SAVONIA

TEKNIIKAN JA LIIKENTEEN ALA

Thesis – Keinänen Niilo

INTEGRATION OF 3D CHARTS IN LIGHTNINGCHART JS

KÄSITELTY: Paikka, aika

HYVÄKSYTTY: Paikka, aika

Koulutusala Tekniikan ja liikenteen ala	
Koulutusohjelma/Tutkinto-ohjelma Sähkötekniikan tutkinto-ohjelma	
Työn tekijä(t) Niilo Keinänen	
Työn nimi 3D-Kuvaajien Integrointi LightningChart JS Kirjastoon	
Päiväys 20.11.2019	Sivumäärä/Liitteet 43
Ohjaaja(t) Keijo Kuosmanen, lehtori Arto Toppinen, yliopettaja	
Toimeksiantaja/Yhteistyökumppani(t) Arction Ltd. Pasi Tuomainen, Toimitusjohtaja	

Tiivistelmä

Opinnäytetyön tavoitteena oli suunnitella, kehittää ja integroida uusi 3D-kuvaajaominaisuus LightningChart JS-datavisualisointikirjastoon. Tarkoituksena oli laajentaa tuotteen asiakasryhmää, sekä luoda pohjaa muille 3D-ominaisuuksille.

Opinnäytetyö perustui internetistä löydettyihin WebGL oppaisiin sekä viralliseen dokumentaation. LightningChart JS koodikirjasto kirjoitettiin TypeScript-kielellä. Myös GLSL-kieltä hyödynnettiin shaderien toteuttamiseen.

Opinnäytetyön tuloksena saatiin toimiva ominaisuus, joka tulee johtamaan uuteen laajaan LightningChart JS päivitykseen. Tulos toimii myös tärkeänä valmistelu-askeleena muille tuleville 3D-ominaisuuksille. Kehityksen aikana löytyi myös optimointitekniikoita, joita voidaan tulla hyödyntämään aiempien ominaisuuksien parantamiseen.

Avainsanat

Ohjelmistokehitys, WebGL, LightningChart JS, TypeScript, laitteistokiihdytetty piirtäminen

SAVONIA UNIVERSITY OF APPLIED SCIENCES

THESIS
Abstract

Field of Study Technology, Communication and Transport			
Degree Programme Degree Programme in Electrical Engineering			
Author(s) Niilo Keinänen			
Title of Thesis Integration of 3D Charts in LightningChart JS			
Date	20 November 2019	Pages/Appendices	43
Supervisor(s) Mr. Keijo Kuosmanen, Senior Lecturer Mr. Arto Toppinen, Principal Lecturer			
Client Organisation /Partners Arction Ltd. Mr. Pasi Tuomainen, CEO			

Abstract

The goal of the thesis was to integrate new 3D rendering features to LightningChart JS, a WebGL based visualization library written in TypeScript. The fundamental requirements were a 3D scene that can be shared for future 3D features as well as an end user feature for rendering large datasets as triangulated 3D cubes.

The thesis was widely based on knowledge learned from online tutorials and official documentation of WebGL. LightningChart JS codebase was written in TypeScript. Additionally, GLSL was utilized for implementing shaders.

The result of the thesis was a functioning feature implementation which will be the basis of a new major feature release of LightningChart JS. The result will also be an important foundation for future core features. During the development, new optimization techniques which can be utilized for improving the previous features, were found.

Keywords

Software development, WebGL, LightningChart JS, TypeScript, hardware-accelerated-rendering

TABLE OF CONTENTS

1 INTRODUCTION

The goal of the thesis project was to design, implement and integrate a new software feature into an existing product. The product being LightningChart JS, a charting library written in TypeScript, that utilizes WebGL for hardware-accelerated rendering. TypeScript is a superset of JavaScript, which should be more familiar to most people, as it is the default scripting language inside web pages. WebGL is an open source JavaScript API that allows executing programs in Graphics Processing Units of computers, to allow the creation of demanding rendering applications. The feature to be implemented was 3D-Chart, and 3D-PointSeries. In combination these can be used to visualize large sets of 3D locations for analysis or similar purposes, inside any web page / web application that supports WebGL.

2 LIGHTNINGCHART JS AND 3D CHARTS

LightningChart JS is a high-performance data visualization library for web applications. Currently, LightningChart JS features a handful of XY series types for displaying up to tens of millions of data points, by using WebGL for hardware-accelerated rendering, as well as some more traditional chart types, such as Pie, Donut and Gauge-charts. Figures 1 and 2 present a line series and donut chart rendered with LightningChart JS.

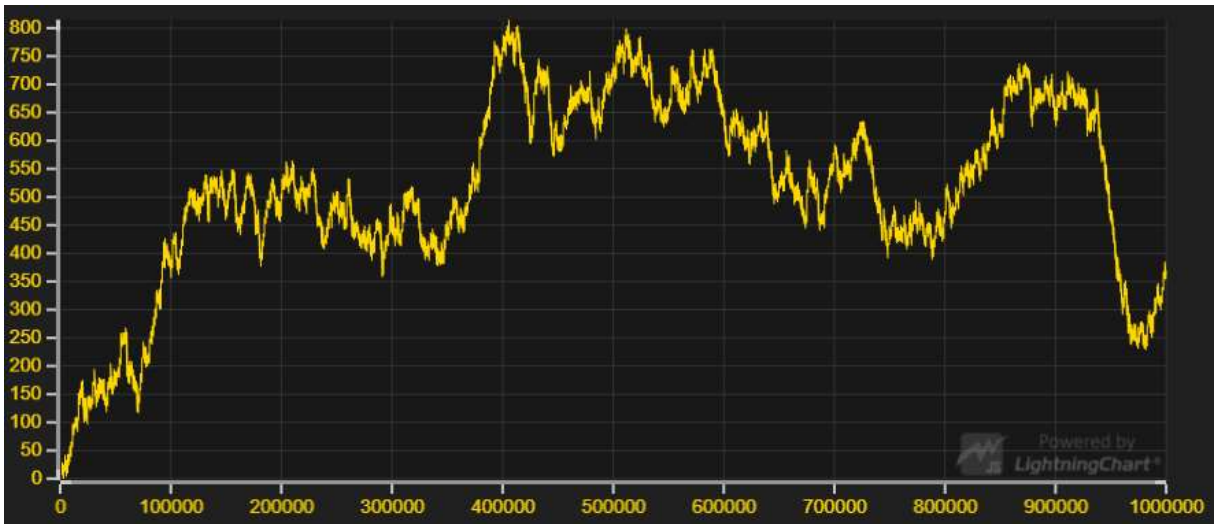

FIGURE 1. ChartXY and LineSeries rendered with LightningChart JS Community edition. (Arction Ltd. 2019)

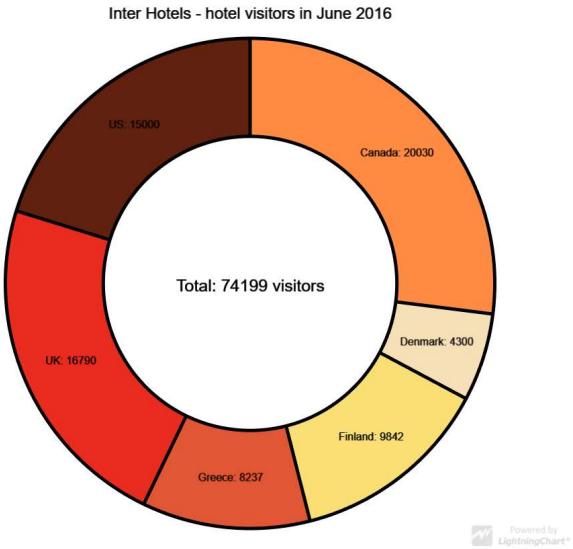

FIGURE 2. Donut Chart rendered with LightningChart JS. (Arction Ltd. 2019)

2.1 Usage cases for 3D charts

Generally, the need for a 3D chart comes from one of two possibilities. First is the need to add a third factor to each data point. For example, a geographical map, where each location has a value to be visualized – in

this case, the geographical location of the point takes two dimensions (eq. X, Z), and the actual value will be the third one (Y). See Figure 3 for an example of this.

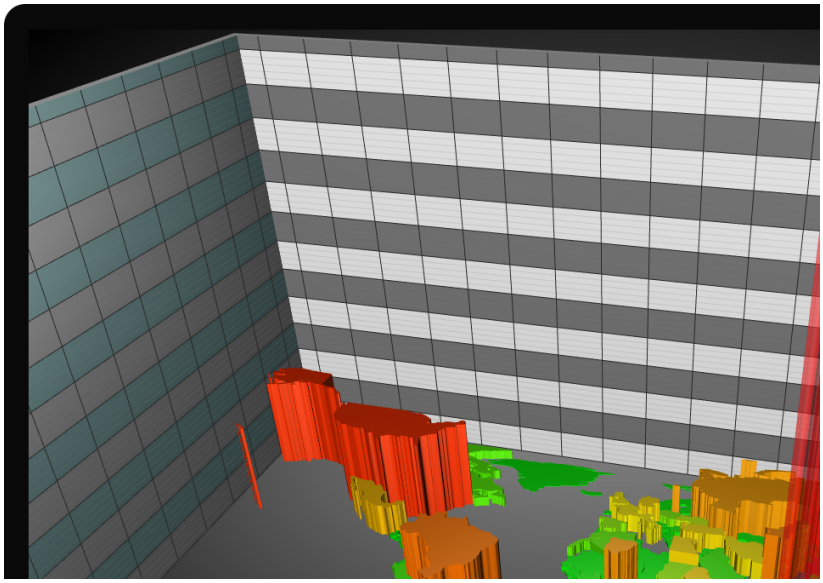

FIGURE 3. 3D world population. (Arction Ltd. 2019)

Otherwise, the motive should be simply visual preference. The 3D scene provides more ways to customize the view and gives a sense of "space". For example, in Figure 4, there is no value associated to the Z coordinate – it is purely visual.

