Billboarding

What should be done: The billboarded object behaves like a normal object, except rotation: It is always normal to the viewer's direction.

Idea

The billboarded node (QuadRenderNode) has a normal vector (1,0,0) (we chose this as default).

We define 2 angles (groundAngle, heightAngle) where xAngle represents the horizontal rotation (around upgoing y-axis) and the yAngle represents the “up-and-down-rotation” (around the x-axis). While the xAngle can have an arbitrary value, the yAngle has to be between -90 and +90 degrees.

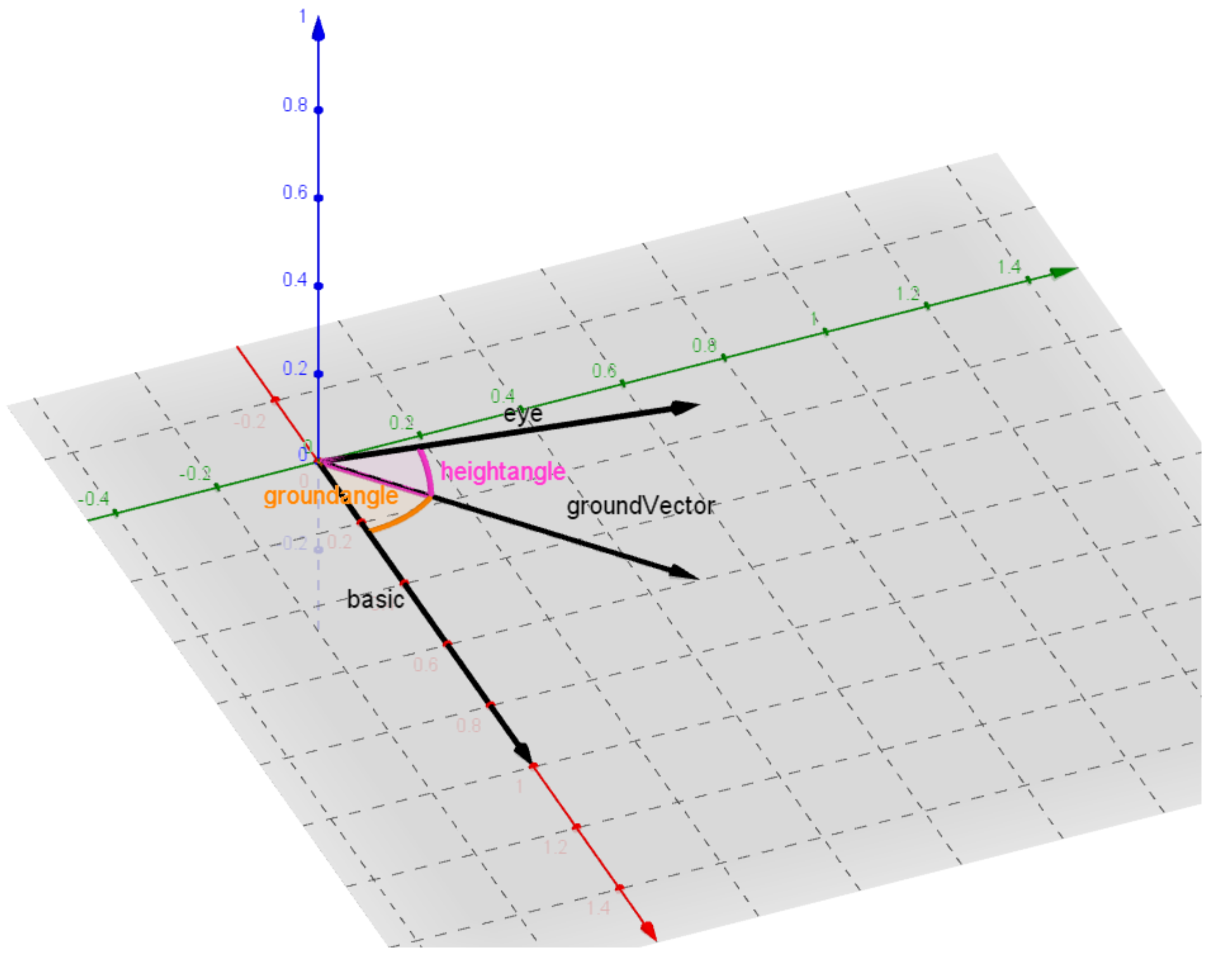


Figure 1: Calculation of rotation angles – GeoGebra screenshot

- The eyeVector (pointing from the object to the camera or the other way round) can be copmuted by subtracting the viewer’s position from the object’s position.
- For the groundVector we take the eyeVector and set the height coordinate (y) to 0.
- The basic vector is (hard coded) (1,0,0).
- The GroundAngle is the angle between basic and groundVector, the heightAngle the angle between eye and groundVector.

Finally, we rotate the billboarding object at first by the heightAngle, and then we do the horizontation rotation by the groundAngle.

Code

We let the billboard node class inherit from the TransformationSGNode (because the rotation is a transformation), therefore we only had to calculate the transformation matrix and call the render function from the parent class. Because texture images are only square-sized, we also provided a scale transformation node to get e.g. a rectangular QuadRenderNode.

class BillboardNode **extends** TransformationSGNode {	Class inherits from TransformationSGNode

constructor(xScale) {
super();
this.**alpha** = 1;
var quadRenderNode = **new** QuadRenderNode();
var scaleNode = **new** TransformationSGNode(***glm***.scale(xScale, 1, 1), quadRenderNode);
this.append(scaleNode);
this.**absPosition** = [1, 0, 1];
}
	Constructor:
	• Create a scale node for rectangular billboards
	• Set absolute position of billboard

render(context) {
var dir = vec3.create();
vec3.**sub**(dir, ***eye***, **this**.**absPosition**);
vec3.scale(dir, dir, 1 / vec3.length(dir));
var dirGround = [dir[0], 0, dir[2]];
var stdVec = [1, 0, 0];
	RENDER function
	create the vectors that are metioned above (stdVec = basis, dirGround = groundVector, dir = eyeVec)

var xAngle = vec3.angle(dirGround, stdVec);
var yAngle = vec3.angle(dir, dirGround);
xAngle = *convertRadiansToDegree*(xAngle);
yAngle = *convertRadiansToDegree*(yAngle);
	Compute the 2 rotation angles

//cos(alpha)=cos(360-alpha): xAngle value is 0;180].
if (***eye***[2] < **this**.**absPosition**[2]) {
xAngle = xAngle + 90;
} **else** {
xAngle = 90 - xAngle;
}
	The angle formular contains a arccos function which returns an angle between 0 and 180 degrees. At billboarding we need 360 degrees. Therefore we need a case distinction:
	If we move below a billboard object, then rotation has to be done by the negative angle. Therefore we ask if we are above or below the object (eye[1]-absPosition[1]) and then add or subtract our angle to/from the 90-degree-offset.
	For the horizontal (x-)angle this principle is the same.

//cos(alpha)=cos(360-alpha): yAngle value is 0;180].
if (***eye***[1] < **this**.**absPosition**[1]) {
yAngle = yAngle + 90;
} **else** {
yAngle = 90 - yAngle;
}	

this.**matrix** = mat4.**multiply**(**this**.**matrix**, ***glm***.rotateY(xAngle), ***glm***.rotateX(yAngle));	Calculate our transformation (=billboard rotation) matrix

super.render(context);
}
	Call the render function (here our matrix is used)

setPosition(x, y, z) {
this.**absPosition** = [x, y, z];
}
	A setter function to set the absolute position in the world.

}	//End of class

User camera movement

When a ‚c‘ on the keyboard is pressed, the user camera mode is enabled. Now the user can define where to go.

Idea

For manipulation the normal camera view, we have to manipulate only the 3 given camera vectors: eye, center and up.

