A Geographic 3D Visualization and Browser of the World Wide Web

Lars-Olof Rydgren

2015

Master's Thesis

Department of Design Sciences
Lund University

Abstract

Internet is a vast digital network. Evergrowing in its complexity with frequent newly developed protocols extending the technology and changing the way we communicate and share information over the world.

Trends, news, videos and other information spread quickly over the world, making web browsers an incredible tool for users who listen to the right feeds or news networks. Web browsers are a necessity for structuring our obtained information and are needed for convenient surfing of the web. But for the non-experienced user it may not be completely self-explanatory what the Internet is, where to go or how far its borders stretches, the depth isn't shown and have to be individually explored mainly through the use of search engines. The culture of Internet can seem transparent the first time you open the traditional web browser. Traditional web browsers lack an introductional guidance into the culture and to the popular sites.

By visualizing public websites of the Internet as objects in a 3-dimensional world, the user will be able to see the boundaries of the web and explore it in a new fashion where the complexity of its structure is greatly reduced. The user will get an overview of the web in which activity and the culture can be shown in a game-alike environment.

This thesis will be about visualizing a crawled subset of websites as a city of buildings, where each building represents a website that the user can visit inside the world, the building's height reflecting its website's popularity. A multi-threaded crawler is used to find and store the relations between the set of websites, namely how many outgoing and which outgoing links each site has. This data in turn will be used by a force-directed algorithm that will position these buildings relatively geographical by attaching imaginary strings between them that either repulse or attract websites with high respective low relational attraction. The attractional forces being defined by the calculated PageRank that are passed between the sites.

The world will be rendered in WebGL and use the JavaScript library Three.js, and will thus be accessed through a web browser that connects to a web server and download all the necessary resources, e.g. textures, objects and JavaScript functions.

Trying to render a subset of Internet's websites as objects obviously puts a lot of stress on standard web browsers. Graphical features and tricks such as Level of Detail have been necessary to implement in order to make it appear a larger part is visualized than what actually is.

Sammanfattning

Internet är ett stort globalt nätverk som fortsätter växa i komplexitet med nya tekniker och protokoll som utvidgar dess användbarhet och sättet vi sprider information.

Trender, nyheter, videor och annan information sprider sig snabbt över världen vilket gör Internet till ett fantastiskt redskap för de användare som prenumererar på rätt informationsströmmar eller nyheter. Webläsare är en nödvändighet för att struktuera den information vi hittat och för att surfning av webben ska bli bekväm.

De områden som en traditionell webläsare saknar är främst en vägledning in i den digitala världen för en icke-erfaren användare. Det är svårt att få en uppfattning av Internets storlek, djup och var man är respektive alla andra användare. Internet har en stor kultur som kan anses transparent i första ögonblicket då man öppnar en webläsare för första gången. Traditionella webläsare saknar en vägvisande introduktion till Internets kultur och dess populära hemsidor.

Om man visualiserar alla Internet's publika hemsidor i en 3-dimensionell värld så ger det möjlighet till att se dess storlek och utforska det på ett alternativt sätt. Vilket därmed också simplifierar dess komplexitet. Användaren kan då få en överblick av webben där dess aktivitet och kultur kan visas i formen av en mer spelliknande miljö.

Denna avhandling handlar om att visualisera en delmängd av Internet's publika hemsidor som byggnadsobjekt i en värld där användaren interaktivt kan besöka varje hemsida i världen, där höjden av varje byggnadsobjekt återspeglar hemsidans popularitet. Ett bot-program med flera trådar har används för att lagra relationerna mellan hemsidorna vi hittat för att senare använda denna data i en så kallad force-directed algorithm. Denna algoritm kommer skapa attraktions och repulsions krafter mellan noder som representerar hemsidorna, där nodernas position ändras iterativt av dessa krafter i mån om att få dem geografiskt placerade. Krafterna mellan noderna bestäms av hur mycket PageRank en sida har gett en annan.

Världen kommer renderas i WebGL med hjälp av en 3D JavaScript motor som heter Three.js. En användare kommer kunna nå den visualiserade världen via sin webläsare som ansluter till min hemsida där de laddar ner alla nödvändiga resurser t.ex. texturer, objekt och skript. Jag använder mig av olika tekniker bl.a. en variant av Level of Detail för att kunna rendera en så stor mängd av hemsidor som möjligt i världen.

Acknowledgements

At times it have been both tough and frustrating solving the numerous technical problems this thesis have brought. I would like to thank my supervisor Joakim Eriksson, my examinator Gerd Johansson, my friends, my family and my relatives for their support and motivational feedback. Who all have been of great help through the process of making this thesis. Without them this wouldn't have been possible.

Stackoverflow, Google, Wikipedia and other information hubs on the web I owe my thanks to as well.

Contents

Abbrevations

CPU	Central Processing Unit
GPU	Graphics Processing Unit
FPS	Frames per second
RAM	Random Access Memory
URL	Uniform Resource Locator
HTTP	Hypertext Transfer Protocol
AJAX	Asynchronous JavaScript XML
API	Application Programming Interface

1 Introduction

3 dimensions is the way we see the real world in. It is an environment we're used to. 3 dimensions makes it easier for us to comprehend and sort our large and complex world. But most of the information you obtain over the web are 2 dimensional; text, pictures, videos, forums, chat rooms, social medias or news. So why do we need 3 dimensions?

If we imagine the analogy where each webpage on each website were a page in a book (the book being a website): would it then be useful to have a library filled with these books in which you may browse around? Most libraries have a database in which you can search the archive for a particular book (which may be compared to search engines), but sometimes you don't know what you're searching for and you just want to browse the library by their sorted genres or check the bookshelf that reveal the most popular books of this month. So visualizing and sorting Internet's websites is definitely worthwhile if it can be presented correctly and give a good overview of all the websites that exist, in a similar fashion to how a person can gain an overview of which books exist when browsing a library. Ultimately this means to use the third dimension to provide the meta data (position, size, shape etc) about the 2 dimensional information that you can find on the web.

The vision is to create an environment to support both an overview of the web and a large multiplayer game with the web browser users being the players. The main objective for this thesis is to create an infrastructure of the 3D world and provide an overview of a subset of the web and a way to browse the websites of that subset inside the 3D world. When websites become sorted and recognizable 3D objects in different shapes, colors, sizes and representative of their content - it will put the web into a perspective.

This objective brings us the questions:

How do we visualize and represent each website as a 3D object?

How do we structurally position a big number of websites in the 3D world?

What will the world containing the visualized websites look like?

How we proceed with these questions will be discussed throughout the thesis.

When we have an infrastructure of the 3D world the thesis will shift focus towards the functionality and the way the user interacts with the application. A group of testers will be given and perform a usability test within the world, the tests and the results will be presented in Chapter 10.

2 Conceptual design

2.1 Vision

The vision stretching beyond the timeline for this thesis is a massively multiplayer online game integrated with a web browser. The 3D world shall contain and visualize all the web's websites as building objects. The color, shape and size of the building objects being representative of each individual website. The buildings being extruded on the surface of the inside of a large sphere. Newly crawled websites shall be dynamically added to the 3D world, the exploring users for example assisting in *Distributed web crawling* [1]. The world shall be functional and effective in spreading the culture of Internet, and with options to spread new content through an integrated content feed.

Social minigames, chat channels, world events, easter eggs, explorative browsing in the world, ability to organize and store found content or structure and sort it for readability, visual feedback to which web servers you have trafficking data between, relevant content flying in the atmosphere around its sources. The users deciding what the world will look like as much as possible, being given enough freedom to change the world.

The world shall also be a geographical map of where websites are respective to each other, which are related to which and where you are in respect to every website. Helping the user to get a sense of direction of where to start browsing, possibly with a guide explaining and showing the user around the big websites and where they are located in the world. This kind of introductionary information may not be completely obvious to obtain for non-experienced Internet users, the traditional web browsers tends to offer a few URL suggestions of where to start but aside from that the Internet may seem completely dark and unknown.

The world shall have a cosmic theme with mostly toon shaded objects, giving the world a more playful look. Optimized for high end graphic cards so that a large part of the websites can be visualized at the same time. The graphics among other things being able to be adjustable through an interface.

The buildings shall strive to be uniquely fashioned towards the website that it represents. The corresponding website favicon shall be visible on top of the building so that the user can recognize which website a building represent from a distance.

The world shall have a star casting light upon the world, orbiting the center of the world. A stream of content, thumbnails of documents, in a grid-like pattern, shall flow out of it. It will be a real time content feed of what is currently the most attractive documents between the userbase. The users will be able to follow the stream and quickly gaze through the titles and thumbnails of the documents, and then have the ability to open a tab with them, sort them, share them or bookmark them if a document is of interest.

Each user (or a very large amount of them depending on the client's hardware) shall be visualized and represented as an imaginative shining sparkle flying around in the world. They shall not show a user's exact position but rather anonymously fly around the content that the user is browsing. The purpose being to simulate the real life scenario of the assembling process of a crowd around a point of interest (e.g. breaking news), which more than often tends to lead to more interest by nearby observers [2]. The hope is to enable digital crowd psychology in real time so that mass media may spread even faster among the userbase.

The world will ultimately be an overview of the web and a game built into the web browser, but the web browser should still have its core functionalities left, such as tabs, bookmarks, cookies, history, themes and plugins. However, these functionalities shall be slightly altered towards a 3D design. For example the Home button shall bring the user to an actual 3D designed home located somewhere on the outside surface of the spheric world. The home shall be fully customizable to the user, where the user may store and sort downloaded files, bookmarks, mails, news feeds, links among other things - represented as 3D objects. It shall be the home of the web browser. Along with its customizable part it shall show incoming and outgoing traffic and various statistics about the state of the web browser. The home's interior shall be private to the user while its exterior shall contain a place to put files that the user wants to share to the public.

However, the redesign of the web browser has to be carefully thought-out so that these additions do not to hinder its current capabilities, for example the bookmarks should still be searchable, sortable and quickly obtainable without being a hassle to use.

2.2 The first prototype

In order to start we have to design our first prototype by cherry picking features from the vision explained above. The first core features and challenges that I've decided to include in this prototype are:

- The world structure
- The websites represented by building objects
- The website objects being characterized by the following three things: its respective favicon, the ability to see its HTML content in the 3D world and its height representing its popularity.
- A geographically based location of the websites in the world
- Cosmical themed environment
- Spaceship-alike controls

- A star which for now will only be a light source

These features hide behind a lot of back-end implementation before they are able to be visible to a user. But lets first conceptualize these features together to get a picture of what we're striving for.

2.2.1 Concept and motivations for the features

The main feeling that I want the user to experience from entering the world is to feel like the Internet is an intranet - to be able to see it all in one scope and feel connected to all its users. For that reason I decided the world structure should be a 3D sphere in which the users and all the objects will be put.

I want the user to feel free even if its inside a restrictive space, although big. It seems best to give the user a point of view that is unattached from any gravitational force towards the surface. A free moving camera that lack the same kind of sense of direction that a space ship does would add to the "bigness" feeling and to the cosmical theme over the world. I decided the roof texture of the buildings will show similarity to a star, so when the user looks from one side of the world to the other the building objects that are very remotely distanced will show similar resemblance to that of stars in the night sky.

To characterize and identify the building objects so that they can relate to a website I will use the corresponding favicon to hover above the building to make it easier to both identify and scan through the horizon and recognize a website from a distance.

Being able to load the corresponding website's content inside the 3D world also help to identify the building. I'll use a frame that hovers next to the building so that the user may be able to explore the website within the frame. Let's call these website elements. But also as an alternative the user shall be able to open the website in a new tab in the web browser.

The building height correlating to a website's popularity would also help in identifying a website from a distance. I think the user would also seek and find an interest in the higher buildings rather than the lower. As is my intention, since my vision aims to reveal internet activity in different ways, which will generally be higher among the websites with high popularity.

The geographical positioning of the buildings are meant to cluster websites with similar interrelations nearby each other which hypothetically would create downtowns with skyscrapers (websites with high popularity spreads to its relatives) and suburban districts with low popularity in the city. I want to calculate this positioning with a geographic property in mind and evenly spread out the buildings over the spheric surface so that there aren't vast empty areas in which no buildings are located. I think a world that contains buildings that aren't sorted in some way is more confusing for the user, especially when one of the core visions is to make the world an overview of the web, it has to have some

underlying structure that in one way or another sorts this massive amount of information for the user.

In Figure 1 I've drawn the concept for our first prototype where we have buildings that represent websites that are extruded on the inside of the surface of the sphere which size reflects the website's calculated popularity. The building position are evenly spread over the spheric surface and geographically put to create downtowns and suburban areas. The favicons obtained from corresponding website are hovering as billboards above their respective building. The orbiting star will just be a light source for the world for now. The website elements aren't shown in Figure 1.

Most of the ideas have been pitched with friends and colleagues who have been interested in my work during the timeline of the thesis, this have helped me to refine both the concept and vision of the project.

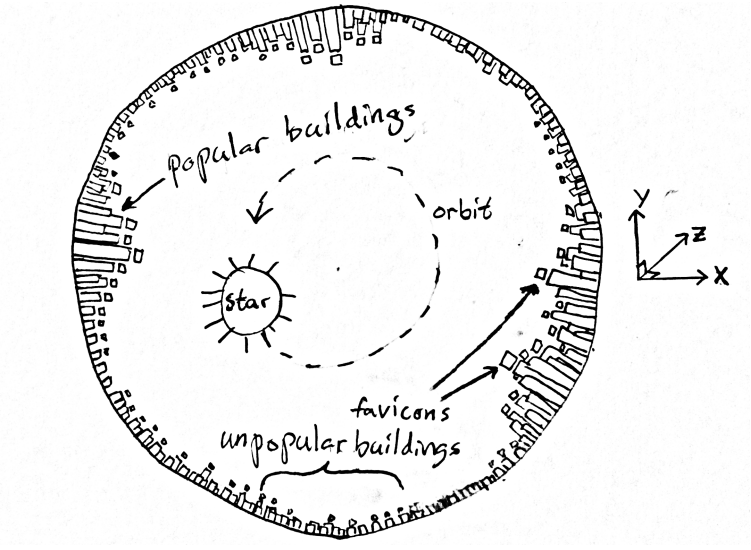


Figure 1: The main concept of the 3D rendered world (2D representation).

3 Technical Background

In this chapter we'll go through several important concepts and components that were used in the project and also explain briefly about what Three.js is, the basics of computer graphics and how one can render a 3D world inside the web browser.

3.1 The World Wide Web

The World Wide Web (shortened *www* and commonly called *the web*) is the information system of interlinked documents accessible via Internet [3]. It was invented in early 90s by Tim Berners-Lee and have grown rapidly ever since with more information being interlinked to the web. These documents are also referred to as web pages and can be accessed by web browsers such as Chrome, Firefox and Opera. The web pages are typically written in HTML (HyperText Markup Language) contain information about how the web browser shall construct the web page. The web page simply contain information about how to present text, media, links to resources or other web pages to the user browsing that web page.

When the user input an address (called an URL) to a website in the web browser, the web browser typically fetches the main web page file (often called index.html) on the given address through a protocol called HTTP, and display it to the user.

3.2 Web crawler

A web crawler is a program often referred to as a bot that browses the web in automation by following specific policies. These policies tell the crawler where to browse, what to download, whether it should periodically re-visit certain sites for changes or how polite it should be to avoid overloading websites [4].

The crawler often searches for specific information or metadata on/of a website, such as e-mail addresses, images or certain HTML-tag elements.

If we start the crawler at the root path of a website which is typically at the index.html, the crawler can be configured to search to a certain depth relative to that page. This means it will find the href HTML-tags[1] that are either relative with respect to the current page or absolute paths and parse these pages until the specific depth have been reached (or it found what it searched for).

The depth value is very important because it largely impacts the time complexity of the crawler. Sometimes it's needless to go beyond a certain depth to find information, it heavily depends on what information the crawler is searching for.

[1] The href-tags are hyperlinks which is used to link from one page to another.

3.3 WebGL and computer graphics

WebGL is a cross-platform API[2] that is based on the OpenGL Embedded System (OpenGLES) 2.0 interface which gives a 2D or 3D context within a canvas element in a HTML document [5]. Since it is for embedded systems it is mainly adopted for use on smaller devices such as mobile phones or tablets, where as its big brother OpenGL have more relevance on larger rendering platforms. WebGL have been adopted on a number of different web browsers, i.e. Google Chrome, Firefox and Opera. In realistic scenarios there are few cases where large complex programs or games are downloaded through a web browser because it would take too long time for the user to wait, which is why web browsers are perfectly suitable for OpenGLES instead of OpenGL. WebGL also includes OpenGL Shading Language (OpenGLSL) with a similar syntax to C that is used for creating shaders for the objects. What shaders are and what you can do with them, how to draw and render a 3D program in HTML will be explained in this chapter.

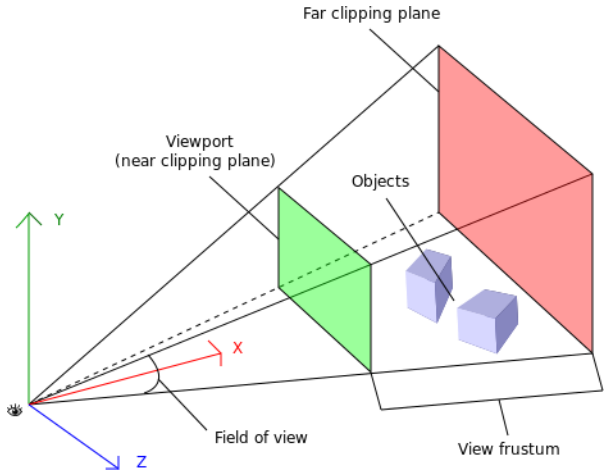


Figure 2: An illustration of where the viewport is. The viewport is where the objects will be put to their 2D screen space [6].

3.3.1 Shaders

In OpenGLSL there are two types of shaders: *vertex shader* and *fragment shader*. The shader programs are linked to the objects that we want to create in the world.

[2] An API is an application program interface that exposes a set of routines or protocols for software to be built on.

A vertex shader handles each individual vertex of an object, that vertex operates on its own but with help of user defined inputs such as *attribute* variables which can be a position, a normal or texture coordinates. Attributes are variables per vertex.

Another user defined input are *uniform* variables which are global variables for the entire object, accessed by all the vertices of that object. The vertex shader expect the 4-element vector *gl_Position* to be written to before the program ends, it will be the resulting position of the vertex [7]. This also means that you can displace the vertex of the object in the vertex shader such that the scale, shape and the location of the object changes from that of the original design and position - opening up for powerful physical motions of the objects.

The fragment shader's purpose is to obtain a color at every pixel of the rendering area, it uses the transformed vertex positions to generate the shading of a particular object on the corresponding pixels - taking material, light equations and different shading techniques into account to calculate the color. The uniform variables are also available in the fragment shader, the attributes such as the 2D texture coordinates also called *UV-coordinates* are commonly made into *varying* variables in the vertex shader. The varying variables are a bridge between the vertex shader and the fragment shader. The varying variable will be interpolated over the primitive that is being rendered and you can access the interpolated value in the fragment shader [8]. The UV-coordinates are used in the fragment shader, mainly to lookup a color from a texture map, but can be used to draw gradients, fractals and other effects over the primitive.

3.3.2 Drawing in HTML

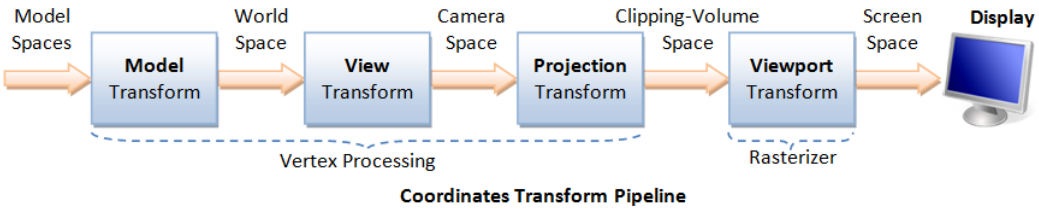


Figure 3: The process of transforming an object's vertices until they reach the screen space [9].

