Visualisation of Resource Flows
Technical Report

Jesper Karjalainen (jeska043) Erik Johansson (erijo926)
Anders Kettisen (andke020) Tobias Nilsson (tobni908)
Alexander Eriksson (aleer034)

Contents

Glossary

Emscripten Emscripten is a toolchain for compiling to asm.js and WebAssembly, built using LLVM, that lets you run C and C++ on the web at near-native speed without plugins. Usually also refers to the Emscripten Software Development Toolkit (EMSDK), a complementary set of tools to build c++ apps on the web.

GLSL GL Shading Language. A programming language similar in syntax to C that is executed on the GPU in the different render stages.

OpenGL Open Graphics Library, a cross-language, cross-platform application programming interface (API) for rendering 2D and 3D vector graphics..

SVG Scalable Vector Graphics. A language for describing two-dimensional graphics. SVG allows for three types of graphic objects: vector graphic shapes (e.g., paths consisting of straight lines and curves), images and text. Graphical objects can be grouped, styled, transformed and composited..

1. Introduction

Visualization systems for the import and export of different resources are used as a tool by researchers in climate science. These applications are usually limited to a 2D view and viewing one specific resource at a time.

The goal of this project is to improve upon these existing applications by creating a 3D version able to visualize multiple resources at a time. In addition to just resources, indices such as GDP will be displayed using a choropleth map. This combined visualization of both resources and indices aims to further aid the researchers, allowing them to come to better conclusions.

1.1. Problem description

The project will investigate and implement methods for the visualization of import and export data between countries, called trade flows, onto a model of the earth rendered in 3D. This will result in a web-based application in accordance with the requirement specification.

1.2. Tools

The application source code is written in C++ and JavaScript, using the Emscripten toolchain and the Emscripten Software Development Kit (EMSDK). EMSDK allows for easy porting of OpenGL and standard library routines to the web from C++ source. The EMSDK compiler, emcc, compiles C++ code to WebAssembly and JavaScript glue code, allowing modules in C++ to be run in a browser environment. The application is mostly written by the project developers, but it has some dependencies.

Table 1. Software dependencies of the application.

lodepng	PNG file loading utility in C/C++
rapidxml	Generic XML-document parser in C++
glfw	Window and context library for debug
glew	GL extension loader library for debug
VectorUtilities	Linear algebra toolbox
GL_utilities	Utility library for common OpenGL operations
jQuery	Standard JavaScript library for extended GUI functionality

All of the dependencies listed in table 1 are open source software.

2. Background/Theory

This section will highlight important background information and the essential theory needed to do the project.

2.1. Sphere models

There are two common ways of creating a sphere in computer graphics. The first way is to compute vertical and horizontal lines, creating surface elements defined by four sides. These quadrangles can then be subdivided into triangles.

This sphere model is called the UV sphere, shown in figure 1.

The problem with this model is that the polygons around the equator are much bigger than the polygons around the poles, creating an uneven distribution of vertices.

The second model is called the icosphere, which originates from a so called icosahedron, shown in figure 2.

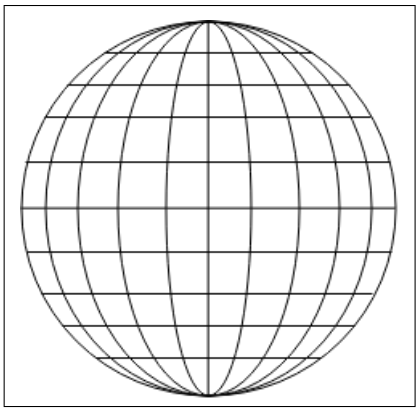

Figure 1. The structure of a UV sphere.

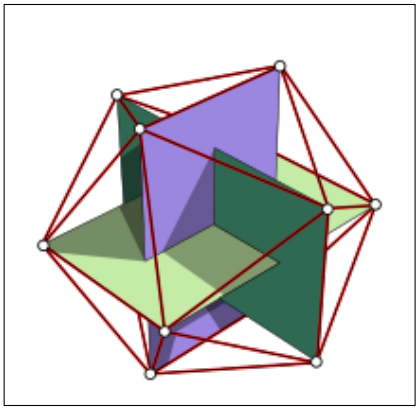

Figure 2. An icosahedron and the golden rectangles defining its vertices [1].

The icosahedron consists of 20 equilateral triangles, defined by a total of 12 vertices. Each vertex is surrounded by five neighbouring triangles.

The vertices can be seen as the corners of three orthogonal *golden* rectangles intersecting each other [1]. In other words, the ratio between the long side and the short side of each rectangle is the golden ratio (equation 1).

$$\phi = \frac{1+\sqrt{5}}{2} \tag{1}$$

This property can be used to find the 12 vertices, and this procedure will be described further in section 3.4.1.

To create a better approximation of a sphere, the icosahedron can be subdivided into smaller triangles, creating a so called icosphere. This process will be described in detail under section 3.4.2.

The icosphere does not have the problem of unevenly distributed vertices, so the ”resolution” is the same wherever you are on the sphere. And all polygons have the same area.

2.2. Vector Graphics

Vector Graphics is a type of image that, instead of containing pixel values, contains commands or mathematical statements to convey information forming an image.

Vector graphics have many use cases, but one of the more interesting ones for this project is that country borders can be stored in a geometric file like this.

One such file format is the Scalable Vector Graphics (SVG) format [2], which contains what is known as ”paths”, strings of characters containing information that is read as sets of 4 points forming a cubic bezier (it can store simpler shapes and other data than paths, but these are of little use to this project).

These paths are used to to draw different vector graphics containing map data. Open source examples can be found at for example [3].

2.3. Cubic Beziers

Bezier curves have many interesting properties, but the most useful property for an application looking to draw bezier curves is the one described by De Casteljou’s algorithm.

The property is described in [4], and can be used to find points on a Bezier curve through subdivision by simply calculating the middle points between the control points of the Bezier curve, as shown in figure 3.

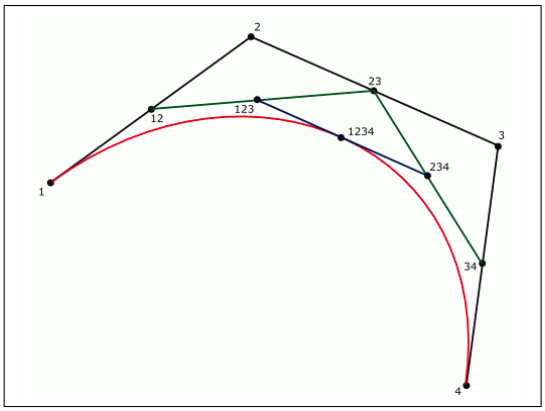

Figure 3. Subdividing a cubic Bezier curve by its midpoints. [5]

2.4. Cartesian Mapping of Earth

When working with coordinate systems of the earth it is often a choice of working with different projections. The equirectangular projection is an especially useful projection for the case of computing points on a plane corresponding points on a rectangle.

$$x = (\lambda - \lambda_0)cos(\phi_1) \tag{2}$$

$$y = \phi - \phi_1 \tag{3}$$

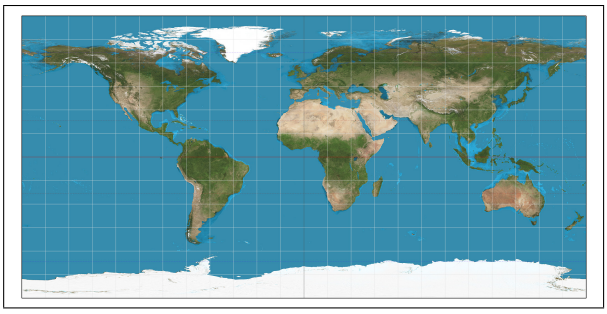
Figure 5. Equirectangular projection of Earth. Also known as the Mercator Projection.

Equations 2 and 3 describe the forward mapping from the earth as a sphere into a rectangle with coordinates x, y shown in figure 5. λ is the longitude of the projection location, ϕ the latitude. ϕ_1 is the standard parallels from south pole to north pole where the scale of the projection is true. λ_0 is the central meridian of the map.

2.5. Lines in 3D

The OpenGL ES 2.0 and by extension The WebGL 1 API is highly optimized to draw triangles but it does have functionality to draw lines with the *GL_Line* primitive. However, this implementation suffers from some problems that give disappointing visual results.

Many current browsers rarely implement the GL directives such as *glLineWidth* with any other option than 1.0, i.e lines can only have a width of 1 pixel on the screen.