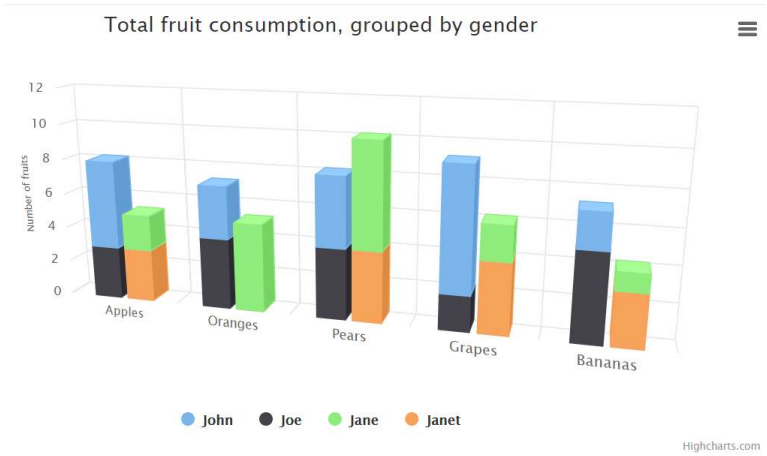

FIGURE 4. Highcharts 3D example (Highcharts 2019)

3 AGREEING ON REQUIREMENTS

The scope of the thesis project was decided together with the author, the lead developer of LightningChart JS and the product owner. This holds within the features that must be performable with the finished result. The author was given complete freedom over the actual design and implementation. In the finished product, the 3D-Chart …

- …clearly displays the X, Y and Z Axes, with coloured lines in the 3D space. The boundaries around the Axes are further defined with a bounding box drawn with thin lines.
- …can have Point Series added to it. Point Series can be used to render any supplied amount of datapoints (combination of X, Y, and Z values) as 3D cubes. The size and colour of cubes can be set by the user (per Series, not individually). Point Series scale according to the Charts Axes that can be configured by the user.
- …is viewed as a *perspective projection*, whose location and orientation in the 3D-space can be altered with a mouse, or touch-monitor.
- …has a static light source that illuminates the scene.
- …can be created as a standalone component, or inside a Dashboard component. The Dashboard is a LightningChart JS component that allows for multiple Charts to exist inside an adjustable grid container.

4 THEORY OF 3D RENDERING IN WEBGL

This chapter introduces and explains of all the WebGL techniques and features that are used in the scope of the thesis work. The ideas and techniques presented in this chapter are based on the *WebGL fundamentals* documentation (WebGL Fundamentals 2019).

4.1 Basic rendering

The chapters are in order of appearance from the implementation point of view, where the goal is to render a single cube.

4.1.1 WebGL context

To start with WebGL rendering, it is necessary to get a reference to a WebGL context. This is an interface that exposes all the capabilities of WebGL. It can be taken in JavaScript from a *canvas* element (code snippet 1).

```
const gl = canvas.getContext( 'webgl' )
```

Code snippet 1. Getting reference to a WebGL context

4.1.2 Vertex buffer

Whether it is 2D or 3D, modern hardware accelerated rendering is done by using Vertex and Index-buffers. They allow for optimized rendering of complex objects, by minimizing duplication of drawing properties and required communication between application code and the computer hardware. Let's look at an example of rendering a single filled square (Figure 5):

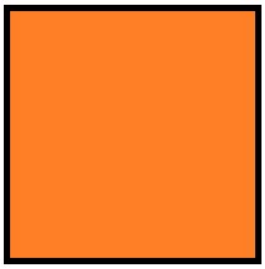

FIGURE 5. Filled square

The square is built of four points, each located at a corner (Figure 6). The Vertex buffer is a list of all points that are used to construct a shape.

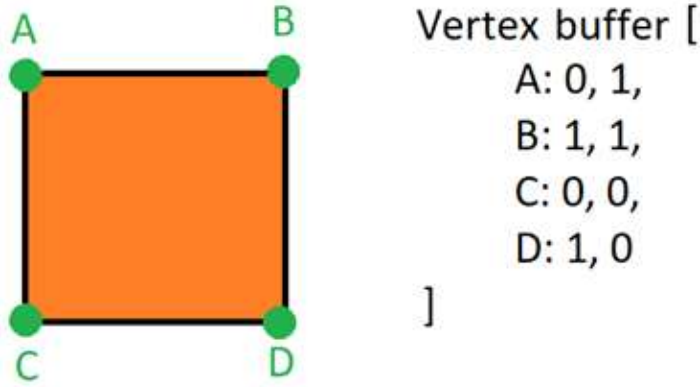

FIGURE 6. Vertex buffer for a filled square

The Vertex buffer lists all *attributes* of points in order. In the example above, these attributes are X and Y coordinates. Code snippet 2 shows how to create a Vertex buffer and load data into it.

```
const vertexData = [ 0, 1, 1, 1, 0, 0, 1, 0 ]
const vertexBuffer = gl.createBuffer()
gl.bindBuffer( gl.ARRAY_BUFFER, vertexBuffer )
gl.bufferData( gl.ARRAY_BUFFER, new Float32Array( vertexData ), gl.STATIC_DRAW )
gl.bindBuffer( gl.ARRAY_BUFFER, null )
```

Code snippet 2. Creation of a Vertex buffer

4.1.3 Index buffer

The Index buffer is a list of rendering instructions for the computers Graphics Processing Unit (GPU). The GPU is optimized to render *triangles*, as such it is the responsibility of the rendering application, to prepare all shapes as a set of *triangles*. This is exactly, what the Index buffer is – a list of *triangles* to be rendered (Figure 7).

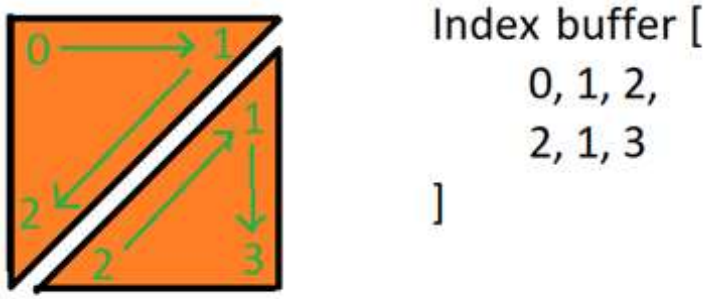

FIGURE 7. Index buffer for a filled square

The numbers inside Index buffers refer to the *index* of an item inside the Vertex buffer, so in this case 0 would be equal to "A". This might seem overly complex to just render a single square – and it is – but this

approach is optimal when the rendered shapes are large and detailed. Shapes can consist of as many as millions of triangles, connected to each other to form a polygon-like shape, that can be used to build pretty much any realistic shape. This kind of shapes have a lot of vertices that are used multiple times through the Index buffer, which results in much better performance than if for every triangle there would be three vertices supplied. Code snippet 3 shows how to create an Index buffer and load data to it. Note how the only differences from Vertex buffer creation are the changed WebGL target enumeration (ELEMENT_ARRAY_BUFFER), and the used *typed Array* constructor (Uint16Array).

```
const indices = [ 0, 1, 2, 2, 1, 3 ]
const indexBuffer = gl.createBuffer()
gl.bindBuffer( gl.ELEMENT_ARRAY_BUFFER, indexBuffer )
gl.bufferData( gl.ELEMENT_ARRAY_BUFFER, new Uint16Array( indices ), gl.STATIC_DRAW )
gl.bindBuffer( gl.ELEMENT_ARRAY_BUFFER, null )
```

Code snippet 3. Creation of an Index buffer

4.1.4 Vertex/Fragment shaders

Now that the data is prepared to be rendered, what is left is to tell the GPU how to render it. In Figure 6, there are some made-up XY coordinates in the range [0, 1], as well as a bright orange colour. To process these as desired, *shaders* are required*. Shader* is a computer program that runs on the GPU. They are written in a special language that is compiled down to GPU instructions. In the case of WebGL, this language is called GLSL (Open**GL S**hading **L**anguage). In WebGL version 1, as used in this project, there are two types of *shaders*: the *vertex shader* and the *fragment shader.* Figure 8 describes both in brief.

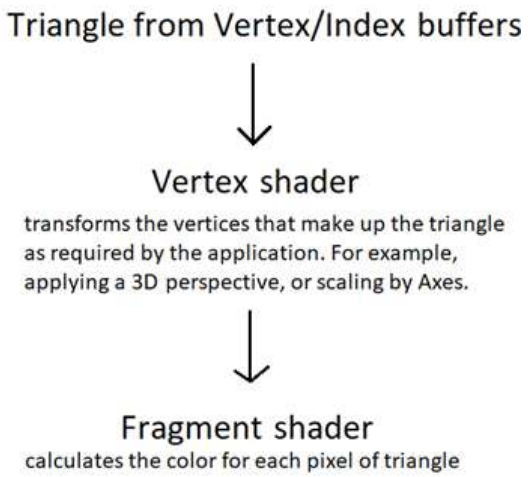

FIGURE 8. Vertex / Index buffers

One of the special features of WebGL is that shaders can be compiled during run-time, simply by passing a string containing the shader source code to a compiler function (Code snippet 4).

```
const shaderSource = `
attribute vec2 aPos;
void main(void) {
    gl_Position = vec4( aPos.xy, 0.0, 1.0 );
}
`


const shader = gl.createShader( gl.VERTEX_SHADER )!
gl.shaderSource( shader, shaderSource )
gl.compileShader( shader )
```

Code snippet 4. Compiling GLSL shaders

4.1.4.1 Attributes and uniforms

Shaders have two ways of receiving input values from applications: *attributes,* and *uniforms*. As mentioned in section "Vertex/Index buffers", the items of the Vertex buffer are called *attributes*. The defining feature of an *attribute* is that it is defined "per vertex". Each vertex has the same number of *attributes*, and they are all available in the *shaders*. Contrary to this, a *uniform* is an input variable accessible in *shaders*, that is defined "per shape". Table 1 demonstrates some usage cases for both.

TABLE 1

Attribute	Uniform
Vertex location (X, Y, Z)	3D perspective
Vertex normals	Shape transformation (location, rotation, scale)
Per-vertex colour	Per shape colour
Texture coordinate	Light sources

4.1.4.1.1 Loading uniform values to shaders.

Uniform values are loaded by using a combination of *shader* reference and the name of the uniform property in shader source code (Code snippet 5). Each possible uniform data type (in this case, four-dimensional float vector) has its own method.

```
gl.uniform4f( gl.getUniformLocation( shader, 'uColor' ), 1, 0, 0, 1 )
```

Code snippet 5. Loading uniform value to shader

4.1.4.1.2 Loading attribute values to shaders.

Attributes are read from the bound Vertex buffer, using *Vertex attribute pointers*. This is what links the items in the Vertex buffer to the *attribute* variables in *shaders* (Figure 9).