- Because “rolling” is not allowed, the up vector is always [0,1,0]. (done)
- For the remaining vectors we first calculate the direction vector. Once we have done this, we can
  - Set the eye vector: no zoom eye vector does not change. Otherwise, we have to add a multiple (can also be negative if zooming out) of the direction vector to the current eye vector.
  - Set the center vector: eye+direction = center

Computing the direction vector

Given: camera rotation (x: 0 to 360 degrees, y: -90 to 90 degrees), which is calculated by mouse movement of x and y coordinate (taken 1:1).
To Find: direction vector [dirX, dirY, dirZ] with certain fixed length

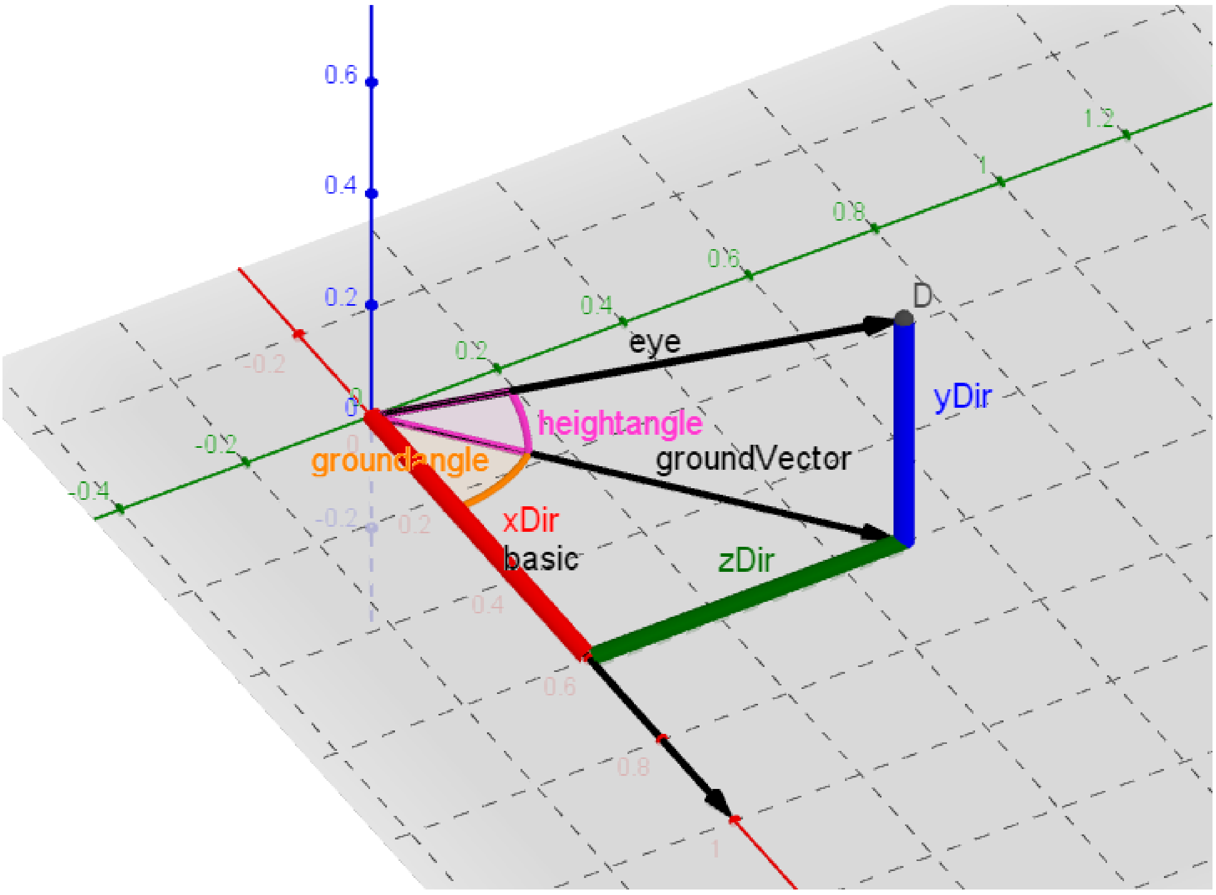


Figure 2: direction vector components (dirX, dirY and dirZ) - GeoGebra screenshot

- dirY (simplest): this vector component is only dependent of the y rotation. Therefore we get this value by taking the sin of the y camera rotation. (sin(0)=0, sin(90°)=1)
- dirX: the more the y rotation differs from 0, the more the dirX gets shorter. When the x roation is 0, then the xDir reaches its maximum. Therefore we get dirX by multiplying cos(x rotation) and cos(y rotation)
- dirZ: the same as dirX, but now the dirZ reaches its maximum when (x rotation+90°) is 0. Therefore we get dirX by multiplying sin(x rotation) and cos(y rotation)