The WebGL API is accessed from a set of JavaScript interfaces that is used to eventually draw 3D objects onto the canvas element of HTML. Before rendering there are certains steps at minimum that you need to do to initialize a 3D world [10]:

1. **Create a canvas element** - This is done by creating a <canvas>-element in your DOM-structure that defines the position and dimensions of it.

2. **Obtain a WebGL context from the canvas element** - Done in JavaScript by calling *getContext* on the canvas element.

3. **Define the viewport** - Along with the canvas element you typically want a WebGL viewport that is equal in its dimensions. The content between the viewport and the far clipping plane is called the view frustum and is what will be shown to the user after the objects have been transformed to 2D space see Figure 2. The dimensions of the canvas and the viewport can vary as how the user defines them, they can be fully sized and stretched to a web browser's window or be contained in a smaller div-element on a website. For example in this project it is stretched to the full size of the window and will be automatically adjusted if the user changes the size of the web browser.

4. **Create a vertex buffer of buffers for the objects to be drawn** - All objects in the world are made up of *primitives* which can be a square, triangle, point or line. All these primitives use arrays of their vertices' positions, these arrays are called buffers. So a triangle's vertex buffer can be explained by a 3x3 matrix with a (x, y, z)-coordinate corresponding to each of the triangle's corners.

5. **Create matrices that will define the transformations of the vertex buffer(s) to screen space** - The object have a *ModelView matrix* that combines the different transformations (translation, rotation and scaling) of the model and the camera. Since all is relative to the camera you actually move the world and its objects if you move the camera. The second matrix we need is the *projection matrix* that is multiplied together with the ModelView matrix to convert our 3D camera space coordinates of the object into 2D screen space coordinates so that they can be displayed on the viewport. The projection matrix defines the size of the viewport, the aspect and hence the field of view.

6. **Create and initialize a shader for the object(s) to be drawn** - Now we initialize which objects shall have which shaders and send in their respective ModelView matrix, along with the camera's projection matrix. The matrices are combined in the vertex shader to transform the vertices to the 2D screen space. Figure 3 show the vertex transformation pipeline.

7. **Draw** - Draw the resulting world in the viewport of the WebGL context by telling it the object's buffer, shader, matrices and which primitives to use.

3.3.3 Rendering loop

Step 7 in Chapter 3.3.2, drawing, is the step that you preferably want to do as often as possible to get high FPS. It is the main part of the rendering loop

together with various variable updates. The render loop can be summarized and simplified to three steps in WebGL:

1. The function call *requestAnimationFrame* which you request from the window object in JavaScript. It will tell the browser that you wish to perform an animation, in which it recalls the current render method and request a new frame to draw. Usually the FPS will try to match the display's refresh rate, but will drop if there are a lot of calculations in between each frame or if you switch tab in the web browser or alternatively minimize the web browser [11].
2. Update object's position, rotation, shader attributes or uniforms. Everything that should change with time.
3. Draw the scene.

WebGL can be considered low level because it works very closely with the GPU and the core of computer graphics. It can be hard for an inexperienced 3D programmer to adopt, and it will take a bit of a time to produce anything substantial such as a game using only the WebGL API. There are tools that help you create a 3D world faster and with less effort, a JavaScript library such as Three.js does this easier for you, explained further in chapter 3.4. But having a low level fallback such as WebGL is very powerful if you need to alter specific changes in the engine.

3.4 Three.js

Three.js is a lightweight cross-browser 3D JavaScript engine that enwraps the WebGL API which let you use and manipulate the basic objects involved in computer graphics. These objects can be renderers, cameras, lights, shaders, particle systems, meshes and geometries such as spheres, cubes or icosahedrons[3]. There is a variety of mathematical tools such as quaternion[4] that can be used for rotating 3D objects, or splines[5] in which you can translate your 3D object along a smooth curve. It's a wide and helpful library that have continued a rapid development since April 2010. It's located on their GitHub[6] repository where the public can review and contribute to the engine. Three.js can run in all web browsers supported by WebGL. It comes with a minified JavaScript library when downloading it from their repository [14].

Three.js simplifies WebGL API and contain many more features such as:

[3] A spherically shaped object that generally have 20 triangular faces.

[4] A quaternion is a rotation vector of the form (x, y, z ,w) in which x, y and z represent the axis of which to rotate around and w is the amount of rotation [12].

[5] *A spline is a numeric function that is piecewise-defined by polynomial functions, and which possesses a sufficiently high degree of smoothness at the places where the polynomial pieces connect [13].*

[6] GitHub is an open repository for source code management where users are able to share and upload their code for others to see.

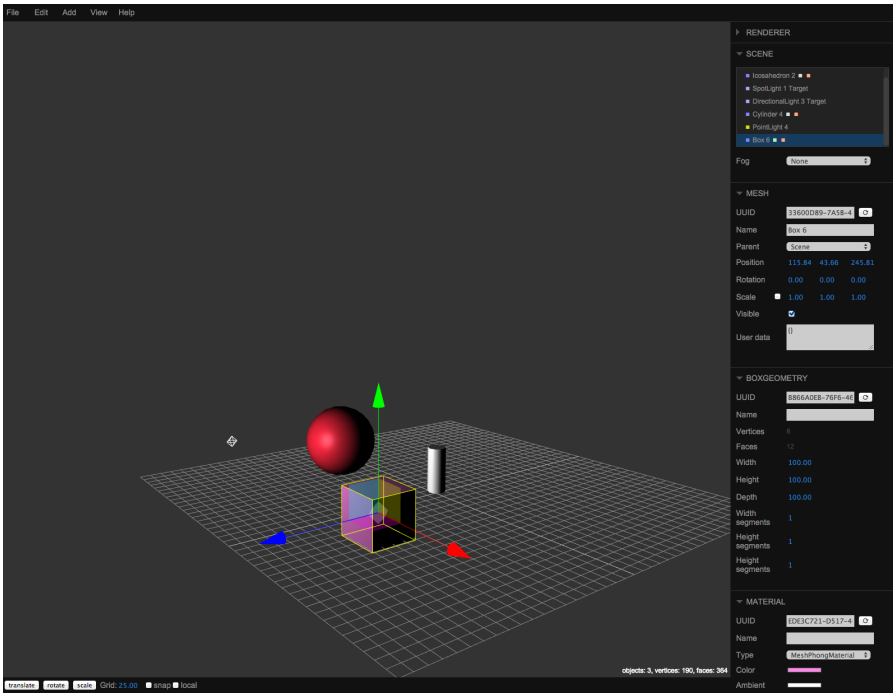


Figure 4: The new Three.js beta development tool.

- **Renderers**: Canvas, SVG and WebGL.
- **Scenes:** very easy adding and removal of 3D objects.
- **Cameras:** different ways of shaping the projection matrix, i.e. perspective or orthographic.
- **Lights:** point lights or spot lights with shadows cast over the 3D objects.
- **Materials** (shader)**:** predefined shading techniques such as Lambert or Phong, that contain the light equations for the 3D objects.
- **Data loaders:** for loading images, binary data or JSON objects.
- **Import:** capability to import JSON files from modelling programs such as Blender.
- **Support:** API documentation, active developers listening to feedback and working on frequent updates for the engine.
- **Examples:** widely spread examples over the internet of every feature the library contains.

However, Three.js is not a game engine but rather a tool to easily create a 3D environment in the web browser. But it is an ongoing project, at the moment there is a beta version of a Three.js IDE[7] in which you can build your 3D world visually by with adding/removing objects that you can position and change various parameters of, as shown in Figure 4.

3.5 Octree

An octree is a tree data structure that is heavily used in computer graphics to partition 3D space. Each node in the octree is a cube which have exactly eight cubic children or *octants* subdivided in its space. Each node stores the center point of its cube which its children uses to find the bounding box of their cube [17]. Figure 5 show how the octree subdivides a cube into 8 individual octants, the subdivision can be infinitely deep. If we store the objects in the octree we'll be able to traverse the octree in purpose to find the closest objects around a specific object, or shoot a ray to intersect the first object that the ray hits.

There are a lot more applications for octrees but in this project it will mainly be used to intersect our world object and used in our force-directed algorithm. The octree implementation that we use is written by Colin Hover [16] in JavaScript and is nicely adapted to fit Three.js. Figure 6 shows the octree structure (pink lines) formed over the world object that have created octants down to the level of 8 triangles per octant.

[7] An IDE is an Integrated Development Environment, a tool and source code editor that assits the programmer in building software.

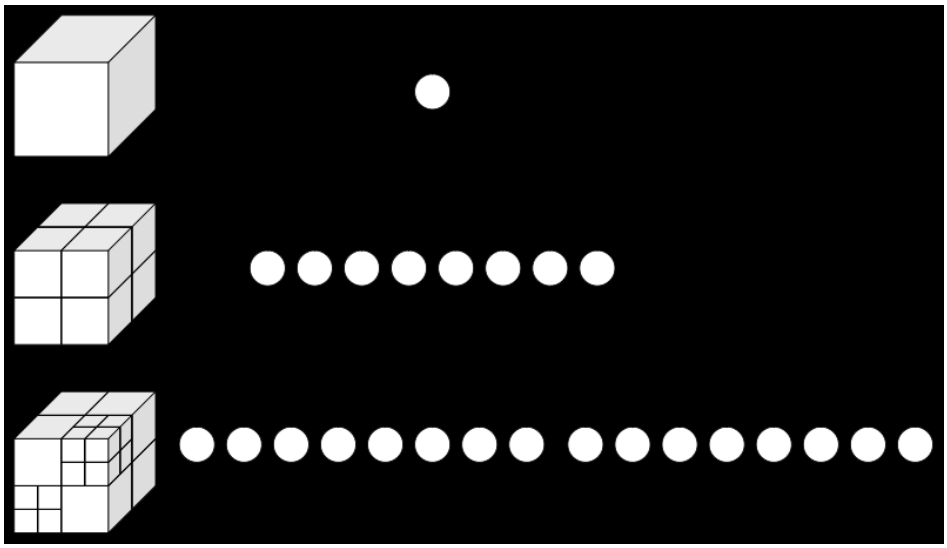


Figure 5: Left picture is the recursive subdivision of a cube into octants. Right picture is the corresponding octree in its tree representation [15].

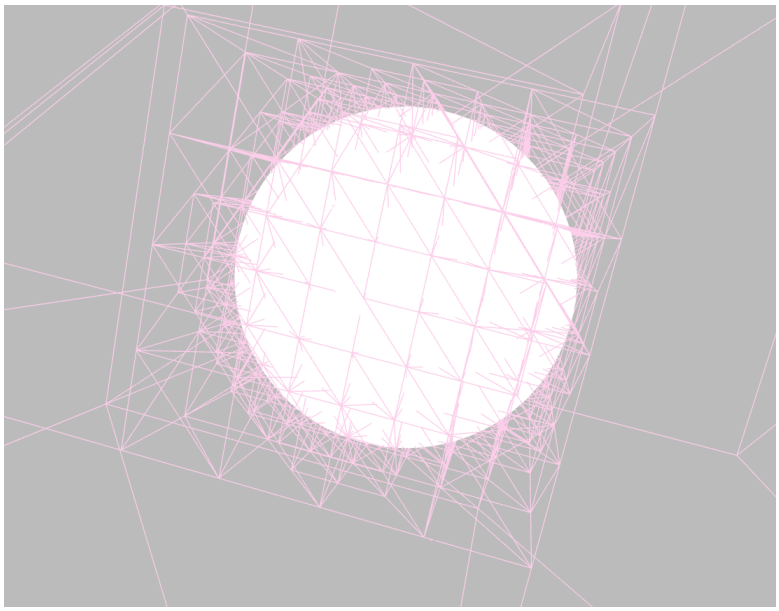


Figure 6: The world icosahedron object added to the octree visualized. The pink lines being the octree structure [16].

3.6 PageRank

PageRank is a probability distribution which represent the likelihood of a person randomly clicking on links over the Internet will arrive upon a particular site. The PageRank can be calculated on any quantity or size of documents, either internally on the same website or among many websites. The computation of the algorithm can be done iteratively where the sum of all PageRanks will converge close to 1 [18], and the PageRanks values will be close approximate values to that of the theoretical values.

Since we deal with probabilities, a document's PageRank of 0.5 would represent there being 50% chance of a person randomly clicking to end up on that particular document.

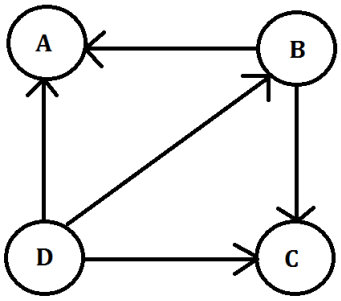


Figure 7: The outgoing links between four different documents, the directed arrows representing which document links to another document.

Suppose we have the set of documents shown in Figure 7 where document B links to C and A, and D links to all other documents. First we initialize all documents with the PageRank $\frac{1}{N}$ where N is the number of documents in our set, which in this example is 4. This means all documents are given the PageRank 0.25 fair and square. Each document's PageRank is then calculated from the sum of each inbound-linking document's PageRank divided by its total number outgoing links. 1 shows the PageRank formula for document A. Where L(B), L(C) and L(D) are the number outgoing links from each respective document, assuming each outgoing link to a specific document only counts once.

$$PR(A) = \frac{PR(B)}{L(B)} + \frac{PR(C)}{L(C)} + \frac{PR(D)}{L(D)} \tag{1}$$

Formula 1 can be expressed as 2 which is the PageRank for any document **u,** the summation of the receiving PageRank fractions.

$$PR(u) = \sum_{v \in B_u} \frac{PR(v)}{L(v)} \tag{2}$$

To give the person that is randomly clicking a bit more character; a *dampening factor* denoted **d** is often added to the PageRank formula. The dampening factor is the probability at any step in the algorithm that the person eventually stops clicking. There have been various studies testing different values for this factor, but the factor is usually set to around 0.85. A document's PageRank is mostly derived from the PageRank of other documents, the dampening factor adjust the derived value downward.

Another aspect when you are calculating PageRank properly is handling *sinks.* In the case when the person surfing reaches a page which have no outgoing links it will cancel the surfing process, these pages are called *sinks,* such as A and C in Figure 7. So when calculating PageRank to deal with the sinks and be fair with pages that are not sinks, the pages with no outgoing links will share their PageRank equally with the rest of the documents in the set.

To account for the dampening factor and the sinks, the new formula can be expressed as in 3 [18].

$$PR(p_i) = \frac{1-d}{N} + d \sum_{p_j \in M(p_i)} \frac{PR(p_j)}{L(p_j)} \tag{3}$$

Where $p_1, p_2, ..., p_N$ are the inbound documents from the set of $M(p_i)$ and $L(p_j)$ is the set containing the number of outbound links from p_j.

So when calculating the PageRank for the first iteration of document A in the example Figure 7, we do as follows:

$$PR(A) = \frac{1-d}{N} + d(\frac{PR(B)}{L(B)} + \frac{PR(C)}{L(C)} + \frac{PR(D)}{L(D)})$$

$$PR(A) = \frac{1-0.85}{4} + 0.85(\frac{0.25}{2} + \frac{0.25}{4} + \frac{0.25}{3})$$

$$PR(A) \approx 0.2677$$

Because C is a sink its $L(C) = N = 4$ as shown in above calculations, and thus the resulting PageRank for A for the first iteration become 0.2677. This process is repeated for each of the documents and through several iterations until the values reach a good approximation.

3.7 Force-directed algorithm

Force-directed graph drawing algorithms is a way to draw 2 or 3-dimensional vertices in a space so that they may be distanced at an equal length between each other in an ideal scenario. The edges are assigned forces that will act upon the vertices and the forces are either used to simulate the motion of the edges or to minimize their energy [19]. The forces are typically spring-like attractive forces that are based on *Hooke's law*[8] and repulsive forces that are based on *Coulomb's law*[9].

There are different works and approaches when it comes to drawing graphs with the force-directed algorithm for example: *A Heuristic for Graph Drawing* by P. Eades [23], *An Algorithm for Drawing General Undirected Graphs* by T. Kamada and S. Kawai [24] and *Graph Drawing by Force-Directed Placement* by T. Fruchterman and E. Reingold [25].

Eades in his model which he calls the *spring embedder* the edges act as springs with a logarithmic force and the vertices are electrically charged with a square-inverse repulsion force law. Eades initially randomly draw the graph and continue with a fixed number of steepest descent iterations that for each iteration move the vertices by the force exerted upon them.

Kamada and Kawai modifies Eades' spring embedder model in their optimizations algorithm by instead using spring forces based on Hooke's law on both attraction and repulsion. The springs between the vertices strives try to attract and repel until the spring reaches its rest length. For each pair Kamada and Kawai define the rest length of the spring proportional to the shortest path between two connected vertices and the constant or the stiffness for the spring being inversely proportional to its rest length. Instead of moving all the vertices every iteration the algorithm moves the vertex with the greatest force to a position of locally minimal energy.

Fruchterman and Reingold's model is more similar to Eades when it comes to defining the force laws. They set the attraction force to be the squared distance between the endpoints of an edge for two vertices. The repulsion force being the being inversely proportional to the distance between two vertices. As such becoming less computational than that of Eades model but still similar.

For example Fruchterman and Reingold's have their attraction law defined as:

$$f_a(d) = \frac{d^2}{k}$$

[8] A physical law that states that the force F needed to compress a spring by a distance X is proportional to that distance. The distance being multiplied by a constant k that is the stiffness of the spring. Such that $F = -kX$ [20].

[9] Coulomb's law is an *inverse-square law* which states that the electrostatic force between two sources v_1 and v_2 is an intensity or constant k multiplied by the inverse squared distance $d(v_1, v_2)$ between these sources. Such that $F = k\frac{1}{d(v_1,v_2)}$ [21, 22].

and their repulsion law as:

$$f_r(d) = -\frac{k^2}{d}$$

where d is the distance between two vertices and the optimal distance between two vertices are defined as k:

$$k = C\sqrt{\frac{area}{number\,of\,vertices}}$$

where C is a constant found experimentally.

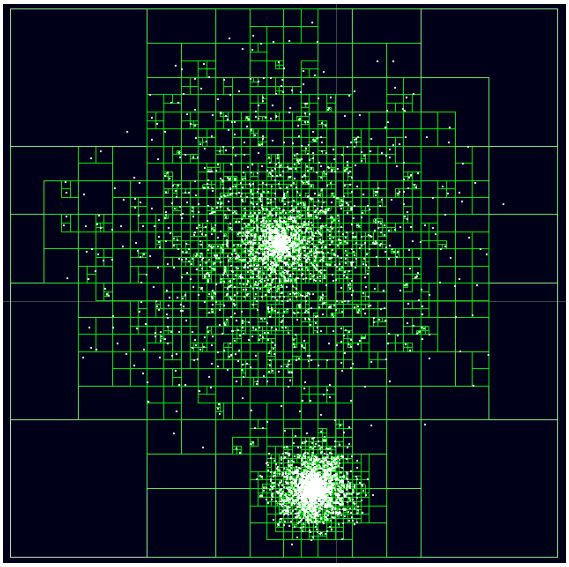


Figure 8: The Barnes-Hut algorithm applied to a set of vertices which forms the quad tree.

The force laws shows a certain elegancy and can vary in their characteristic. Many of the force-directed algorithms are also designed with an annealing character that can be thought of as a cooling process in that the movement of the vertices gradually decreases as the iterations goes.

The mentioned approaches have complexities varying between $O(n^2)$ and $O(n^3)$ where n is the number of vertices in the simulation - which quickly can become unsuitable and time-consuming for when drawing large graphs. In physics this type of problem is referred to as the *n-body problem*[10]. A way to reduce the time complexity and solve the problem in $O(n\,log(n))$ is through the *Barnes-*

[10]The n-body problem is typically found in galaxies where each body represent a star or object, each object attracting every other object through the gravitational force they exert.

Hut algorithm [26] which typically divides the vertices in a quad or oct-tree as shown in Figure 8.