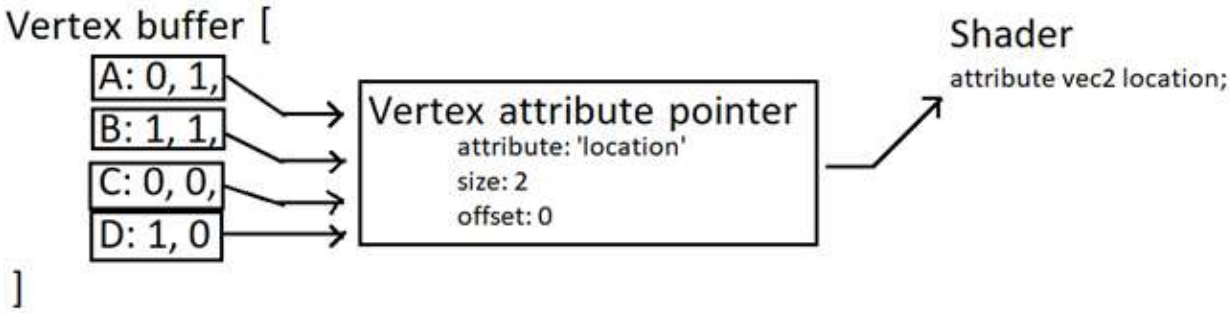

FIGURE 9. Vertex attribute pointer

Code snippet 6 shows how this looks in code.

```
gl.bindBuffer( gl.ARRAY_BUFFER, vertexBuffer )
const attribLocation = gl.getAttribLocation( shader, 'location' )
gl.vertexAttribPointer(
    attribLocation,
    // Total attribute values amount.
    2,
    gl.FLOAT,
    false,
    // Amount of values for this attribute (vec2 = 2).
    2,
    // Offset from previous attributes.
    0
)
gl.enableVertexAttribArray( attribLocation )
```

Code snippet 6. Defining vertex attribute pointers

4.1.5 Drawing elements

After loading the buffers, attribute pointers and uniforms to WebGL, all that is left for rendering is to call the correspondent draw method – for triangulated rendering with an Index buffer, the method is "drawElements" (Code snippet 7).

```
gl.drawElements( gl.TRIANGLES, vertexData.length, gl.UNSIGNED_SHORT, 0 )
```

Code snippet 7. Drawing prepared data

4.2 Advanced techniques

Covering theory behind additional techniques that are by now basically always expected in most 3D rendering applications.

4.2.1 Perspective projection

The computer monitor is a 2-dimensional display of pixels. As such, it cannot display anything with additional "depth" – which is traditionally regarded to as the Z component. 3D rendering simulates the sense of depth, by mapping the locations inside the virtual 3D space to the 2D screen according to some projection technique. Particularly, in this thesis a "perspective projection" will be used, which is designed to replicate the way the human eyesight looks. Most notably objects further away from the projection should be visually smaller, compared to ones close, as can be seen in Figure 10.

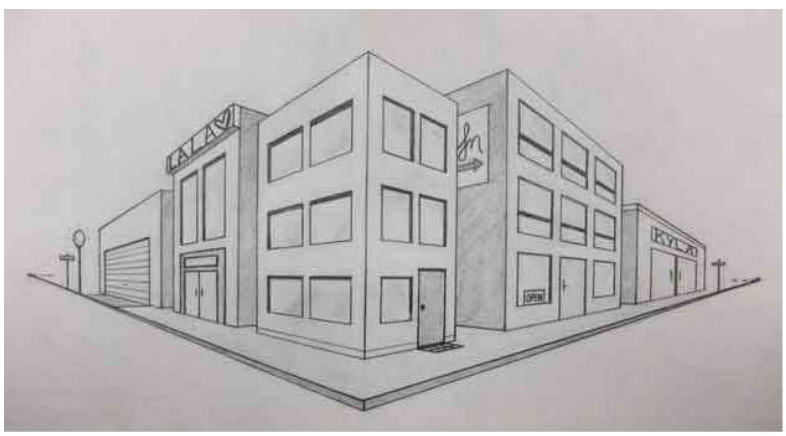

FIGURE 10. Perspective projection (Civilseek 2018)

In graphics, a projection can be applied by computing a *transformation matrix* for the projection and multiplying each *vertex* with this *matrix*. The properties that define a *perspective matrix* are its field of view, aspect ratio, near clipping plane, and far clipping plane. The formula of a *perspective matrix* is:

$$\begin{bmatrix} \frac{f}{aspectRatio} & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & \frac{near + far}{near - far} & -1 \\ 0 & 0 & \frac{near * far * 2}{near - far} & 0 \end{bmatrix}$$

…where

$$f = \frac{1}{\tan\left(fieldOfViewRad * 0.5 \right)}$$

$$aspectRatio = \frac{displayWidth}{displayHeight}$$

Formula 1. Perspective matrix. (WebGL Fundamentals, 2019)

The official WebGL tutorial has great documentation on the effects of each property of the *perspective projection* (Figure 11). The following chapters give a brief explanation of each property.

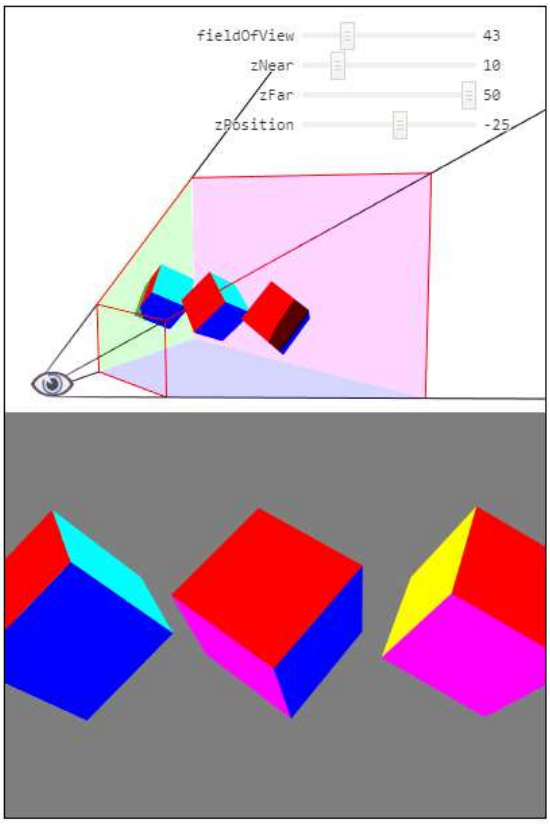

FIGURE 11. Properties of perspective projection.
(WebGL Fundamentals, 2019)

4.2.1.1 Field of view

Field of view controls the angle that is exposed to the "viewers eye". In normal cases 90 degrees is always used for this, as it feels most "human". Using values as high as 180 degrees would result in a "fisheye effect".

4.2.1.2 Aspect ratio

Aspect ratio is always equal to the ratio between display width and height.

4.2.1.3 Near and far clipping planes

The *near* and *far clipping planes* define the range of Z-values that can be rendered with the projection. Any Z-value outside the range [*zNear, zFar*] is clipped out and not rendered.

4.2.1.4 Applying the perspective matrix in shaders

Since the *perspective matrix* applies to each vertex, it is a uniform – as it is a 4x4 matrix, it can be represented in GLSL as **"uniform mat4"**. To transform each vertex according to the perspective is as simple as multiplying each vertex with the *perspective matrix* (Code snippet 8).

```
attribute vec3 aPos;
uniform mat4 uProjectionMatrix;

void main(void) {
    gl_Position = uProjectionMatrix * vec4(aPos.xyz, 1.0);
}
```

Code snippet 8. Applying perspective in vertex shader

4.2.2 Camera

After implementing the *Perspective Matrix* from the previous chapter, the next feature that comes to mind is moving the "viewpoint" around in the 3D world. This concept is known as the "camera" (imagine that the view is interpreted through a camera lens, that is moved and rotated around in the world. Similarly, as the *projection*, the camera is also represented as a *transformation matrix*. A popular fun-fact in 3D rendering is the fact that the camera is never actually moved, but instead the world around it is moved according to the "imaginary camera location". The *camera matrix* is a *matrix* that transforms a given point in relation to the desired camera location, and orientation.

The *camera matrix* is constructed from four 3D-*vectors*: the *right vector,* the *up vector*, the *forward vector*, and the *position vector*, according to the following formula:

$$\begin{bmatrix} right.x & right.y & right.z & 0 \\ up.x & up.y & up.z & 0 \\ forward.x & forward.y & forward.z & 0 \\ position.x & position.y & position.z & 1 \end{bmatrix}$$

Formula 2. Camera matrix. (WebGL Fundamentals, 2019)

In Figure 12, the orange *vector* is **right**, the white *vector* is **up**, and the blue *vector* is **forward**. Note, that they are interpreted looking away from the target ("F").

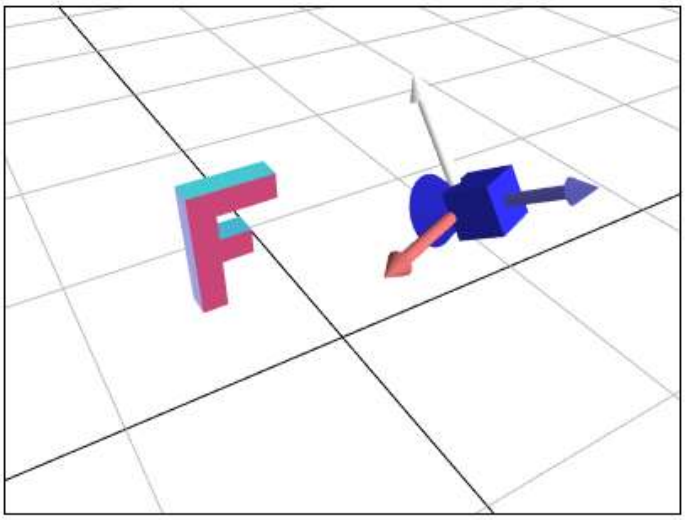

FIGURE 12. The 3D camera visualized. (WebGL Fundamentals, 2019)

The *camera matrix* is applied on top of the *perspective matrix*, by *matrix multiplication*:

$$viewProjectionMatrix = projectionMatrix * cameraMatrix$$

Equation 1. Combination of projection and camera matrices.

The "viewProjectionMatrix" is a matrix that transforms a 3D location according to both the *projection matrix* and the *camera matrix*. In the shaders, it replaces the *projection matrix*, while its use remains exactly the same.

4.2.3 Light source

In previous 3D pictures, the objects have been coloured very vividly. The reason for this is that the form of 3D-objects is very difficult to perceive if the object is coloured with a single colour. Example of this can be seen in Figure 13:

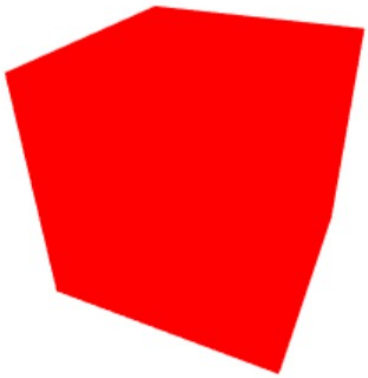

FIGURE 13. Single coloured cube

This problem is solved with *light shading*, a technique that mimics the behaviour of light in the real world – illuminating parts of objects based on the direction and intensity of the scenes light source. As one might guess from the name, *light shading* is handled in the *shaders*. There are, however, some additional steps that must be included in the preparation of data. For this case, let's consider a single, static light source, that distributes its light evenly in all directions. The information of the light source is loaded to the *shaders* as a *uniform* (as it is the same for the whole shape).