Code

This segment is at the position where the lookAt vectors are calculated for rendering

`if (userCamera) {`	
`    if (jumpToUserCamera) {`	Here the camera rotations are calculated if it is switched from any camera mode to the userCamera mode (in order to have no “jump”)
`        //calculate direction`	
`        let directionOffset = vec3.normalize(vec3.create(),`	
`vec3.subtract(vec3.create(), center, eye));`	
`        camera.rotation.y = -directionOffset[1] * 360 / Math.PI;`	
`        let acosParam = directionOffset[0] / Math.cos(camera.rotation.y`	
`* Math.PI / 360);`	
`        camera.rotation.x = Math.acos(acosParam) * 360 / Math.PI;`	
`        camera.rotation.z = -camera.rotation.x;`	
`        camera.zoom = 0;`	
`        jumpToUserCamera = false;`	
`    }`	
`    //calculate lookat direction`	Calculate the vector components like discribed above
`    var dirX = Math.cos(camera.rotation.x * Math.PI / 360) *`	
`Math.cos(camera.rotation.y * Math.PI / 360);`	
`    var dirY = Math.sin(-camera.rotation.y * Math.PI / 360);`	
`    var dirZ = Math.sin(camera.rotation.z * Math.PI / 360) *`	
`Math.cos(camera.rotation.y * Math.PI / 360);`	
`    //round in order to neglect rounding mistakes`	Sin(Math.PI) should be zero but it is not (rounding mistakes by the processor), therefore we round the values to 9 digits after the comma
`    dirX = Math.round(dirX * 1000000000) / 1000000000;`	
`    dirY = Math.round(dirY * 1000000000) / 1000000000;`	
`    dirZ = Math.round(dirZ * 1000000000) / 1000000000;`	
`    var direction = [dirX, dirY, dirZ];`	Create direction vector
`    //calculate new lookat vectors`	Calculate the new values like discribed above
`    vec3.add(eye, eye, vec3.scale(vec3.create(), direction,`	
`camera.zoom * zoomspeed));`	
`    vec3.add(center, eye, direction);`	
`    up = [0, 1, 0];`	
`}`	

Materials and Phong Shading

All objects in the scene have a material node and are phong shaded.

Idea

Add material nodes to the scene graph. At the shading process, compute the fragment color by using the phong shading function.

Code for fragment shading

vec4 calculateSimplePointLight(Light light, Material material, **vec3** lightVec, **vec3** normalVec, **vec3** eyeVec, **vec4** textureColor) {	
lightVec = normalize(lightVec); normalVec = normalize(normalVec); eyeVec = normalize(eyeVec);	Normalize vectors to length=1
//compute diffuse term **float** diffuse = max(dot(normalVec,lightVec),0.0);	Calculate the diffusion part: the hightest diffusion can be reached if the incoming light is normal to the surface, which means that the dot product will be maximum (=1) if the normal vector of the surface and the incoming light vector have the same direction.
//compute specular term **vec3** reflectVec = reflect(-lightVec,normalVec);	The reflect vector is computed by “mirroring” the incoming light vector at the normal vector.
float spec = pow(max(dot(reflectVec, eyeVec), 0.0) , material.shininess);	The specular part is high if the turned reflection vector and the viewer’s vector have the same direction
// replace diffuse and ambient material color with texture color material.diffuse = textureColor; material.ambient = textureColor;	Set the texture color for the material
vec4 c_amb = clamp(light.ambient * material.ambient, 0.0, 1.0); **vec4** c_diff = clamp(diffuse * light.diffuse * material.diffuse, 0.0, 1.0); **vec4** c_spec = clamp(spec * light.specular * material.specular, 0.0, 1.0); **vec4** c_em = material.emission;	Clamp values (“post-values” are between 0 and 1)
return c_amb + c_diff + c_spec + c_em;	
}	

Spotlight

For the spotlight we only have to manipulate the shader function that computes point lights.

Idea

Normal fragment phong shading for diffusion part: compute angle between the (incoming) light direction vector and the normal vector of the surface. A lower angle results in a higher diffusion value (dot product, and set to 0 if it is <0).