4 Module: the 3D environment

With the first prototype characterized we can start to break down the features that make up this prototype. I will group up the features and infrastructure challenges into *modules* where this Chapter and module will be about the 3D environment and where we shall render our 3D world. An overview of all the modules used and which part they play are shown in Figure 9. It makes it easier to group up closely related features so that each module will be able to be revisioned and tuned separately. Surely there are dependencies between the modules but I think for both the reader and me that it will be easier to structure the prototype in this way. These modules will encapsulate the discussion and implementation of the first prototype that was explained in Section 2.2.

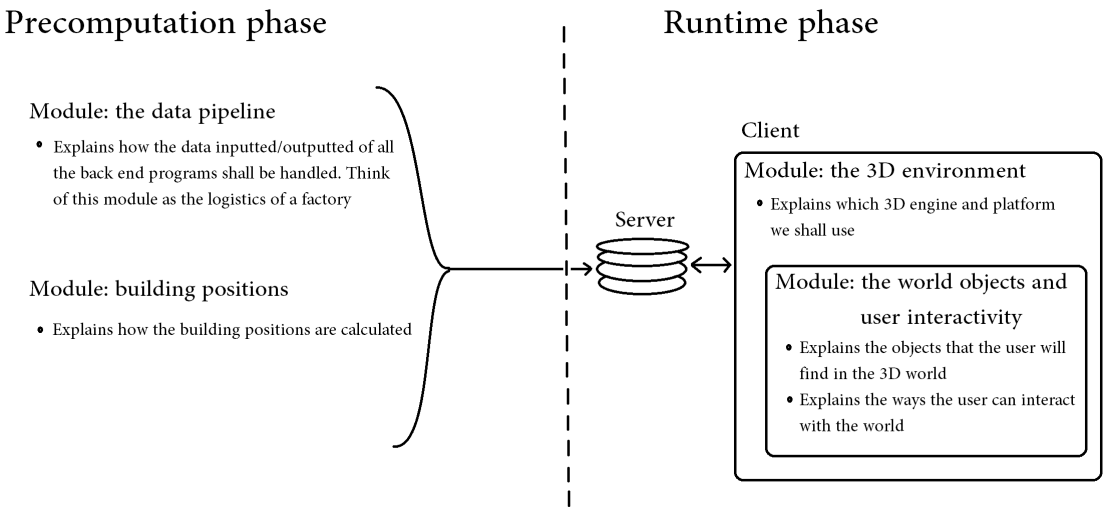


Figure 9: The overview of all the modules and which role they partake.

4.1 Where do we visualize and render our concept?

Obviously the user will have to open the 3D application in some way and start exploring the world. Since this is the first prototype there are challenges in setting up the environment around the mentioned features and also show respect to the future development of this project.

The vision is to create a really large world with great space-alike graphics and a widespread city of websites. For performance heavy 3D applications there are game engines that have been developed over many years such as Unity3D[11] or Unreal Engine[12]. This project would benefit greatly from those mainly because

[11] Unity3D is a cross-platform game engine, released in 2005 by Unity Technologies[27].
[12] Unreal Engine is a cross-platform game engine, released in 1998 by Epic Games[28].

we're rendering a huge city of buildings with an effectful and animated environment with high-polygon objects. But this project is also largely about using web browser functionality along with an interactive 3D Internet, and the majority of the data transmission inside the 3D world will be HTTP-based where we render web pages and display the HTML content of the Internet.

The WebGL context inside web browsers could offer a web-based platform for this project and have upon its arrival kicked off a series of other web browser-based 3D projects with heavy graphics. It's a context for my project that may offer a lot of potential in the future, but it still feels like it is in its experimental phase and haven't been too widely adopted yet, I believe it just needs more time and attention.

The vision strives towards two different solutions, those are:

1. Rendering the 3D environment inside a common web browser with a client-server model. Using WebGL as rendering context that would be enwrapped by the web browser's interface and thus use a webserver to store all the necessary resources on, such as scripts and textures. The negative part is that a lot of resources would have to be sent between the server and client, but then you would be given the advantage of having an incredibly lightweight client side that only store data for the session.

2. A 3D application built with one of the larger game engines such as Unity3D or Unreal Engine with an integrated web browser-alike interface able to render HTML content and with textures and data stored locally at client side. The negative part with this solution is that it would take a lot more time to design and implement, but with the benefit of having a customized interface to my fitting and less data transferring between the server.

I decided to go with the first alternative simply because its more lightweight for the client side which would mean an easier adoption onto other devices such as tablets if the development ever reaches that stage, and also because WebGL is a fascinating technology that I wish to bring more projects to.

To open the application the end user will type in the URL to my webserver in a web browser that supports WebGL, which then loads all the necessary resources and renders the 3D world. Since a lot of people use web browsers in their daily life, typing in the URL to reach the application would be a very convenient way.

4.2 What 3D engine shall we use?

After choosing to continue the development in WebGL we have the choice to utilize one of many 3D JavaScript engines that enwraps the WebGL API, such as

GLGE[13], PhiloGL[14], SceneJS[15], SpiderGL[16] or Three.js. After looking into each one of them and considering different factors such as: how easy it was to create a WebGL scene and add different objects of different sizes to it, which engine had good examples, which engine had a large community, active developers, convenient code and naming conventions, and many other factors. I decided Three.js had most of my needs and it felt easy to learn.

Three.js also felt like a tool that you could easily extend with your own customized code on top of, in case the provided functionality wasn't enough. For example you can add custom shaders [33] in which you can insert predefined code snippets in your vertex and fragment shaders, such as shadow mapping and other standard routines.

[13] GLGE is a JavaScript library that eases the use of WebGL[29].

[14] PhiloGL is a WebGL framework for data visualization and game development among other things[30].

[15] SceneJS is a WebGL 3D visualization engine [31].

[16] SpiderGL is a JavaScript computer graphics library enwrapping the WebGL context [32].

5 Module: the world objects and user interactivity

This Chapter is about the world objects that the user can see in the world and the first user interactivity that we implement.

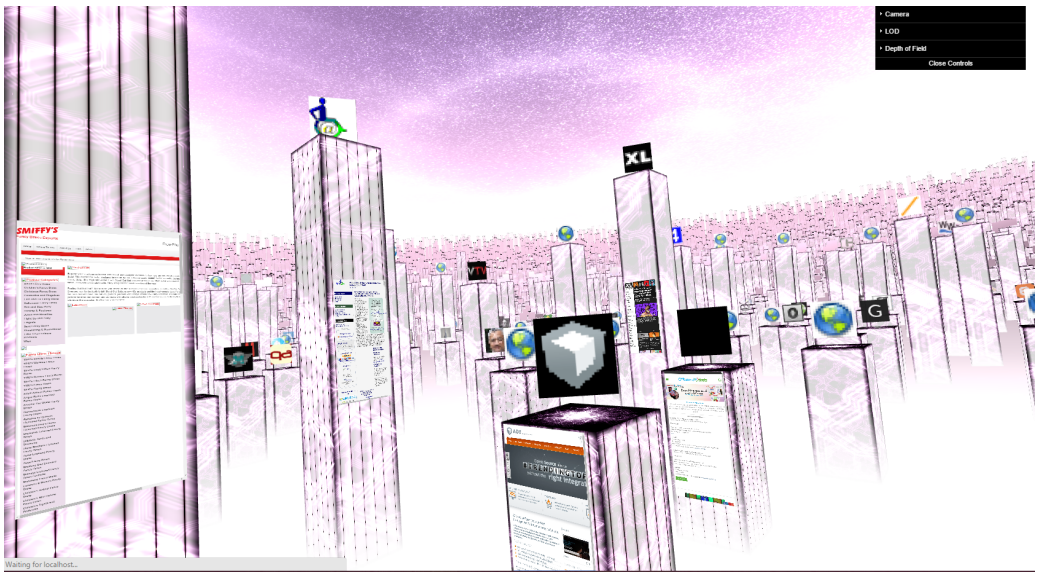


Figure 10: A screenshot of the world. The favicons hovering above the buildings and the website elements hovering next to the buildings.

A quick peek at the resulting world is shown in Figure 10 and Figure 11. As seen in these screenshots we can distinctly recognize the few objects explained in the concept description, mainly:

- The spheric world object that encapsulates everything in the 3D world. See Section 5.1.
- The building objects. See Section 5.2.
- The website favicons hovering above the buildings. See Section 5.3.
- The website elements hovering on the sides of the buildings. See Section 5.4.

In order to be able to render a large amount of buildings and objects at the same time I had to design a system in which we divide the buildings into three groups, all with different details, called *Level of details.* This optimization is designed and discussed in Section 5.7.

Another optimization I chose to implement early is the technique to merge groups of objects, both for geometry and texture. This reduces the load on the CPU. I used this technique on the favicons as well. The optimizations are discussed in Section 5.7 and Section 5.8.

5.1 The spheric world object

The world shall be round according to the concept description. For that we need a spheric geometry that the camera shall be restricted within. Ideally this object should have a level of detail depending on how close the camera is. Because otherwise; relative to all other objects, the larger the world gets the larger the polygons and the edges between them will seem to be, making it more noticeable for the user to see that the sphere is in fact not round. The shape of the polygons matters as well because it impacts the distribution of the vertex points of the spheric world object.

One of the most common techniques to tesselate a sphere is through the *Gaussian* or *longitude/latitude-grid* shown in Figure 12. Another technique is the *Geodesic grid* shown in Figure 13 which is essentially a subdivided polyhedron, usually an icosahedron [34].

The positive traits of the geodesic grid compared to the longitude/latitude grid is many; such that the resolution can easily be increased with subdivision, and it suffers from less over sampling near the poles. For these reasons I made the decision to use a geodesic grid icosahedron for the world geometry, and also

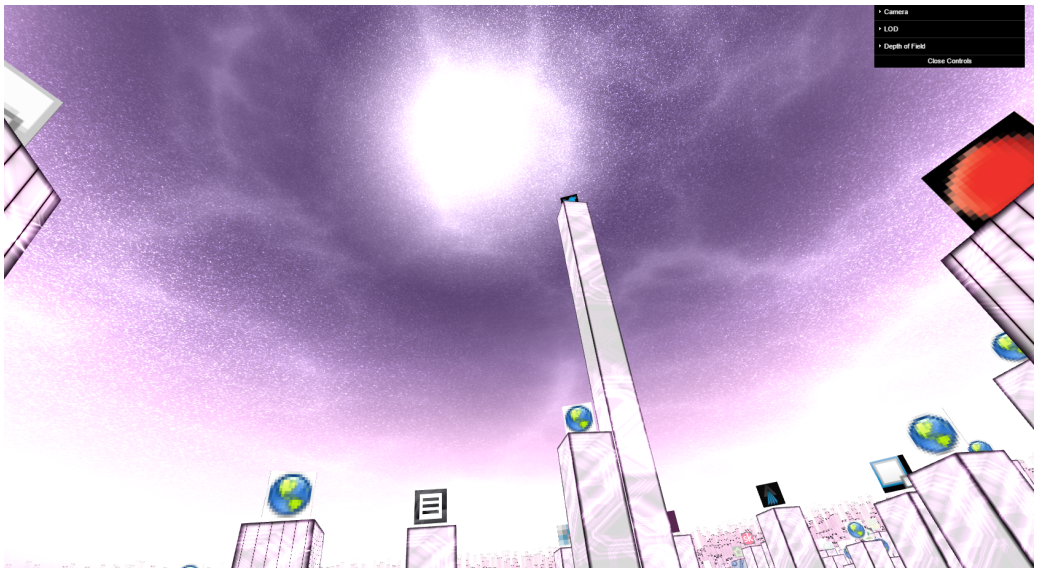


Figure 11: Another screenshot of the world with the star in sight.

because of the beneficiary factor that the triangles are roughly the same size - and through this, by giving each triangle an index we can use the world object's triangles as a tile system [17] for the geometry that we are going to place on its surface as shown in Figure 14.

Thus when navigating around in the world, we can continuously test which tiles are nearby in respect to the camera's position and then load the data of those tiles. In the rest of this thesis I will refer each triangle on the icosahedron as a *tile.*

We ultimately need to implement some sort of tile system to manage the data fetching between the client and the server. And this is one way to do it. Another tiling system would be to alternatively subdivide the world space by creating an octree with a certain depth but that would create irregularly sized tiles when the octree's tile's frames collide with the sphere's surface.

By having an icosahedron subdivided as a tile system we may easily be able to increase the complexity of the icosahedron through further subdivision in case we need more tiles. The icosahedron that we'll use for now is made of 5120 triangles. This would for example mean that with a set of 500 000 buildings, each icosahedron triangle would include roughly 97 buildings if the buildings were spread roughly equal across the surface.

If we load data from each tile respectively, the complexity of the icosahedron would preferably have to be increased the more websites we have in our set since we don't want to load too large chunks of data when moving around with the camera. The web server could suffer from too many HTTP-requests if each tile's data was sent separately, and this is a bad practice when designing a web server [35].

If we render all the fetched data in one go, it will impact our rendering capabilities as well. So depending on how big we make the tiles there is clearly a balance between what's optimal for rendering purposes and what's best for the web server's performance.

In order to load lets say the data from the nearest 100 tiles of where the camera is located it means that every tile needs to know which neighbouring tiles they have. This could be precomputed and stored on the web server. I made a JavaScript program that solved it by for each tile storing a file (named after the tile index) containing the closest 500 tile indices ordered by smallest distanced tile. The distances were computed by intersecting from the middle of each tile to every other tile's middle.

This program would in brute force have the time complexity $O(n^2)$, but for intersecting I used an octree which sorted the icosahedron's triangles into the tree structure and brought the complexity down to $O(n * log(n))$.

[17] A tiled system is a virtual space which you subdivide in a convenient way, usually in a grid, in purpose to be able to load local data from respective tile easier.

For now the files will contain the closest 500 tile indices, loading data from more than the 200 closest tiles creates performance issues on my computer with the hardware specified in 8.1, so even with a few optimizations I think 500 will be enough buffer for now.

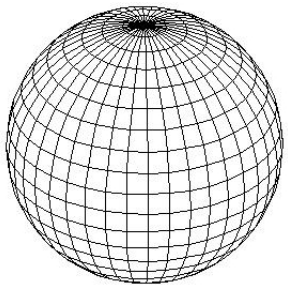


Figure 12: A longitude/latitude-grid sphere [36].

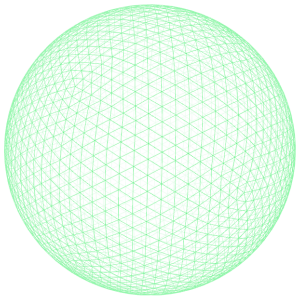


Figure 13: A geodesic grid icosahedron made of 5120 triangles that are roughly the same size.

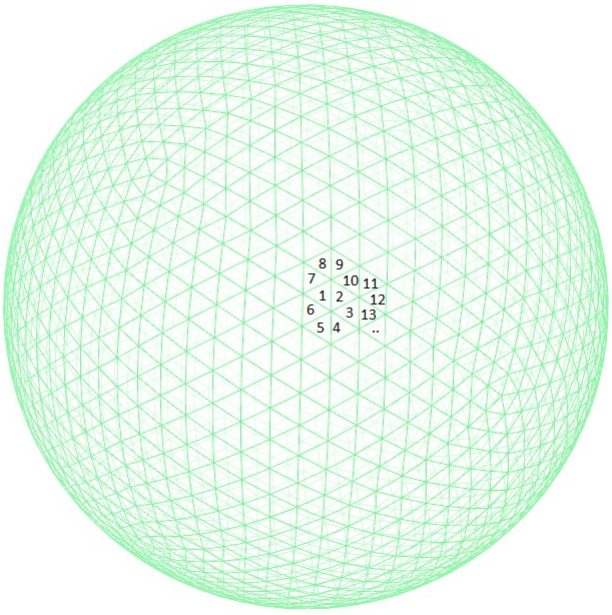


Figure 14: An indexed geodesic icosahedron that will be used as a tile system for loading the geometry.

5.2 The buildings

To be able to render as much buildings as possible we want as few polygons for each building and as big building objects as possible. You could have very complex and spectacular objects polygon-wise, but then get performance issues before you reach an amount enough to call it a city. For now I will use low polygon placeholder buildings and nascent textures that individually aren't spectacular but put together they will form a very large city that can be considered stunning - focusing on rendering as much buildings as possible.

The simplest form of what one with a bit of imagination could call a building would be one with 8 corners, as shown in Figure 15.

The 8-vertex building have a roof top, which I will be giving a different texture than the one to the sides of the building, shown in Figure 16. The building consist of 12 triangles, but we may discard the bottom two ground triangles as they won't be seen to the viewer.

To increase performance even further I experimented with making a billboard out of this type of 8-vertex building, so that we can make a level of detail out of our building geometry depending on how far it is from the camera. What this implies is to make a 2D plane object of the building that always faces the camera, and to the viewer it would look like a 8-vertex building object.

It wouldn't be completely trivial because there is a roof texture and a side texture, the billboard would need these two texture lookups and then a split interpolation depending on where the viewer is located angle-wise in respect to the building.

In addition, the dimensions of the billboard would have to be adjusted depending on the angle towards the viewer. If the viewer is above the billboard (90 degrees in respect to the plane the building is standing on) then the billboard size should change its dimensions to match that of the roof top dimensions, vice versa if the viewer is located on the side of the building (the roof top out of sight) then the billboard dimension should match the side dimensions.

However, since the billboard version of the buildings will only be used for the building objects that are far away from the viewer, it isn't necessary that they look perfect, just good enough to trick the viewer's perception of them.

The billboard's mesh is a simple plane with two triangles as shown in Figure 17. When textured with respect to the angle to the viewer it will look like Figure 18 when the viewer is slightly above the building, and when directly above the plane the dimensions will be equal to the roof top of the 8-vertex building and will thus look like Figure 19. The adaptive dimensions of both texture and vertices can be calculated in the shaders. And while the billboards may be considered to look very different from the 8-vertex buildings, we can see that in the resulting picture in Figure 10 that in the far distance it is very hard to distinguish the billboards from the 8-vertex buildings.

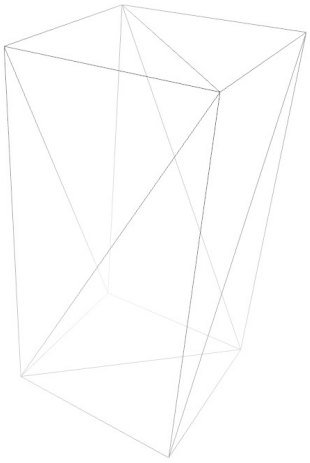


Figure 15: The full detail building object with 8 vertices, 12 triangles.

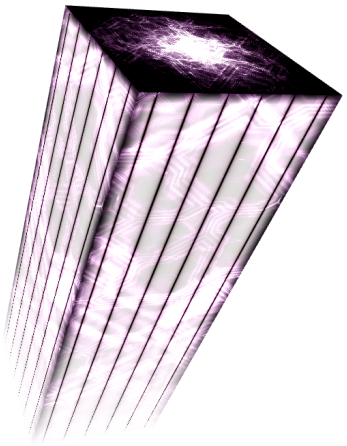


Figure 16: Textured 8 vertex building, with two different textures, one for the roof and one of the sides.

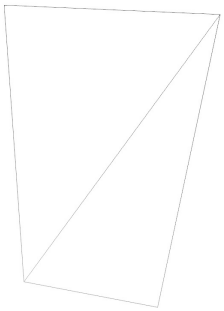


Figure 17: A billboard mesh with 4 vertices, 2 triangles.

Figure 18: The billboard building with adaptive dimensions for both the mesh and textures. When viewed from an angle slightly above the building.

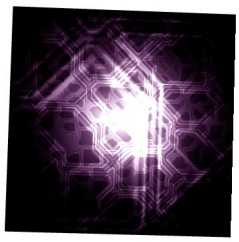


Figure 19: The billboard building when looked directly from above. The dimensions now being more square like than rectangular compared to Figure 18.