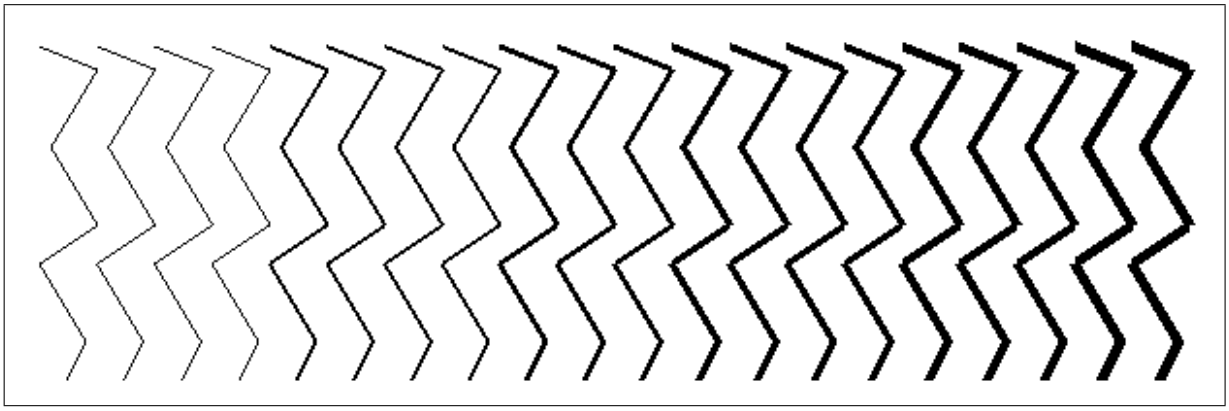
Figure 4. A set of rendered line primitives, displaying the disconnect and aliasing problems present. [6]

No anti-aliasing or smoothing is used in the draw implementation which means the line looks jagged and bumpy. The draw call further makes no attempts to join the line segments, meaning there is a clear disconnect.

Figure 4 displays all of these problems with a set of test lines.

2.6. Rendering Polygons

The common problem of rendering non convex polygons by their contour have many solutions. This documentation presents some of these ideas relevant to the project.

2.6.1 Splitting polygons

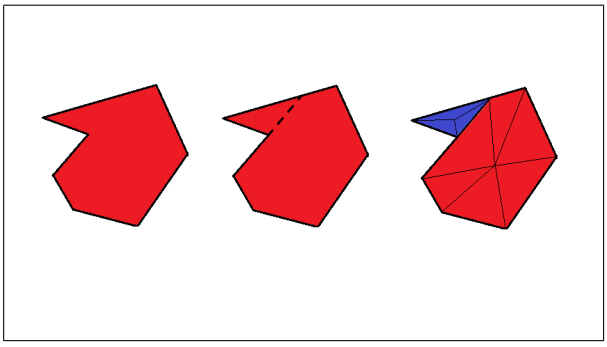

Figure 6. The basic idea of polygon splitting.

To generate a mesh from polygons, the polygons first have to be split up into several convex parts, were they initially are concave. This is a limitation of OpenGL, which prefers the triangle primitive, and triangulation of concave polygons is best formulated by splitting them. The basic method is shown in figure 6.

Every polygon that is not convex is split into several parts [4]. Figure 6 shows that at every turn that is concave, the polygon is split into two new, smaller polygons. This is done until the original polygon is split up into only convex polygons.

When this is done, each convex part of the polygon is split into many small triangles, based on a center of mass point. The algorithms used in the splitting procedure are the following.

- The signed area of a polygon:

$$\frac{1}{2}\sum_{i}^{n}(x_i y_{i+1} - x_{i+1} y_i) \tag{4}$$

The sign of this sum tells if the polygon goes clockwise or counter clockwise.

- Crossproduct between two following lines:

$$(x_1 y_2 - x_2 y_1)\hat{z} \tag{5}$$

Here the sign of the z component tells if the turn between two lines is clockwise or counter clockwise.

- Intersection between two lines:

$$t = \frac{(x_1-x_3)(y_3-y_4)-(y_1-y_3)(x_3-x_4)}{(x_1-x_2)(y_3-y_4)-(y_1-y_2)(x_3-x_4)} \tag{6}$$

$$u = -\frac{(x_1-x_2)(y_1-y_3)-(y_1-y_2)(x_1-x_3)}{(x_1-x_2)(y_3-y_4)-(y_1-y_2)(x_3-x_4)} \tag{7}$$

Given two lines, one can calculate for which scalar value, t for line 1 and u for line 2, the intersection takes place on each respective line. If $0 \leq t \leq 1$ the point of intersection takes place somewhere on line 1, and if $0 \leq u \leq 1$ the same is true for line 2.

2.6.2 Tessellate polygons

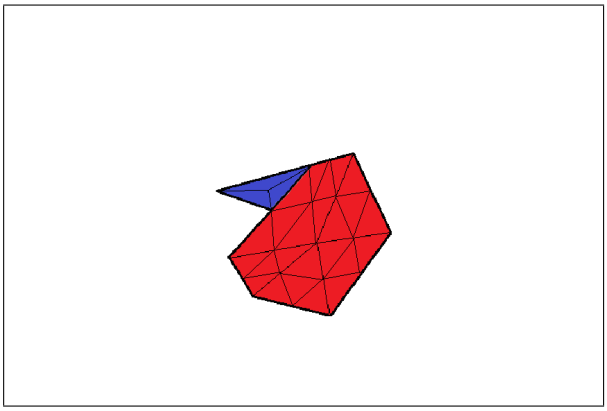

Figure 7. Example of tesselation to smaller polygons.

The splitting is done in a 2D-environment. What is not taken into consideration however, is the fact that the polygons need to be mapped to spherical coordinates on the globe. This causes longer lines between points in polygons to intersect the globes crust.

Therefore, the polygons need to get higher resolution. In this case, this is done by tessellating the triangles that has lines that are longer than a certain threshold.

The idea of this is displayed in figure 7. The triangles in the red polygon are getting tessellated, while the tessellation is not done in the blue polygon, when comparing figure 7 with figure 6.

Each triangle that has a line that is larger than the threshold is split up into 4 new smaller triangles, by connecting each of the original triangles sides middle points.

2.6.3 Filling Polygon by the Stencil Method

A completely different way of approaching the problem of rendering arbitrary polygon shapes is the Stencil Buffer method.

The stencil buffer is a (often) binary bitmask of the surface on which drawing is done selectively, dependent on the bit values in the buffer. It supports stencil functions that allow different logic to be applied when rendering onto the stencil buffer.

The method of polygon filling is described in [7], and involves rendering a triangle fan from an arbitrary point onto the contour points of the polygon. If any point in the stencil buffer has been rendered onto an even number of times, the test is failed. If the point has been rendered an odd number of times, it means the point is inside the polygon.

2.7. Picking

Picking is the common problem in computer graphics. Given user input on the screen such as a touch or a mouse-click, the expectation is to be able to interact with the applications world in some way.

While this problem can sometimes be very involved, it can also be remarkably easy to solve. Some methods are described in [4].

The relevant method related to this application is Ray-Casting in model coordinate. This involves transforming a clicked pixel coordinate into model coordinates. However, since the geometry in the application is easily described by function expressions, ray-sphere intersection tests and ray-plane intersection tests can be used instead of the suggested triangle tests to determine where on the model a click was made.

Furthermore, point-in-polygon tests are relevant to determine if a picked point falls within a polygon contour. Since country borders are closed polygons, the use of such a method should be obvious.

In [4] there are 2 suggested methods. First, the odd-even test, a method that simply counts the number of intersections. If the number ends up odd, the point is inside. Second, the zero winding number test, which adds or subtracts 1 depending on the ”winding”, or direction of the contour, to a total sum. If the sum is non-zero, the point is inside.

3. Method

This section will go through the most important steps taken in the project and describe solutions to the different problems.

3.1. File Loading

When it comes to data there are two main categories; trade and index. Trade data contain information about import and export of resources between countries whereas index data correlates

indices and countries, displayed in a choropleth map.

Files are loaded asynchronously through an Emscripten file system API. And in order for the API to know which files are to be loaded it needs their pathways, it cannot just load arbitrary files in a folder. This is solved by specifying which files to read by adding their filenames/pathways to a separate text file.

3.2. Formatting new files

To add new data, a formatted Excel file is saved as a comma-separated value file (.csv), stored in a specific data folder with its filename added to the former mentioned text file with the others.

Regarding formatting, each file requires a separate header at the top where trade or index is specified, name of resource or index, unit if it was trade and lastly which year is concerned.

In the case of a trade file, following its header shown in table 2, it contain ISO 3 codes of countries that import the specified resource along the next row, countries that export along the first column and the amounts in between.

Table 2. Header used in a trade file.

Type:Trade	Name:Soy	Unit:Ton	Year:2018

An index file with its header displayed in table 3 is followed by all ISO 3 codes in the first column and corresponding index values in the second column.

Table 3. Header used in an index file.