Spotlight: Basically also phong shading, but the diffusion part is calculated in another way:
Here we compute the angle between the incoming light, and the “direction” of the spot light source. In particular, we take the dot product of the incoming light vector and the “light direction” at the source, and set it to 0 if it is negative.
In contrast to the normal phong shading (where the diffusion value is the clamped dot product), we now set the diffusion part to 1 if this value is above a certain threshold, or to 0 if it is below (full or no lighting). Afterwards we let the diffusion part become less if the distance between fragment and spotlight source gets greater.

Code in fragment shader

Normal shading (provided here for comparison)

```
float diffuse = max(dot(normalVec,lightVec),0.0);
```

spotlight shading:

```
float diffuse = max(dot(lightSpotVec,spotDirectionVec),0.0);
diffuse = diffuse < 0.9 ? 0.0 : 1.0; //threshold = 0.9
//diffuse==1 <=> fragment is lighted by spot
  diffuse = diffuse/(lightObjectDistance); //distance
```

Minimap

The minimap in our project is a map that shows the world (still in 3D) from the bird-perspective.
Our minimap is shown on the right top corner with one third of the size of the normal view.
It overlays the normal view, so that in the corner only the minimap is shown and nothing from the nomral view.
Adittionally there is a Path drawn as a red line on the minimap that schows the camera movement of the last 10 seconds.

