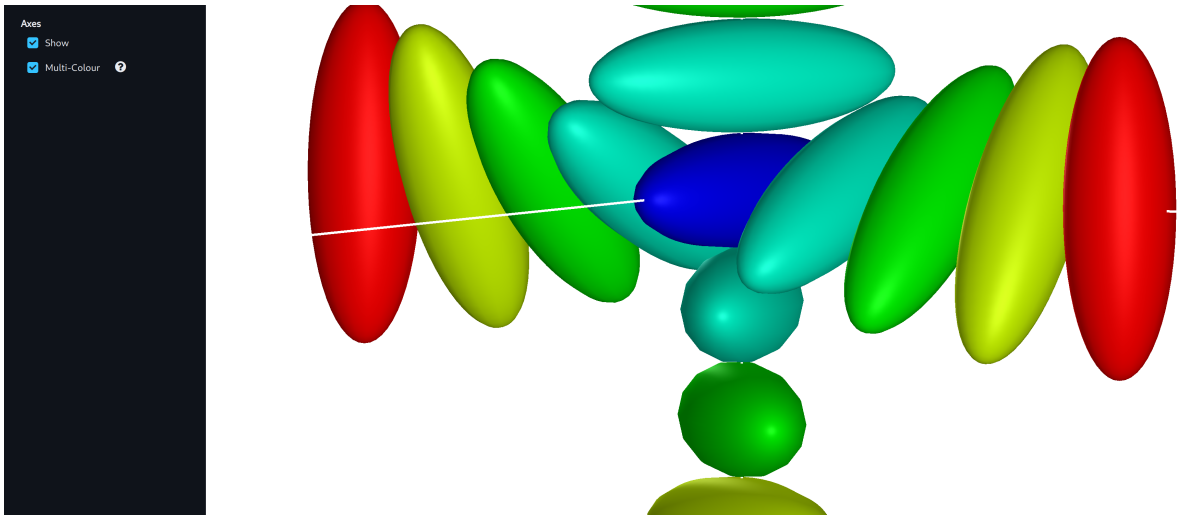


Figure 4.2: Bug where axes are not coloured despite the "Multi-Colour" toggle being enabled when axes are first enabled

4.2.3 Improvement Goals

- Axes are unlabelled
  - Axes should be changed to extend only in the positive direction
- Director(TODO label) is not shown
  - An additional line should be shown indicating director direction
- Colours should be labelled or meaningful
  - Colour axes according to angle with director (as in TODO REFERENCE)
- Axes should not be obscured
  - Place axes in screen corner rather than centre
- Axes should be clearly distinguishable
  - Ensure axes retain contrast with background under light and dark views

4.2.4 WebMGA 3.0 Implementation

Axes Positions

The existing implementation was found to be needlessly convoluted so was largely stripped out. For example, coloured and uncoloured axes were implemented entirely separately, resulting in a large amount of duplicated code and convoluted logic flow. The bug identified in WebMGA 2.0 regarding uncoloured axes showing with "Multi-Colour" enabled, for example, was found to occur due to incorrect colour object initialisation, meaning what should be "Multi-Colour" axes showed as uncoloured since the colours are not defined when these lines are loaded the very first time.

In the new axes code, they are simply defined in terms of an axes centre point, three axes vectors, and an axis length scale. The axes vectors are handled in world coordinate space so are trivially defines as $x = (1, 0, 0)$, $y = (0, 1, 0)$, and $z = (0, 0, 1)$.

Since the axes centre needs to remain in a fixed position on screen at all times, it needs to be defined relative to the camera. Three.js provides a method on any world object which converts from object relative coordinates to world coordinates, so this is used to trivially place the axes centre into world space as required. Since this relationship changes when an object, in this case the camera, moves, the axes centre must therefore be redefined on any camera movement. This process also does not account for changes to intrinsic camera properties, importantly camera zoom. The axes therefore need to be scaled proportionally to the camera's zoom level on any zoom change.

Using the world space centre point and the axes vectors and scales, axis lines are trivially defined as:

$$l_0 = c \tag{4.1}$$

$$l_1 = c + szv \tag{4.2}$$

Where l_0 and l_1 are the axis line start and end, c is the axes centre, v is the axis vector, s is the axis scale factor, and z is the zoom factor of the camera. These can be recalculated and applied on every camera change. A Three.js Line object is constructed for each axis using the line start, end, and a colour.