Type:Index	Name:GDP	Year:2018

3.3. Data Management

When all files have been loaded, processing of their content begin. This is handled differently depending on if the given file has trade or index data, but ultimately results in the processed content being stored in a data registry. This registry will then contain trade and index information between each and every country in a specialized STL map that can be accessed and then visualized.

3.4. Creating the sphere

When it comes to the choice of sphere model, the icosphere was chosen over the UV sphere, mostly because of the uniform distribution of vertices.

The vertices of the sphere are used in many calculations (for example the cube mapping), so having an evenly distributed set of vertices, the level of detail will be constant across the whole globe.

3.4.1 Defining the first vertices

As described in section 2.1, the icosphere starts off as an icosahedron (thus the name), which then can be subdivided. So the vertices of the icosahedron must be defined to begin with.

If we look at one of the icosahedron's golden rectangles (shown in figure 2) from the side, we get something like figure 8. And to make things easier, the rectangle is centered in the origin, with a distance to each corner equalling to 1.

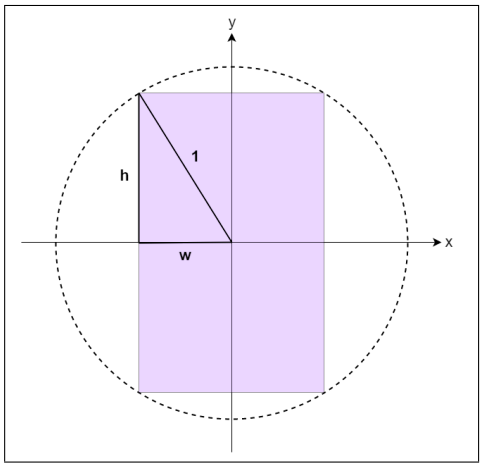

Figure 8. A golden rectangle, touching the unit sphere.

Then we can imagine a unit sphere touching all corners of the rectangle. This is the sphere we want to create (or get close to).

Since we know that the rectangle has the golden ratio (equation 1), we directly get the relationship in equation 8. And also, we get equation 9 from Pythagoras' theorem.

$$\frac{h}{w} = \frac{1+\sqrt{5}}{2} \tag{8}$$

$$h^2 + w^2 = 1 \tag{9}$$

Solving this system of equations, we get the exact dimensions of our rectangle (equations 10 and 11).

$$h = \frac{1+\sqrt{5}}{\sqrt{4+(1+\sqrt{5})^2}} = 0.85065... \tag{10}$$

$$w = \frac{2}{\sqrt{4+(1+\sqrt{5})^2}} = 0.52573... \tag{11}$$

Now, with figure 2 as guidance, the vertices of the icosahedron can easily be determined. And then you just need to define the connectivity of the triangles.

Figure 9 shows the icosahedron with an applied texture.

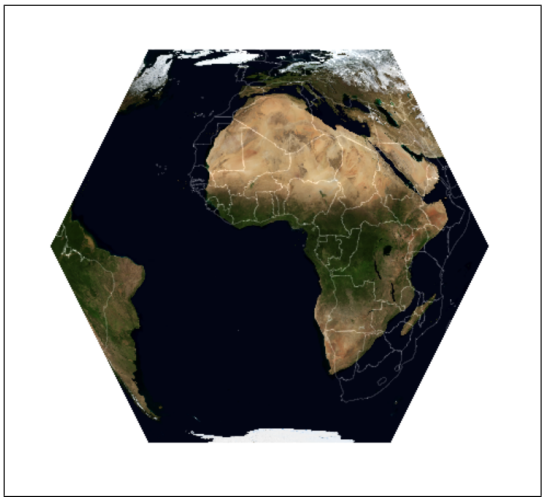

Figure 9. Earth as an icosahedron.

3.4.2 Making the sphere smooth

When the icosahedron is all set up, the next step is to turn it more into a sphere. This is done by subdividing every triangle into four smaller triangles, as seen in figure 10.

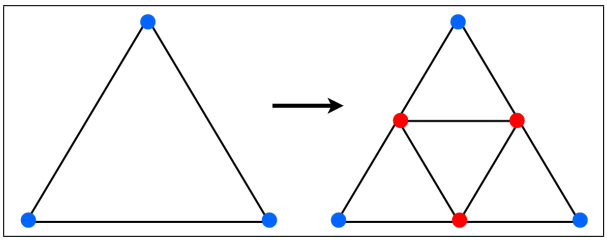

Figure 10. The idea of subdividing a triangle.

The new vertices are calculated as the midpoints of the edges in the original triangle. Then, these vertices need to be normalized (in order to end up on the unit sphere). So the vertex we're looking for is given by equation 12, where v_0 and v_1 are the two vertices spanning up the edge.

$$v_{mid} = \frac{v_0 + v_1}{|v_0 + v_1|} \tag{12}$$

Now, if we were to loop through all the triangles and create the midpoint for every edge, all the new vertices would be duplicated (since every edge is shared by two neighbouring triangles). This would not be runtime efficient.

The problem is illustrated in figure 11, here only for four triangles.

The vertices marked in yellow would be duplicated, if no control was made.

This was solved by using a lookup table that keeps track of all the new vertices and what edge they belong to. And if we come to an edge where a midpoint has already been created, we reuse that vertex, instead of creating a new one.

When all of the icosahedron's triangles have been subdivided, the starting number of 20 has now turned into 80. See figure 12 for the result.

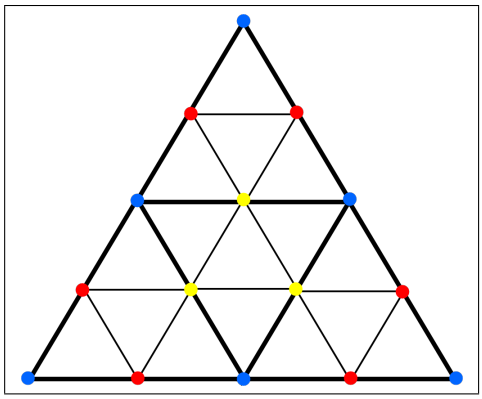

Figure 11. The duplication problem.

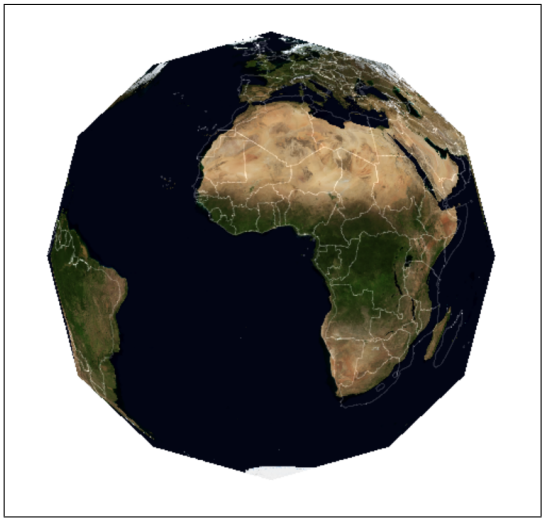

Figure 12. The icosphere after one subdivision.

Then you can use these new triangles to subdivide again, multiplying the number of triangles by four every time.

We found that about seven iterations gives us the result we want, while still not being too computationally heavy. This gives us a total of $20 \cdot 4^7 = 327\,680$ triangles. See figure 13 for the result.

3.5. Texturing a Sphere

When texturing a sphere there are two alternatives, wrapping a 2D texture or using environment mapping. As the 2D approach will create a squeezed appearance on the top and bottom of the sphere the environment mapping approach was used.

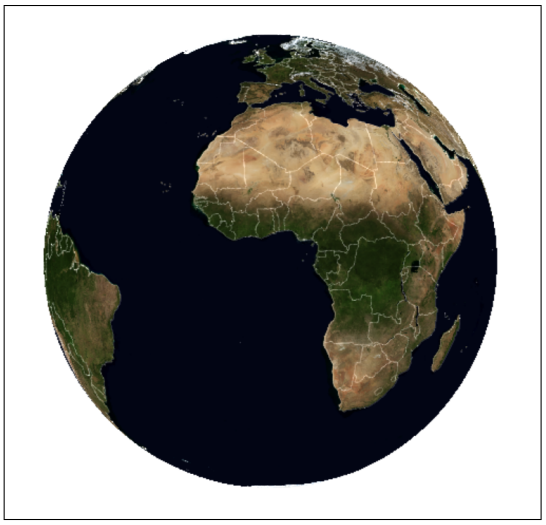

Figure 13. The icosphere after seven subdivisions.

Environment mapping is done by placing a virtual bounding box around the sphere. This box is then textured and the texture is then rendered onto the sphere by calculating a ray from the projection point to the bounding box by reflecting it on the sphere, as seen in figure 14 [4].