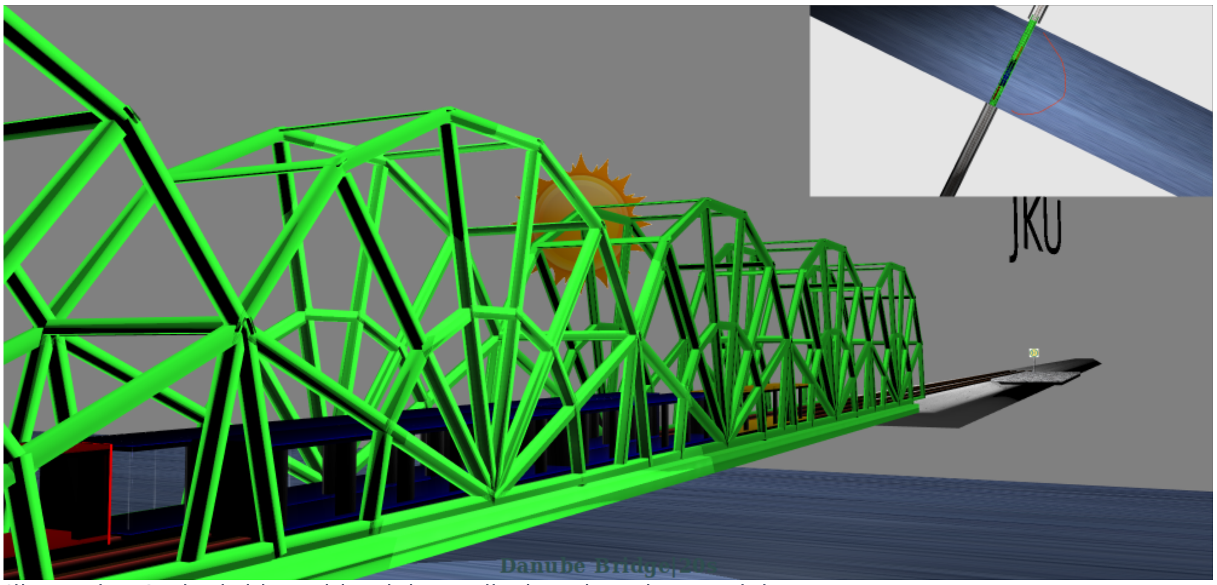


Illustration 1: the bridge with minimap displayed on the top right