4.2.3.1 Vertex normals

The most crucial information for shading an individual vertex is the direction of the light source, **relative to the orientation of each *face***. In Figure 14, the *vector* **"N"** is the *normal vector* of the *face's* orientation. In rendering terms, this is simply referred to as "the *normal*". The *vector* **"L"** is the offset from the shaded vertex, to the location of the light source.

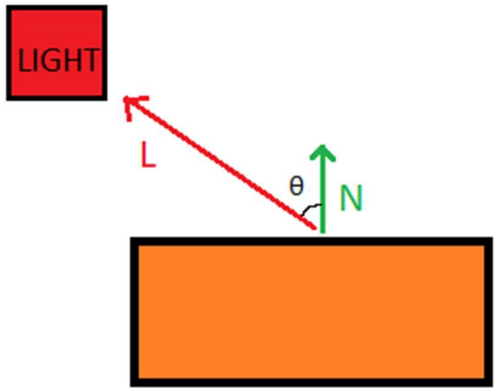

FIGURE 14. Surface and light normals

To save processing power from the CPU, this computation is traditionally moved to the shaders. For this, two variables need to be added (Code snippet 9): the normal of each vertex (attribute) and the location of the light source (uniform).

```
attribute vec3 aNormal;
uniform vec3 uLightLoc;
```

Code snippet 9. Vertex normals and light location variables

To compute the intensity of the light on a vertex, the angle between the *vectors* L and N*,* (**θ**, **theta**) is used, which can be calculated with *vector dot product* (Code snippet 10).

```
vec3 lightDir = normalize( uLightLoc - aPos );
float cosTheta = clamp( dot( aNormal, lightDir ), 0.0, 1.0 );
```

Code snippet 10. Angle between vertex normal and light source in vertex shader

To apply colour, the information must be passed on to the fragment shader with a *varying* variable (Code snippet 11).

```
...
uniform vec4 uColor;
varying vec4 vColor;

void main(void) {
    ...
    vColor = vec4( cosTheta * uColor.rgb, uColor.a );
}
```

Code snippet 11. Passing vertex colour on to fragment shader

The same *varying* variable must then be used in the fragment shader (Code snippet 12).

```
varying vec4 vColor;

void main(void) {
    gl_FragColor = vColor;
}
```

Code snippet 12. Applying per-vertex colour in fragment shader

In order to perform these computations in the shader, the vertex *normals* must be added to the list of *attributes* in the Vertex buffer. Now, for each 3D-vertex, there will be a total of 6 *attributes*: location (X, Y, Z), *normal* (X, Y, Z). This shading technique is called *Gouraud shading*. Its definitive feature is that the colour computation is done in the vertex shader.

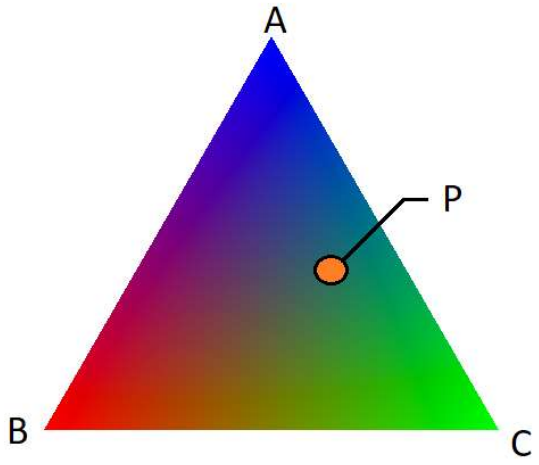

FIGURE 15. Gouraud-shaded triangle

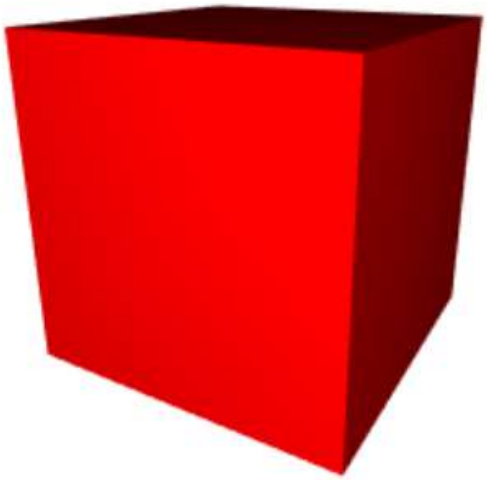

FIGURE 16. Gouraud-shaded cube, with light source at cameras location

Pay attention to Figure 16 and how the *faces* that are facing away from the camera (and the light source) are darker than the *face* in front – this is the effect of "cosTheta" multiplication.

4.2.3.2 Effect of distance.

Let's add a couple of cubes to the scene, each being further and further away from the light source (Figure 17).

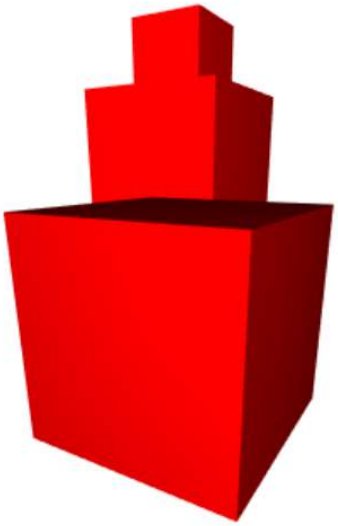

FIGURE 17. Cubes increasingly further away from light source

This doesn't look very realistic, because all the *faces* (that are facing the same way) are illuminated equally intensively. In real life, the intensity of light fades, as it travels longer distances. This can be replicated by adding the distance to the light source as a factor of colour shading (Code snippet 13).

```
void main(void) {
    ...
    float lightDistance = length( uLightLoc - aPos );
    vColor = vec4( uColor.rgb * cosTheta / ( lightDistance * lightDistance ), uColor.a );
}
```

Code snippet 13. Light source distance as a factor of colour shading

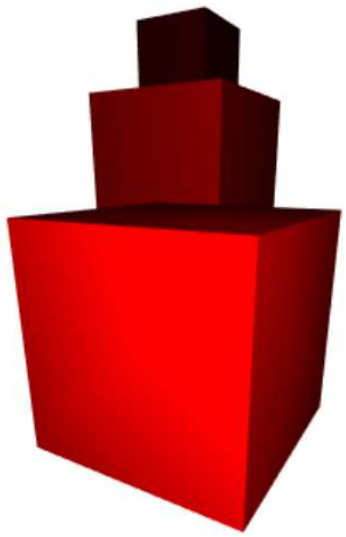

FIGURE 18. Distance shading

4.2.3.3 Ambient lighting

In Figure 18 the *faces* that are not directly facing the light or are the furthest away are dark – perhaps even too dark. There is a light shading technique that is designed to combat this problem: ambient lighting. Ambient lighting works by simply adding a minimal lighting intensity to all vertices, so that no pixel is ever fully dark (Code snippet 14).

```
float ambient = 0.30;
vColor = vec4( uColor.rgb * ( ambient + cosTheta / ( lightDistance * lightDistance ) ), uColor.a );
```

Code snippet 14. Ambient lighting in shaders

Figure 19 shows the same scene with an added ambient-component in the Vertex-shader. Note, how the previously dark *faces* are now also slightly illuminated.

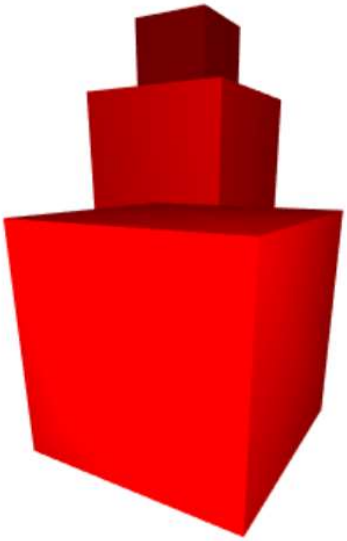

FIGURE 19. Ambient shading

4.2.3.4 Specular lighting

If you look at an object in the real world, if it's remotely shiny then the light will be reflected directly at you, almost like a mirror (Webglfundamentals.org).

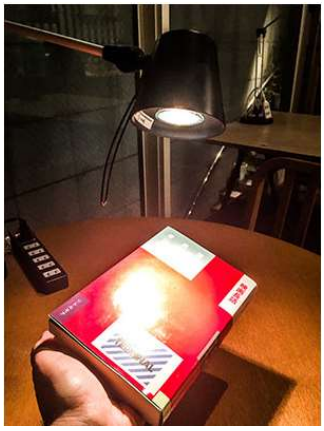

FIGURE 20. Specular light in the real world. (WebGL Fundamentals, 2019)

This can be simulated in *shaders*, by again checking the sharpness of some *vectors*. In this case, we're concerned about the angle between the *vector* of the reflected light (**"R")**, and the *vector* of the camera **("C")**. The *reflected vector* can be calculated with in-built functions of *GLSL* (Code snippet 15).

```
...
uniform vec3 uCameraLoc;
uniform vec3 uColorSpecular;

void main(void) {
    ...
    vec3 cameraDir = normalize(uCameraLoc - aPos);
    vec3 reflectDir = reflect(-lightDir, aNormal);
    float cosAlpha = clamp( dot( reflectDir, cameraDir ), 0.0, 1.0 );
    vec3 colorSpecular = uColorSpecular.rgb * cosAlpha / lightDistance2;
```

Code snippet 15. Specular colour component.

As the colour computation gets increasingly complex, it becomes necessary to group it up more logically in source code (Code snippet 16).

```
vColor = vec4(colorAmbient + colorDiffuse + colorSpecular, uColorDiffuse.a);
```

Code snippet 16. Final colour equation.

In Figure 21, a white specular colour is applied to a cube, resulting in a *shiny* effect.

FIGURE 21. Specular shading

4.2.3.5 Phong shading

Phong shading is a newer technique than Gouraud shading, which allows for more detailed light effects to be visible, by shading per pixel, instead of per vertex. In GLSL, this simply means passing the factors of light shading to fragment shader with *varying* variables and doing the same colour computation there. With a shape as simple as the cubes above, the difference between Phong shading and Gouraud shading is hardly visible. However, strong *specular light* will make it very clear (Figure 22).