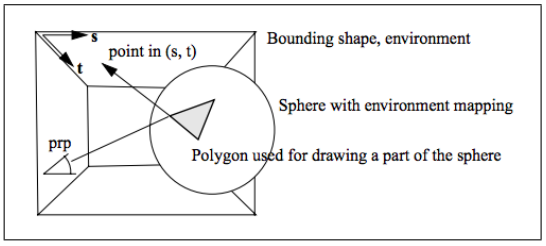

Figure 14. Enviroment mapping from PFNP [4].

OpenGL has a environment mapping method called cube mapping built in. This means that the bounding box takes the shape of a cube and simply requires a texture to function. This texture consists of six images, one for each face of the cube, where the textures have a slight distortion to compensate for the shape of the sphere, as seen in figure 15.

3.6. Generating Country Borders

The country borders that are generated for the application is done in 2 steps: the initial raw vector

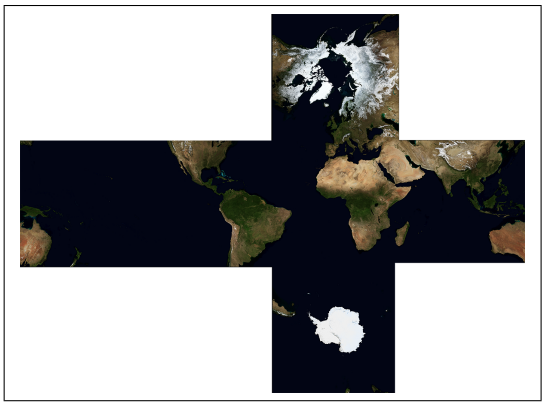

Figure 15. The cubemap texture.

data is read from the SVG file paths. This is followed by subdividing the resulting bezier curves using a distance error as stop criterion.

To parse this SVG file, a simple parser was implemented conforming to a useful (to the project) subset of the SVG standard. This parsing results in a dataset loaded into the application containing geometric country data, expressed as sets of cubic bezier curves forming closed polygons.

Since a country's border often consists of multiple closed polygons, an abstraction was put in place to gather these polygons in a simple binary-search-tree, where related polygons share the same key. The key in this case is the ISO-3 code as a string.

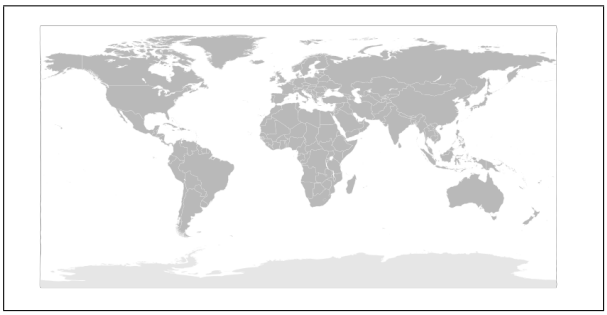

Figure 16. The border data set visualized as an SVG image rendered in HTML. The map projection is the equirectangular projection from the set of SVG files at [3].

As has been mentioned, the parsed data that describes the image in figure 16 is further processed. The program employs an implementation of subdividing the curve data points to generate smooth looking line segments for rendering in OpenGL.

Table 4. Variables and notation in algorithm 1.

$\mathbf{p}_1$, $\mathbf{p}_2$, $\mathbf{p}_3$, $\mathbf{p}_4$	Control points
$\mathbf{p}_{12}$, $\mathbf{p}_{23}$, $\mathbf{p}_{34}$, $\mathbf{p}_{123}$, $\mathbf{p}_{234}$, $\mathbf{p}_{1234}$	Mid-points (*see figure 3)*
ϵ^{min}	Error tolerance
$addPoint(\cdot)$	Method to add new point
$\epsilon(\cdot)$	Error measure
$subdivide(\cdot)$	subdivision method

Algorithm 1 Recursive Subdivision of Bezier Curve

for all curves: $\{\mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3, \mathbf{p}_4\}$ **do**
    $addPoint(\mathbf{p}_1)$
    $subdivide(\mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3, \mathbf{p}_4)$
    $addPoint(\mathbf{p}_4)$
end for
procedure $subdivide(\mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3, \mathbf{p}_4)$
    $\mathbf{p}_{12} = (\mathbf{p}_1 + \mathbf{p}_2)/2$
    $\mathbf{p}_{23} = (\mathbf{p}_2 + \mathbf{p}_3)/2$
    $\mathbf{p}_{34} = (\mathbf{p}_3 + \mathbf{p}_4)/2$
    $\mathbf{p}_{123} = (\mathbf{p}_{12} + \mathbf{p}_{23})/2$
    $\mathbf{p}_{234} = (\mathbf{p}_{23} + \mathbf{p}_{34})/2$
    $\mathbf{p}_{1234} = (\mathbf{p}_{123} + \mathbf{p}_{234})/2$
    if $\epsilon(\mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3, \mathbf{p}_4) < \epsilon^{min}$ **then**
        $addPoint(\mathbf{p}_{1234})$
    else
        $subdivide(\mathbf{p}_1, \mathbf{p}_{12}, \mathbf{p}_{123}, \mathbf{p}_{1234})$
        $subdivide(\mathbf{p}_{1234}, \mathbf{p}_{234}, \mathbf{p}_{34}, \mathbf{p}_4)$
    end if
end procedure

Algorithm 1 operates by recursively subdividing the cubic beziers by its midpoint. The only detail worth exploring further in this algorithm is the error measure $\epsilon(\cdot)$. It is needed to describe how well the points from algorithm 1 matches the analytic polynom, and be robust.

Preferably it should also be fast, as any lengthy

initial computations punishes the load time of the application. The L^1-norm of the difference of the control points $\mathbf{p}_2, \mathbf{p}_3$ and the midpoints $\mathbf{p}_{13}, \mathbf{p}_{24}$ proved to be sufficient.

$$\epsilon = |\mathbf{p}_1 + \mathbf{p}_3 - 2\mathbf{p}_2|_1 + |\mathbf{p}_2 + \mathbf{p}_4 - 2\mathbf{p}_3|_1 \quad (13)$$

Equation 13 provides the implemented error function. Note that the terms are multiplied by 2. The motivation for equation 13 is as follows: the optimal approximation of the curve as a line segment occurs when enough subdivision has taken place so that all 4 control points lie evenly spaced on a straight line.

When this flatness is achieved, further subdivision has little meaning. The choice of ϵ^{min} depends on the original coordinate system boundaries used in the SVG file, which are 2752.766×1537.631 'units' (SVG images are not defined in pixels).

That said, the choice was mostly dependent on the number of points to be provided to the triangulation of the overlay mesh, as the computational load (not surprisingly) of that implementation far exceeded any computational time that the implementation of algorithm 1 takes. Therefore, ϵ^{min} was set to 1.994.

3.7. Drawing 3D lines

Given a set of points forming line segments, a solution to the thickness problem is to tessellate a mesh of triangles consisting of duplicates of the original points.

From a line segment l represented as two 2D points a_0, b_0, a *wide line segment* is generated by duplicating these points producing the set a_1, a_2, b_1, b_2 forming the triangles (a_1, a_2, b_1) and (b_1, b_2, a_2).

To attain width, the points are "pushed" along the line segments normal direction **n** with half the desired line width $w/2$.

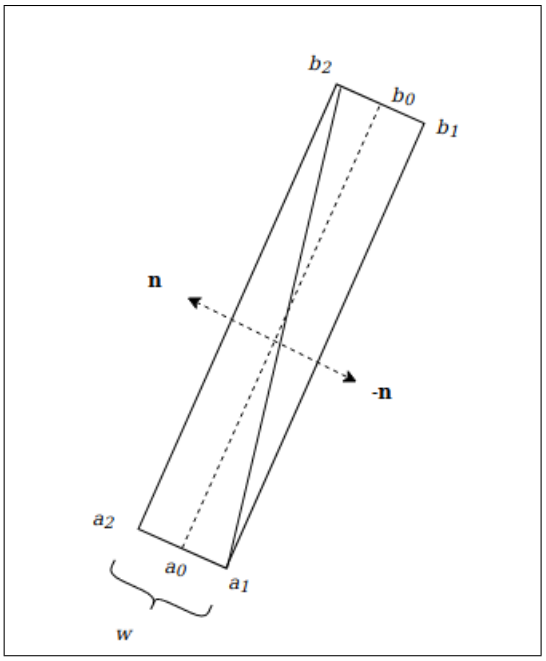

Figure 17. Tessellation of a line segment.

With this simple procedure illustrated in figure 17, line meshes can be generated from the same points that defined an openGL line primitive.

If the normals were calculated using (a_0, b_0), the problem of disconnect apparent in 4 would still be present. As a matter of fact, so would the aliasing. The method only achieves arbitrary width, which only (partly) solves one of the problems with the original primitive.