Director

Plotting the director is made simple using the above setup. A new axis is simply defined using Equations (4.1) and (4.2), with v set to the already computed director vector (TODO show where this was done).

Axes Colouring

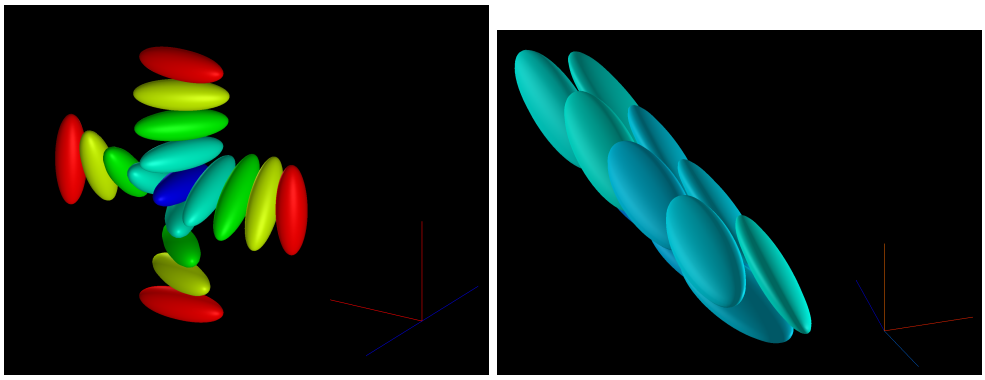
It was decided that a meaningful colouring for the axes lines (including the director) would be using the same colour scheme as for molecule colour (TODO show where). This can be done easily since all axes have a defined direction vector which can be passed to the TODO COLOURFROMDIRECTORNAME function (TODO show where). The resulting colour is simply passed as part of the Line object constructor.

Axes Summary

Some result can be viewed in Figure 4.3.

GUI Implementation

TODO discuss UI



(a) (b)

Figure 4.3: Axes in WebMGA 3.0

4.2.5 WebMGA 3.0 Bugs

TODO

4.3 Shapes

4.3.1 WebMGA 2.0 Implementation

WebMGA 2.0 implements the following molecule shapes:

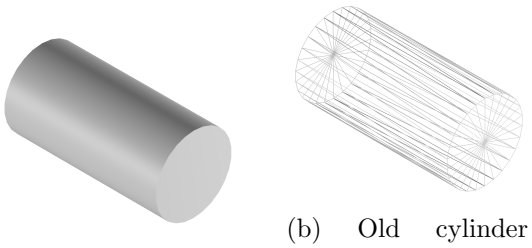
- Sphere (Figure 4.10)
- Ellipsoid (Figure 4.12)
- Spherocylinder (TODO)
- Spheroplatelet (Figure 4.13)
- Cut Sphere (**??**, implemented as a double cut sphere)
- Cylinder (Figure 4.4)
- Torus (Figure 4.5)

Notably missing but useful are the regular cut sphere, the spherical cap, and the lens. The cylinder and torus shapes are present since the three.js library provides easily callable predefined meshes, however serve little practical purpose since no realistic molecular configuration would model using these.