5.3 The website favicons

A favicon is known as the website icon or bookmark icon that is usually based on a company logo. The icon tends to be in the size of 16x16 pixels and typically populate most web browsers in the tab bar or bookmark bar as shown in Figure 20. The file is usually located on most websites at *http://<domain>/favicon.ico.* I found out that many favicons had the dimension 64x64 as well so we shall store the favicons in this size instead.

According to the concept description the favicons shall be textured onto billboards that hover above their respective building.

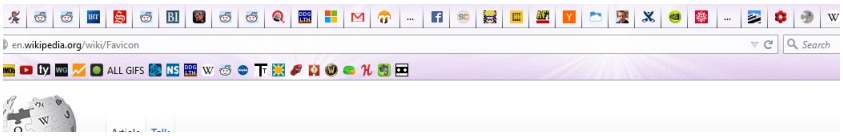


Figure 20: Favicons populated in the tab bar and bookmark bar in Firefox.

There are various of ways we could fetch the favicon from a website. We could fetch them on the fly through a servlet or other free query service, one which crawls the requested website for the favicon and returns the ico-file to the client, or we could pre-download the favicons and store them on the web server for easy fetching. Alternatively we could use both: fetching dynamically for websites that gets dynamically spawned in the world, and pre-downloading for all the websites that we know will exist in the world.

For convenience we could pre-download all the favicons by using the free public services that crawls the favicon for us, such as *http://www.google.com/s2/favicons?<url>* or *http://g.etfv.co/<url>*.

A Java-program that I created starts multiple threads that each separately takes an URL from our list of websites, appends it to the favicon crawling service URL and downloads the icon, for example:

```
File favicon = new File("google.ico");
URL url = new URL("http://g.etfv.co/http://www.google.com");
ReadableByteChannel rbc = Channels.newChannel(url.openStream());
FileOutputStream fos = new FileOutputStream(favicon);
fos.getChannel().transferFrom(rbc, 0, Long.MAX_VALUE);
```

This Java code snippet will give us the favicon corresponding to the given URL. However, in the case of *http://g.etfv.co,* if a favicon failed to be found they will return a placeholder favicon instead which can be seen as an Earth-like icon in Figure 24.

So when we have all the favicons we can easily create corresponding billboard meshes, displace them above the buildings and send in the favicon texture to their shaders.

5.4 The website elements

The website elements are a way for the user to explore the content of a website inside the 3D world, the result can be seen in Figure 21 which is a screenshot taken in front of the building "thedrum.com". The user was intended to be able to explore the website within this 2D frame, but this is heavily prototype code and I did not manage to implement that functionality. These website elements however can be used anywhere to load HTML content, for example in advertisement as well.

Figure 21: The website element attached on the side of the building for "thedrum.com".

Being able to render a HTML document inside a WebGL context is not completely trivial, the technicalities are explained in Section 5.4.1.

5.4.1 The technicalities of the website elements

In order to be able to visualize web pages and the content of the Internet inside the world we may use the CSS3DRenderer from Three.js. Why we need different renderers is because the WebGL context which uses the canvas element is isolated from the DOM-elements and HTML world, such that you can't put DOM-elements and render them inside the canvas. A solution to make it appear that HTML documents exist within the WebGL context is to combine the CSS3DRenderer with the WebGLRenderer [37].

The CSS3DRenderer can contain a scene with CSS styled elements that can be transformed in a two-dimensional or three-dimensional space. The trick is to put the CSS3DRenderer context behind the WebGLRenderer context using the HTML zIndex property of the elements. After that we enable alpha values on

the WebGLRenderer so that the canvas may render transparency and will in that way show the content of the CSS3DRenderer.

What this enable us to do next is to create a WebGL object in the WebGL scene and along with it create a CSS object within the CSS3DRenderer scene that share the same transformation at all times. Then we specify the WebGL object with zero opacity and render from both the CSS and WebGL scenes each render loop.

The result will be that the WebGL object create a transparent window through the WebGLRenderer which have mapped transformations (same size, rotation, translation etc) to the CSS object that is rendered in the CSS scene behind. In this way we may make it look as if HTML websites were inside the WebGL scene.

5.5 Camera controls

The camera that we use is a first person camera. So we need to define a behavior to that camera when the user gives a certain input on their keyboard or mouse.

The concept explains a camera that feels like a spaceship. A spaceship benefits from having many controls that let is maneuver in all directions with ease. The most basic controls used for todays first person games is the *WASD* keyboard setting. Where W moves the camera forward, S moves the camera backward, A strafes the camera left and D strafes the camera right. Along with a mouse that allows the user to look around in any direction by rotating the camera - the user will have the ability to move in any direction in the 3D world.

I'm using a file called FlyControls.js bundled within the Three.js zip which allows us attach this kind of keyboard setting to the camera. FlyControls.js also defines the keys Q to roll the camera left, E to roll the camera right, R to increase altitude and F to decrease in altitude. All which I think are beneficial controls for our camera.

The mouse steering is also defined in the FlyControls.js file. The mouse rotates the camera in corresponding direction after you hold down the left click button anywhere inside the viewport, and depending on how far off the mouse is from the center of the viewport the faster the camera will rotate. It is not how a standard free look camera usually works because a normal 3D application would typically fixate the mouse in the middle of the screen, but the web browser can't control the mouse in that way. It is a clever design for web browser 3D applications, so I decided to use the mouse controls in the way it was implemented.

The things I have had to adjust experimentally with the controls is how fast the camera will move and rotate when the user steers the camera.

5.6 Building interaction

When we have a lot of buildings inside the 3D world the user would most likely expect some kind of interaction with them. The features I decided to implement is the ability to right click a building, and receive a popup menu that show the name of the clicked site and two options. The options being *Preview* and *Browse in a new tab,* as can be shown in Figure 22.

Clicking the Preview option will attempt to load a website element on the wall of the building as can be seen in Figure 10. Clicking Browse in a new tab will simply open the website in a new tab in the web browser.

Figure 22: Right clicking a building will show a popup menu corresponding to that website.

5.7 Optimization: Levels of detail (LOD)

When we have different details on our building geometries we can utilize these by having lower detailed objects (billboards, 4 vertices) further away from the camera and fully detailed objects (8 vertices) close to the camera. In addition, since there will be little user interaction with the remote buildings we can bunt them up in a single merged geometry with the same shader, which can be calculated on the GPU. If we want to have better user interaction with the close fully detailed buildings we'll have to displace their geometry on the CPU, making it easier to intersect them in case we want collision detection or similar functionality.

We can even add another level of geometry between the fully detailed buildings and the billboards, and that is: fully detailed buildings that are vertex displaced on the GPU instead of the CPU.

So we'll have three levels of details, or rather three levels of handling the geometry (since the closest and second closest buildings both have the same detail, 8

vertices). We shall alias and summarize the three levels, starting with the level that will be closest to the camera:

1. B1: 8-vertex buildings. Vertex positions calculated on the CPU. Color, animated texture and a wind-like motion of the roof top vertices calculated on the GPU.
2. B2: 8-vertex buildings. Vertex positions calculated on the GPU. Color, non-animated texture and a wind-like motion of the roof top vertices calculated on the GPU.
3. B3: 4-vertex buildings. Vertex positions calculated on the GPU. Color and non-animated texture calculated on the GPU.

Each tile on the icosahedron will contain roughly 97 buildings as mentioned in 5.1. The B3 buildings will have a single merged geometry for all its tiles, which is very good performance-wise. Exceptions will have to be made for B1 and B2 which I chose to have merged geometries for each tile instead. The reason for this is that the icosahedron's spheric surface isn't completely round and this will create corners and crevices between the triangles such that both B1 and B2 - which are relatively close to the viewer - will need more proper vertex displacement so that they don't clip the surface of the icosahedron.

For B1 and B2 we can use the normal of the icosahedron's tiles to do the vertex displacement more properly, but this means B1 and B2 will have a tile dependency, and that implies making the shader unnecessarily complex if the geometry buffers are to contain buildings from more than one tile. This is less of a case for B3 because its geometry are far away and will not be seen if it clips the ground, so in that case it's sufficient enough to input the world radius to the B3 shader to get a good vertex displacement, thus removing any tile dependency for the B3 geometry (which is why it can be a single merged geometry).

We won't be using the traditional way of creating levels of details, which is to create multiple geometry buffers for each object and swap buffers as the camera moves close respective far away from the object so that the detail changes. Not only would it take more memory consumption and create long startup times just to initialize the objects, load textures, create shaders, input uniforms and attributes to them - it wouldn't be efficient with the way our buildings are located. Because we use merged building geometries for each tile and the buildings are located irregularly within each tile. Which means we would have to create a static building object for each tile, stored on the web server, which goes against the principle of making the world as dynamic as possible.

So instead, as an alternative, we'll initialize a big number of vertices that will be dynamically recycled from the current buildings and displaced to new buildings as the camera moves (for every tile separately). By assigning a fixed number of vertices for each of the different LODs we can adjust how much of the full detail buildings we want respective the lower detailed buildings.

In order to start vertex displacing our big merged geometries we need to assign which vertex will be linked to which corner of a building, this goes for both B1, B2 and B3. This is done when initializing the objects by sending in an attribute for every vertex in the geometry.

To clarify the above I'll give an example:

> Lets say that we have a B3 geometry enough for 50 buildings, this means $50 * 4 = 200$ vertices which we initialize in the beginning. And since the geometry will be recycled to newer buildings all the time as the camera moves, it means the first 4 vertices in our buffer may represent Google in one second and Youtube in the next, with relevant data (such as building height, position and texture) updated to the shader during this transition.
>
> For every building the shader also needs to know which of these 4 vertices will be the roof vertices respective ground vertices, so for each group of four vertices (modulo 4) in our vertex buffer we send in an attribute index ranging between [0, 3] which I call *displacementType*. displacementType 0 being the right ground vertex, 1 being left ground vertex, 2 being left roof vertex and 3 being right roof vertex.
>
> In this way if we look at our vertex buffer we can say that vertex 0 shall be the right ground vertex of the Wikipedia building and vertex 6 shall be the left roof vertex of the Yahoo building. The same goes for our B1 and B2 buildings except the vertex groups are of size 8.

In principle our LODs will be structured according to Figure 23 but as mentioned

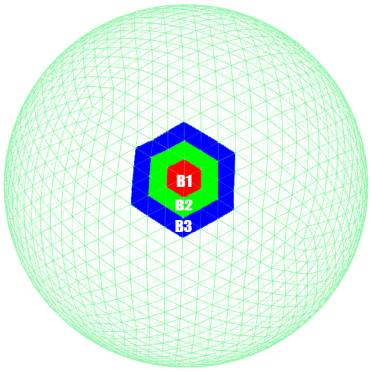


Figure 23: The design of the three building geometry level of details that will be dynamically loaded, the camera being located in the center of B1 in picture.

above we'll initiate the world with a fixed number of vertices for each level of detail, meaning that we decide how many tiles each level shall occupy, and thus we may have larger or smaller areas for B1, B2 and B3 as what is seen in Figure 23.

5.8 Optimization: merged favicon textures

In the same way we merge the geometry of the buildings for each tile, we can merge the geometry and the textures of the favicons into tile-wise groups too. The view should be pleasantly filled with favicon billboards that are neither too large or too small, doesn't clutter the view or are too far away the user is unable to tell their difference. So I decided they will only be drawn above the B1 buildings.

So again to reduce the number of draw calls we merge the geometry buffers of the favicon billboards to the same sizes to that of B1 - a merged geometry for each tile (˜97 buildings and hence ˜97 favicons). However, this complicates the procedure to draw them because of the following reasons:

1. The vertices doesn't know which favicon they should draw.
2. The vertices doesn't know which specific corner of the billboard they belong to.

We can solve these problems in the same fashion as we did for the B3 buildings by indexing the vertices in groups of 4 with the attribute variable *displacementType,* as mentioned in 5.7. And then each group of 4 vertices being an index itself to a certain favicon.

The favicons are very small individually and it would be inefficient to send them separately from the web server. To improve the performance of the fetching process I made the decision to merge the favicons into big mosaic textures, one for each tile, converted into uncompressed PNG files and stored on the web server.

Once the mosaic textures have been fetched by the client they will be inputted to the respective shader.

As shown in Figure 24 the favicons are aligned next to each other. However, for the shader to know which vertices belong to which favicon we need to input the following two things:

1. Shader attribute: the index of a particular favicon in the mosaic texture to each group of 4 vertices in our mesh. The indices starting at the left side of the mosaic texture at 0. Denoted *uvOffset.*
2. Shader uniform: the number of favicons the mosaic texture contain. Denoted *uvLength.*

By doing this we can easily transform the UV-coordinates to return the correct favicon to be drawn on the billboard. The new UV-coordinates in the shader becoming:

```
vec2 vUv = vec2((Uv.x + uvOffset) / (uvLength + 1.0), Uv.y);
```

Only the x-value (the width coordinate of the texture) will have to be transformed with the offset value, the y-value stays the same. The uvOffset value will have to be stored for each building along with the rest of the tile's data on the web server.

Figure 24: The favicon mosaic texture for a tile's 79 buildings. The texture was too long to fit this page so I had to break it down into three levels as you can see in the figure.

For the vertex displacement: since we calculated the B1 buildings on the CPU it means we have stored the building height variable of these buildings which we can input as an attribute to each group of 4 vertices to the favicon billboard shader so that it can correctly displace the vertices above a building.

6 Module: building positioning

There are different ways of positioning the buildings. So the first question is: how do we position the buildings geographically? It is certainly not practical to position millions of buildings manually, so that alternative got excluded very early. The idea of a force-directed algorithm is very attractive because it becomes much easier to iteratively calculate a geographical position on a set of nodes than to calculate it in one go. In fact, I have not found a solution to calculate a geographical distribution of equally spread points that isn't iterative.

The second question is: what will dictate how attracted one node will be to another in the force-directed algorithm? How much PageRank one website give another can be compared to how much a website likes another website. So naturally that seems like a good attraction factor to use in the force calculations between two nodes. However, in order to calculate the PageRank on our set of websites we need data about how many and which outlinks a website has. A crawler is needed to obtain that data.

This module will be divided into three programs:

- Crawler program written in Java. See Section 6.1.
- PageRank program written in Java. See Section 6.2.
- Force-directed algorithm written in HTML/JavaScript. See Section 6.3

These three program must follow a certain data pipeline because the force-directed algorithm is dependent on data from the PageRank program and the PageRank program is dependent on data from the crawler program, as shown in Figure 33. More about the data pipeline will be discussed in Chapter 7. It felt reasonable to refactor these three programs into its own module because they are the programs that decide where the buildings will be positioned.

6.1 The crawler

The crawler's main purpose is to traverse parts of the World Wide Web, finding unique valid website URLs and the data necessary to calculate PageRank over our set of websites. The data that the crawler have to find is the factor $F(p_j, p_i)$ from Equation 4, which is the number of links a website have to each of its unique outgoing link. Where $O(p_j)$ in Equation 4 is simply the size of the set $F(p_j, p_i)$ for each website p_j.

Figure 25 shows how I've designed the crawler. I chose to write it in Java because it's a language I'm quite familiar with.

For the crawler I use the standard classes HTMLEditorKit.Parser and HTMLEditorKit.ParserCallback for parsing the HTML code for href tags.

The crawler is multi-threaded in order to let us minimize the idle time that we have to wait for each HTTP connection to be established. Each crawler that is given a new URL will create threads to parse that domain without jumping to external domains, they will follow the relative href paths and stay within the same domain until a certain depth have been reached (I had a max depth set on 30), and while crawling they will store the number of times and to which external domains there are outgoing links to. Our purpose is to find the PageRank of entire domains, not to documents within domains, so our crawler may differ a bit from the design of typical crawlers.

Each of the SharedData objects are paired with a crawler and their purpose is to synchronize the deposit and fetching of crawled URLs respective URLs to crawl between the threads, and also to store the count of outgoing links in a datastructure. And because we store the data domain-wise not document-wise it means our set becomes rather small and we're able to keep this data in memory (at least for now).

In the future of this project this may be changed so that we only store a small part of the data in memory while the rest is continuously saved in files on the harddrive as the crawler program is running. For now the program only store the data in files when it has finished, such that we can load the data from previous sessions in order to start the crawler from where it left off.

The program is set to stop after we've crawled through a certain number of

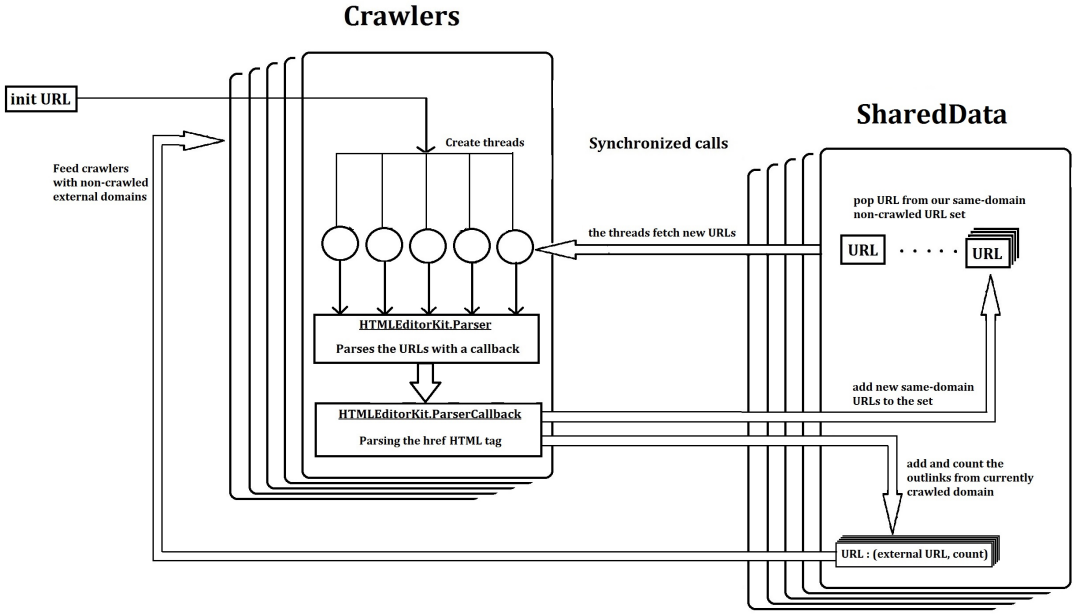


Figure 25: The flow of the multi-threaded crawlers.

websites.

The newly found external URLs that had the highest count and that we haven't crawled before we input to the crawlers, with the purpose to only crawl each domain exactly once.

6.2 PageRanks

The PageRanks of our websites are needed to calculate the height of the buildings in the world, and also for the force-directed algorithm to know how much a website is attracted to another website.

I've implemented the PageRank algorithm as described in Section 3.6 in Java but with a slight modification: $L(p_j)$ in Equation 3 in Section 3.6 assumes that all outgoing links only counts once. So if a document had several links to a particular website it would still split its PageRank equally among each unique outgoing link. Put in the perspective of a person randomly surfing a particular website this would imply that there is an equal probability of that person pressing either outgoing link, which is false. For example if a website have 90% of its outgoing links to wikipedia.com and 10% of its outgoing links to yahoo.com, there is certainly a much higher probability of the person randomly surfing to end up on wikipedia.com (heavily depending on website layout of course). So with this slight modification Equation 3 can be changed into the following:

$$PR(p_i) = \frac{1-d}{N} + d \sum_{p_j \in M(p_i)} \frac{F(p_j, p_i) PR(p_j)}{O(p_j)} \tag{4}$$

Where $O(p_j)$ is the total number of all outgoing links from website p_j, and $F(p_j, p_i)$ is the number outlinks that p_j has to p_i. $F(p_j, p_i)$ divided by $O(p_j)$ is then the fraction of outlinks that p_j has to p_i.

Once we've calculated all the PageRanks we can find the attraction between two websites, or rather how much PageRank a particular website gives to an outlink in the last iteration of the PageRank algorithm. I calculate this as $DR(p_j, p_i) = \frac{F(p_j, p_i) PR(p_j)}{O(p_j)}$ without the dampening factor involved. $DR(p_j, p_i)$ is needed in the force-directed algorithm in order to calculate the attraction force that the node representing website p_j will exert upon the node representing website p_i, and thus how close these websites will be to each other in the world. More of this will be further explained in Section 6.3.

