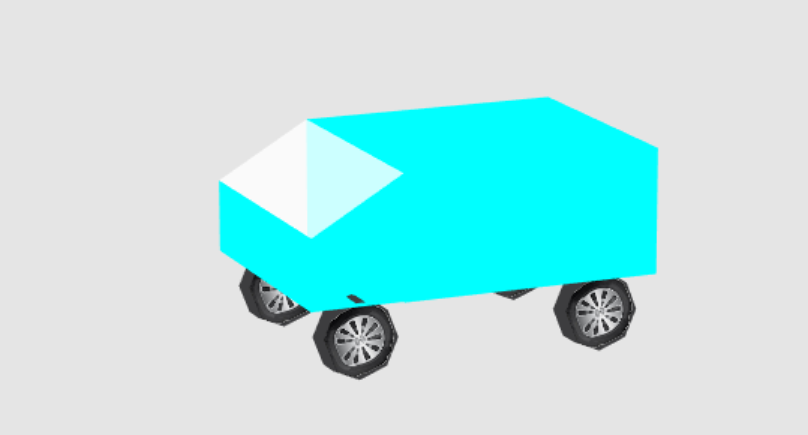
**Basic movie effects**

1. **Composed object**  
   a) Add at least one manually composed object that consists of multiple scene graph nodes, e.g., roboter with arms and legs.  
   b) Animate separate parts of the object and also move the composed object itself in the scene.  
   c) Use at least two clearly different materials with specular properties for the various parts of your composed object. Note that a material is different from a texture and that just setting different vertex colors is not sufficient.
2. **Hand-crafted object**

The tires of the car represent the hand-crafted object. The whole car is implemented in the car.js file, which makes use of the initRegularOctagonalPrism.js. The initRegularOctagonalPrism.js returns all model properties(vertices, indices, normals, texture).

**Car  
initRegularOctagonalPrism in 2D** (front view)  
  
  
We assume, that the length of all sides are equal. Now, we want to compute the length m, if the side length s is given. We also assume, that the height and width of the whole box is equal to 2. The figure bellow depicts the relationsship between the sidelength s and the length m.



2

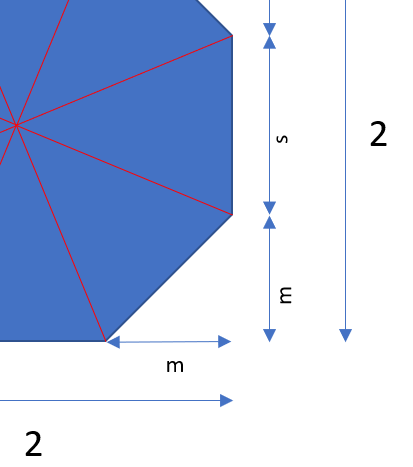
2

s

m

m

m



Thus,

The equation should also hold.

If we combine the two equations together, we get

Thus, the following equation must be true.

It follows:

So, we first compute the sidelength s. Afterwards the length m can also be computed by the following formula.

The length is equal to the variable octagonSideLength in the program.

**Compution in Javascript**

**let** octagonSideLength = 2/(2/***Math***.sqrt(2)+1);  
**let** m = octagonSideLength/***Math***.sqrt(2);

**Composition**

The model contains the vertices of the front regular octagon, the back regular octagon and the side areas. The inner method getHexagonVertices(z) generates the appropriate vertices of a hexagon with the property, that the third dimensional value is the given z. The inner method getHexagonIndices returns the indices. The parameter offset of the method getHexagonIndices should refer to the index, where the first vertex of the hexagon is stored. The inner method getTextureCoords returns the texture coordinates of a hexagon as an array.

**function** getHexagonVertices(z){  
 **return** [  
 0,0,z, *// 0 - middle point* 1,-octagonSideLength/2,z, *// 1 right bottom* 1,octagonSideLength/2,z, *// 2 right top* 1-m,1,z, *// 3 top* -1+m,1,z, *// 4 top* -1,1-m,z, *// 5 left* -1,-1+m,z, *// 6 left* -1+m,-1,z, *// 7 bottom* 1-m,-1,z, *// 8 bottom* ];  
}

**function** getHexagonIndices(offs){  
 **let** arr = [  
 0,1,2,  
 0,2,3,  
 0,3,4,  
 0,4,5,  
 0,5,6,  
 0,6,7,  
 0,7,8,  
 0,8,1  
 ];  
 **for**(**let** i=0;i<arr.**length**;i++)  
 arr[i] += offs;  
 **return** arr;  
}

a) Create one scene graph node that renders a hand-crafted 3D shape (5-25 vertices; not a cube, sphere, quad, or loaded model). Fully specify properties for this object, i.e., vertices, normals, and texture coordinates.   
b) Apply a texture to your self-created complex object by setting proper texture coordinates.