4.3.2 WebMGA 2.0 Bugs

4.3.3 Improvement Goals

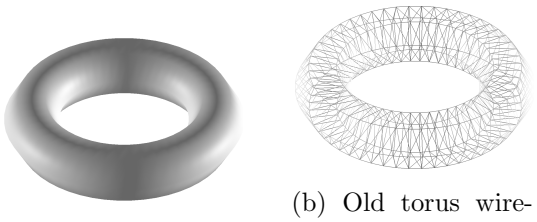
- Problem
  - Fix



(a) Old cylinder

(b) Old cylinder wireframe

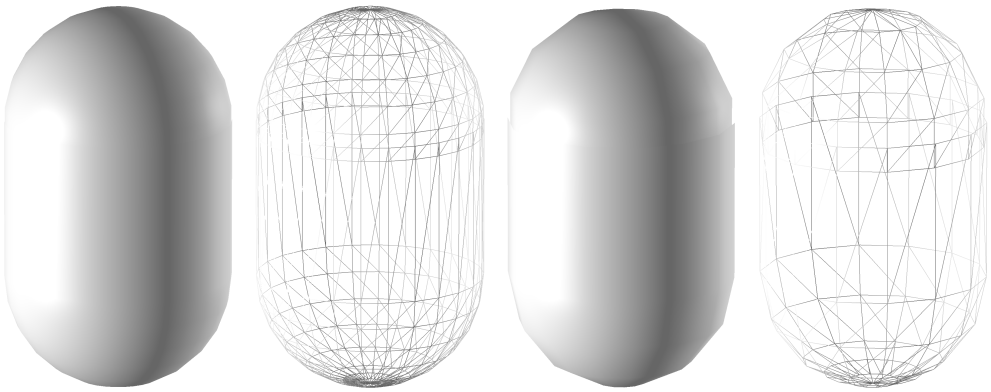
Figure 4.4: Cylinder



(a) Old torus

(b) Old torus wireframe

Figure 4.5: Torus

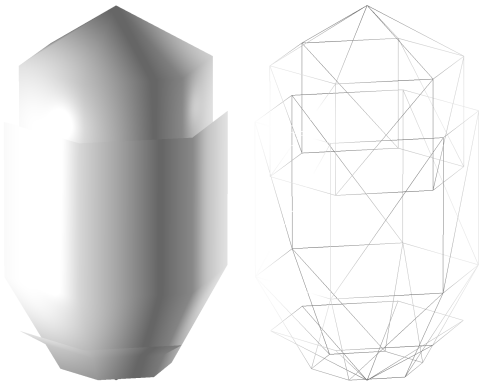


(a) WebMGA 2.0 high detail shape

(b) WebMGA 2.0 high detail mesh

(c) WebMGA 2.0 medium detail shape

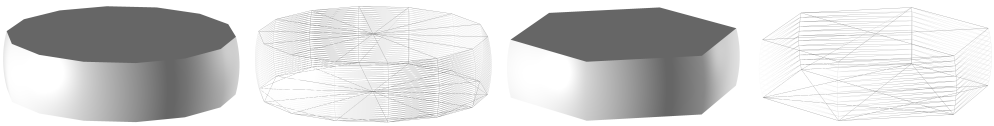
(d) WebMGA 2.0 medium detail mesh



(e) WebMGA 2.0 low detail shape

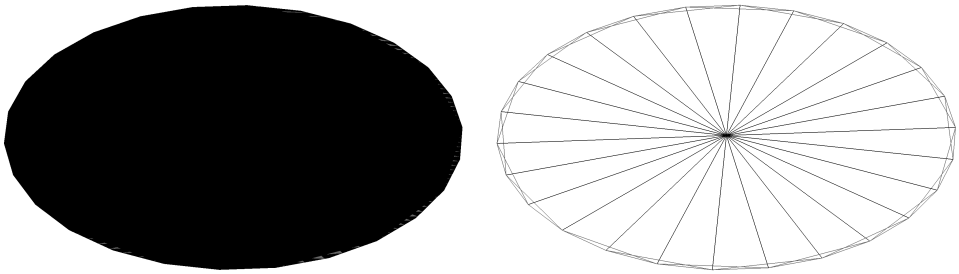
(f) WebMGA 2.0 low detail mesh

Figure 4.6: Bad spheocylinder mesh generated by WebMGA 2.0

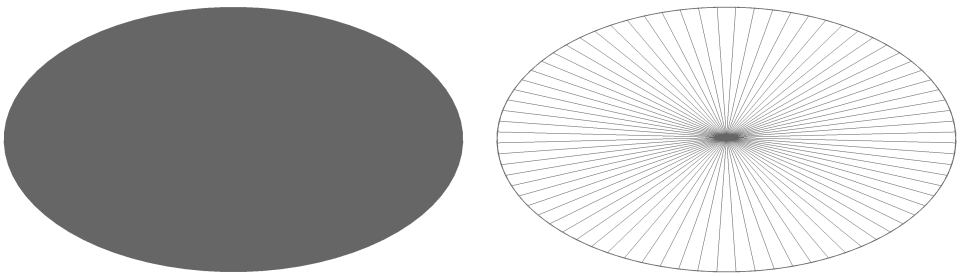


(a) WebMGA 2.0 (b) WebMGA 2.0 (c) WebMGA 2.0 (d) WebMGA 2.0

Figure 4.7: Notably higher mesh quality vertically for double cut sphere with WebMGA 2.0