To attain all of the desired traits of the lines, the method of line drawing requires multiple implementation steps:

1. Attain arbitrary width.
2. Join the segments.
3. Perform anti-aliasing.

The line widths could really be defined in any coordinate system. Our implementation defines the line widths in *pixels on the screen*. This choice was to make the zooming function of the application not make the lines hard to see on the screen. The implementation is similar to that of the billboard described in [4].

Table 5. Variables and notation in algorithm 2

$\mathbf{v}$	Projected vertice coordinate
$\mathbf{v}^{ndc}$	Vertice in NDC
$\mathbf{v}^{in}$	Input vertice
$\mathbf{v}^{out}$	Output vertice
$\mathbf{n}$	Normal of line
MVP	Model-View-Projection matrix
P	Projection Matrix
w^{px}	Line width in pixels
w^{ndc}	Line width in NDC
a	Pixel size ratio

Algorithm 2 Line width computation

1: **for all** vertices: $\mathbf{v}^{in}$ **do**
2: $\quad \mathbf{v} = \mathbf{MVP}\mathbf{v}^{in}$
3: $\quad \mathbf{v}^{ndc} = (\frac{v_x}{v_w}, \frac{v_y}{v_w})$
4: $\quad w^{ndc} = \frac{w^{px} v_w P_{11} a}{2}$
5: $\quad \mathbf{v}^{out} = (v_x^{ndc} + w^{ndc} n_x, v_y^{ndc} + w^{ndc} n_y)$
6: **end for**

The computations constituting algorithm 2 is performed in the vertex shader. By projecting $\mathbf{v}$ to normalized screen coordinates, the width computation can be performed by the method in figure 17, if the width w is also computed in this coordinate system, which is done on line 4 in algorithm 2.

To join the line segments, the normals $\mathbf{n}$ have to be calculated differently dependent on if they are the end of a line or a segment of a line. To perform the joining of lines, the vertex shader needs to be aware of if it is assigning width to a line segment or line end.

Shader programs are executed completely parallel, which means there is no way to peek at the next or previously processed vertex in the program. The solution to this problem is, given a set of duplicated vertices V forming a tessellated line mesh, make right and left shifted versions of these points, V_{next} and $V_{previous}$. Provide these point sets to the vertex shader as well, and determine if the vertex under consideration is a segment or an end.

To resolve the sign of the normal, a set of alternating scalar signs $d \in \{1, -1, 1...1, -1\}$ are provided to the vertex shader.

Table 6. Variables and notation in algorithm 3.

$\mathbf{v}$	Vertex coordinate
$\mathbf{v}^{previous}$	Left shifted vertex coordinate
$\mathbf{v}^{next}$	Right shifted vertex coordinate
$\mathbf{n}$	Normal of line
$\mathbf{t}$	Tangent to line
d	Direction scalar

Algorithm 3 Line join (normal) computation

1: **for all** vertices: $\{\mathbf{v}, \mathbf{v}^{next}, \mathbf{v}^{previous}\}$ **do**
2: $\quad$ **if** $\mathbf{v} = \mathbf{v}^{next}$ **then**
3: $\quad\quad \mathbf{t} = \mathbf{v} - \mathbf{v}^{previous}$
4: $\quad$ **else if** $\mathbf{v} = \mathbf{v}^{previous}$ **then**
5: $\quad\quad \mathbf{t} = \mathbf{v}^{next} - \mathbf{v}$
6: $\quad$ **else**
7: $\quad\quad \mathbf{t} = \frac{\mathbf{v}-\mathbf{v}^{previous}}{||\mathbf{v}-\mathbf{v}^{previous}||} + \frac{\mathbf{v}^{next}-\mathbf{v}}{||\mathbf{v}^{next}-\mathbf{v}||}$
8: $\quad$ **end if**
9: $\quad \hat{\mathbf{t}} = \frac{\mathbf{t}}{||\mathbf{t}||}$
10: $\quad \mathbf{n} = (-\hat{t}_y, \hat{t}_x)d$
11: **end for**

Algorithm 3 is also implemented in a vertex shader. Note that all the computations are performed in NDC space. The 2D vector $\hat{\mathbf{t}}$ is the normalized tangent to the line in NDC space at $\mathbf{v}$. The tangent is then used to find the normal.

An implementation detail appears on lines 2-6 in algorithm 3. If a vertex is an end, the vertex in V has been copied into the same position in either V_{next} or $V_{previous}$ where the shift would otherwise be out of range, allowing for the equality check.

Achieving anti-aliasing of the correctly joined and arbitrarily wide meshes attained by applying algorithms 2 and 3 is a simpler matter. Utilizing the varying pass into the fragment shader to interpolate between the alternating scalars d gives us

the signed and normalized distance from the fragment to the middle of the line mesh.

By assigning an opacity gradient value at some distance from the the line mesh edge, the edge will appear smooth.

Table 7. Variables and notation in algorithm 4.

w^{px}	Line width in pixels
$\tilde{d}$	Interpolated direction scalar
f	Gradient width value in pixels
f^{max}	Gradient max position in pixels
f^{min}	Gradient min position in pixels
α	Opacity
δ	Distance from line middle in pixels

Algorithm 4 Anti-alias of line

for all fragments **do**
  $\delta = |\tilde{d}|w^{px}$
  $f^{min} = w^{px} - f$
  $f^{max} = w^{px} + f$
  if $\delta < f^{min}$ **then**
    $\alpha = 1.0$
  else
    $\alpha = 1.0 - \frac{\delta - f^{min}}{f^{max} - f^{min}}$
  end if
end for

The anti-aliasing in the fragment shader is applied with an f value of 0.5, which means the width of the fading out gradient is 1 pixel. This choice means that only when jagged edges from rasterization are present, the opacity α will be not 1.0.

The combination of algorithm 2, 3 and 4 becomes the implementation of line drawing used to produce country borders and lines representing flow in the final application.

3.8. Assigning widths to flow lines

To visualize differences in trade amounts, the flow lines are scaled depending on the amount. The width in pixels for a flow line is computed with equation 14.

$$w^{px} = w^{min} + q(w^{max} - w^{min}) \tag{14}$$

Here, w^{min} is the minimum width a line should be able to get, and w^{max} is the maximum. These were set to 1 and 8 pixels respectively.

The variable q is a quotient, saying how big the trade amount is in relation to the biggest trade amount for that resource.

3.8.1 Linear scaling

The application offers two ways of scaling the flow lines: linear and logarithmic scaling.

The scaling determines how the quotient q, mentioned above, is calculated.

For the linear scaling, q is given by equation 15, where V is the current trade amount and M is the biggest trade amount for the resource.

$$q = \frac{V}{M} \tag{15}$$

3.8.2 Logarithmic scaling

For the logarithmic scaling, the quotient q is given by equation 16, with V and M defined in the same way as for the linear scaling.

$$q = \frac{ln(1+V)}{ln(1+M)} \tag{16}$$

In short, the logarithmic scaling makes the smaller trades appear bigger, while the linear scaling can be seen as the true scaling.

3.9. Drawing Flow Lines

To generate the necessary set of points $\{\mathbf{p}_n\}_1^N$ as an arc on the surface of a sphere two things are needed: a starting point **a** and an end point **b**. By

normalizing $\mathbf{a} = \frac{\mathbf{a}}{|\mathbf{a}|}$ and $\mathbf{b} = \frac{\mathbf{b}}{|\mathbf{b}|}$ and then taking the cross-product, $\mathbf{a} \times \mathbf{b}$, a third vector $\mathbf{c}$ is generated, this is the rotation axis.

$$\theta = \frac{\mathbf{a} \cdot \mathbf{b}}{|\mathbf{a}||\mathbf{b}|} \tag{17}$$

The angle to be rotated is given by equation 17. Point $\mathbf{p}_n$ can then be found by rotating point $\mathbf{a}$ by the angle $\frac{\theta}{n}$ around $\mathbf{c}$.

To find an arc over the sphere surface each point on the line had a "lift" term added to it.

The curve is parameterized by $s \in [0, 1]$. This s can then be used to find an elevation term for each point. However, the line should be an arc and land safely on the sphere at its end. This means that the latter half has to decrease in height.