1. **Illumination**  
   a) Use multiple light sources.  
   b) One light source should be moving in the scene.  
   c) Implement at least one spot-light by extending the existing light node LightSGNode and a Phong shader. Apply Phong shading to all objects in the scene.
2. **Camera**  
   After the automatic camera flight, it should be possible to freely control the camera using the mouse and keyboard to move through the scene.   
   a) Use the WASD-keys to manually control the camera along the viewing direction. (W-key: forward movement, S-key: backward movement, A-key: leftward movement, D-key: rightward movement)  
   b) Use the mouse to control the heading (mouse-x; rotation around y-axis) and pitch (mouse-y; rotation around x-axis) of the camera relative to the ground (no roll!).

The control of the camera is implemented in the javascript file ”cameraControl.js”, which is in the src directory. The camera object contains the data about the placed camera.  
Important  
The angle pitch and yaw are stored in the camera object. The variable position in the camera object, represents the position of the camera from the point of the rotated world. According to this definition, the mouse and keyboard interactions can be very easily implemented.  
Thus, we first rotate the world and then do the translation for computing the viewmodel.

**function** *getViewMatrix*(){  
 **let** viewMatrix = mat4.create();  
 viewMatrix = mat4.**translate**(mat4.create(), viewMatrix, vec3.negate(  
vec3.create(), ***camera***.**position**));  
 viewMatrix = mat4.**rotateX**(mat4.create(), viewMatrix, ***glm***.deg2rad(-***camera***.**angle**.**pitch**));  
 viewMatrix = mat4.**rotateY**(mat4.create(), viewMatrix, ***glm***.deg2rad(-***camera***.**angle**.**yaw**));  
 **return** viewMatrix;  
}

***document***.addEventListener(**'keypress'**, **function**(event){  
 **let** k = event.**key**.toUpperCase();  
 *// W/S control forward and backward, while A/D control strafing left and right.  
 // source: https://en.wikipedia.org/wiki/Arrow\_keys#WASD\_keys* **let** diff = vec3.fromValues(0,0,0);  
 **if**(k == **"W"**){  
 diff = vec4.fromValues(0,0,-1,0);  
 }  
 **if**(k==**"A"**){  
 diff = vec4.fromValues(-1,0,0,0);  
 }  
 **if**(k==**"S"**){  
 diff = vec4.fromValues(0,0,1,0);  
 }  
 **if**(k==**"D"**){  
 diff = vec4.fromValues(1,0,0,0);  
 }  
 diff = vec3.scale(vec3.create(),diff,0.1);  
 ***camera***.**position** = vec3.add(vec3.create(),***camera***.**position**,diff);  
});

1. **Animations**  
   a) Need to start automatically.  
   b) Need to be framerate-independent.  
   c) The camera needs to be animated without user intervention.

The animation is implemented as a Scene Graph Node. We implement the animation in the file   
”AnimationSGNode.js”. The class contains the keyframes of the animation. Between the keyframes, we use the linear interpolation along the start and end value. The Scene Graph Node AnimationSGNode extends from the provided class “TransformationSGNode” and sets the matrix of the super class in der render function.

**Linear Computation**

*// interpolate between startM and endM linearly***let** scale = (timeInMs-startM.**timeInMs**) / (endM.**timeInMs** - startM.**timeInMs**);  
  
**let** diff = vec3.**sub**(vec3.create(),endM.**position**,startM.**position**);  
diff = vec3.scale(vec3.create(),diff,scale);  
res.**position** = vec3.add(vec3.create(),diff,startM.**position**);  
  
res.**angle**.**pitch** = (endM.**angle**.**pitch**-startM.**angle**.**pitch**) \* scale + startM.**angle**.**pitch**;  
res.**angle**.**yaw** = (endM.**angle**.**yaw**-startM.**angle**.**yaw**) \* scale + startM.**angle**.**yaw**;

We can also see, that this technique is framerate-independent, because the next frame is not constructed from the previous frame.