(a) WebMGA 2.0 Shape (b) WebMGA 2.0 Wireframe



(c) WebMGA 3.0 Shape (d) WebMGA 3.0 Wireframe

Figure 4.8: Buggy shape representation when double cut sphere has 0 height

4.3.4 WebMGA 3.0 Implementation

Sphere

Key to the new shape implementations is the implementation for the sphere (Figure 4.10). The sphere mesh is generated by sampling points across the sphere's surface in such a way as to split it into a finite number of flat, triangular sub-faces as shown in Figure 4.9. This sampling is performed with the spherical coordinates system for some specified sphere radius (r) and a set of azimuthal (θ) and a set of polar (ϕ) angles derived from the number of points required, converted to an equivalent Cartesian form.

$$p_{spherical} = (r, \phi, \theta) \tag{4.3}$$

$$p_{cartesian} = \begin{bmatrix} r \sin\phi \cos\theta \\ r \sin\phi \sin\theta \\ r \cos\phi \end{bmatrix} \tag{4.4}$$

Any unique point on the origin centred r sphere can be uniquely defined by some θ,ϕ pair. Therefore, to evenly space points across the surface, a set of θs and ϕs is generated by taking n (essentially a measure of mesh quality) evenly spaced values over the interval of a full circular rotation ($[0, 2\pi)$). Each unique pairing of ϕ and γ, along with r, is used to produce the full set of Cartesian vertices using Equation (4.4). This method is sufficient to produce a sphere mesh as in Figure 4.10a from WebMGA 2.0. The sampling is modified slightly for WebMGA 3.0 to produce a mesh as in Figure 4.10c TODO FILL THIS BIT IN, ALSO DISCUSS ORDERING, ALSO DISCUSS HOW CHANGING SPHER EGENERATION TO BE MORE EFFICIENT SHOULD TRICKLE DOWN TO ALL OTHER METHODS TOO

Some optimisations can be implemented to efficiently generate a full set of vertices while sampling only $\frac{1}{4}$ of the points around the sphere's surface. This uses the fact that the origin centred sphere is symmetrical in each of the x, y, and z planes. TODO FINISH THIS BIT

TODO FACE GENERATION

Ellipsoid

The ellipsoid shape (Figure 4.12) can be represented as an origin centred sphere of radius 1 scaled in the x,y and z directions by some scalar value in each direction. This can be represented by a slightly modified form of the Cartesian sphere equation in Equation (4.4):

$$p_{ellipsoid} = \begin{bmatrix} scale_x \times r \sin\phi \cos\theta \\ scale_y \times r \sin\phi \sin\theta \\ scale_z \times r \cos\theta \end{bmatrix} = scale \odot p_{cartesian} \tag{4.5}$$

From this formulation, it can be seen that an ellipsoid can be generated by slightly modifying the vertex sampling process for a sphere, whilst leaving the rest of the mesh building process unchanged. A sphere point can be sampled using Equation (4.4) with radius 1 and then multiplied

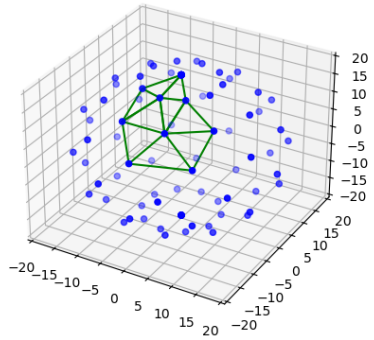
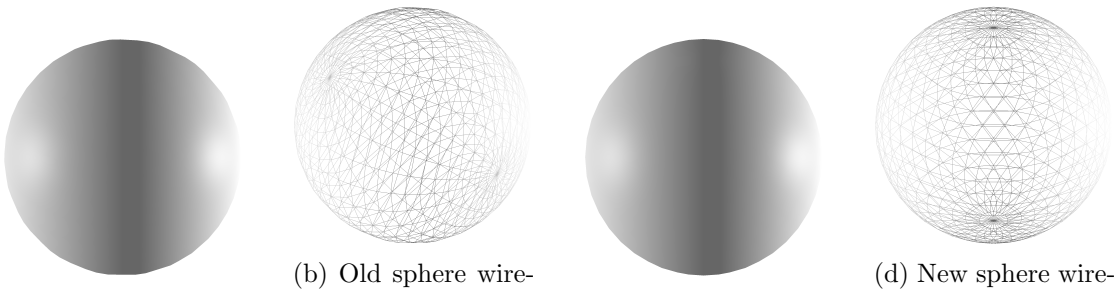


Figure 4.9: Example sphere vertex distribution (9×10 vertical×horizontal samples). Some mesh edges shown to demonstrate mesh construction from vertices.