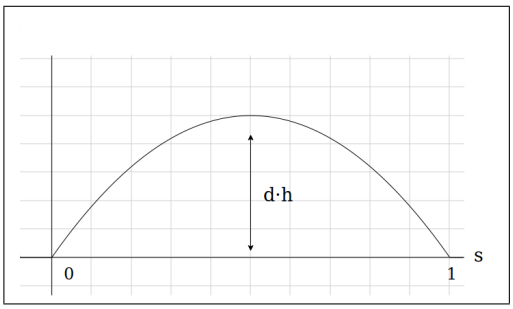

Figure 18. The "lift" function. Strikingly similar to the quadratic bezier blending function for the middle control point.[4]

The function of s is shown in figure 18. The resulting function output should be added to the points $\mathbf{p}_n$ in the radial direction of the sphere. Due to how $\mathbf{p}_n$ is computed, this is a simple multiplication.

$$\mathbf{p}_n^{lifted} = \mathbf{p}_n(1 + dh(1 - s_n)s_n) \tag{18}$$

In equation 18, h is a static height factor of 0.25. The distance between points $\mathbf{a}$, $\mathbf{b}$ is also added as a factor d to lift longer lines further away from the sphere. N was set to $51d$, meaning the number of points to sample the curve relates linearly to the distance between $\mathbf{a}$ and $\mathbf{b}$. $\mathbf{a}$ and $\mathbf{b}$ are found by computing the centroids of the point sets generated by algorithm 1.

3.10. Picking

Picking is implemented as a chain of transformations and intersection tests to determine where on the globe the user tried to interact. Figure 19 displays the complete idea.

First, the interaction point is found on the screen and transformed into model coordinates. This is followed by an intersection test of the model. This point is then further transformed into map coordinates. Lastly, the coordinate is tested against the polygons of the country borders.

3.10.1 Picking on Simple Geometry

$$\mathbf{m}_{ray} = \mathbf{M}^{-1}\mathbf{P}^{-1}\mathbf{n}_{coordinates} \tag{19}$$

To be able to interact with the globe, e.g. to be able to select a country or hover over it to highlight it, an interaction ray casted from the mouse was implemented. The creation chain for the ray is done in the order *screen-coordinates→normalized-screen-coordinates→eye-view-coordinates→world-view-coordinates*, equation 19.

Where $\mathbf{m}_{ray}$ is the mouse ray, $\mathbf{n}_{coordinates}$ are the mouse coordinates normalized between -1 and 1, and $\mathbf{P}^{-1}$ and $\mathbf{M}^{-1}$ are the inverted projection and world to view matrices.

After the multiplication with the projection matrix, the z value in the result are set to -1, before multiplying with the world to view matrix, to represent the ray going into the screen.

$$\begin{aligned} a &= \mathbf{m}_{ray} \cdot \mathbf{m}_{ray} \\ b &= 2(\mathbf{m}_{ray} \cdot (\mathbf{p}_C - \mathbf{c}_S)) \\ c &= \mathbf{c}_S \cdot \mathbf{c}_S + \mathbf{p}_C \cdot \mathbf{p}_C - 2(\mathbf{c}_S \cdot \mathbf{p}_C) - \mathrm{r}^2 \end{aligned} \tag{20}$$

$$t = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \tag{21}$$

The casted mouse ray is then used to find where in the world the mouse is currently hovering. A *line-spehere*-intersection is used to find if, and in that case where, the mouse ray intersects the globe, equations 20, 21.

The variables $\mathbf{p}_C$ is the position of the camera, $\mathbf{c}_S$ is the center of the sphere and r is the radius of the sphere. While a, b and c are used to see if a line intersects a sphere according to the following conditions:

$b^2 - 4ac > 0$, intersection
$b^2 - 4ac = 0$, tangency
$b^2 - 4ac < 0$, no intersection

In equation 21, t describes the scale factor for the traverse along the mouse ray from the cameras position to the intersection(s).

3.10.2 Picking on Closed Polygon

For testing if a point is inside a country polygon, it is first transformed into normalized coordinates u and v, using the intersection coordinate $\mathbf{p} = \mathbf{m}_{ray}t$ by

$$\begin{aligned} u &= \tfrac{1}{2} - \frac{arctan(p_y, p_x)}{2\pi} \\ v &= \tfrac{1}{2} - \frac{sin^{-1}(p_z)}{\pi} \end{aligned} \tag{22}$$

These coordinates are then tested by the zero winding rule test mentioned in section 2.7. The implementation relies on an "is left" rule, where the point (u, v) is tested against every edge $(a_x, a_y), (b_x, b_y)$ by computing

$$l = (b_x - a_x)(v - a_y) - (u - a_x)(b_y - a_y) \tag{23}$$

granted that $a_y < v < b_y$, the sign of l is added to the winding number sum.

3.11. Rendering Polygons

Below the implementations of techniques previously discussed for rendering polygons of arbitrary shape are presented.

3.11.1 Concave Polygon Splitting into Convex Parts

The common approach of divide and conquer, splitting a concave contour into its convex subdivisions. Each country is based on a set of points, representing the borders of the country. This data comes from the SVG-file mentioned in previous sections

Table 8. Variables and notation in algorithm 5

$\{\mathbf{c}_n\}_1^N$	Coordinate set for a polygon
$\overline{\mathbf{c}_n\mathbf{c}_k}$	The edge $\mathbf{c}_n - \mathbf{c}_k$
$\mathbf{s}$	Splitting Vector
i_s	Splitting index
i_i	Intersect index
t	Scalar value for splitting line (eq 6)
u	Scalar value for intersecting line (eq 7)
$\mathbf{p}_{intersect}$	Point of intersection

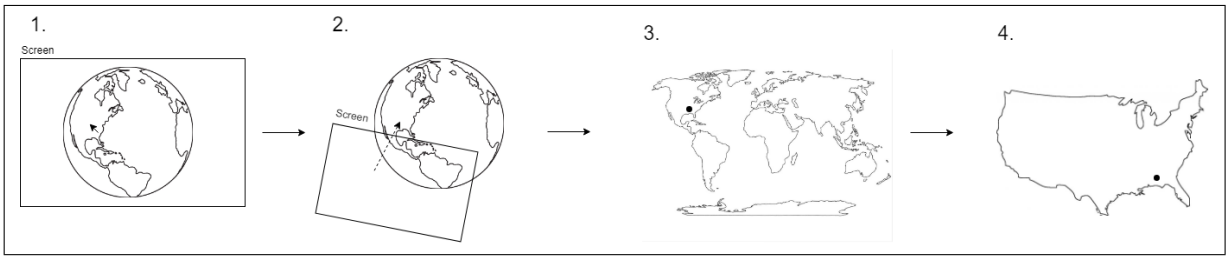

Figure 19. The chain of operations required to perform picking.

Algorithm 5 Split polygon

for all $n \in [1, ..., N]$ **do**
  if $\overline{\mathbf{c}_n\mathbf{c}}_{n-1} \times \overline{\mathbf{c}_{n+1}\mathbf{c}}_n > 0$ **then**
    $\mathbf{s} = \overline{\mathbf{c}_n\mathbf{c}}_{n-1}$
    $i_s = n - 1$
    break
  end if
end for
for all $n \in [1, ..., N]$ **do**
  if s $\notin [\overline{\mathbf{c}_{n-1}\mathbf{c}}_{n-2}, \overline{\mathbf{c}_n\mathbf{c}}_{n-1}, \overline{\mathbf{c}_n\mathbf{c}}_{n+1}]$ **then**
    if $t > 1$ **and** $0 \leq u \leq 1$ **then**
      $\mathbf{p}_{intersect} = \mathbf{c}_{n-1} + u * \overline{\mathbf{c}_n\mathbf{c}}_{n-1}$
      $i_i = n - 1$
    end if
  end if
end for

Firstly, investigation is done, with the help of equation 4, whether the country polygon is defined clockwise or counter clockwise in terms of point order. If it is not clockwise, the coordinate-list is flipped so that the polygon is clockwise.

Secondly, if that country or country part is not convex, this is investigated with the help of equation 5, that part is run through the splitting algorithm 5.

Table 9. Variables and notation in algorithm 6

$\{\mathbf{c}_n\}_1^N$	Coordinate set for a polygon
i_s	Splitting index
i_i	Intersect index
$\mathbf{p}_{intersect}$	Point of intersection
$polygon_1$	New polygon 1
$polygon_2$	New polygon 2

Algorithm 6 Create new polygons

if $i_i < i_s$ **then**
  add $\mathbf{c}_{i_s}$ **to** $polygon_1$
  add $\mathbf{p}_{intersect}$ **to** $polygon_1$
  for all $n \in [i_i + 1, ..., i_s - 1]$ **do**
    add $\mathbf{c}_n$ **to** $polygon_1$
  end for
  for all $n \in [0, ..., i_i]$ **do**
    add $\mathbf{c}_n$ **to** $polygon_2$
  end for
  add $\mathbf{p}_{intersect}$ **to** $polygon_2$
  for all $n \in [i_s, ..., N]$ **do**
    add $\mathbf{c}_n$ **to** $polygon_2$
  end for
else if $i_i > i_s$ **then**
  for all $n \in [i_s, ..., i_i]$ **do**
    add $\mathbf{c}_n$ **to** $polygon_1$
  end for
  add $\mathbf{p}_{intersect}$ **to** $polygon_1$
  for all $n \in [0, ..., i_s]$ **do**
    add $\mathbf{c}_n$ **to** $polygon_2$
  end for
  add $\mathbf{p}_{intersect}$ **to** $polygon_2$
  for all $n \in [i_i + 1, ..., N]$ **do**
    add $\mathbf{c}_n$ **to** $polygon_2$
  end for
end if

The variables i_s, i_i and $\mathbf{p}_{intersect}$ are used to create the new polygons according to algorithm 6

The new polygons are in turn checked if they are convex, if they are not, they are each run through algorithm 5 and thus algorithms 5 and 6 works in a iterative manner. If a polygon is convex it is saved to a list of convex parts that the country will consist of.