6.3 Force-directed algorithm

The force-directed algorithm that we need to design is different from that of previous works mentioned in Section 3.7. The design concept suggest a city that

have equally spread buildings along the inside surface of the spheric world and on top of that being geographically located relative to each other. The crawling data obviously don't contain all the websites in the world, but the amount of websites the world shall contain may surely increase in future prototypes so it is necessary to make a model that support scalability. So this separates our force-directed algorithms from others that I've seen, our algoritm have very specific criterias to fulfill and therefor we have to analyze them carefully before going ahead. To isolate the criterias in a more structured way I alias them:

C1: The nodes shall be geographically representative of the crawled relational website data.

C2: The nodes shall be equally distanced.

C3: The nodes shall be displaced to form a spheric shape.

C4: The algorithm need to handle scalability in both time complexity and aesthetics for up to several millions of nodes.

We can utilize the spring laws to a certain degree but we need to adapt the general spring embedder model in order to meet those criterias. For C1 and C2 the spring model is enough to provide aesthetical results for smaller graphs but a big problem with force-directed algorithms in general is scalability, which C4 requires. The force-directed algorithm can be time consuming and the aesthetics can be heavily affected the larger the graph gets.

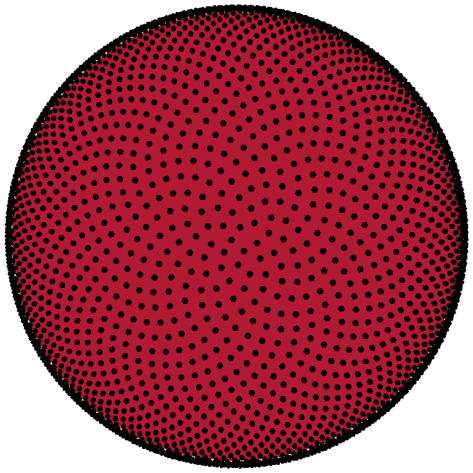


Figure 26: Equally distanced points along the surface of a sphere. Using the golden ratio to calculate the positions.

The ideal model that C2, C3 and C4 are striving for are spherical fibonacci points shown in Figure 26. The points are equally distanced and have an aesthetically

pleasing pattern, they are easy to calculate but the hard part is accomplishing C1 with the fibonacci point construction.

C1 is the critera that strives for a force-directed algorithm so we'll have to go with that. However, to get a better result in our force-directed algorithm we can use data from the fibonacci point structure, such as pre-calculating the average distance between two nodes and use this data to shape our model. That data may be helpful in fulfilling C2 and C3. How we design the force-directed algorithm is discussed in Section 6.3.1 and how the flow of the algorithm will behave is discussed in Section 6.3.3.

6.3.1 Designing the model

Lets start with a spring model that has the following attraction force law:

$$f_a(v_{j,}, v_i) = \frac{d(v_j, v_i)^2}{k}$$

Where node v_j is representing the website p_j and v_i is representing the website p_i. $d(v_j, v_i)^2$ being the squared distance between v_j and v_i, and $k = C\sqrt{\frac{area}{number\ of\ nodes}}$ which is the equilibrium distance between the two nodes/vertices as explained by Fruchterman and Reingold [25]. And a repulsion law:

$$f_r(d) = -\frac{k^2}{d}$$

If we imagine an example with the above laws where we have thousands of nodes randomly distributed over a 3D space. They are all equally attracted to one single node v_0, meaning they have the same attraction constant a towards v_0. These nodes will create a cluster around v_0 and will at the same time try to repel each other while being repelled by v_0. As a result of the repulsion some nodes may be repelled further away from v_0 than the others because there most likely isn't enough space to fit a thousands nodes at an equal distance around v_0.

What happens then if we have an attraction formula that scales with the squared distance towards v_0? Such that the further away a node is from v_0 the more force it will be given to reach v_0. The result will be that the nodes furthest out will become very aggressive to reach v_0 and the cluster will create a repulsion field blocking the nodes furthest out to get closer, but at the same time the cluster will be pushed closer and closer to v_0. Eventually the repulsion field may collapse if the attraction is too great (which is very bad).

The other worse case is that a node with higher attraction towards v_0 than other nodes may become blocked by the repulsion field from getting closer to v_0. Clustering is fine if it can be controlled.

Suggestively to fix the mentioned problems would be to start with low repulsion and high attraction values so that the nodes can reach their target before they repulse away from each other. This is where we experimentally decide the constant C. A lower C will increase the attraction while decreasing the repulsion.

The constant k assist in fulfilling the aesthetical side of C4 and partially solves the scalability issues since we input the area of our sphere and the number of vertices/nodes in our graph, which both are crucial factors for the algorithm.

To solve C1 we need to change the attraction law so that nodes with higher attraction get more force to move to their targets. We can do this by introducing a multiplier $a(v_j, v_i)$ into the attraction law f_a, which we can call the *attraction scalar*, being defined as:

$$a(v_j \mapsto p_j, v_i \mapsto p_i) = \frac{DR(p_j, p_i)}{\max\limits_{k,l} DR(p_k, p_l)}$$

Where we normalize $DR(p_j, p_i)$ by dividing it by the largest in the set we get a scalar in the interval [0, 1]. Ultimately what this means is that the node representing the website that gave the largest portion of PageRank to another website in the last iteration of the PageRank algorithm will receive $a = 1$, and thus gain the max attractive force a node can get. This should help to fulfill C1. The attraction force law now defined as:

$$f_a(v_j, , v_i) = a(v_j, v_i)\frac{d(v_j, v_i)^2}{k}$$

The nodes that we use in this algorithm will later be replaced by buildings once we have the positions, and the formulas we have obtained so far doesn't account for the radius or diameter of the buildings, such that two buildings could overlap each other. But since the building sizes will be set during the runtime of the world program we can adjust the dimensions dynamically later and negate this issue for now.

C2 and C3 combined with C1 calls for a more methodical force-directed algorithm.

The nodes only has to be displaced on the spheric surface in the end of the iterative process and only then needs to be equally distanced. We don't want the nodes getting blocked by other nodes when moving to their targets, that would damage the geographical property of the graph. And rather than keeping the nodes fixed on a spheric 2D surface through all iterations (which will root a lot more congestions); instead we may let the nodes move more freely in a 3D space at the beginning of the algorithm.

However, that will make it harder to transition between C1 and the rest of the

criterias, because the nodes will be clustered all over the place if we let them move completely free in a 3D space.

So suggestively we create a boundary space between two spheres; one inner sphere centered around origo with radius r_1 and one outer sphere also centered around origo with radius r_2 where $r_1 \leq r_2$.

The inner sphere will thus be inside the outer sphere, the nodes restricted between their surfaces, such that if $u_i = (x, y, z)$ is the position of node v_i and the distance it is from origo is $|u_i|$, the restrictive space for the node will be where $r_1 \leq |u_i| \leq r_2$.

The boundary space for the nodes is shown more clearly in Figure 28. In this way we will have the nodes in a more advantageable position where they early on resemble the shape of a sphere and will be much better organized to displace towards the spheric surface. And that can easily be done by incrementally increasing r_1 for each iteration until $r_1 = r_2$, and thus $r_1 = r_2 = |u_i|$.

So to summarize: we first focus on calculating the nodes into a geographical position in a slightly strict 3D space - giving the nodes enough iterations and force to travel across the entire space to find their attractive nodes, then we displace them correctly to the spheric surface by increasing the radius of the inner sphere until it has the same radius as the outer sphere (the nodes being pressed outwards by this), and at last we can distance the nodes equally among each other. This satisfies the criterias well so far.

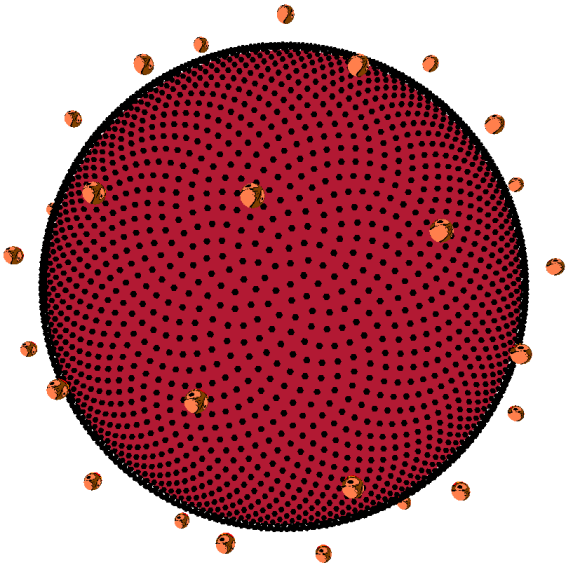


Figure 27: The orange gravity balls located above and around the spheric surface. Their positions being fibonacci generated points.

6.3.2 Optimizations to the model

However, if we start with low repulsion and high attraction we will be brought into another problem; and that is that the nodes may cluster up so much that the graph will become heavily weighted towards one end. That will make it harder for us to displace the nodes equally distanced over the spheric surface. It will take many iterations to make the nodes repel each other to spread over the surface by the repulsion force law.

Increasing the repulsion force too much and too fast if the nodes are heavily clustered may result in a repulsion explosion where nodes gain so much force they scatter all over the space, which will result in the geographic property of the graph being damaged. To solve this I introduce what I call *gravity balls.*

The gravity balls will be put around the spheric space using fixed fibonacci coordinates (so that they are equally spread around the surface) as shown in Figure 27. The gravity ball's purpose is to increase/decrease their gravity (attractional force) in order to balance the nodes around the space so that they don't get weighted towards one end of the space. E.g. if the majority of the nodes are located towards the right side of the space, the gravity balls on the right will decrease their gravity while the gravity balls on the left will increase theirs. The nodes will not move outside the spheric space so the gravity balls will have full control of them.

The gravity balls will help to fulfill C2, C3 and C4 but using them at the wrong time in the algorithm will inflict damage on C1. So only once we have displaced the nodes to the spheric surface and decreased the attraction scalar will we use the gravity balls to quickly wrap the nodes around the spheric surface. The number of gravity balls needed is an open question, the more we have the more evenly spread the nodes will get but the time complexity increases (each iteration each gravity ball need to calculate its gravitational force on each node). As a start I'll use 50 gravity balls in my algorithm.

To reduce the time complexity of the algorithm and ease C4 we can use an octree to our advantage. With means to change our solution closer to that of the Barnes Hut solution.

Where as instead of each node applying a repelling force to every other node in the set we would use the octree to search the local area of nodes and only repel those within close proximity. The number of attractive nodes that a node has also increases the time complexity. In order to reduce time complexity one could reduce the max number of attractive nodes that a node can have, or discard the low attraction forces that nodes have towards other nodes - but I chose to include all attraction force calculations since the set of nodes we use isn't extremely large. The octree will have to be created and destroyed each iteration to accommodate the node's new positions.

6.3.3 The stages of the algorithm

So far we've discussed the characteristics of our spring model and what our algorithm needs to fulfill the criterias. The majority of the details in the model have to be found out experimentally; such as tweaking the constant k in the spring model, the influence of the gravity balls or other parameters in order to keep the simulation stable.

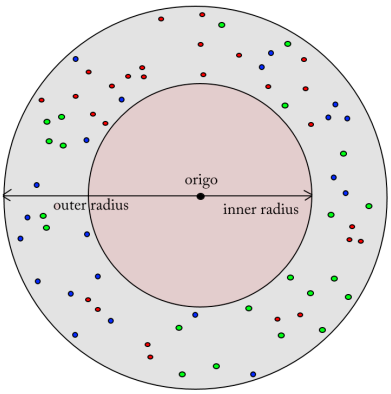


Figure 28: Stage 1: The nodes are positioned randomly inside the boundary space between the outer spheric radius and the inner spheric radius.

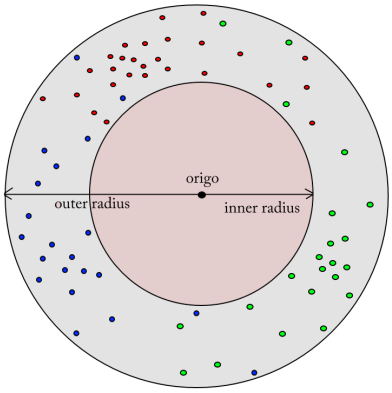


Figure 29: Stage 2: Nodes that have attractions towards other nodes will position themself close to them.

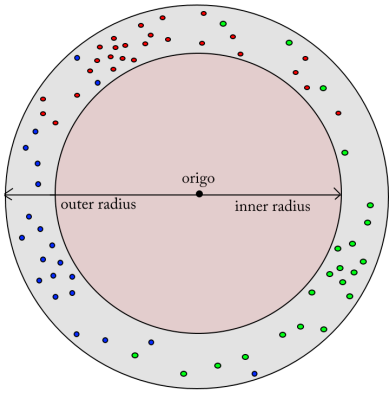


Figure 30: Stage 3a: The inner radius being increased to meet the outer radius. Pushing the nodes into a smaller space.

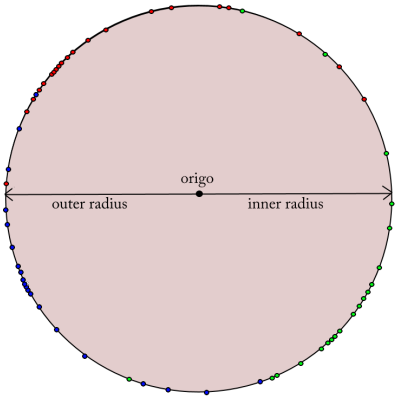


Figure 31: Stage 3b: The inner radius being equal to the outer radius. The nodes now displaced on the surface of the sphere.

We can still create a template for the algorithm which we break down into different stages, because it would be too hard to focus on fulfilling all the cri-

terias at once. At what iteration we'll start a new stage have to be figured out experimentally.

Note that Figure 28, 29, 30, 31 and 32 are 2D representations of the 3D environment where the same colored nodes represents nodes attracted to each other. The stages we have are:

Stage 1: Initialization stage. Place the nodes randomly in the spheric boundary space. See Figure 28.

Stage 2: Then we give the nodes high attraction and low repulsion in purpose to let the nodes position themself geographically. See Figure 29.

Stage 3: We turn off attraction calculations completely and gradually increase the repulsion. At the same time we increase the inner radius to slowly meet the outer radius as seen in Figure 30. We also let the gravity balls influence the nodes during this stage (not seen in Figures) so that they spread around the surface faster. When the inner radius equals the outer radius the nodes may still be clustered as seen in Figure 31.

Stage 4: The last iterations we decrease the influence of the gravity balls, and we mainly let the repulsion force law serve its purpose to push the nodes at an equal distance towards each other and give the result as seen in Figure 32. With hope of resemblance to the ideal node structure in Figure 26.

In the transition between Stage 2 and Stage 3 I will store a snapshot of the denseness of the graph. Evaluating how many nodes every node has within a certain proximity. I will use this data in combination with the PageRank to calculate the building heights. Because the denseness also reflects how tough it is for nodes to get into the middle of the cluster. So this will further enable the concept of having skyscrapers vs suburban districts.

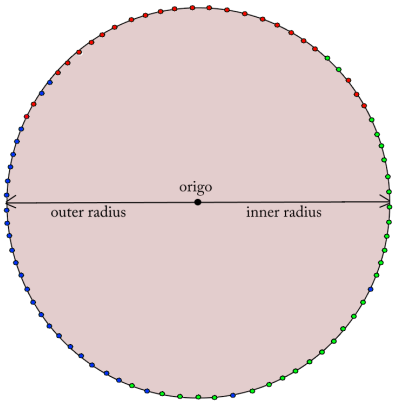


Figure 32: Stage 4: The nodes geographically positioned and equally distanced around the spheric surface.

6.3.4 An overview of the force-directed algorithm program

The program is written in JavaScript and HTML. Mainly because it simplifies the experimentational part of developing a good force-directed algorithm, and the octree among other JavaScript libraries are necessary for the visualization.

In the program there are numerous things that we have to do apart from running the actual algorithm itself. The most mentionable things are:

- We setup a couple of DOM-elements in which we will present data about the algorithm when its running. In this way we get feedback about the status of the algorithm while it's running.
- We parse a file that I call *dependencies_pageranked* that were outputted from our PageRank program (more about its relevance is discussed in Section 7.1). The file contains all the $DR(p_j, p_i)$ for our entire set of websites which we use to calculate the attraction scalar between our set of websites. Parsing this file will also give us knowledge of all the websites that exist in our set, so we create a node object for each unique URL that we find in the file.
- We also setup the gravity balls and their fibonacci positions. And for the gravity balls to know when they should attract or repulse, they need to know their equilibrium state. To find out the equilibrium state, or rather: "equilibrium distance", we can temporarily create the ideal node fibonacci structure shown in Figure 26 with the same number of nodes used in the real simulation, and calculate the average distance that each gravity ball have towards all the nodes. Then we average those values between the number of gravity balls that we have, and that will be our equilibrium distance constant. In this way our gravity balls will know for each iteration when the nodes as a whole are too far away or too close. This is done by subtracting the equilibrium distance from the current iteration's average distance of all the nodes, if the value is positive the gravity ball will attract and if its negative the gravity ball will repulse.
- The octree have to know how large its search radius shall be when we use it to search the close proximity for nearby nodes (in order to calculate the repulsion force exerted upon them). To calculate the value for the search radius constant of the octree I chose to utilize the ideal node fibonacci structure from Figure 26 again, where we aim to have the search radius around a specific node to contain a fixed and relatively low amount of nodes. This may not be optimal because when the nodes are heavily clustered a too wide search radius may contain a lot of nodes and will thus increase the time complexity for us. So in an optimal solution I would design this value to be adaptive to the state of the graph. But as designed now it's easy to calcuate, we only have to do it one time and it isn't completely arbitrary.

- Each and every iteration of the algorithm we want to backup the node positions in order to safely be able to restore the session in case the program crashes and we've wasted many hours running the program. This data will also be used to be able to replay the nodes and their position in a replay-program so that we can see the results of the full scale simulation in a visual form in order to confirm that the criterias of the force-directed algorithm hold their standard.

7 Module: the data pipeline of the back end programs

As explained earlier on it is a lot of data for a client to fetch from a web server at once, this means we have to partition the data in some way. All the data has to be stored and structured on the web server in a fetchable format for the client. So we need to define a data pipeline that explains how the data that we obtain/calculate in the different programs moves to the next program until it eventually is fully processed and ready to be put on the web server.

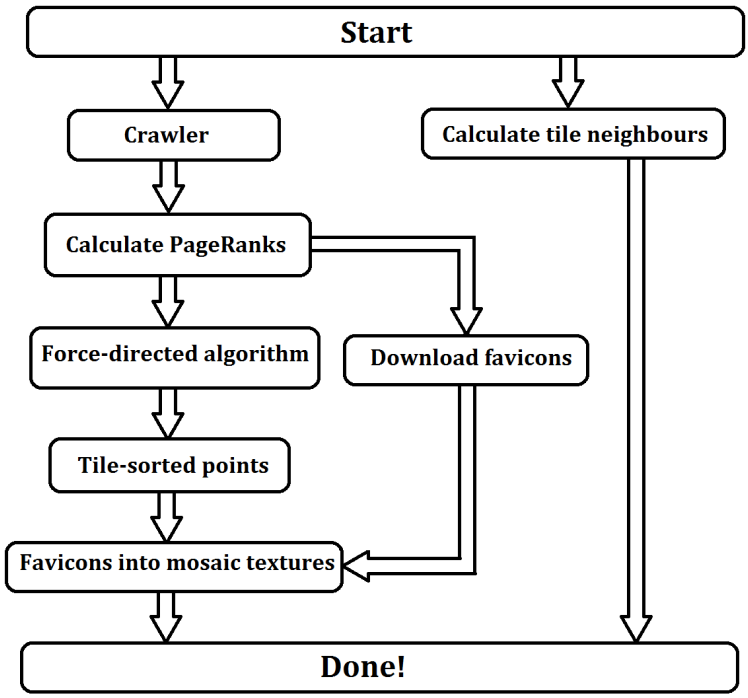


Figure 33: The data pipeline.