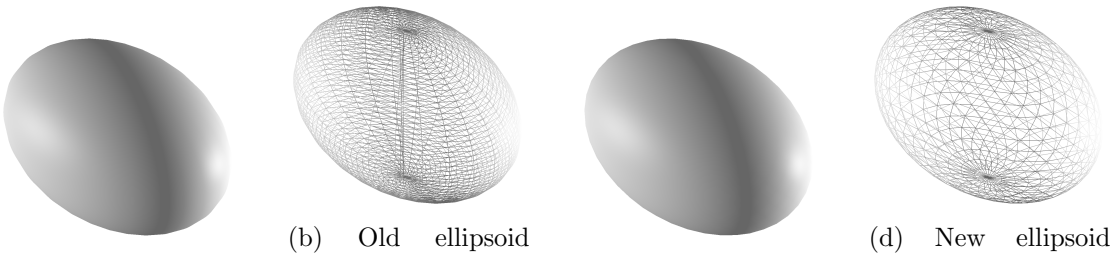
(a) Old sphere

(b) Old sphere wireframe

(c) New sphere

(d) New sphere wireframe

Figure 4.10: Sphere molecule mesh implementation



(a) Old ellipsoid

(b) Old ellipsoid wireframe

(c) New ellipsoid

(d) New ellipsoid wireframe

Figure 4.11: Ellipsoid molecule mesh implementation

by the scaling vector $\begin{bmatrix} scale_x \\ scale_y \\ scale_z \end{bmatrix}$ to give an equivalent result to Equation (4.5).

In the program this is implemented by creating an "Ellipsoid" class as a child of the "Sphere" class and overriding the "sample_sphere()" method. Since this implementation is so simple, the JavaScript code is provided below:

```
//Ellipsoid mesh generator
export class Ellipsoid extends Sphere {
    //Scale factor in [x, y, z] directions
    scale: number[];

    constructor(x: number, y: number, z: number) {
        //Derive from origin centred sphere of radius 1
        super(1);
        this.scale = [x, y, z];
    }

    //Samples from ellipsoid instead of sphere
    sample_sphere(radius: number, theta: number, phi: number): number[] {
        //Multiply origin centred sphere coordinates by scale vector
        return math.dotMultiply(super.sample_sphere(radius, theta, phi), this.sc
    }
}
```

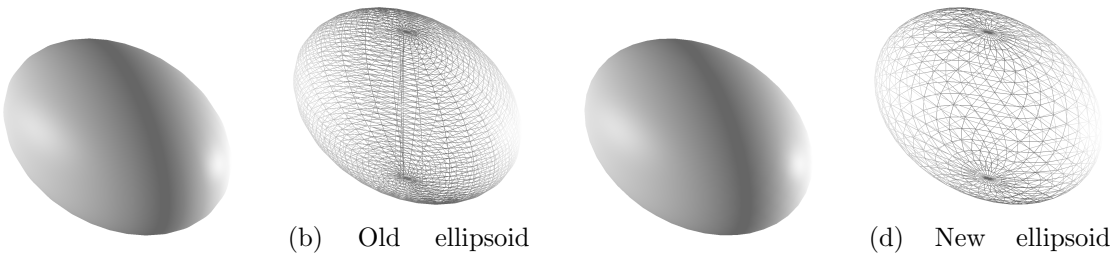
Spherocylinder

The spherocylinder shape (TODO REFERENCE) can be represented as an origin centred sphere of radius r scaled in the z directions by (half of) some length value in each z direction (positive/negative). This can be represented by a slightly modified form of the Cartesian sphere equation in Equation (4.4):

$$p_{ellipsoid} = \begin{bmatrix} r\sin\phi\cos\theta \\ r\sin\phi\sin\theta \\ r\cos\theta + n \end{bmatrix} = p_{cartesian} + \begin{bmatrix} 0 \\ 0 \\ n \end{bmatrix} \tag{4.6}$$

$$n = \begin{cases} \frac{\text{length}}{2} & \text{if } r\cos\theta > 0 \\ -\frac{\text{length}}{2} & \text{if } r\cos\theta < 0 \\ 0 & \text{otherwise} \end{cases} \tag{4.7}$$