**Render Function**

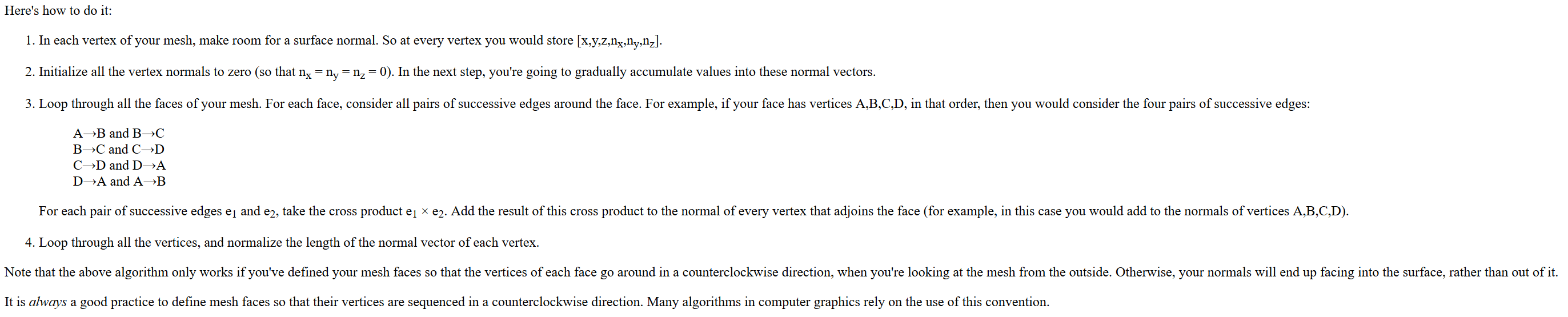
render(context) {  
 **let** timeInMilli = context.**timeInMilliseconds**;  
 **if**(!timeInMilli){  
 timeInMilli = 0;  
 }  
 **let** cam = **this**.computeCameraInfo(timeInMilli);  
 **let** viewMatrix = mat4.create();  
 viewMatrix = mat4.**translate**(mat4.create(), viewMatrix, vec3.negate(vec3.create(), cam.**position**));  
 viewMatrix = mat4.**rotateX**(mat4.create(), viewMatrix, ***glm***.deg2rad(-cam.**angle**.**pitch**));  
 viewMatrix = mat4.**rotateY**(mat4.create(), viewMatrix, ***glm***.deg2rad(-cam.**angle**.**yaw**));  
 **this**.**matrix** = viewMatrix;  
 **super**.render(context);  
}

Terrain Heightmap

**Normal Calculation**We decided to precompute the normal vectors and store them into a json file. Therefore, we wrote a small javascript file, which computes the normal vectors and prints them on the console. We copied the output from the console and save them into a JSON file. The javascript file is in the folder “terrainNormalMapGeneration”.

The most interesting part is the algorithm for the computation of the normal vectors, which is stated bellow.

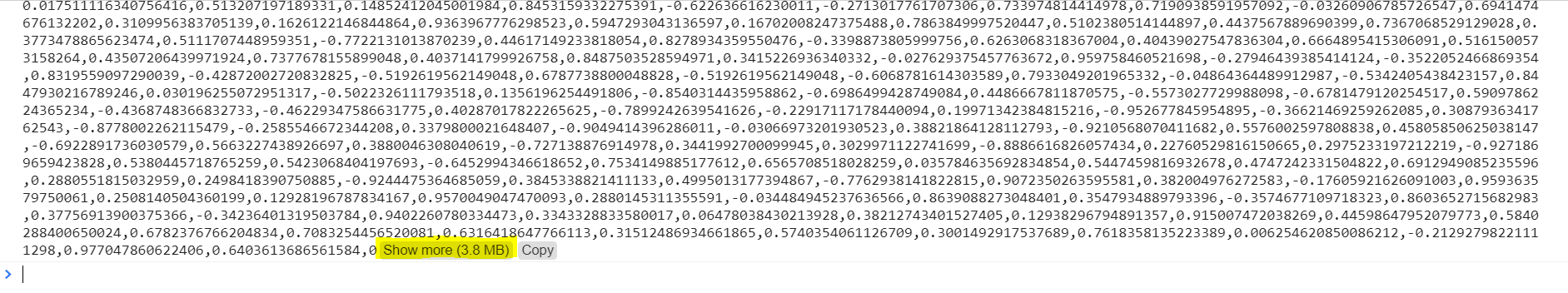
Algorithm for Normal Calculation



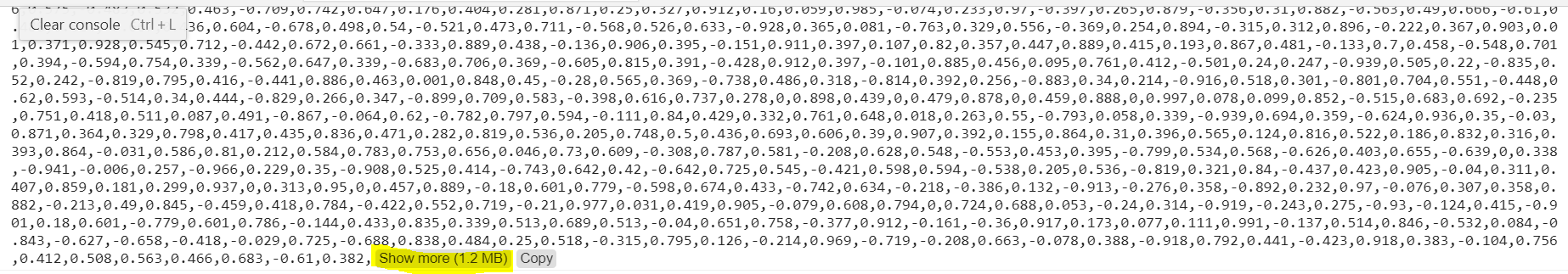
Reference of the algorithm <https://mrl.nyu.edu/~perlin/courses/fall2002/meshnormals.html>