The tiles that we divided the world icosahedron object into will be the partitions that we use. The final output of the entire data pipeline is one file for each tile. In that file we need a list of all the websites that exist within that tile and their corresponding data. The data each website has is its (x, y, z)-position in the world, its PageRank, the value for how dense the surrounding area is with other websites, and its index for which favicon in the mosaic texture it shall fetch.

How we reach that final output is a combination of a lot of different programs working together. The process, the input and output of these programs are

explained in Section 7.1. How the back end side looks while the client is fetching this data and running the main program is explained in Section 7.2.

7.1 The input and output of every program in the pipeline

The pipeline I've designed is shown in Figure 33 which reveals the dependencies between the programs as well.

All the involved programs have an input, an output and a role that we'll have to define. In the following subsections we define these.

7.1.1 Calculate tile neighbours

The only independent program in the pipeline is the *Calculate tile neighbours* program shown in Figure 33 which purpose was explained in Section 5.1. This program would produce a file for each tile, labeled after the tile's index. Each file containing the 500 closest tile indices ordered by shortest distance. I call these *neighbouring tile files*. The format I've chosen for the files are:

$$t_1;t_2;t_3;t_4;\ \ldots\ \ t_{499};t_{500};$$

Where the tile indices t_x are separated by semicolons to make it easier parsing the content of the files later.

For example the file for tile indexed 101 looked like (with a few rows omitted):

```
75;100;102;76;74;103;124;126;99;77;73;125;48;46;104;127;123;
98;47;78;128;72;147;122;49;45;105;50;149;97;145;79;44;71;148;
146;17;129;121;19;15;106;18;51;150;96;16;80;144;130;43;151;70;
....
....
....
244;1253;1635;261;4346;1;1060;1146;1191;1170;298;4347;303;300;
1825;1210;1543;242;1644;1636;296;1059;1241;293;1147;245;1824;
1092;1093;302;278;1647;1542;277;1605;243;1604;1227;1062;294;1633;
```

5120 of these neighbouring tile files are outputted by the program and directly stored on the web server since the program has no dependencies.

7.1.2 Crawler

The data pipeline also begins at the *Crawler* program which do not require any input other than data from itself that it saved from previous sessions. Either

you could start it by initializing an URL or you load some of the popular non-crawled URLs from previous sessions and let it start from there. The resulting data is a file of all the outlinks and the number of times they have been referenced from every site that has ever been crawled. I call this file *dependencies* and the format I chose for this file is:

$;p_0$
$p_{99} : F(p_{99}, p_0)$
$p_5 : F(p_5, p_0)$
..
..
$p_{321} : F(p_{321}, p_0)$
$;p_5$
$p_1 : F(p_1, p_5)$
$p_{13} : F(p_{13}, p_5)$
..
..
$p_{87} : F(p_{87}, p_5)$
..

Where the semicolon marks the site that have been crawled p_i and the rows beneath are the corresponding outlinks p_j and the number of times they are referenced $F(p_j, p_i)$. Structured in this way makes it both comprehensible to read and easily parsed by the next program by using the semicolons to identify the crawled sites.

A snippet and shortened version of this file looked like:

```
;downloadsquad.com
scribol.com:1375
apple.com:625
google.com:500
engadget.com:375
;google.com
twitter.com:587
sverigesradio.se:238
doi.org:230
nyu.edu:165
svenskafans.com:99
nydailynews.com:93
facebook.com:81
;lendingclub.com
twitter.com:139
facebook.com:125
flickr.com:33
thedailybeast.com:24
123rf.com:20
barrons.com:16
```

The file became very big very fast. In total the program crawled 10881 websites and 515606 unique domains names were found.

7.1.3 Calculate PageRanks

The PageRank program will then take the dependencies file as input and output a file called *pageranks* and also a file called *dependencies_pageranked.* The pageranks file is just a list of all the websites and their corresponding calculated PageRanks sorted by highest PageRanks first. The dependencies_pageranked looks like the dependencies file except rewritten so that every outlink count $F(p_j, p_i)$ was replaced by the outlink PageRank $DR(p_j, p_i)$. The pagerank file was given the format:

p_0:$PR(p_0)$
p_1:$PR(p_1)$
p_2:$PR(p_2)$
..
..
p_N:$PR(p_N)$

The top 15 rows of this pagerank file looked like:

```
twitter.com:0.0025322202304912775
facebook.com:0.0015728024566919675
youtube.com:9.45567068854955E-4
google.com:8.774591834390169E-4
blogger.com:5.85864525584561E-4
t.co:5.10446423482761E-4
feedburner.com:3.6822434391256434E-4
linkedin.com:2.953042233835132E-4
apple.com:2.7352858933482263E-4
flickr.com:2.63343408510231E-4
wordpress.org:2.139109367816497E-4
pinterest.com:1.993901954737891E-4
addthis.com:1.8819293579710512E-4
wikipedia.org:1.8200003576211926E-4
amazon.com:1.7784445889018703E-4
```

The dependencies_pageranked file was given the format:

;p_0
p_{99} : $DR(p_{99}, p_0)$
p_5 : $DR(p_5, p_0)$
..
..
p_{321} : $DR(p_{321}, p_0)$
;p_5
p_1 : $DR(p_1, p_5)$
p_{13} : $DR(p_{13}, p_5)$
..
..
p_{87} : $DR(p_{87}, p_5)$
..
A snippet and shortened version of the dependencies_pageranked file looked like:

```
;downloadsquad.com
scribol.com:2.267211824060025E-7
apple.com:1.0305508291181932E-7
google.com:8.244406632945544E-8
engadget.com:6.183304974709158E-8
;google.com
twitter.com:1.319673432433264E-4
sverigesradio.se:5.3506350412115305E-5
doi.org:5.1707817625153446E-5
nyu.edu:3.709473873108834E-5
svenskafans.com:2.2256843238653004E-5
nydailynews.com:2.090794364843161E-5
facebook.com:1.821014446798882E-5
;lendingclub.com
twitter.com:2.975602063752133E-7
facebook.com:2.675901136467746E-7
flickr.com:7.064379000274848E-8
thedailybeast.com:5.137730182018071E-8
123rf.com:4.281441818348392E-8
barrons.com:3.425153454678714E-8
```

Which syntax-wise looks exactly the same to that of the dependencies file.

7.1.4 Download Favicon

The *Download Favicon* program goes through the pageranks file and download the favicon file for each website and saves them in their standard ico-format. Since the pageranks file is sorted by highest PageRanks the program can for ex-

ample also be set to download only the top 20 000 favicons from their respective websites. This program can be run independently, the favicons are not needed until later in the pipeline which means we can run it in parallel with the next program.

The favicons were named after the domain they were obtained from. A small part of the folder is shown in Figure 34.

7.1.5 Force-directed algorithm

The *Force-directed algorithm* program takes the dependencies_pageranked file which it parses to obtain all the $DR(p_j, p_i)$ which are going to be the main factor in calculating the attraction force between two websites. The force-directed algorithm program will then output the 3D position for each of the websites in a file, the positions will be vectors outgoing from the origo. We also store what I call the *density* value which is - as mentioned briefly in Section 6.3.3 - how many nodes there are in the close proximity. I called the output file *force_points* which were given the format:

$p_0 : x_0 ; y_0 ; z_0 ; d_0$
$p_1 : x_1 ; y_1 ; z_1 ; d_1$
$p_2 : x_2 ; y_2 ; z_2 ; d_2$
. .
. .
$p_N : x_N ; y_N ; z_N ; d_N$

Where (x_0, y_0, z_0) is the position for website p_0. Every website p_i up to p_N

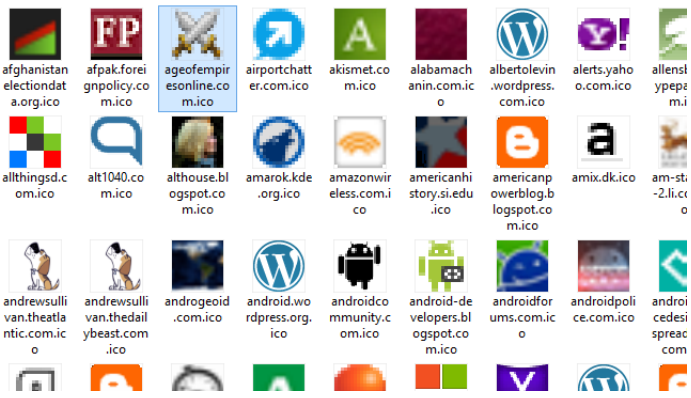


Figure 34: The favicons saved in their ico format and named after the domain they were obtained from.

have their position appended after them with their corresponding density value d_i.

A snippet of the force_points file looked like:

```
privatehimalaya.com:1.413518538730341;-0.7734547175520732;4.733258195002897:39
kenyasafarisholiday.com:1.6167006846267884;-1.041328882194241;4.615399555340292:21
tudorinnhotelvictoria.co:1.2909349767164988;-1.0899537171104756;4.504409544996605:8
vietlandexplore.com:1.4699592694805839;-0.7652309773969987;4.717376526979814:36
apyork.com:1.626277444091639;-1.0984664330578033;4.5987599600635845:12
perugrandtravel.com:1.3126267175518336;-0.8309869724337513;4.689326639482708:25
```

But it is one huge file that isn't partitioned in any way, so that is for the next program to solve.

7.1.6 Tile-sorted points

The *Tile-sorted points* program takes both the pageranks and force_points files as input and as its name states it sorts the websites by its position into its respective tile. The program will create one file for each tile (assuming all tiles contain website positions which the force-directed algorithm should have handled by distributing them relatively equally), the file named after its index, the index ranging between [0, 5119] as explained in Section 5.1. Each file will contain the website data that we've calculated gathered so far. I called these files *website tile files*.. The format for each of these files looked like:

$p_0\ \ x_0\ \ y_0\ \ z_0\ \ PR(p_0)\ \ d_0$
$p_1\ \ x_1\ \ y_1\ \ z_1\ \ PR(p_1)\ \ d_1$
$p_2\ \ x_2\ \ y_2\ \ z_2\ \ PR(p_2)\ \ d_2$
..
..
$p_N\ \ x_N\ \ y_N\ \ z_N\ \ PR(p_N)\ \ d_N$

Which contain the position, PageRank and density value for each website within that tile.

A snippet of the output for the website tile file named 3326 looked like:

```
lighthouse.org -0.027611263252377883 -0.6038875802763519 -0.7965911175311838
1.8884609543452492E-6 0.013358778625954198
rotinrice.com -0.0607583995530293 -0.5961176595832102 -0.8005948743383201
1.8730126686738426E-6 0.03244274809160305
marioarmstrong.com -0.043496308524130174 -0.5828299499290801 -0.8114291839775297
1.8772635234956574E-6 0.015267175572519083
sleeveshirtconsulting.com -0.05372305170408592 -0.6033727937342573 -0.7956476013266913
1.874019038572604E-6 0.02480916030534351
telesud.com -0.0224742011854665 -0.5961616122489666 -0.8025498379301985 1.873350265806847E-6
0.015267175572519083
pprotect.org -0.029761251133130162 -0.598510960663493 -0.8005616140539425
1.8728669773118518E-6 0.013358778625954198
```

The program resulting in 5120 files, each file containing the data for roughly 90-100 websites. I took the chance to normalize the density values in this program as well.

7.1.7 Favicons into mosaic textures program

Last in the data pipeline we input the favicons given by the Download Favicon program and the force_points file to the "Favicons into mosaic textures program". This program goes through the websites in each website tile file and merge all the corresponding favicons into a horizontal mosaic PNG texture, an example shown in Figure 24. The mosaic texture file inheriting the name of the website tile file.

As mentioned in Section 5.3 the uvOffset which is the favicon's index in the mosaic texture needs to be stored in the website tile files after we've created the mosaic textures. If we denote the uvOffset for a particular website as $U(p_i)$ and append them with a whitespace to the website tile files, the final format of the files will become:

$$
\begin{array}{l}
p_0 \;\; x_0 \;\; y_0 \;\; z_0 \;\; PR(p_0) \;\; d_0 \;\; U(p_0) \\
p_1 \;\; x_1 \;\; y_1 \;\; z_1 \;\; PR(p_1) \;\; d_1 \;\; U(p_1) \\
p_2 \;\; x_2 \;\; y_2 \;\; z_2 \;\; PR(p_2) \;\; d_2 \;\; U(p_2) \\
.. \\
.. \\
p_N \;\; x_N \;\; y_N \;\; z_N \;\; PR(p_N) \;\; d_N \;\; U(p_N)
\end{array}
$$

Figure 35: The mosaic favicon texture for tile 3326.

A snippet of the website tile file 3326 now looking like:

```
lighthouse.org -0.027611263252377883 -0.6038875802763519 -0.7965911175311838
1.8884609543452492E-6 0.013358778625954198 5
rotinrice.com -0.0607583995530293 -0.5961176595832102 -0.8005948743383201
1.8730126686738426E-6 0.03244274809160305 9
marioarmstrong.com -0.043496308524130174 -0.5828299499290801 -0.8114291839775297
1.8772635234956574E-6 0.015267175572519083 7
sleeveshirtconsulting.com -0.05372305170408592 -0.6033727937342573 -0.7956476013266913
1.874019038572604E-6 0.02480916030534351 15
telesud.com -0.0224742011854665 -0.5961616122489666 -0.8025498379301985 1.873350265806847E-6
0.015267175572519083 29
pprotect.org -0.029761251133130162 -0.598510960663493 -0.8005616140539425
1.8728669773118518E-6 0.013358778625954198 25
```

With the corresponding mosaic favicon texture 3326.png shown in Figure 35. The results can be confirmed by checking in a web browser that *lighthouse.org* which have uvOffset 5 have the favicon shown at the 6th position in 3326.png, or that *rotinrice.com* with uvOffset 9 have the favicon at the 10th position in 3326.png. A uvOffset -1 indicating that there is no favicon for the given website.

And at this stage we're done, we have all the data that we need: the neighbouring tile files, the website tile files and the favicon mosaic textures. Which we now can store on the web server.

7.2 What happens in back end when running the main program?

From the moment the user connects to the web server from the web browser there are numerous events happening along the way until any frame of the world is drawn.

As the Figure 36 shows we start by downloading all the libraries used in the program, this included libraries such as Three.js, *jQuery*[18], *Tween*[19], *dat.gui*[20] and all the self made libraries. The shaders and textures for the world and the building objects are downloaded as well.

At the initialization phase we utilize the Three.js library to create the the scenes, renderers, controls, materials and the different geometry shapes that we need. We attach the camera to the scene which will receive the position (x, y, z) = (0, 0, 0) that we translate to a distance a bit less than that of the world's radius.

Along with the WebGL scene that we put all our buildings and world objects in we'll also create a CSS-scene (same scene-class from Three.js, it is needed so that we can separate the scenes and their content for the different renderers) that we are going to put our website elements in as explained in Section 5.4. Both scenes will need different kinds of renderers for their different purposes, THREE.CSS3DRenderer for the CSS scene and THREE.WebGLRenderer for our WebGL scene.

We give the CSS3DRenderer's zIndex the value -3 and the WebGLRenderer's zIndex to -2. Such that the CSS3DRenderer have the lowest stack order with WebGLRenderer being on top. Then we load the controls THREE.FlyControls which gives the camera the feeling of controlling a spaceship with the ability to move freely in 3D.

[18] jQuery is a JavaScript library with an API that simplify things such as document manipulation, event handling, AJAX requests and animation for HTML.

[19] Tween is a JavaScript written tweening engine for easy animation. Tweening is an interpolation technique where an animation program generates extra frames between the key frames that the user has created. This gives smoother animation without the user having to draw every frame. https://github.com/sole/tween.js/

[20] dat.gui is a lightweight controller library for JavaScript. https://code.google.com/p/dat-gui/

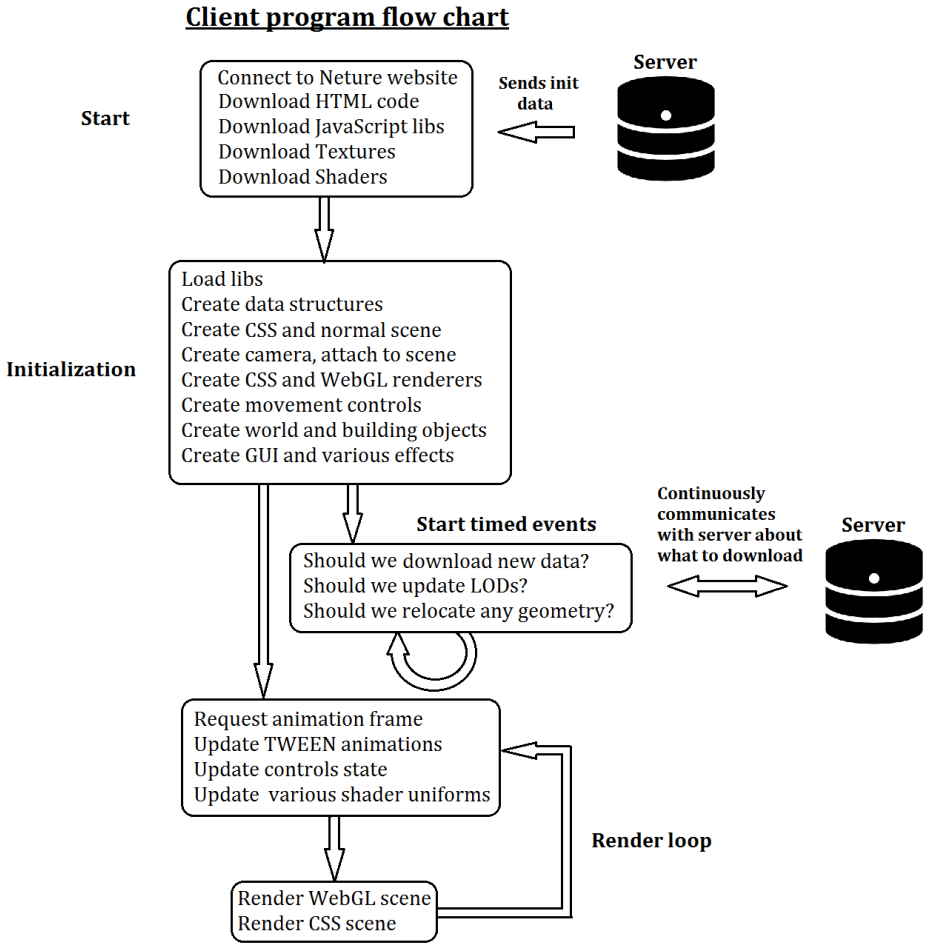


Figure 36: The technical flow of the program from the client point of view.

After we've created and attached the objects for the website elements, the star, the world to respective scene - we immediately start fetching the first neighbouring tile data since the camera is ready in its initialized position.

In the end of the initialization phase we start numerous setInterval timed events which with their specified interval will execute a function in a timed manner. These timing events will continuously check our state in the world and see if something needs to be downloaded or updated in our data structures.

In the render loop we do what is described in Section 3.3.3. All the shaders that change with time and are dependent of it need the time uniform updated from the render loop. This includes for example our B1 and B2 buildings because their textures are animated. Apart from uniform updates we also need to update the controls that handles the camera state. The objects that are connected to the TWEEN library need their state updated for eventual tweening.

Once everything is updated we first render the WebGL scene with the WebGLRenderer and then our CSS scene with the CSS3DRenderer. We let the render loop continue and request new animation frames to draw as fast as possible.

The most important timed event that runs in the background is the one that checks when the camera fly above a new tile from that of the previous tile, because that is when we should construct three new LODs and displace relevant building geometry. To check whether we've entered a new tile we can create a vector between the origin of the world icosahedron and the camera position and then intersect the icosahedron triangles (using an octree here too) to find the tile's index and compare this index to the previously stored index. This timed event doesn't need to happen very often but I've set it to every 200ms for now.