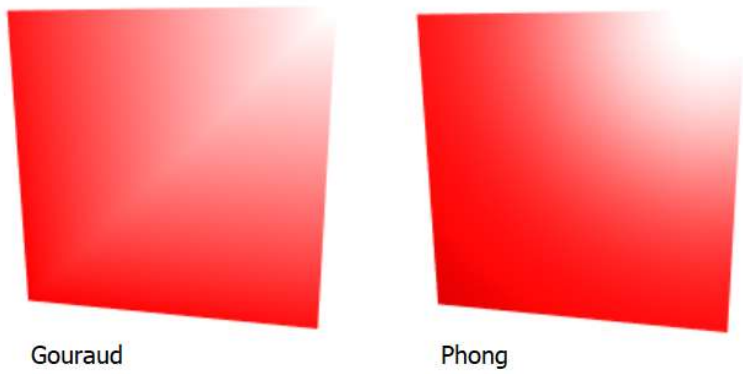

FIGURE 22. Gouraud vs Phong shading with strong specular light

4.2.4 Back face culling

Back face culling is a very widely used performance-boost technique in 3D rendering, which is based on skipping the rendering of *faces* that are facing away from the *camera* – effectively halving the amount of *faces* that need to be rendered. In *WebGL*, implementing back-face culling is as simple as setting two flags:

$$gl.enable(\ gl.CULL\ FACE);$$
$$gl.cullFace(\ gl.BACK\ )$$

Code snippet 17. WebGL enabling back face culling

This tells *WebGL* to cull the backs of *faces*. However, for *WebGL* to know, which *faces* are facing back, they must be constructed in a very specific manner. As explained previously in section: "Vertex and Index buffers", *faces* are defined as triangles formed by three 3D locations. What *WebGL* does, is it looks at the order of these triangles when rendering each *face*, and decides based on their order:

- If the triangles are in counterclockwise order, the *face* is rendered as if it was facing back.
- If the triangles are in clockwise order, the *face* is rendered as if it was facing front.

This order is interpreted as if the triangle was laid in front of the user, in a 2D plane.

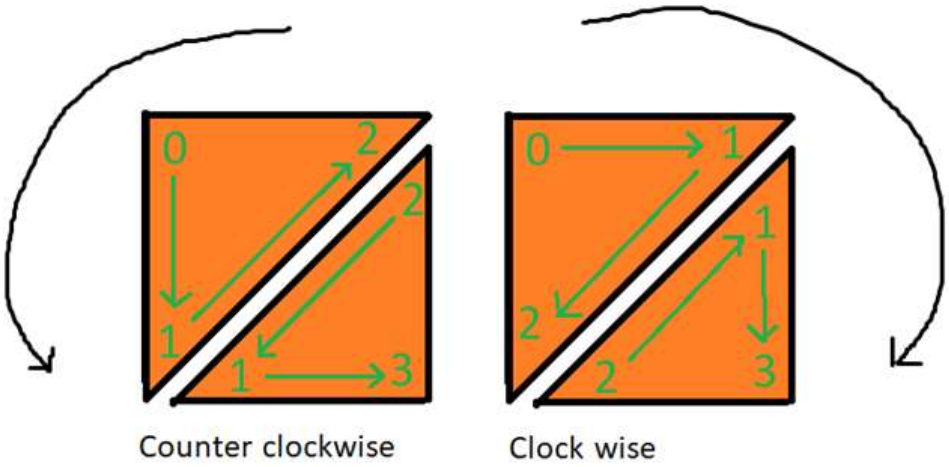

FIGURE 23. Order of vertices

Now, normally back face culling can't be seen, as the back facing *faces* can't be seen, but to make sure that it is working as should, we can render a *Cube*, with "cull mode" set to *gl.FRONT* – this does the exact opposite, skipping the rendering of front facing *faces*.

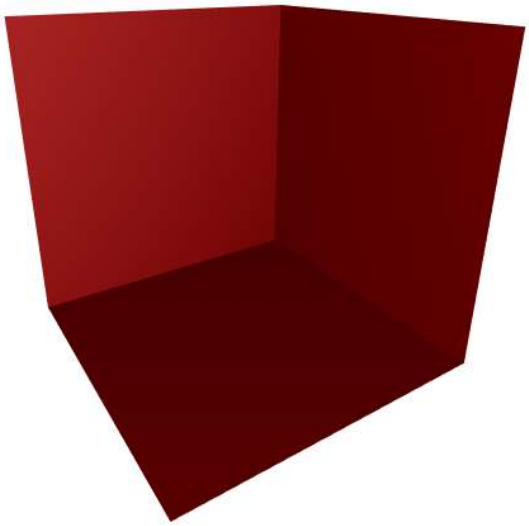

FIGURE 24. Cube with **front** face culling enabled

4.2.5 Mouse picking

In many applications, it is necessary to create some ways that the users can interact with 3D objects on the screen – for example, dragging a 3D object around with the mouse. This is known as *mouse picking*. However, as the mouse coordinates are 2 dimensional, one might ask how we can translate it to the 3D world coordinates, for checking collisions.

4.2.5.1 Unprojecting

Unprojecting is a technique of translating a 2D location on the display to the 3D space. It is a two-step process.

4.2.5.1.1 Translate coordinates on display to *clip space*

Clip space is the final coordinate system in the rendering pipeline (eq. the output of Vertex-shader). This can be done with equation 2.

$$clipCoords = \begin{bmatrix} -1 + 2 * \dfrac{p.x - viewport.x}{viewport.width} \\ -1 + 2 * \dfrac{p.y - viewport.y}{viewport.height} \end{bmatrix}$$

…where

$$p = mouse\ location\ on\ display$$

$$viewport = boundaries\ of\ the\ WebGL\ viewport$$

Equation 2. Display to clip space translation (XY).

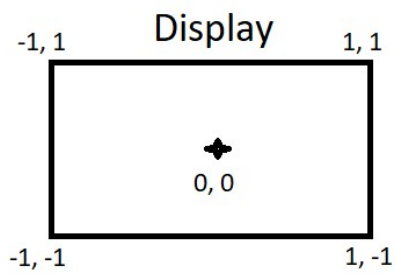

FIGURE 25. Clip space coordinate system

When using a perspective projection, the display is imagined existing on the XY plane, that is located at the Z coordinate: zNear (near clipping plane). With this we can conclude the clip space coordinates Z coordinate as:

$$clipCoords.z = zNear$$

Equation 3. Display to clip space translation (Z).

4.2.5.2 Backwards transformation

Continuing from previous chapter, backwards transform the *clip space* coordinates to a 3D world coordinate. This is done by multiplying the *inverse* of *viewProjectionMatrix* with the *clip space* coordinates.

$$mWorldCoords = viewProjectionMatrix^{-1} * clipCoords$$

Equation 4. Backwards transform clip space coordinates to 3D world

Afterwards, the X, Y, and Z components must be divided by the fourth "W" component, to correctly translate the resulting matrix into a vector.

$$wInverted = \frac{1}{mWorldCoords[3]}$$

$$worldCoords = \begin{bmatrix} mWorldCoords[0] * wInverted \\ mWorldCoords[1] * wInverted \\ mWorldCoords[2] * wInverted \end{bmatrix}$$

Equation 5. Transform coordinate matrix to a 3D vector

4.2.5.3 Ray casting

Continuing from the previous computed 3D world coordinate, what still needs to be done in order to collide the mouse with 3D objects, is to find the respective 3D direction, where the mouse is pointing at. When using a *perspective projection*, this is very simple. The direction can be represented as a 3D *ray*, that traverses through the previously found location, as well as the location of the *camera*.

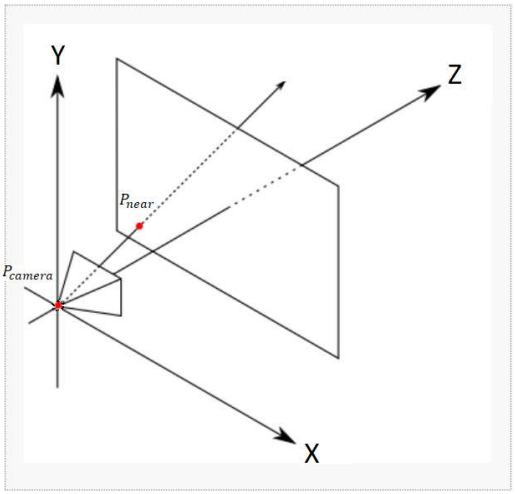

FIGURE 26. Casting a ray from camera location, through the mouse location. (Gabriel Gambetta 2019)

5 IMPLEMENTATION OF A 3D CHART

This chapter goes through the process and challenges of implementing a 3D chart using WebGL in addition to the already covered rendering techniques.

5.1 Triangulation of a 3D cube

The process of triangulating any shape starts by listing the unique vertices that are the minimal number of vertices that are needed to render the shape – for a *cube*, there are eight unique vertices (Figure 27). As covered in chapter Vertex normals, each vertex is associated with a *normal vector* that is used for shading. A cube has 8 vertices, then that means it also has 8 normals – Figure 27 shows how that looks.

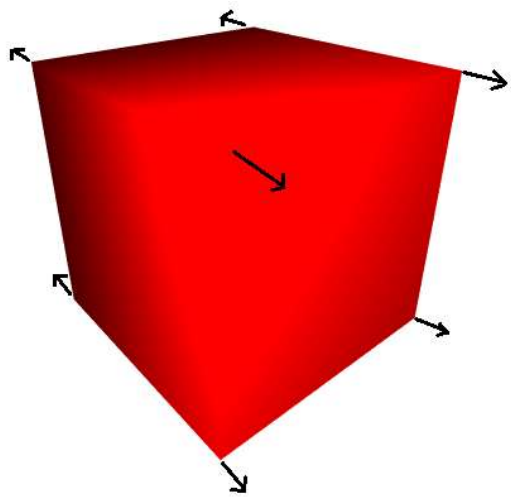

FIGURE 27. Cube with 8 vertices and 8 normals (black arrows)

That doesn't look very good, and it makes total sense – the cube only has enough normals to face two of eight possible directions. This is a case where the number of vertices should be increased above that of the unique vertex count, to get better shading results. True enough, almost always 3D cubes are rendered with 24 vertices and 24 normals (Figure 28).