It is important to say, that this algorithm works only correct, if the vertices are in counterclockwise order.

In order to compute the normal vectors, it is very important to have a good structure. In our case, we decided to define an one dimensional vertex array. Additional we define an one dimensional array, which contains a 3 values, which refer to the vertex array. Every item in the array, states that the 3 values, which refers to the vertex array, form a triangle. With this defined datastructure, it is very simple to compute the normal vectors. After the computation, we decided to round all normal values with 3 digits. If we don’t round the values, the json file will be 3.8MB. This is depicted in the figure bellow. We can also see, that the values contains to much digits.



After, we defined the fixed precision, the file was only 1.2 MB. This is also pictured in the figure bellow.



Source code of the computation of the normal vectors

**function** *computeAndPrintNormalsVectors*(vertices, indices){  
 **var** normals = [];  
 vertices.forEach(() => normals.push(vec3.create()));  
  
 **for** (**let** i = 0; i < indices.**length**; i++) {  
 **let** vertexA = vec3.normalize(vec3.create(),vec3.**sub**(vec3.create(),vertices[indices[i][1]],vertices[indices[i][0]]));  
 **let** vertexB = vec3.normalize(vec3.create(),vec3.**sub**(vec3.create(),vertices[indices[i][2]],vertices[indices[i][0]]));  
 **let** triangleNormal = vec3.cross(vec3.create(),vertexA,vertexB);  
 normals[indices[i][0]] = vec3.add(vec3.create(),normals[indices[i][0]],triangleNormal);  
 normals[indices[i][1]] = vec3.add(vec3.create(),normals[indices[i][1]],triangleNormal);  
 normals[indices[i][2]] = vec3.add(vec3.create(),normals[indices[i][2]],triangleNormal);  
 }  
 **for** (**let** i = 0; i < normals.**length**; i++) {  
 normals[i] = vec3.normalize(vec3.create(), normals[i]);  
 }  
 **let** res = [];  
 **for** (**let** i = 0; i < normals.**length**; i++) {  
 res.push(***Number***.parseFloat(normals[i][0].toFixed(***accuracy***)));  
 res.push(***Number***.parseFloat(normals[i][1].toFixed(***accuracy***)));  
 res.push(***Number***.parseFloat(normals[i][2].toFixed(***accuracy***)));  
 }  
 ***console***.log(***JSON***.stringify(res));  
}

For the visualization of the terrain heightmap we have only to draw a mesh and apply the precomputed normal vectors, which are stored in the json file.

Mesh  
In the first step, all vertices and texture coordinates are pushed in two separate arrays. We go through all cells, compute the vertex position, texture coordinate and push them. Our models goes from -1 to 1 in every dimension.

Algorithm:

**let** cellSize = 2/meshSize;  
**for** (**let** i = 0; i < meshSize; i++) {  
 **for** (**let** j = 0; j < meshSize; j++) {  
 vertices.push(cellSize \* j - 1);  
 vertices.push(height[i \* meshSize + j]);  
 vertices.push(cellSize \* i - 1);  
 texture.push(j/meshSize);  
 texture.push(i/meshSize);  
 }  
}

After computing the coordinates, we continue with the computation of the indices, which represents the triangles and refers to the vertex array.  
For every Cell we now push the indices.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | 1 … col + 1 + row \* meshSize  2 … col + row \* meshSize  3 … col + (row + 1) \* meshSize  4 … col + 1 + row \* meshSize  5 … col + (row + 1) \* meshSize  6 … col + 1 + (row + 1) \* meshSize |

**for** (**let** row = 0; row < meshSize-1; row++) {  
 **for** (**let** col = 0; col < meshSize-1; col++) {  
 **let** vertexIndex = col + row\*meshSize;  
 indices.push(vertexIndex+1);  
 indices.push(vertexIndex);  
 indices.push(vertexIndex+meshSize);  
 indices.push(vertexIndex+1);  
 indices.push(vertexIndex+meshSize);  
 indices.push(vertexIndex+meshSize+1);  
 }  
}