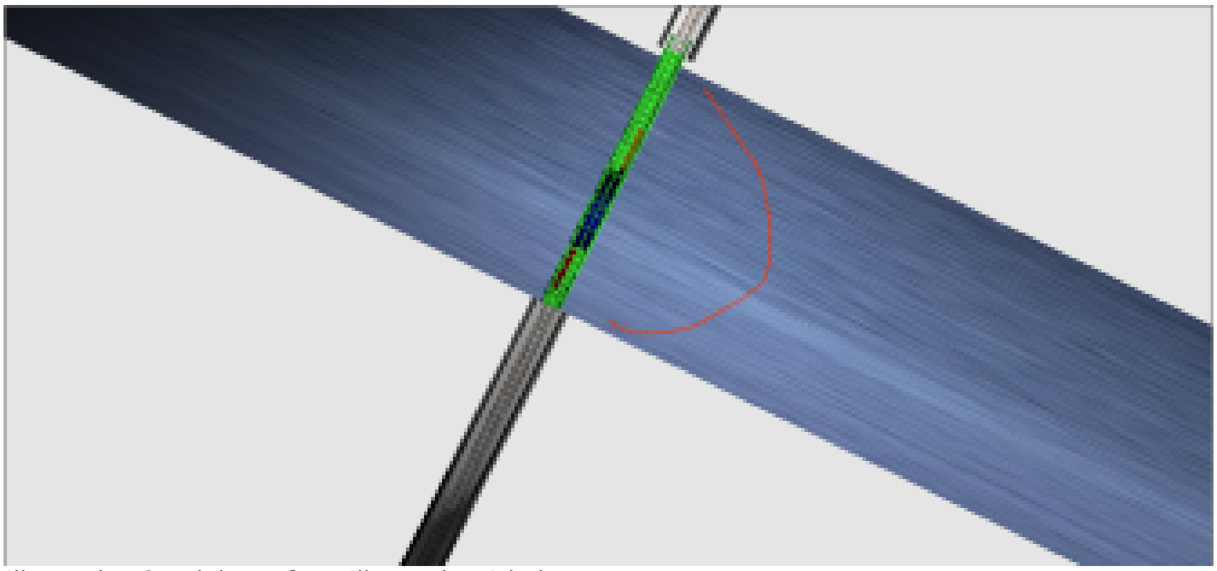
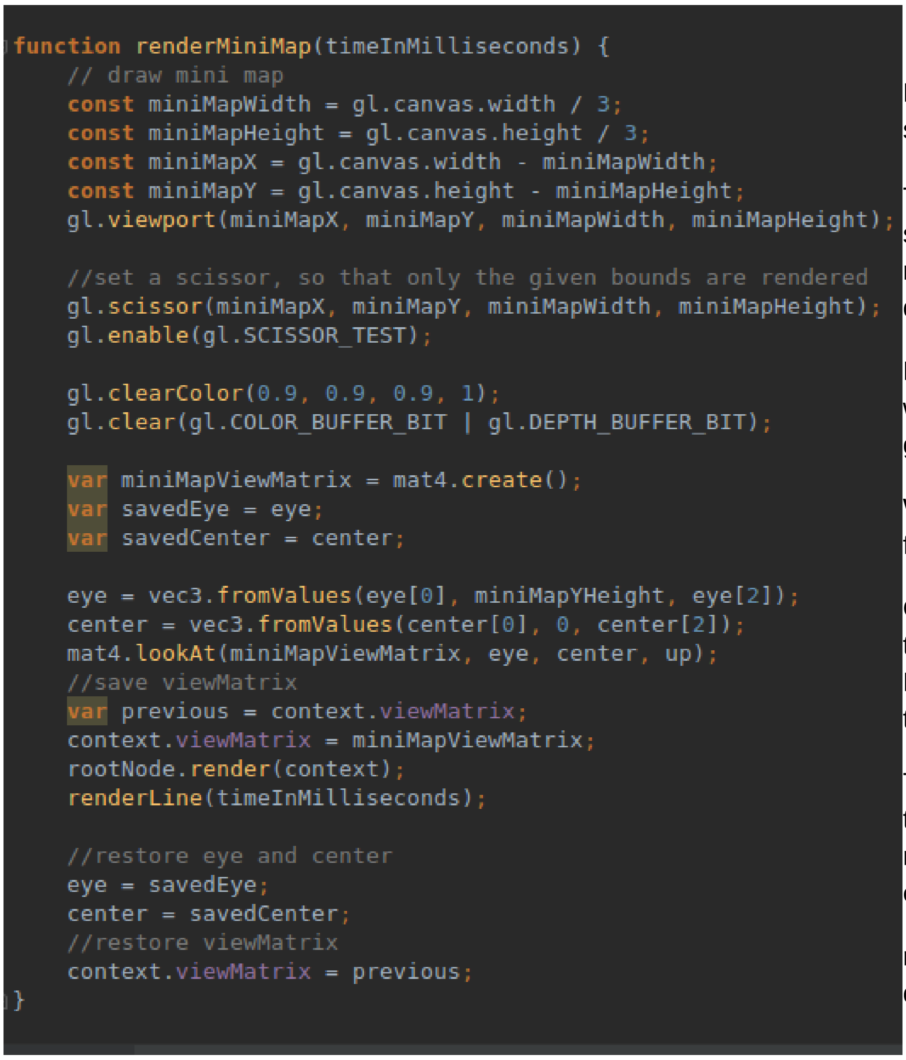