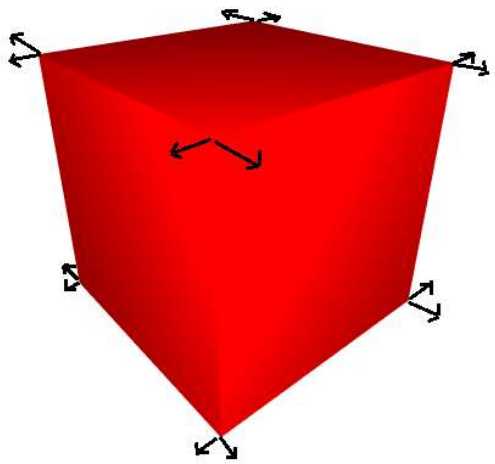

FIGURE 28. Cube with 24 vertices and 24 normals (black arrows)

5.2 Rendering the bounding box and axes

A 3D chart usually needs a visible bounding box for the scene. The primary purpose of the bounding box is to have a visual reference in the otherwise invisible 3D space. It can also be useful as a measuring tool, as there is no clear unit for the dimensions in the 3D scene. This makes sense, as if you look at Figure 29, all the sides of the frame are of equal length, but thanks to the *perspective projection* it looks as if they are different.

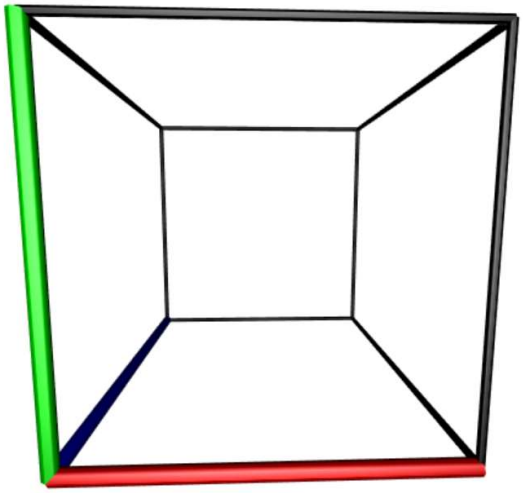

FIGURE 29. Bounding box and axes

The measurement unit in a 3D scene is known as *World Unit*, particularly known for the fact that it has no standard range of values and is unsuited for any measurement purpose.

5.2.1 Triangulation of a 3D line

To render the bounding box and axes shown in Figure 27, vertex and index buffers must be created to represent lines at arbitrary start *(A)*, end *(B)* locations as well as thickness *(T)* of the line. A traditional way of achieving this is to present the line as an *imperfect cylinder*.

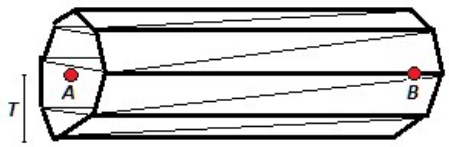

FIGURE 30. Triangulation of an imperfect cylinder

Because the *cylinder* is *"imperfect",* its' surface is not smooth, but has some amount of slight edges. This amount of edges can be represented as a *resolution* value. More edges will result in more rendering work, but a smoother looking line. For the purpose of this thesis, the filling of the ends of the line is not covered – this could, however, be done with a variety of methods. For one, by rendering another cylinder at each end, that is facing the end of the line. Figure 31 shows a line rendered with high resolution (edge amount is set to 100). Figure 32 shows the same line with a much lower resolution (edge amount is set to 8)

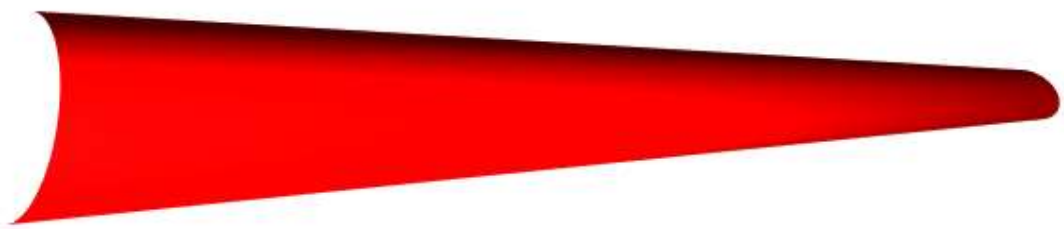

FIGURE 31. 3D line rendered with WebGL

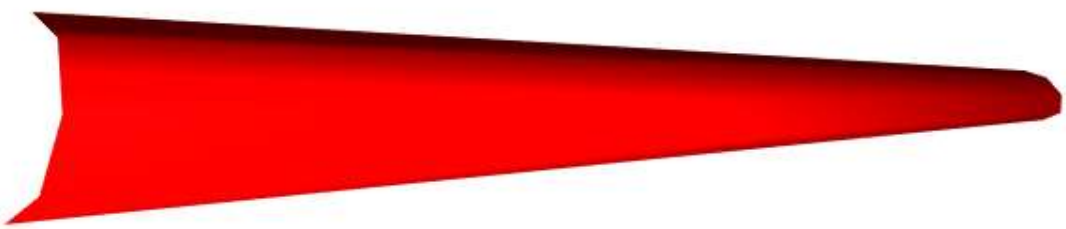

FIGURE 32. 3D line with low resolution

5.3 Translating datapoints based on Axes

A core feature of any charting library is the *Axis*, allowing the user to select a range of interested values on a dimension *(X, Y or Z).* The data will be rendered accordingly, so that the selected range is best visible. In

the 3D chart, each dimension has its own *Axis*, whose range can be configured arbitrarily (Figure 33). For the point series example, the cubes locations are translated according to these axes.

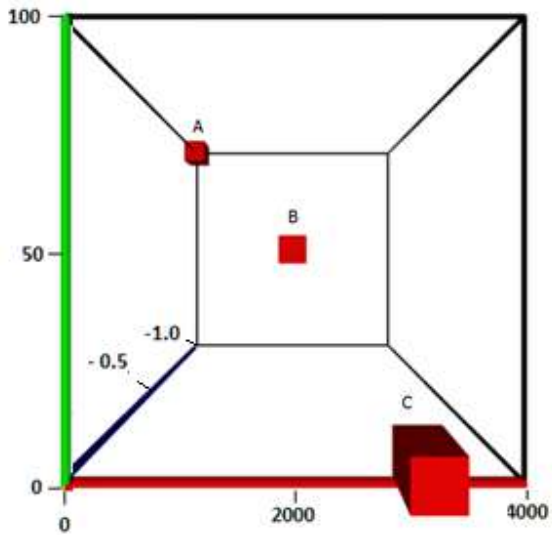

TABLE 2

Cube label	X	Y	Z
A	0	100	-1.0
B	2000	50	-0.5
C	3000	10	0.1

FIGURE 33. Cubes translated according to Axes

Each *Axis* has three configurable properties: start value, end value, and the size that the *Axis* represents in the 3D scene *(axis size)*. The effects of *Axis* transformations can be split into *scaling* and *translation* – these are most conveniently applied as *shader uniforms* (Code snippet 18).

```
uniform vec3 uScaling;
uniform vec3 uTranslation;
```

Code snippet 18. Axis translation uniforms

These *uniforms* are used to translate a position along 3D *Axes* to the 3D world space *(World Units)*.

```
attribute vec3 aPos;

void main(void) {
    vec3 posWorld = ( aPos - uTranslation ) / uScaling;
    gl_Position = posWorld;
}
```

Code snippet 19. Using the Axis translation uniforms in vertex shader

Uniforms for *Axes* that are centred in the scene can be calculated as follows:

$$uScaling = \frac{(end - start)}{size}$$

$$uTranslation = start + \frac{end - start}{2}$$

Equation 6. Calculation of Axis uniforms

5.4 Camera controls

The greatest asset of a 3D scene is the freedom of moving the perspective around. In this implementation, the camera can be moved by the user by pressing the right mouse button down on the canvas and moving the mouse – this rotates the camera around the center of the scene (so that the camera is always pointed at the center). Additionally, the mouse wheel can be used to move further/closer to the center of the scene. For computing camera transformation, first the vectors relative to the camera location, and the target that the camera is looking at (center of scene) must be found (Figure 34) (learnopengl.com).

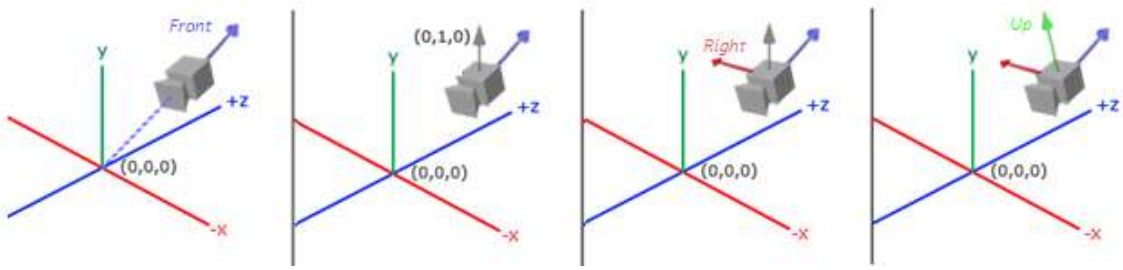

FIGURE 34. Finding "look-at" vectors. (Learn OpenGL 2014)

$$front = normalize(\, cameraLocation \; - \; target \,)$$
$$right = normalize(\, Y \; \times \; front \,)$$
$$up = normalize(\, right \; \times \; front \,)$$

Formula 3. Look-at vectors

5.4.1 Horizontal mouse rotation

Horizontal mouse movement rotates the camera location vector around the *up* vector (Figure 34). Vector rotation around an arbitrary *Axis* can be calculated using *Rodrigues' rotation formula* (formula 4).

$$vRot = \mathrm{v}\cos\theta + (1 - \cos\theta)(\mathrm{axis} \; \cdot \mathrm{v}) * \mathrm{axis} + (\mathrm{axis} \; \times \; \mathrm{v}) \sin\theta$$

Where

$$v = vector\ to\ be\ rotated$$
$$\theta = \mathrm{rotation\ amount\ in\ radians}$$
$$axis = vector\ to\ rotate\ around$$

Formula 4. Rodrigues' rotation formula. (Wikipedia 2019)

$$newCameraLocation = rotate(\, cameraLocation, up, rotateAmount \,)$$

Equation 7. Horizontal camera rotation

5.4.2 Vertical mouse rotation

Vertical mouse movement rotates the camera location vector around the *right* vector (Figure 34).

$$newCameraLocation = rotate(\, cameraLocation, right, rotateAmount \,)$$

Equation 8. Vertical camera rotation

5.4.2.1 Preventing problematic camera angles

Rotating to completely vertical angles will break the camera controls. This is because of how the "look-at" vectors are found (chapter 5.4 camera controls). Specifically, because of the temporary Y vector that is used. The algorithm will not work if any of the three camera vectors is parallel to the temporary vector. This problem angle can be prevented by limiting the vertical rotation angle to a minimal degree relative to the used Y vector. The angle between two vectors can be calculated with equation 9.

$$\cos\theta = \frac{a \cdot b}{|a| * |b|}$$

Equation 9. Angle between vectors

5.4.3 Zooming with mouse scroll wheel

Zooming simply moves the camera location along the direction towards the scene center.

$$newCameraLocation = cameraLocation + zoomAmount * (center - cameraLocation)$$

Equation 10. Camera zooming

6 OPTIMIZING PERFORMANCE

The goal in LightningChart JS implementation of a Point Series, is the ability to render as much data as possible, and as fast as possible. To understand the bases of optimizations, we must first understand the runtime environment in a web application and the used resources.