From this formulation, it can be seen that a spherocylinder can be generated by slightly modifying the vertex sampling process for a sphere, whilst leaving the rest of the mesh building process unchanged. A sphere point can be sampled using Equation (4.4) with radius r and



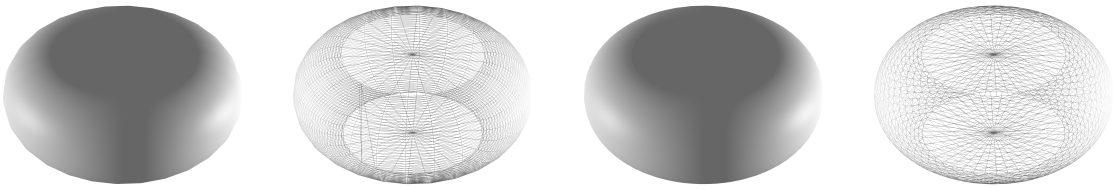
(a) Old ellipsoid (b) Old ellipsoid wireframe (c) New ellipsoid (d) New ellipsoid wireframe

Figure 4.12: Ellipsoid molecule mesh implementation

added to the scaling vector $\begin{bmatrix} 0 \\ 0 \\ n \end{bmatrix}$ as defined in Equation (4.7) to give an equivalent result to Equation (4.6).

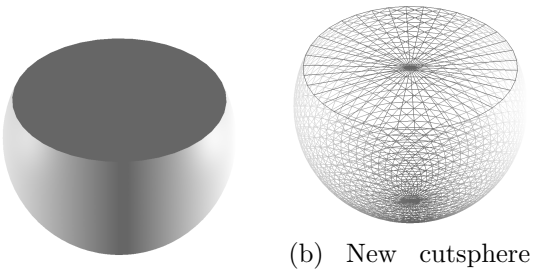
In the program this is implemented by creating a "Spherocylinder" class as a child of the "Sphere" class and overriding the "sample_sphere()" method. Since this implementation is so simple, the JavaScript code is provided below:

```
//Spherocylinder mesh generator
export class Spherocylinder extends Sphere {
    //Scaling vector (either side of centre) to stretch sphere into spherocylind
    length_scaling_vector: number[];

    constructor(radius: number, length: number) {
        //Derive from origin centred sphere of chosen radius
        super(radius);
        this.length_scaling_vector = [0, 0, length / 2];
    }

    //Samples from spherocylinder instead of sphere
    sample_sphere(radius: number, theta: number, phi: number, epsilon: number =
        let sphere_coordinate: number[] = super.sample_sphere(radius, theta, phi
        //Stretch point in z direction by scale vector, matching stretch directio
        //Unchanged if z is (approximately) 0
        if (Math.abs(sphere_coordinate[2]) < epsilon) {
        } else if (sphere_coordinate[2] > 0) {
            sphere_coordinate = math.add(sphere_coordinate, this.length_scaling_
        } else if (sphere_coordinate[2] < 0) {
            sphere_coordinate = math.subtract(sphere_coordinate, this.length_sca
        }
        return sphere_coordinate;
    }
}
```



(a) Old spheroplatelet (b) Old spheroplatelet wireframe (c) New spheroplatelet (d) New spheroplatelet wireframe

Figure 4.13: Spheroplatelet molecule mesh implementation



(a) New cutsphere (b) New cutsphere wireframe

Figure 4.14: Cutsphere

Spheroplatelet

Cut Sphere

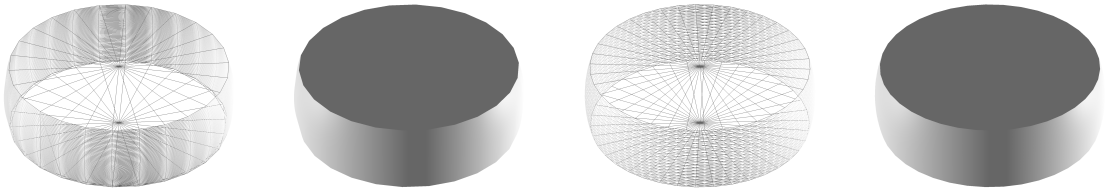
The cut sphere shape (Figure 4.14) is implemented simply by sampling the sphere as before but over a reduced range of ϕ values. Since the sphere will not be completed, an empty circular face is left which can be filled by generating an additional vertex by averaging the coordinates for each vertex on the circle's edge, splitting it into triangles. This can be observed on Figure 4.14b.