Illustration 2: minimap from Illustration 1 in large

Theory:
The theory behind it is that we want to draw the same world from another perspective into the original view.

Implementation:

The code to draw the minimap is this this one:



```
function renderMiniMap(timeInMilliseconds) {
    // draw mini map
    const miniMapWidth = gl.canvas.width / 3;
    const miniMapHeight = gl.canvas.height / 3;
    const miniMapX = gl.canvas.width - miniMapWidth;
    const miniMapY = gl.canvas.height - miniMapHeight;
    gl.viewport(miniMapX, miniMapY, miniMapWidth, miniMapHeight);

    //set a scissor, so that only the given bounds are rendered
    gl.scissor(miniMapX, miniMapY, miniMapWidth, miniMapHeight);
    gl.enable(gl.SCISSOR_TEST);

    gl.clearColor(0.9, 0.9, 0.9, 1);
    gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);

    var miniMapViewMatrix = mat4.create();
    var savedEye = eye;
    var savedCenter = center;

    eye = vec3.fromValues(eye[0], miniMapYHeight, eye[2]);
    center = vec3.fromValues(center[0], 0, center[2]);
    mat4.lookAt(miniMapViewMatrix, eye, center, up);
    //save viewMatrix
    var previous = context.viewMatrix;
    context.viewMatrix = miniMapViewMatrix;
    rootNode.render(context);
    renderLine(timeInMilliseconds);

    //restore eye and center
    eye = savedEye;
    center = savedCenter;
    //restore viewMatrix
    context.viewMatrix = previous;
}
```

First we change the viewport to the size and position we want it to be.

Then we use gl.scissor() with the same parameter to tell WebGl that nothing beond this bounds should be drawn.

Now we can execute gl.clearColor() which only clears the color for the given bounds.