So for every new tile we enter we need to download that tile's neighbouring tiles as explained in 5.1, and for each newly found neighbouring tile that we haven't discovered we need to download that tile's building data.

The large problem with constructing the LODs is to dynamically assign which tiles will be given to which LOD. The assignment of tiles to geometries need several data structures to keep track of the different indices. For structuring our LODs we only need to know the neighbouring tile indices, but for the vertex displacement we need to know the positions of the buildings.

For the interested I'll give an example of how I handle this procedure in 7.2.1.

7.2.1 Example: Loading the server data and constructing the LODs

Lets say we start the world and decided the client's computer have enough performance for 13 B1, 24 B2 and 36 B3 tiles, we initialize these objects and the world, we have indexed the tiles of the icosahedron as in Figure 37 and the camera is located above tile 1. The client do not know where tile 1 is in respect to

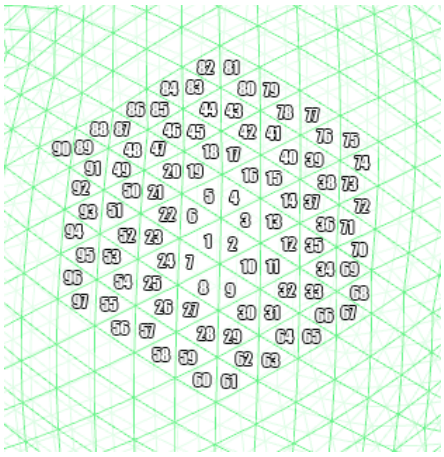


Figure 37: The indexed tiles of the world icosahedron.

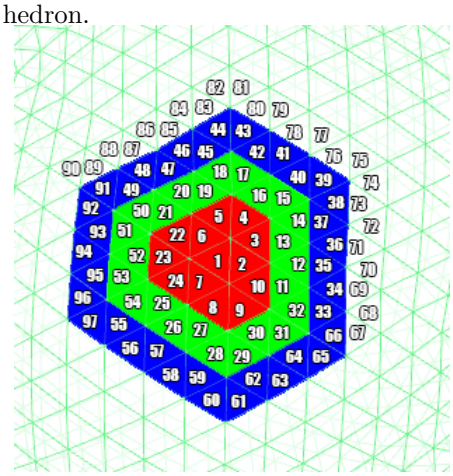


Figure 38: The LOD regions when the camera is located on tile 1. Red: B1, Green: B2, Blue: B3.

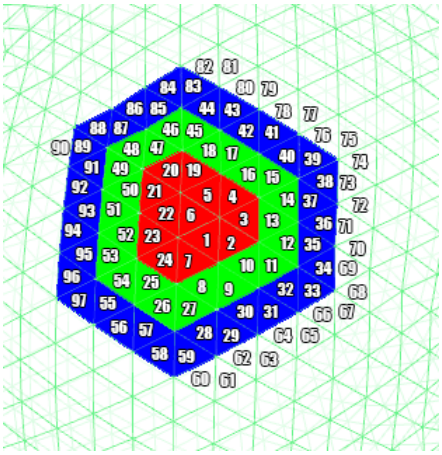


Figure 39: The LOD regions when the camera is located on tile 6. Red: B1, Green: B2, Blue: B3.

every other tile. When our setInterval routine that continuously intersects the camera's position and notices this is our first time visiting tile 1 and we have not downloaded the neighbouring tiles for tile 1, then we request the neighbouring tiles for tile 1 from the web server. The web server will for example return a file that looks like:

```
2;3;4;5;6;7;8;9;10;11;12;13;14;15;16;17;18;19;20;21;22;23;24;
25;26;27;28;29;30;31;32;33;34;35;36;37;38;39;40;41;42;43;44;
45;46;47;48;49;50;51;52;53;54;55;56;57;58;59;60;61;62;63;64;
65;66;67;68;69;70;71;72;73;74;75;76;77;78;79;80;81;82;83;84;
85;86;87;88;89;91;92;93;94;95;96;97;90;
```

Which in this example contains the 96 neighbouring tile indices sorted by smallest distance (which outside this example would contain the closest 500 tiles). We save the neighbouring tile data in our data structures so that we don't have to download this file more than once.

Before displacing any geometry we need to know what buildings every newly discovered neighbouring tile contain, so we send requests to the web server again to download each individual neighbouring tile's data. At the same time we send requests to download each newly discovered tile's favicon mosaic texture. Here again we save which tile we've downloaded what from so that we don't need to download it more than once. The content of website tile 1's data may for example look like:

```
facilitatedigital.com -0.59570 0.797168 0.098297 1.8832545513680896E-6 3
sweetlemon.me -0.557937 0.823783 0.100433 1.874159172733451E-6 2
almostinfinite.com -0.558842 0.824305 0.090633 1.8722077954075248E-6 4
gbn.org -0.590177 0.80187 0.093179 1.8766017795884329E-6 0
bigelow.org -0.575576 0.811080 0.104213 1.8741939740419658E-6 1
```

Every line in this file represent the data for a building with the format "website x y z PageRank densityValue uvOffset". In this example there are only 5 buildings in tile 1, but outside this example we would normally have ˜97 buildings in every tile when the buildings are distributed well.

The client saves the building data into data structures. So now we have all the data needed for displacing our geometry; the position of the building, the PageRank and density value which we'll use to calculate the building height, the uvOffset which we use to draw the corresponding favicon above the building.

Since this is our first time displacing there is nothing to relocate from a previous tile position. So we assign the closest 13 tiles inclusive tile 1 to the B1 buildings, the next 24 tiles to B2 and the next 36 tiles to our B3 buildings as shown in Figure 38.

The data about which tiles belongs to which LOD and hence which tile belongs

to which geometry object at any given time needs to be saved in data structures. In short: we link the tile indices to geometry indices creating tuples such as {geometryIndex : tileIndex}.

After the assigning is complete we push the tile tuples into a relocation queue to prepare the new buildings. Every once in a while we pop the entire relocation queue to start the displacement and update all the attributes and uniforms for the B1, B2 and B3 buildings and for the favicon objects. Then we have our buildings visualized the next render cycle.

When the camera moves and the setInterval routine that check which tile the we are in notices we have moved above for example tile 6 instead, we repeat the procedure. In short:

1. Download the neighbouring website tile data for tile 6.
2. Download newly discovered neighbouring tile's indices.
3. Download the favicon textures.
4. Assign which tiles are in which LOD.
5. Assign the tile indices to geometry indices inside their respective LOD, creating the tuple {geometryIndex : tileIndex}.
6. Push the tile tuples that needs to be relocated into the relocation queue.
7. Pop the queue, displace the geometries and update attributes and uniforms.

Note: In step 4 we obtain the new LOD regions from tile 6 as shown in Figure 39. But there is a collision between the LOD regions compared to where the buildings are at the current, the current being shown in Figure 38. For example tile 9 which were a B1 should now be a B2, and tile 30 which were a B2 should now be a B3. So our algorithm have to find out which tiles needs to be switched to another LOD since we don't want to relocate geometry that already have the correct LOD. And this switch have to happen in the same render cycle or we'll notice empty tiles that stay empty until it have a geometry that have filled them, which is why we pop the entire relocation queue at the same time.

So that is how we relocate our geometry to its second location around tile 6.

8 Development and equipment

What to keep in mind for this project is that it shall be a game world where multiple users can join and explore the world in a more adventurous way of web surfing. So it is important the world have an aesthetic side and compete with modern game graphics. It is not reasonable to expect every user to have the latest hardware technologies in their main computer, there is tradeoff between good graphics and availability that this project will have to meet. I decided that my home computer was a fitting benchmarking system to meet this criteria. The hardware as specified in 8.1 have manufacture dates ranging between 2009 and 2011, it's enough to run most modern games on medium or high graphical settings at 30 or more FPS.

8.1 Hardware

I chose to optimize the application on my home computer, the specifications is as follows:

- CPU: Intel Core i5 2500k 3.3GHz
- GPU: AMD Radeon HD 5850
- Memory: Corsair 16 GB, 1600 MHz

8.2 Software

The working environment have its root in Windows 8 and using Eclipse Kepler/Luna [38] which have support for Java projects and Dynamic Web projects. The languages varied between Java, JavaScript, WebGL, HTML and PHP, such that a multiple language IDE as Eclipse with compiler and interpreter have been mandatory to have a good workflow. Eclipse also comes along with a great number of different installable plugins such as Subversive (SVN) which integrates version control and have a systematic way of storing your project files in a repository, SVN is also very easy to use once it is integrated. TortoiseSVN [39] was used to create a SVN repository on Dropbox [40] to have a backup of the project at all times on the internet. However, in the later phase of the project the sheer number of files and big file sizes impacted both version control and committing/pulling to/from a repository heavily because it made Eclipse crash frequently due to out of memory issues. After a certain point of which I switched to Sublime Text 3 [41] which is a minimalistic lightweight source code editor with smart plugins and features to support programmers, I continued to use SVN along with Sublime Text instead.

The main program along with the other web-based programs I deployed on an Apache HTTP Server by setting my workspace as document root, making it

easy to run all the web projects on localhost. I used XAMPP [42] by Apache Friends for this. XAMPP is an acronym for **X**-platform (read as cross-platform), **A**pache HTTP Server, **M**ySQL, **P**HP and **P**erl. All parts except Perl was used. Very early in the project I used MySQL to create a database locally and store the crawler data, but this later turned out to be ineffective rather than just storing the data in simple text files on the web server.

Google Chrome have been the main browser for testing and the most reliable to run my project. But Firefox, Safari and Opera among other web browsers have support for WebGL. Waterfox which is a 64-bit web browser was needed to run the memory-heavy force-directed algorithm, in which standard 32-bit web browsers would have become unstable when reaching above 3-4 GB RAM. However, Google came with a stable release of 64-bit Chrome in the summer of 2014 [43] which I adopted for my project for when it was needed to run the force-directed algorithm again.

9 Algorithms testing and validation

During the development the force-directed algorithm I had to test it thoroughly and validate the results that the program outputted. Section 9.1 explains the process that I had when developing the algorithm. Section 9.2 discusses the tweaks that had to be done after validating a test run and also shows the final back end results of the algorithm.

9.1 The experimentation process

The tough part with the force-directed algorithm was to make the small scaled simulation create a good and stable outcome for the large scaled simulation. Since the details of the algorithm had to be found out experimentally it meant we couldn't spend too much time running the simulations, we had to run the algorithm in a rather small scaled environment before the larger simulations were executed. And these experimental simulations had to be displayed visually in order to be able to interpret the result, so I chose to represent the nodes as low polygon spheres that were colored after how attracted they were to nodes with similar attractions. The gravity balls were also displayed - changing in size depending on how much gravitational force they exerted.

When doing the full scaled simulation (over the entire set of websites) there was no need to visualize the environment at runtime, so in that case the code that created the meshes could be turned off,. The rendering of the WebGL environment could be turned off completely. Only periodic feedback about the progress of the simulation was turned on, such as how many iterations that were left.

Typically a simulation that were rendered and displayed with 5000 nodes took 3 minutes to run while a simulation with 500 nodes only took 15 seconds. So the key part to develop a good model was to follow these steps:

1. Tweak the model
2. See the result in the 500-node simulation. If the results looked good then continue, otherwise back to step 1.
3. Try the 5000-node simulation.
4. Evaluate results and go back to step 1.

These steps were repeated until a good stable model had been found. If we were confident about the 5000-node simulation we could try the full-scale simulation. In this way the number of iterations needed and at which iteration to do what could be found out experimentally.

9.2 The final revision and its result

Through extensive experimentation with the force laws I managed to get a decently stable simulation in both the small scale and the large scale environment. I gave the full-scale simulation 400 iterations which took 31 hours to complete. Stage 1 was the initialization stage (iteration 0), stage 2 I gave 130 iterations, stage 3 I gave 40 iterations and stage 4 I gave 190 iterations.

In the initialization phase I set the octree search radius as $octree\_search\_radius = 2\sqrt{\frac{area}{number\ of\ nodes}}$. At every iteration in the algorithm for each node we would do an octree search with the calculated radius and most likely find a couple of nodes nearby which would then exert a certain repulsion force. This would be the main factor in scaling our time complexity towards $O(n\,log(n))$, but when introducing gravity balls among other things I sacrificed a bit of time to get a better resulting graph.

At iteration 130 when the nodes had reached their highest geographical point I stored the density value of each node. The density value being the number of nodes within the octree search radius from each respective node.

Many times the clustering of the nodes became too dense and the repulsion formula were too aggressive which caused the nodes to explode in chaotic ways and damage the graph. So I had to find the right balance between attraction and repulsion at the right time in the algorithm. Alongside this I introduced a mechanic on the node's forces so that each node wouldn't be able to turn abruptly around 180 degrees from one iteration to the next. However, this calculation would also increase the time complexity but at a cost of a better resulting graph.

At the beginning of the algorithm I set the attraction force to be high (and thus repulsion low) by having a low equilibrium distance between two nodes as defined as $k = C\sqrt{\frac{area}{number\ of\ nodes}}$ where $C = 0.1$, $area = 4\pi r^2$ and where $r = 5$ is the outer spheric radius.

I kept the outer spheric radius fixed at 5 units during the whole algorithm. 5 units of radius does not say much, but for comparison the inner spheric radius were set to 4. So until we began increasing the inner spheric radius to be equal to the outer spheric radius the nodes had the 1 unit difference of 3D space to move inside. I experimented with lower inner spheric radiuses, such as starting with inner spheric radius of 0 then increasing it towards a radius of 5 as the iterations passed, but that would result in nodes clustering too much in the middle and the nodes would be pushed too quickly away from each other. In some cases nodes with high attraction towards each other could be unlucky and end up on opposite sides of the space.

I also experimented with having an inner spheric radius very close to 5 but as predicted previously congestion became a problem, where nodes stuck on the other side looked like they moved in traffic, zigzagging endlessly between other

nodes before they could reach their attracted cluster. Having that 1 unit of spheric space helped to reduce congestion and get a better geographic result in the graph.

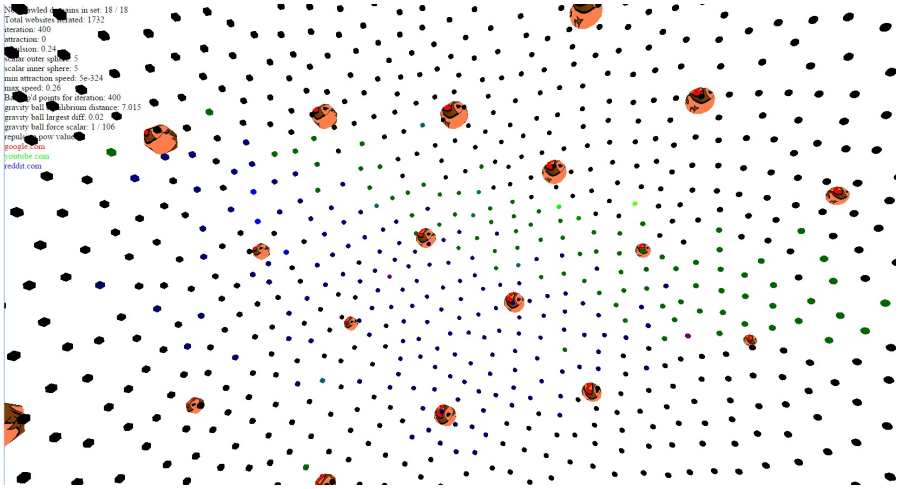


Figure 40: The result of our small scaled force-directed algorithm simulation with 1732 nodes. The point of view is from inside the spheric surface, with the orange gravity balls being on the outside.

Optimally one would have to change k dynamically as iterations goes since the area that we input is the area of a sphere and the formation of the nodes is only spheric in the end of the algorithm. But the results turned out good even without this optimization so I did not proceed in implementing it.

I slightly altered the repulsion formula to $f_r(d) = -\frac{k^2}{d}$ which worked well for stage 2. But when reaching stage 3 and 4 where we focused on spreading the nodes over the spheric surface the nodes looked aesthetically better when you had a d powered with 2. So the new repulsion formula became $f_r(d, x) = -\frac{k^2}{d^x}$ where I had $x = 1.0$ in the beginning of the algorithm, and once stage 3 began I interpolated x between 1 and 2 until we reached iteration 210. This gave a smoother transition in the increasingly higher repulsion force exerted and also further prevented the system from exploding chaotically.

At the beginning of the algorithm I also forced a minimum speed on all the nodes so even those with very low attraction force could reach to their targets, otherwise they would not have had the chance to travel across the entire space. This minimum speed iteratively decreased towards 0 as the iterations passed during the attraction stage (stage 2). I also had to introduce a maximum speed so that nodes that were given too much force during an iteration wouldn't be able to teleport to the other side of the space in one iteration.

The result of the small scaled simulation is shown in Figure 40 where the coloring represent nodes with similar attractions.

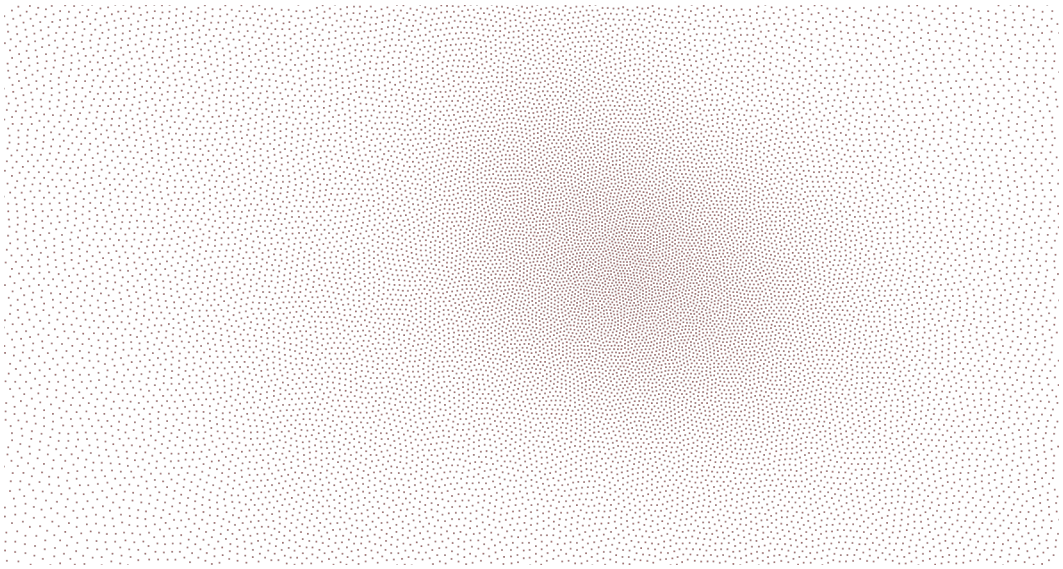


Figure 41: The result of the full scaled force-directed algorithm simulation with 515606 nodes. The point of view is from inside the spheric surface zoomed in on the nodes.