The new range of ϕs can be defined as $[\arcsin \frac{c}{r}, \pi)$, where c is the radius of the circle resulting from the cut and r is the radius of the sphere. TODO THIS SHOULD HAVE A DIAGRAM AND BE ELABORATED ON IMPLEMENTATION

Double Cut Sphere

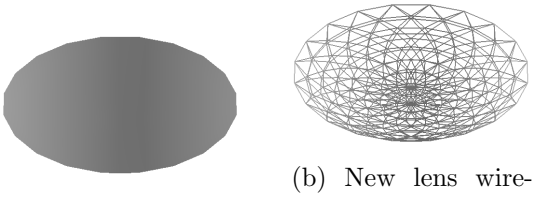
The double cut sphere shape (Figure 4.15) is implemented simply by sampling the sphere as before but over a reduced range of ϕ values. Since the sphere will not be completed, two empty circular faces are left which can be filled by generating two additional vertices by averaging the coordinates for each vertex on the corresponding circle's edge, splitting it into triangles. This can be observed on Figure 4.15d.

The new range of ϕs can be defined as $[\arcsin \frac{c}{r}, \pi - \arcsin \frac{c}{r})$, where c is the radius of the circle resulting from the cut and r is the radius of the sphere. TODO THIS SHOULD HAVE A DIAGRAM AND BE ELABORATED ON IMPLEMENTATION



(a) Old doublecut-sphere (b) Old doublecut-sphere wireframe (c) New doublecut-sphere (d) New doublecut-sphere wireframe

Figure 4.15: Doublecutsphere molecule mesh implementation



(a) New lens (b) New lens wireframe

Figure 4.16: Lens

Cap

The cap shape (**??**) is implemented simply by sampling the sphere as before but over a reduced range of ϕ values. Since the sphere will not be completed, an empty circular face is left which can be filled by generating an additional vertex by averaging the coordinates for each vertex on the circle's edge, splitting it into triangles. This can be observed on **??**.

The new range of ϕs can be defined as $[0, \arcsin \frac{c}{r})$, where c is the radius of the circle resulting from the cut and r is the radius of the sphere. TODO THIS SHOULD HAVE A DIAGRAM AND BE ELABORATED ON IMPLEMENTATION

Lens

4.3.5 WebMGA 3.0 Bugs

TODO

4.4 File types

4.4.1 WebMGA 2.0 Implementation

WebMGA 2.0 supports only its own JSON-based file format as defined in CITEORIGINALDISSERTATION. This could prove an obstacle to users since, in practice, different formats are output when running molecular dynamics simulations (CITE!!).

4.4.2 WebMGA 2.0 Bugs

TODO

Molecule count												
Unit box X length (lx)												
Unit box Y length (ly)												
Unit box Z length (lz)												
Not used	Not used											
Position (rx)	Position (ry)	Position (rz)	Velocity (vx)	Velocity (vy)	Velocity (vz)	Orientation (ex)	Orientation (ey)	Orientation (ez)	Orientational velocity (ux)	Orientational velocity (uy)	Orientational velocity (uz)	Molecule ID
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

Table 4.1: LAMMPS format molecule configuration

Unit box X half length ($lx/2$)	Unit box Y half length ($ly/2$)	Unit box Z half length ($lz/2$)				
Shape parameter	Position X (rx)	Position Y (ry)	Position Z (rz)	Orientation X (ex)	Orientation Y (ey)	Orientation Z (ez)
⋮	⋮	⋮	⋮	⋮	⋮	⋮

Table 4.2: Cinacchi format molecule configuration

4.4.3 WebMGA 3.0 Implementation

WebMGA 3.0 implements two new file formats for defining molecular configurations as defined below.