This method was used to generate the overlay mesh, which in turn is used to draw chloropleth maps ontop of the globe model.

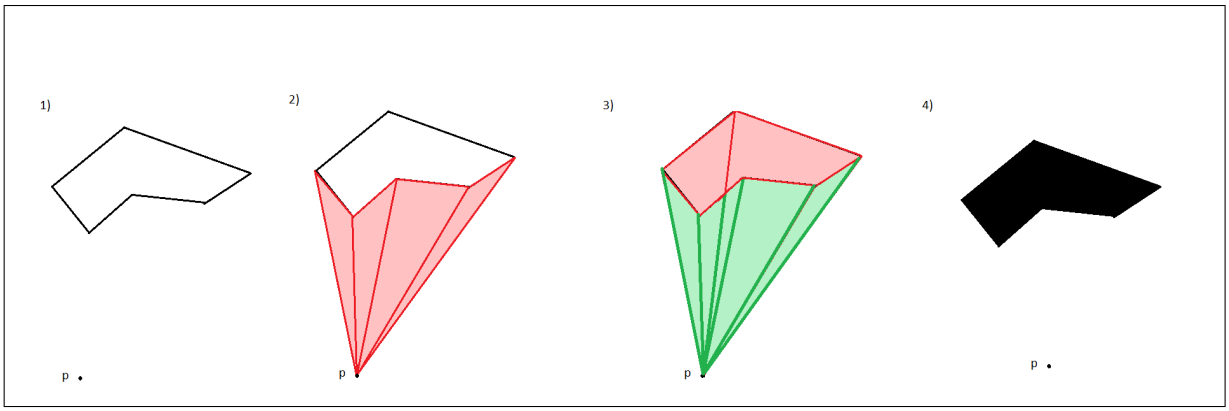

Figure 20. The stencil buffer polygon fill method. Red indicates an odd bit, green an even. 1) The contour of a polygon, and the triangle fan anchor p. 2) Midway of first pass rendering step, the edge of the polygon is used to draw a triangle fan from p. 3) The resulting bitmask generated by the stencil function after entire first pass. 4) The resulting filled in polygon by the second pass.

3.11.2 Filling by the Stencil Buffer Method

Using the contours generated by algorithm 1, the fill method that was briefly mentioned in the background section was implemented to highlight the countries on mouse hover in the application.

The method operates in 2 passes. The first pass renders onto the stencil buffer, applying the stencil functions which assures that whenever a triangle in the triangle fan originating from point **p** is rendered, the bit is set to the opposite value to what it was currently in the buffer.

The result is an odd-even bitmask. This corresponds to the OpenGL method *glStencilOp(GL_KEEP, GL_KEEP, GL_INVERT)*.

The second pass simply uses the mask generated by the first pass, rendering a quad fit over the screen, using the stencil to test for bits set to odd. Figure 20 is a 4 step representation of the method. For this method to work, depth testing has to be disabled.

The last detail to implement the method is the choice of **p**. The only requirement is that **p** lies in the viewing frustum. As such, the camera point-of-interest (or look-at-point) is a natural choice for **p**.

3.12. Shader Animations

All animations of the application are implemented as functions in the shader programs.

3.12.1 2D-3D Transition Animation

The transition animation from the two different views is implemented as a fairly simple vertex shader program. The idea is that since the two geometries that are to be transitioned between are described by very simple mathematical surfaces, the sphere S, and the rectangle P in the xy-plane.

The mapping $f : S \rightarrow P$ is a solved problem and is described in [4], but the application of the mapping is texture mapping of a sphere.

The transition also have the added criterion that the mapping should be an animation, which means the mapping needs to be function that is continuous by some parameter τ.

The solution is found by simple mathemathics. Let a point on the surface $\mathbf{s} \in S$ be described in spherical coordinates. The mapping $f : S \rightarrow P$

of **s** to a point $\mathbf{p} \in P$ is found by

$$\mathbf{p} = f(\mathbf{s}) = \begin{cases} p_x & = arctan(s_y, s_x) \\ p_y & = sin^{-1}(\frac{s_z}{||s||}) \\ p_z & = 0 \end{cases} \quad (24)$$

The $arctan(\cdot, \cdot)$ two-argument function is defined in [4]. Using equation 24, describing any point on the intermediate surface $\mathbf{v} \in V$ between the two surfaces can be done as a function of the parameter τ if $\tau \in [0, 1]$ and the point s by

$$\mathbf{v}(\mathbf{s}; \tau) = \mathbf{s} - \tau(\mathbf{s} - f(\mathbf{s})) \quad (25)$$

As τ nears 1, the surface V becomes more and more reminiscent of P. If τ is 0, V will be equal to S.

This functionality is implemented in all vertex shaders that are required to deal with geometry following the transition animation.

Some care had to be taken with points close to the anti-meridian and poles, as geometry gets stretched and warped in these areas, due to the boundaries of equation 24.

3.12.2 Flowline Directions

The flowlines are animated to give an indication of which direction the flow is heading to make distinguishing between import/export easy.

The animation uses the opacity value of the 4-component color vector to produce a visible change in the appearance of the flows. The opacity value is a position dependent scalar that also varies over time.

$$\alpha = 0.5 + sin(t - p) \quad (26)$$

The exact formula is found in equation 26 and uses the elapsed time t in a sine to create periodicity.

To generate a shift for equation 26 for each position, a position based parameter p is used that assigns each point in the flow to the range $[0, 1]$.

Then to increase the speed at which the line "moves" and how many particles are moving, scale factors can be multiplied with t and p respectively. We chose a factor of 7.5 for t and 50 for p.

The end result of this is the appearance of particles travelling along the flowline.

3.13. User interface

The user interface was implemented using HTML, CSS, JavaScript and a little bit of jQuery.

The HTML creates all the elements the website should contain. The CSS defines the styling of these elements. And the JavaScript can be seen as the engine of the frontend, defining all the functions that should be executed when the user does something.

The jQuery was only used for the filtering slider described in section 3.13.4.

The main components of the interface are:

- The canvas element that OpenGL renders all the graphics into.
- The side menu where the user can make selections and change settings.
- The switch between 2D and 3D view.
- The info-box that displays information when hovering on a country.

3.13.1 Connecting JavaScript and C++

To be able to send information from the frontend to the backend (and vice versa), you need to be able to call functions between the two.

To call C++ functions from JavaScript, we use something called *Embind*, that comes with the Emscripten package. Embind exposes C++ functions to JavaScript, so that they can be called in a normal way.

The compiled Emscripten code also provides a global JavaScript object called *Module*, that can be used to access the exposed C++ functions.

To call JavaScript functions from C++, we use something called *EM JS*, that pretty much creates a JavaScript function library for us. These functions can then be directly executed from the C++ code.

For a more detailed description, see the Emscripten documentation [8].

3.13.2 Filling the drop-down lists with data

Since the data is loaded in the backend, we need to send everything to the frontend (to fill the lists of countries and resources).

So initially, the drop-down lists are empty, and when the loading is complete, the data is received via EM JS, as described in section 3.13.1. The different options are then appended to an HTML div that expands in size, however becoming scrollable after a certain maximum.

3.13.3 Selection handling

When the user makes a selection in the menu, this action is forwarded (via the Module object) to the backend. Then there's a class called *selectionhandler* that keeps track of all the selections, and based on that, calculates the new possible options. These options are then sent back to the frontend, updating the drop-down lists.

An example of this would be when the user selects a resource, and the countrylist should be shortened to only those countries who have data for that resource.

The selectionhandler also makes sure that the flow visualization is only showing trades for the currently selected resources and countries.

3.13.4 Filtering of flows

In the side menu, there is a slider that allows the user to filter out flow lines, which can be useful if you for example only care about the big flows, and there are a lot of smaller flows making it hard to see.

The slider has two handles, so the user can specify an interval in which a trade amount needs to lie within for it to be rendered as a flow line.