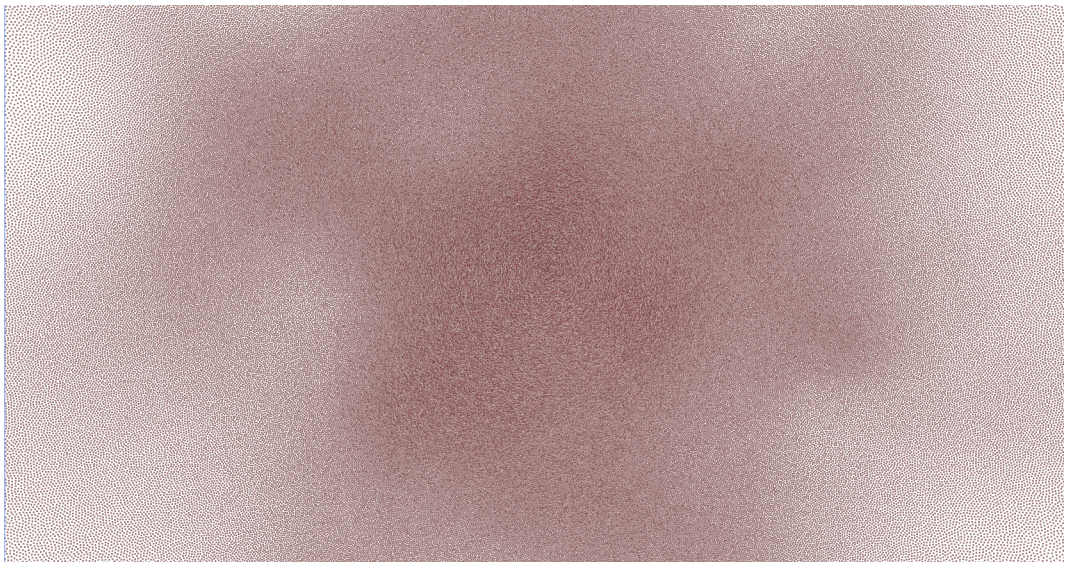


Figure 42: The result of the full scaled force-directed algorithm simulation with 515606 nodes. The point of view is from inside the spheric surface.

Figure 41 and Figure 42 show the result of the full scaled simulation where each dot is a node. Denser patterns between the nodes could be seen where heavy node clusters had a harder time to spread. Even with a little uneven spread I'm quite happy with the result, I don't believe the denser node clusters will impact the aesthetical side too much but rather add to the concept of having downtowns in the world.

10 User testing

I chose the user testing to be exploratory since there aren't many objectives to give the user, the majority of the work done is back end and there are only a few things the end user can test and interact with. So the purpose of this user testing was the following:

- To know if the controls are intuitive.
- Test if they can find out the building interactivity (right clicking a building) and what to do with it.
- General understanding.
- Aesthetical experience and first impressions.

10.1 How the testing was done

I allowed five participants to try my project given 5-10 minutes each for free exploration and attempt to find as much functionality in the world as possible. All of the participants had played games earlier in their life and had at least some experience with first person cameras, which was the demographic I was searching for. I chose that demography because I had not implemented a guide to configure or reveal the controls and wanted more emphasis on the general understanding and impression of the world, it would be hard for users who had no experience with a first person camera to navigate. I observed their free exploration but did not assist them or answer questions. After they were satisfied with the testing I had a questionnaire prepared for them in which they filled out a form that best described their feeling and reaction towards the program. This was to be able to get some feedback from what was in their mind during the testing. The questionnaire can be seen in Figure 43.

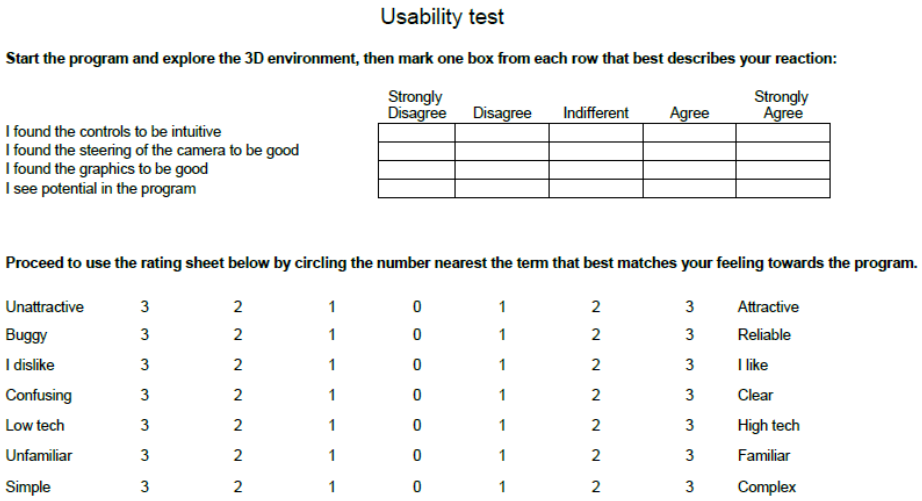
10.2 Observations

The observatory feedback was incredibly useful, much of the functionality that I thought was obvious was not obvious to the participants. The participants also discovered a bug. The preview of a website element appeared only the second time you clicked the preview option.

I also noticed a few inconveniences when the participants were playing. The ones that I took note of were:

- The mouse steering was tough to learn, it took several minutes for some of the participants to use it decently. The sensitivity was too high.

- The participants had no indication of when they were able to right click something. So the right click functionality on buildings prototype was not completely obvious. More feedback should have been given, such as highlighting a building when the mouse hover above it. That could partially have solved and give hints to the existence of this prototype. Otherwise it was more out of luck that the participants discovered it.
- Most of the participants assumed the favicon hovering above the buildings were clickable. But I had only implemented right click on the building structures, not the favicons.
- The website element often took a long time to load its HTML content and sometimes got errors.
- It was hard for the users to navigate back to a place they had been earlier, since all buildings looked the same apart from the height so the world became more of a maze than a map. More variance, colors and shapes of the structures are definitely is needed to create a more true geographic identity.



Usability test

Start the program and explore the 3D environment, then mark one box from each row that best describes your reaction:

	Strongly Disagree	Disagree	Indifferent	Agree	Strongly Agree
I found the controls to be intuitive					
I found the steering of the camera to be good					
I found the graphics to be good					
I see potential in the program					

Proceed to use the rating sheet below by circling the number nearest the term that best matches your feeling towards the program.

Unattractive	3	2	1	0	1	2	3	Attractive
Buggy	3	2	1	0	1	2	3	Reliable
I dislike	3	2	1	0	1	2	3	I like
Confusing	3	2	1	0	1	2	3	Clear
Low tech	3	2	1	0	1	2	3	High tech
Unfamiliar	3	2	1	0	1	2	3	Familiar
Simple	3	2	1	0	1	2	3	Complex

Figure 43: The questionnaire form that was filled in by the participants.

10.3 Test results

The results of the usability test questionnaire can be seen in Figure 44.

The majority seemed to found the controls to be intuitive to learn but not that great or they were indifferent about it. The graphics impression was below average, which I think mostly was because the building placeholders that we

used. Having so many of them gave little uniqueness to the buildings and the city as a whole, which made the world feel monotone. But nevertheless, some potential was seen in the program.

Overall the rating in the second form seemed to land relatively close to 0 in all categories. Only "I like" and "Attractive" were relatively high compared to the rest, which is a positive feat. But as said, overall the rating was close to 0, which tells us that much of the program needs improvement.

	Strongly Disagree	Disagree	Indifferent	Agree	Strongly Agree
I found the controls to be intuitive				4	1
I found the steering of the camera to be good		2	2	1	
I found the graphics to be good		3		2	
I see potential in the program			1	2	2

	Average in interval [-3, 3]		Standard deviation
Unattractive	1	Attractive	1,48
Buggy	0,2	Reliable	1,46
I dislike	1,2	I like	0,75
Confusing	-0,2	Clear	1,22
Low tech	0,8	High tech	0,87
Unfamiliar	-0,6	Familiar	2
Simple	-0,2	Complex	1,37

Figure 44: The results of the usability test. In the first form the results are summed up in respective cell. In the second form I set the average values to range between -3 and 3.

High/low tech and Familiar/Unfamiliar I mostly put there for interest to see if the participants thought it was highly technical and if they had seen something similar before. I expected more of High tech and more of Unfamiliar than what it turned out to be, but the high standard deviation can indicate on confusion to the question or that it simply is more of a yes/no question rather than a rating.

11 Discussion and conclusions

11.1 Optimizations

A stuttering effect comes by popping the entire relocation queue at the same time and moving all of the building geometry. It could be made more elegant by relocating fewer LOD geometries at a time and allow a few render cycles in between the relocation. As the camera moves over the surface the B3 buildings could be brought one at a time from the back to the front, and the tile position switching between B1 to B2 and B2 to B3 could also be done in a more continuous manner so that we don't get spikes of a big amount of geometry that needs to be relocated at the same time.

When it comes to the fetching of data from the web server to the client; we do an AJAX call and download data for each tile separately. If the camera moves very fast in the world the result is that very many AJAX calls will be made and cause much overhead in the communication. Instead we should fetch data in larger sizes. Meaning that we group the tiles and give each group of tiles an ID, and then have metadata stored about which group of tiles the client needs to download at a given location.

In Section 5.1 we discussed the model of the world object in which we decided it to be a geodesic gridded icosahedron. Independent of what type of tesselation and how high polygon count for the sphere we used; we would still get crevices between the polygons. And because the buildings are placed orthogonal to the polygon they are standing on they will cause the crevice of the polygon to be more clearly visible simply by looking at the normal of the buildings. This problem could become less apparent if the surface of the icosahedron was adaptively tesselated. In that way the local polygons would be subdivided and as such give a smoother surface of the sphere.

The force-directed algorithm was not meant to be run through the web browser but since I had an octree implementation in JS it became the most convenient solution, and it also gave me a quick way to verify the result of the small-scaled simulations so that I could effectively adjust the algorithm through the experimental process. Many times the web browser crashed because the algorithm required too much performance and the octree's depth had to be reduced substantially in order to cut down on memory resources.

The force-directed algorithm could be optimized by making the force calculations multi-threaded, thus reducing the time it takes to run the simulation. The number of iterations could also be heavily reduced by utilizing a similar solution to that of Kamada and Kawai's algorithm which quickly generates a decent initial starting position for the nodes in order to early on introduce a geographical property.

11.2 User testing

The explorative user testing certainly provided another perspective, more insight and knowledge of which implementations that needs a new revision. Sadly I did not have time to follow-up with a second prototype but the results certainly pinpointed many of the user interactivity issues.

The building interactivity needs further prototyping. World objects that are clickable need a way of showing that.

The controls were found cumbersome to use. Mainly the mouse control. But it is the least cumbersome first person mouse control that I've found to use in web browsers, so it might be that the testers were very used to a standard 3D application first person mouse control which fixates the mouse in the middle. However, the mouse sensitivity was definitely too high which definitely could have been complemented with an interface in which the user could increase/decrease it.

The graphics and building uniqueness has to be improved. Further increasing the polygon level on the objects will require more resources from the web browser which most likely would be best compensated by decreasing the number of rendered buildings. And reducing the number of seen buildings will have a user impact as well. So it is a trade off. However, the building textures could be improved and made much more unique without much of a performance hit.

11.3 Back end conclusions

The dependency file that the crawler outputted revealed a flaw, which can be seen in the snippet in Section 7.1.2. The flaw is that the crawling data is very location biased. The crawler running on my home computer found a high frequency of Swedish domains such as sverigesradio.se or svenskafans.com when crawling google.com. The reason is because they are geographically specific advertisement. A geographically distributed web crawler would reveal betters results in this case in order to reduce the bias towards Swedish websites.

The implementation of our force-directed algorithm did not output perfectly equally distanced points as can be seen in Figure 41. They are scattered so that they don't collide which is good but they are not equally distanced. It have been both hard and frustrating to get a resembling result in the full-scaled simulation to that of the small-scaled. The spring model repulsion formula is the main factor in distancing the nodes in the small-scaled environment and I believe with further polishing and testing of the spring model formulas we could expect better results in the full-scale simulation as well.

The gravity balls handled the large-scale spread of the nodes really well, decreasing the number of iterations by a large amount. However, little did they impact the small-scale spread of the nodes, as was predicted. Dense node clusters takes many iterations to spread and making the repulsion formula more aggressive tended to create instability.

The building height uses a logarithmic formula that factors in the PageRank and the density value. Since we had crawled a rather small part of the visualized set of websites compared to how many dangling websites we had, it meant the PageRank values would be heavily skewed towards the few websites that we had crawled. So the density value was used to give the nodes that had gotten very deep into the node clusters some "credit" for being able to get into the denser parts of the world, which ultimately is quite an achievement when there are so many other nodes that tries to push themself into the center of the clusters.

11.4 Overall conclusions

The decision to render the application within the web browser context have certainly limited the capabilities in having a fully customizable interface. The application as a whole feels less integrated than it should, because it will still just be a website tab in the web browsers. Much of the concept description in Chapter 2.2 aims towards a standalone and completely revamped web browser. The ideas I have to change the interface and functionality of the web browser would be very different from what you would be able to do with a Firefox addon or Chrome extension. However, due to time limitations it was not reasonable to create a new web browser. But for the future of the project I do have plans to migrate towards a similar solution.

WebGL is an amazing tool for the right purpose, it has a high availability due to being adopted by the biggest web browsers in the world. But when it comes to larger 3D applications that requires a lot of resources such as textures and models - one will start to see the limitations compared to having the resources on the client side from the beginning. I spent much of my time implementing a structure to partition the data and only load chunks of it when necessary, but to do that properly one has to implement a lot of fallbacks for when the client is requesting too much at the same time.

The force-directed algorithm was a fantastic tool to position the buildings and the forces can easily be changed to reflect another geographical property other than the PageRank values between websites. It took a long time to implement correctly and with a proper scalability towards larger simulations.

The Same-Origin policy heavily restricted the concept of having dynamically loaded HTML documents in the world, and using a proxy on the webserver is not an optimal solution. It also means that the client has to put a lot of trust in that the webserver does not send back malicious content to the user, as well as protection for the webserver to not download malicious content to harm itself.

The world became so large with so many buildings that it was hard for the participants to navigate or get a perspective of where the most popular buildings were. So for a future prototype some kind of map to help with the navigation is definitely needed, and much more so if the world gets larger than it currently is.

The geographic property of the buildings were hard to see for a user, there is more to geography than just positioning. The terrain needs variety, colors and different structures for users to be able to relate one place from another, but due to time limits and the amount of backend work that had to be done the tuning of the user interactivity was given less focus.

11.5 Future development

I still have high belief in the idea of this project, a decentralized version of it. But in that future the 3D application would be better off to embed a web browser in it rather than the other way around. The 3D application should be in charge of its resources. Opening HTML content inside the 3D world shouldn't be more of a hassle than opening a new tab in a web browser.

For a future prototype I will most likely focus on trying to do an overview of the web first and skip the game part, because that alone is such a huge project. The overview will include a refined version of the geographical methods used in this thesis. The local navigation in the world at the moment needs a lot of adjusting to be purposeful to the end user, the local interactivity is mainly where I want the game culture to thrive. The overview will benefit the everyday user to see and find - with just a quick glimpse into the sky - which websites are the most trafficked by the users and which documents on those websites are the most popular. The purpose mainly being to show the unfiltered "state" of the web, and with options to filter it to reveal content that may be more interesting to the user. To see where activity starts and ends. I think this is a fundamental step in order to open up Internet and the information flow across languages and countries. Of course there are a lot of challenges to this. Only the graphical technicalities alone have just briefly been touched in this project, trying to render a large amount of anything at the same time needs a solid solution.

There are a huge amount of ideas that I wish I had time to implement. I'll never give up on the vision, but in the meantime I may focus on other projects simply because this project may be too big for my hands alone and the goal is far-fetched. Web development is a tough environment because there are a lot of compatibility and security issues which makes other development areas more attractive for me personally where you can focus on the actual product itself. But it is an idea I've been thinking for over 10 years so I'll never really feel satisfied until I see it in one form or another with a similar vision.

References

[1] Wikipedia: Distributed web crawling. Accessed 2015-09-22.

[2] Wikipedia: Crowd manipulation. Accessed 2015-09-22.

[3] Wikipedia: World wide web. Accessed 2015-09-22.

[4] Wikipedia: World wide web. Accessed 2015-09-22.

[5] T. Parisi. *WebGL up and running book*. OReilly Media, Inc, first edition, 2012. Page 3.

[6] T. Parisi. *WebGL up and running book*. OReilly Media, Inc, first edition, 2012. Page 8.

[7] Lighthouse3d. "`http://www.lighthouse3d.com/tutorials/glsl-core-tutorial/vertex-shader/`". Accessed 2015-09-22.

[8] Opengl. "`https://www.opengl.org/sdk/docs/tutorials/ClockworkCoders/varying.php/`". Accessed 2015-09-22.

[9] Ntu. "`http://www.ntu.edu.sg/home/ehchua/programming/opengl/cg_basicstheory.html`". Accessed 2015-09-22.

[10] T. Parisi. *WebGL up and running book*. OReilly Media, Inc, first edition, 2012. Page 10.

[11] Mozilla. "`https://developer.mozilla.org/en/docs/Web/API/window.requestAnimationFram`". Accessed 2015-09-22.

[12] Cprogramming. "`http://www.cprogramming.com/tutorial/3d/quaternions.html`". Accessed 2015-09-22.

[13] Wikipedia: Spline mathematics. Accessed 2015-09-22.

[14] Wikipedia: Three.js. Accessed 2015-09-22.

[15] Wikipedia: Octree picture. "`http://en.wikipedia.org/wiki/Octree#mediaviewer/File:Octree2.svg`". Accessed 2015-09-22.

[16] Collinhover. "`https://github.com/collinhover/threeoctreel`". Accessed 2015-09-22.

[17] Wikipedia: Octree. Accessed 2015-09-22.

[18] S. Brin and L. Page. *The Anatomy of a Large-Scale Hypertextual Web Search Engine*. Stanford University, first edition, 1998. "`http://infolab.stanford.edu/pub/papers/google.pdf`". Chapter 2.1.1. Accessed 2015-09-22.

[19] S. Kobourov. *Spring Embedders and Force-Directed Graph Drawing Algorithms*. University of Arizona, 2012. "http://arxiv.org/pdf/1201.3011v1.pdf". Chapter 2.1.1. Accessed 2015-09-22.

[20] Wikipedia: Hooke's law. Accessed 2015-09-22.

[21] Wikipedia: Inverse-square law. Accessed 2015-09-22.

[22] Wikipedia: Coulomb's law. Accessed 2015-09-22.

[23] P. Eades. A Heuristic for Graph Drawing. Congressus Numerantium, 1984. Volume 42. Page 149-160.

[24] T. Kamada and S. Kawais. *An Algorithm for Drawing General Undirected Graphs*, volume 31. University of Tokyo, 1989. Page 7-15.

[25] T. Fruchterman and E. Reingold. *Graph Drawing by Force-Directed Placement*, volume 21. University of Illinois, 1991. "http://emr.cs.iit.edu/~reingold/force-directed.pdf". Page 1129-11641. Accessed 2015-09-22.

[26] Wikipedia: Barnes hut simulation. Accessed 2015-09-22.

[27] Unity3d. "https://unity3d.com". Accessed 2015-11-26.

[28] Unreal engine. "https://www.unrealengine.com/what-is-unreal-engine-4". Accessed 2015-11-26.

[29] Glge. "http://www.glge.org/". Accessed 2015-11-26.

[30] Philogl. "http://www.senchalabs.org/philogl/". Accessed 2015-11-26.

[31] Scenejs. "http://scenejs.org/". Accessed 2015-11-26.

[32] Spidergl. "http://spidergl.org/". Accessed 2015-11-26.

[33] Three.js custom shaders. "http://threejs.org/docs/api/materials/ShaderMaterial.html". Accessed 2015-11-26.

[34] Wikipedia: Geodesic grid. Accessed 2015-09-22.

[35] Yahoo. "https://developer.yahoo.com/performance/rules.html". Accessed 2015-09-22.

[36] Thatsmaths. "http://thatsmaths.files.wordpress.com/2012/10/lat-long-grid.jpg". Accessed 2015-09-22.

[37] Learningthreejs. "http://learningthreejs.com/blog/2013/04/30/closing-the-gap-between-html-and-webgl/". Accessed 2015-09-22.

[38] Eclipse. "https://www.eclipse.org". Accessed 2015-09-22.

[39] Tortoisesvn. "http://tortoisesvn.net/". Accessed 2015-09-22.

[40] Dropbox. "https://www.dropbox.com". Accessed 2015-09-22.

[41] Sublimetext. "https://www.sublimetext.com". Accessed 2015-09-22.

[42] Apachefriends. "https://www.apachefriends.org". Accessed 2015-09-22.

[43] Chromium. "http://blog.chromium.org/2014/06/try-out-new-64-bit-windows-canary-and.html". Accessed 2015-09-22.