6.1 Optimization of web applications in general

The definition for how performant a web application is depends on two factors: how *fast* it performs its scripts, and how much resources it consumes while doing so.

6.1.1 Central Processing Unit

The CPU is hardware that performs all computations (mathematics, if checks, etc.). The quality of the CPU is a big factor how fast JavaScript performs. The longer the applications scripts take, the more it stresses the CPU, which shows as reduced interact ability and *lag*. Saving on CPU performance is generally the biggest challenge in a performant web application, as it is hard to utilize multiple threads for a single web application.

6.1.2 Random Access Memory

The RAM is hardware that stores all temporary data of the application (variables, objects, classes) and the whole operating system. Whenever any object is created in JavaScript, the memory is allocated from the RAM. If too many objects are created, the web application can crash when it runs out of memory. Unused objects are automatically unallocated by *Garbage Collection (GC)*. RAM usage performance can be split into two types: static RAM allocation, which is the amount of memory the application allocates and stays static as it runs and dynamic RAM allocation, which represents all the memory that is consumed from temporary variables and fluctuates throughout the life time of the application. These categories do not have any technical differences, but the grouping is useful when thinking how to optimize an application – high static RAM allocation mostly limits the target market by demanding better hardware, whereas high dynamic RAM allocation can have very bad effects on the displayed application performance. This is because *Garbage Collection* is quite a heavy operation, so if the application creates a lot of unused objects it can also show as reduced interact ability and *lag*.

6.1.3 Graphics Processing Unit

The GPU is hardware that is used specifically for heavy graphical applications (such as *WebGL*). It is basically like a dedicated CPU + RAM combination, so it can be utilized both for storing data used for rendering (*buffers* and *uniforms*) as well as processing power (*shaders*). Any resource that can be moved from the CPU to the GPU should normally be moved, because as said previously the CPU is generally the bottleneck in

performance, as well as the fact that the GPU is optimized far better for the rather limited tasks that it can perform. The performance of an application that utilizes the GPU well so well, that the GPU is the bottleneck will run into *fps (frames per second)* issues when the processing capability of the GPU is reached.

6.2 Optimizing PointSeries3D

For the scope of this thesis, only a single series type and its optimizations will be considered. PointSeries3D is a chart component that renders a group of 3D points according to a selected style and its *Axes*. There is no set limit to the amount of data that it can be given, and the data can be given and rendered in parts. For example, first give 1000 points and after a couple seconds 10000 more. In the scope of this thesis, the PointSeries3D can be styled with the following properties: size of all points as *World Units,* fill color of all points as *RGB (red-green-blue)*. The points will always be rendered as triangulated cubes. It is the task of PointSeries3D to perform these tasks as optimally as possible.

6.2.1 Reusing previous vertices

As written in the feature description (chapter 6.2), the PointSeries3D can have points added to it at any time. In this case, the vertex and index buffers need to be recreated to allocate for the new cubes. This can be optimized by *caching* (keeping an object in memory) the previously computed lists of vertices and indices. This way, PointSeries3D can avoid having to compute the previously added cubes again, by only computing the newly added cubes and appending those to the previous vertices and indices. This optimization saves on CPU but increases the amount of RAM usage because all the triangulated cube information is not released. Figures 35 and 36 show the comparison in JS heap size (allocated RAM) for 100000 cubes.

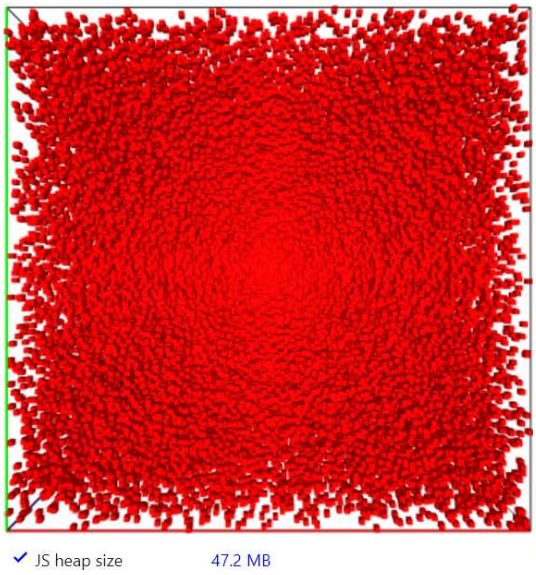

FIGURE 35. JS heap size without caching vertices

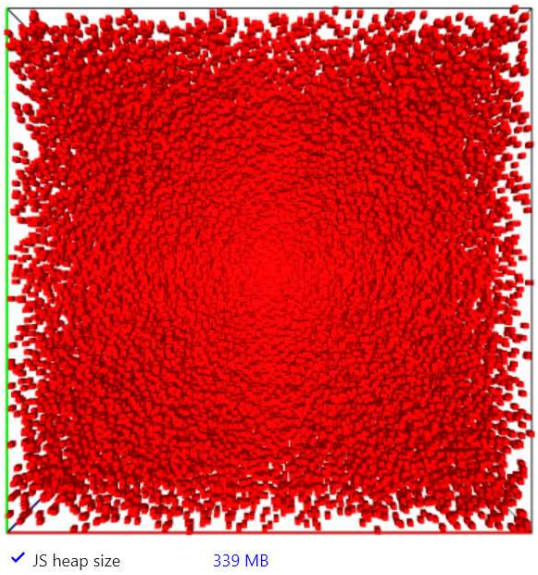

FIGURE 36. JS heap size having cached vertices

6.2.2 Caching previous vertices in WebGL buffers

The caching of vertices in RAM in previous chapter consumes a lot of precious memory and the stored information is rendering information – this is where GPU memory comes in to save the day. The previous vertices can be retained in the GPU memory and extended with the vertices of new cubes. This is made possible by *buffer preallocation*. Instead of making a new GL buffer each time more vertices are added, a large, empty GL buffer is created at the start to which vertices can be added at any time.

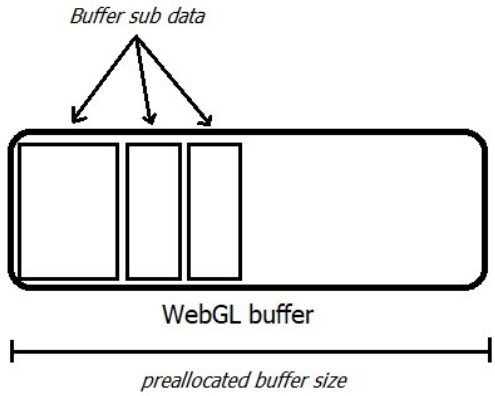

FIGURE 37. Preallocated GL buffer

This moves the large overhead of RAM usage from caching vertices to the GPU memory and as a matter of fact, creating new buffers each time is also a some-what slow operation – using a preallocated buffer also saves some processing time and generates fewer temporary objects. This does not, however, come without its problems. These will be covered in chapter 6.2.4 Mouse picking.

```
// Create a preallocated Vertex buffer.
const size = 1000000
const vertexBuffer = gl.createBuffer( )
gl.bindBuffer( gl.ARRAY_BUFFER, vertexBuffer )
gl.bufferData( gl.ARRAY_BUFFER, new Float32Array( size ), gl.STATIC_DRAW )

// ... Load a chunk of data into the preallocated Vertex buffer.
const dataChunk = [ 0, 1, 2, 3, 4, 5 ]
let offset = 0
gl.bufferSubData(
    gl.ARRAY_BUFFER,
    offset += dataChunk.length,
    new Float32Array( dataChunk )
)
```

Code snippet 20. Creation and usage of preallocated Vertex buffer

6.2.3 Scaling in Vertex Shader

Looking back at the triangulation of a cube and the features of a PointSeries3D, the vertices of the cube will vary based on the value of "*point size*" *(World Units)*. Because the *Axis* transformation in the *vertex shader* expects the vertex positions to be in *Axis* values, not *World Units*, the *point size* will have to be transformed to *Axis* values when the cube is triangulated.

$$sizeAxis = sizeWorld * |\, axisStart - axisEnd \,|$$

Equation 11. Translation of World Units to Axis Units (point size)

However, a problematic result with this approach is that whenever the *Axis* range changes, all the vertices of the PointSeries3D are invalid and will have to be recomputed again. In applications with a lot of cubes and where the *Axis* ranges are dynamically changed, this is not a performant solution. Ideally, a change in neither *point size* nor *Axis* range would have to trigger a recomputation. This can be achieved by moving all related factors to the *shaders* as either *attributes* or *uniforms*. *Point size* is the same for all cubes, so it will be a *uniform* (Code snippet 21).

```
uniform float uPointSize;
```

Code snippet 21. Point size uniform

The *Axis* range is already a *uniform* in the vertex shader (chapter 5.3 translating datapoints based on Axes). Now, the only thing that is left is which direction to apply the *point size*. A cube has eight unique vertices, all of which are defined as a sum of the cubes centre location and a multiplication of *point size* and some direction (Figure 38, equation 12).