If for example the first handle is at 20% and the second handle is at 80%, a trade will only be visualized if the quotient q (described in section 3.8) is in the interval $0.2 < q \leq 0.8$.

3.13.5 Zooming

The zoom function gets its input from the mouse scroll. These values are positive or negative integers.

The input values are used to move the position of the camera by moving its position along the position vector, e.g. the vector from the center of the globe to the position of the camera.

3.13.6 Rotating the camera

The camera is rotated with the change in the mouse's x- and y-coordinates as input. The cameras position $\mathbf{p}$ is changed based on a ϕ- and a θ-angle according to equation 27.

$$\begin{aligned} p_x &= |\mathbf{p}|cos(\phi + \Delta\phi)sin(\theta + \Delta\theta) \\ p_y &= |\mathbf{p}|cos(\theta + \Delta\theta) \\ p_z &= |\mathbf{p}|sin(\phi + \Delta\phi)sin(\theta + \Delta\theta) \end{aligned} \tag{27}$$

Here, $\Delta\phi$ is the change in the mouse's x-position and $\Delta\theta$ the change in the mouse's y-position.

4. Results

In the following section the results will be presented, a variety of the available visualization options will be selected and presented as figures.

As the figures will be static images none of the animations will be availible, these can at the time of writing be seen live on `www.resflow.se`.

4.1. Polygon splitting and tesselation

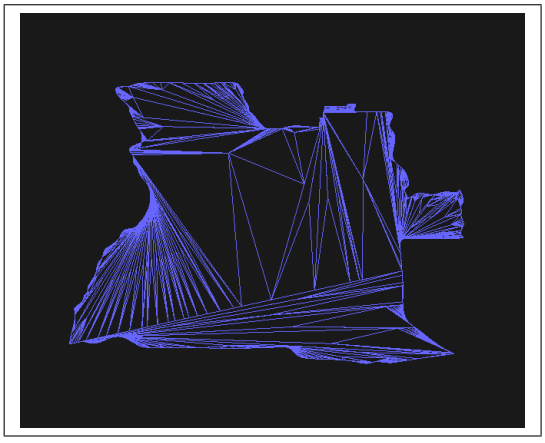

Figure 21. Angola consisting of convex polygons.

In figure 21 the results from the polygon splitting algorithm used on Angola can be seen. Each triangle corresponds to a set of points with indices that are drawn in the program.

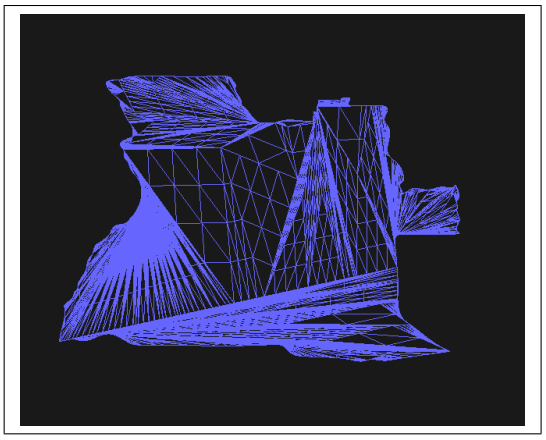

Figure 22. Tesselated version of Angola.

Figure 22 shows the tesselated version of Angola, here can clearly be seen that the number of triangles increased, thus leading to higher resolution.

Because of the disconnect between the polygons after the split, in terms of coordinates, gaps caused by newly created points on the split line after the split can occur. This is currently fixed by having a sufficently high resolution threshold and by making the triangles a bit bigger, so that they overlap and hide these gaps.

4.2. Visualisation of data

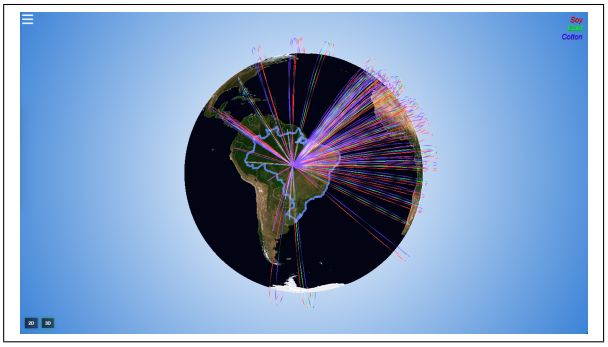

Figure 23. Import flows of soy, gold and cotton for one country

The import and export data is visualized as lines around the globe, called flows. The flows are animated to give an indication of direction for import or export.

If multiple resources are selected, their flows will be shown at the same time with different colors. The colors are matched to a resource in a legend in the lower left corner. An example of this can be seen in figure 23.

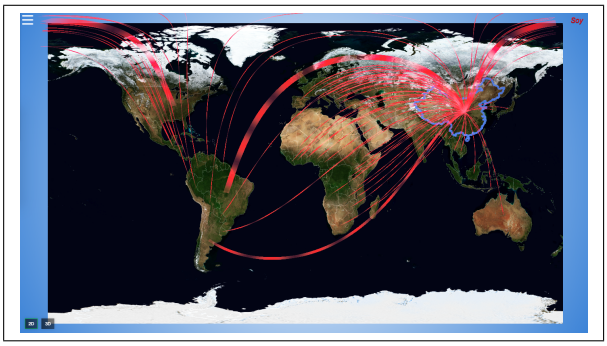

Figure 24. 2D view of the world with import of soy

If an overview of the entire world is desired the 2D view can be toggled. The currently selected flows are retained and remain shown as can be seen in figure 24

4.3. Visualisation of Indices

Figure 25. Transnational Climate Impact index shown with the blue color scale

The ability to visualize different indices as choropleth maps is also possible. If an index is selected the countries are colored accordingly, seen in figure 25

Figure 26. Selection of index color range

The application also allows the user to select between three color scales by clicking on the color legend. The menu drops down as shown in figure 26.

4.4. Combining the visualizations

The visualizations can be combined to draw conclusions about import from potentially unstable countries according to some index.

The previously mentioned color selection for the indices here allows for a clearer view as can be seen in figure 27.

Figure 27. TCI and import of Soy to several countries

5. Discussion

The following sections will discuss some of the issues with the application, what could be done to improve them and implementation of features that lie further in the future.

5.1. Optimization

The current application is at times slow, most noticeably at the start. This first delay is mainly the result of the slow generation of convex countrymeshes.

Another issue that is as visible as this is when the user selects all countries and the corresponding flows has to be drawn, this produces a noticeable stutter on most machines.

An easy way to improve the drawing based delays would be to find a way to reduce the amount of points that are created for the convex meshes while retaining the same visual appearance, previously having fewer points led to gaps in the mesh so this is a problem that would have to be solved.

A possible solution to this could be to have a look up table when generating the meshes, which lets the algorithm know of the connection line between the two newly created polygons during a split.

This has to work in the manner that it reduces the number of points in the total country mesh and connects the indices to the right coordinates.

5.2. Implementation for touch screen devices

The current application runs on all devices that are WebGL compatible, the interface however is severely lacking.

The implementation of a touch-device compatible user interface would increase the number of compatible devices greatly at a small implementation cost. This is because the Emscripten API has touch event handlers that only have to be linked to functions that handle the input, which already exist in the code.

5.3. Exporting the data

Another way to improve the application would be to add the ability to export data and graphics based on the current selections as either separate files or as a single report.

The selected data would be formatted into a set of predetermined plots and diagrams that then will be exported as images. If more detail is requested the raw data could be exported as a text file.

Another type of export is a still image of the visualization with the current selections included. This could make it easier to back up claims that were found by using the application.

References

[1] Wikipedia. (2018) Regular icosahedron. [Online]. Available: https://en.wikipedia.org/wiki/Regular_icosahedron

[2] T. W. W. W. Consortium. (2018) SVG Specification. [Online]. Available: https://www.w3.org/TR/SVG2/intro.html#AboutSVG

[3] wikimedia commons. (2018) Maps of the world. [Online]. Available: https://commons.wikimedia.org/wiki/Maps_of_the_world

[4] I. Ragnemalm, *Polygons Feel No Pain*, 2018.

[5] M. Shemanarev. (2005) Adaptive Subdivision of Bezier Curves. [Online]. Available: http://www.antigrain.com/research/adaptive_bezier/index.html

[6] K. Käfer. (2014) Drawing Antialiased Lines with OpenGL. [Online]. Available: https://blog.mapbox.com/drawing-antialiased-lines-with-opengl-8766f34192dc

[7] T. O. H. Network. (2018) Drawing Filled, Concave Polygons Using the Stencil Buffer. [Online]. Available: https://www.glprogramming.com/red/chapter14.html#name13

[8] E. Contributors. (2015) Emscripten Documentation. [Online]. Available: http://kripken.github.io/emscripten-site/docs/