LAMMPS Format (.cnf)

LAMMPS is a molecular dynamics simulator typically used on highly parallel computers[9]. It uses a specifically designed file format to represent molecular configurations to allow the highest possible performance while preserving some amount of human readability.

Table 4.1 shows the structure of a file of this format. Rows represent lines in the file. Each value is represented by a signed float of format -1.000000, where digits before the decimal are omitted if not present. Values are separated by spaces, padded to align decimal points.

For WebMGA 3.0, a parser script was written in JavaScript which builds a WebMGA JSON configuration from the ".cnf" file provided. Specifically, a unit box is constructed from (lx, ly, lz), and molecule positions and orientations are obtained from corresponding pairs of (rx, ry, rz) and (ex, ey, ez) for some molecule id. All other parameters are dropped since they aren't used by WebMGA. Molecules are ordered in an array according to their id.

Cinacchi Format (.qmga)

TODO WRITE THIS TODO CHECK LETTERS USED FOR ROTATION ETC

See Table 4.2 for the structure of a file of this format. Rows represent lines in the file. Each value is represented by a signed float of format -1.00000000, where digits before the decimal are omitted if not present. Values are separated by spaces, padded to align decimal points.

For WebMGA 3.0, a parser script was written in JavaScript which builds a WebMGA JSON configuration from the ".qmga" file provided. Specifically, a unit box is constructed from (lx, ly, lz), and molecule positions and orientations are obtained from corresponding pairs of (rx, ry, rz) and (ex, ey, ez). The shape parameter is dropped since molecule shape is not defined by the file. Molecules are ordered in an array as they are encountered.

4.4.4 WebMGA 3.0 Bugs

WebMGA ignores the shape parameter from the ".qmga" format configuration. Since some shapes in WebMGA require multiple parameters, and the shape to use is not defined within the

file, I could not see a sensible way to automate applying this. The user must manually enter this value after selecting a molecule shape in the "Models" menu. This is not ideal since a user should expect their configuration to appear correctly as soon as they load the file.

4.5 Optimisations

4.5.1 WebMGA 2.0 Implementation

TODO

4.5.2 WebMGA 2.0 Bugs

TODO

4.5.3 WebMGA 3.0 Implementation

Discrete Levels of Detail

4.5.4 WebMGA 3.0 Bugs

TODO

Chapter 5

Analysis and Testing

Chapter 6

Conclusions and Evaluation

6.1 Achievements

Summarise the achievements to confirm the project goals have been met.

6.2 Evaluation

Evaluation of the work (this may be in a separate chapter if there is substantial evaluation).

6.3 Future Work

Bibliography

[1] Adrian T Gabriel, Timm Meyer, and Guido Germano. Molecular graphics of convex body fluids. *Journal of Chemical Theory and Computation*, 4(3):468–476, 2008.

[2] URL: `https://sourceforge.net/projects/qmga/files/qmga/`.

[3] Juan Pedro Ramírez González and Giorgio Cinacchi. Densest-known packings and phase behavior of hard spherical capsids. *The Journal of Chemical Physics*, 159(4), 2023.

[4] Valerio Mazzilli, Katsuhiko Satoh, and Giacomo Saielli. Phase behaviour of mixtures of charged soft disks and spheres. *Soft Matter*, 19(18):3311–3324, 2023.

[5] Richard James, Eero Willman, FA FernandezFernandez, and Sally E Day. Finite-element modeling of liquid-crystal hydrodynamics with a variable degree of order. *IEEE Transactions on Electron Devices*, 53(7):1575–1582, 2006.

[6] URL: `https://www.ee.ucl.ac.uk/~afernand/rjames/modelling/visualisation/`.

[7] URL: `https://wiki.debian.org/qt3-x11-freeRemoval`.

[8] Eduardo Battistini. Webmga, 2021. URL: `https://students.cs.ucl.ac.uk/2019/group3/WebMGA/diss.pdf`.

[9] Aidan P Thompson, H Metin Aktulga, Richard Berger, Dan S Bolintineanu, W Michael Brown, Paul S Crozier, Pieter J In't Veld, Axel Kohlmeyer, Stan G Moore, Trung Dac Nguyen, et al. Lammps-a flexible simulation tool for particle-based materials modeling at the atomic, meso, and continuum scales. *Computer Physics Communications*, 271:108171, 2022.

Appendix A

Appendices