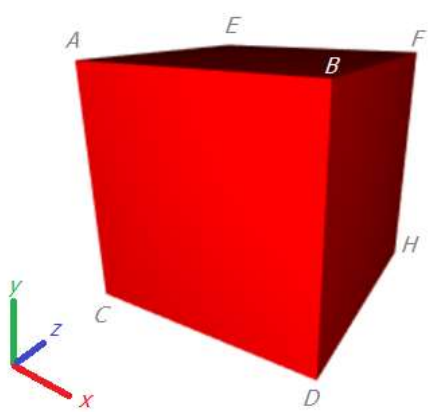

FIGURE 38. Vertices of a cube visualized

$$A = \begin{bmatrix} center.x - pointSize * sizeAxis.x \\ center.y + pointSize * sizeAxis.y \\ center.z - pointSize * sizeAxis.z \end{bmatrix}$$

$$B = \begin{bmatrix} center.x + pointSize * sizeAxis.x \\ center.y + pointSize * sizeAxis.y \\ center.z - pointSize * sizeAxis.z \end{bmatrix}$$

$$C = \begin{bmatrix} center.x - pointSize * sizeAxis.x \\ center.y - pointSize * sizeAxis.y \\ center.z - pointSize * sizeAxis.z \end{bmatrix}$$

$$etc \ldots$$

Equation 12. Computation of cube vertices

Analysing equation 12 will show that the only difference between the vertex calculations is the multiplier of *point size* and *Axis* size. By adding this *"vertex direction"* as an additional *attribute* (Code snippet 22), all the factors that depend on *point size* or *Axis* range are successfully moved to the *shader*.

```
attribute vec3 aVertexDirection;
```

Code snippet 22. Vertex direction attribute

The previously concluded *Axis* transformation code in the vertex shader will then be extended by offsetting each vertex with the multiplication of point size and direction (Code snippet 23).

```
vec3 posWorld = ( aPos - uTranslation ) / uScaling + uPointSize * aVertexDirection;
```

Code snippet 23. Axis translation with point size and direction uniform

This does add three additional *attributes* to the vertex buffer which is not free, but this makes changing *point size* or *Axes* ranges significantly lighter – now there is no recomputation needed, only the *uniform* values must be updated. This optimization unfortunately also contributes to problems with mouse picking.

6.2.4 Mouse picking

Mouse picking on PointSeries3D is done using the techniques explained in chapter 4.2.5 mouse picking. A 3D ray is projected from the camera through the location of mouse on the *near clipping plane.* This ray is then checked for collisions with any of the cubes of the PointSet3D.

6.2.4.1 Region boundary optimization

The collision checking is an extremely heavy operation, especially when there is a lot of cubes to check (say, millions of them). Avoiding it whenever possible is crucial for functional mouse picking performance. One way of limiting this is by first doing a preliminary boundary check for the combined *region* boundaries of all the cubes in the PointSet3D. If the ray doesn't collide with the combined boundary, then it is impossible that it collides with any of the cubes.

6.2.4.2 Challenges of mouse picking with previous optimizations

The previous optimizations improved PointSeries3D performance on multiple fronts (CPU and RAM usage) by moving resource usage to the GPU. However, for collision checking the vertices will be needed and the *point size* will have to be factored in. This essentially means that the previous optimizations are not usable when mouse picking is needed. In the thesis implementation this issue was met with an ambitious approach, where the optimizations are disabled when mouse picking is needed and again enabled when mouse picking is no longer needed. It comes without saying that such logic is complex and comes with its own challenges.

7 FINDINGS AND RESULTS

While it is fast and relatively simple to implement rendering features with WebGL, the necessity for user interact ability proves to be extremely difficult while retaining top performance. This should be considered carefully before starting creation of new applications and when specifying their capabilities.

7.1 PointSeries3D performance

This chapter shows a reference performance with the finished implementation. The application was run in Google Chrome (browser) with AMD Ryzen 5 2400G (CPU) and NVIDIA GeForce GTX 1050 Ti (GPU). Figure 39 shows the maximum number of static cubes rendered with best achievable *fps (frames-per-second)* with the used monitor. At this point, the GPU's processing capability is reached, and it starts bottlenecking the rendering loop. This can be seen by observing the Task Manager (Figure 40).

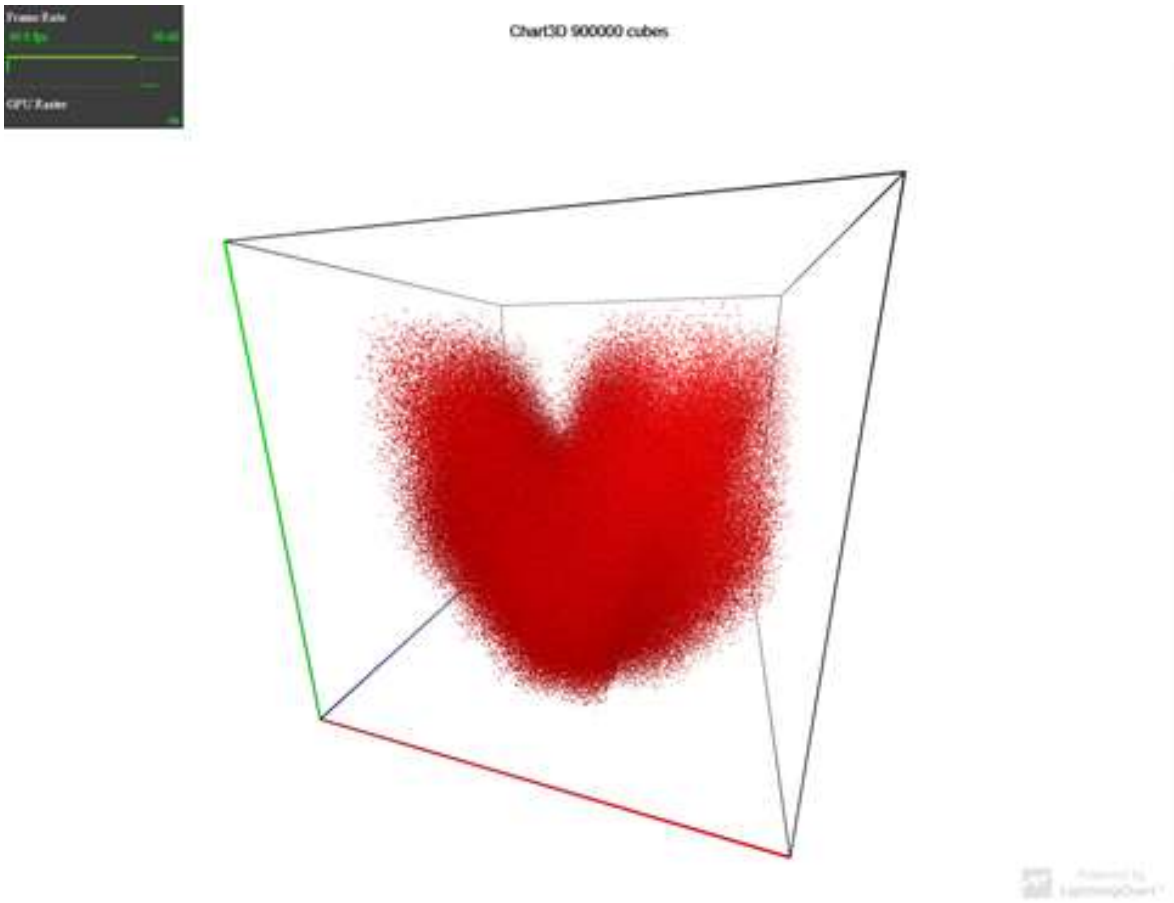

FIGURE 39. PointSeries3D 900000 cubes with 60 fps in Chrome.

Name	S...	19% CPU	55% Memory	0% Disk	0% Network	99% GPU
Google Chrome (7)		5.7%	1,509.9 MB	0.1 MB/s	0 Mbps	96.4%

Figure 40. Task Manager showing maximum GPU usage

Method timing shows that PointSeries3D can prepare 1000 points for rendering in 5-8 milliseconds. A web application running with 60 *fps* renders a new frame every 16.6 milliseconds. This leaves the application with roughly eight milliseconds to prepare for the next frame (considering the time needed for the actual rendering and the browser itself). With this we can conclude that the PointSeries3D implementation can handle approximately 1000 incoming data points every frame, while retaining 60 *fps* (with this running environment).

8 RECAP

To recap the core features that were implemented: a 3D scene was created, where the perspective camera and light source can be moved around with mouse. Inside this 3D scene there are 3D axes and bounding box rendered with triangulated 3D lines as well as Phong shaded cubes that can be created in large numbers by the user. The locations of these cubes are translated according to the configuration of the 3D axes.

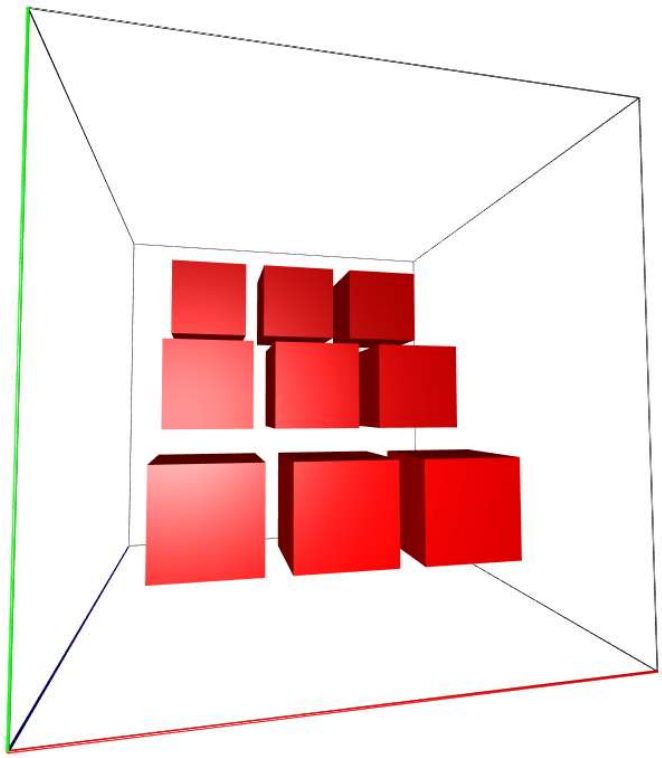

FIGURE 41. Phong shaded cubes inside a 3D bounding box

The created 3D rendering base will be an important steppingstone for the implementation of many more 3D data visualization features. During the implementation phase, powerful optimizations such as *Buffer preallocation* and *point size scaling in shaders* were found, which may also be applicable to other already existing features of *LightningChart JS*.

9 SOURCES

ARCTION LTD. [Referenced 2019-11-21]. Available: https://www.arction.com

CIVILSEEK. [Referenced 2019-11-21]. Available: https://civilseek.com/perspective-projection-drawing/

GABRIEL GAMBETTA. [Referenced 2019-11-21]. Available: https://www.gabrielgambetta.com/computer-graphics-from-scratch/perspective-projection.html

HIGHCHARTS. [Referenced 2019-11-21]. Available: https://www.highcharts.com

LEARN OPENGL. [Referenced 2019-11-21]. Available: https://learnopengl.com/Getting-started/Camera

WEBGL FUNDAMENTALS. [Referenced 2019-11-21]. Available: https://webglfundamentals.org/

WIKIPEDIA. [Referenced 2019-11-21]. Available: https://en.wikipedia.org/wiki/Rodrigues%27_rotation_formula