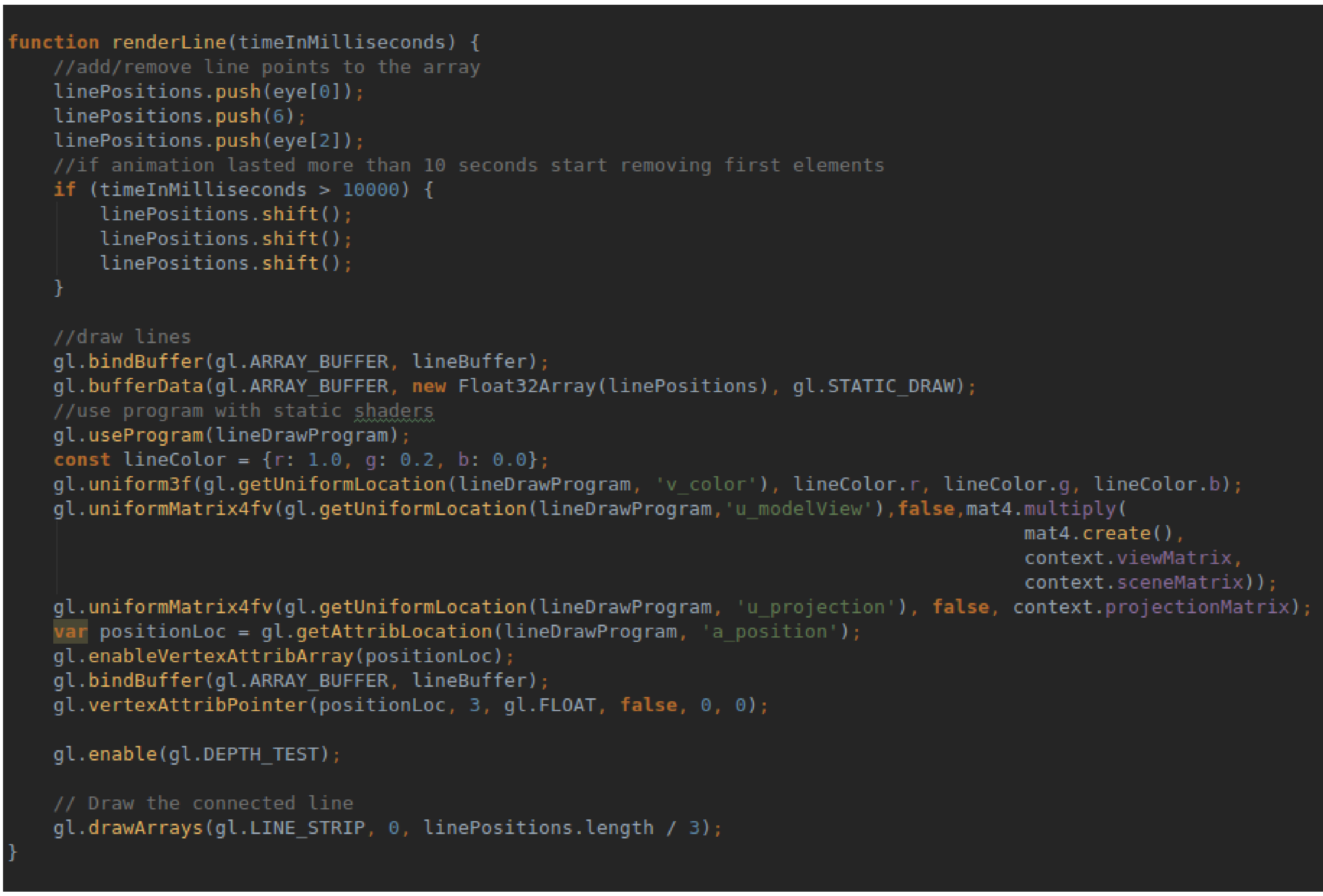
We save the eye and center vectors from the normal view.

Only the y-coordinate (in our world the heigth) is changed. MiniMapYHeight specifies how small the world is drawn.

The next steps are the same as for the normal view (calculating the view matrix of the context with the given eye, center and up)

renderLine(timeInMilliseconds) is described below

Note: eye and center are global variables and are used for both views. With this approach our billboard not only look into the direction of the normal view but also into the direction of the minimapview.



```javascript
function renderLine(timeInMilliseconds) {
    //add/remove line points to the array
    linePositions.push(eye[0]);
    linePositions.push(6);
    linePositions.push(eye[2]);
    //if animation lasted more than 10 seconds start removing first elements
    if (timeInMilliseconds > 10000) {
        linePositions.shift();
        linePositions.shift();
        linePositions.shift();
    }

    //draw lines
    gl.bindBuffer(gl.ARRAY_BUFFER, lineBuffer);
    gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(linePositions), gl.STATIC_DRAW);
    //use program with static shaders
    gl.useProgram(lineDrawProgram);
    const lineColor = {r: 1.0, g: 0.2, b: 0.0};
    gl.uniform3f(gl.getUniformLocation(lineDrawProgram, 'v_color'), lineColor.r, lineColor.g, lineColor.b);
    gl.uniformMatrix4fv(gl.getUniformLocation(lineDrawProgram,'u_modelView'),false,mat4.multiply(
                                                                                      mat4.create(),
                                                                                      context.viewMatrix,
                                                                                      context.sceneMatrix));
    gl.uniformMatrix4fv(gl.getUniformLocation(lineDrawProgram, 'u_projection'), false, context.projectionMatrix);
    var positionLoc = gl.getAttribLocation(lineDrawProgram, 'a_position');
    gl.enableVertexAttribArray(positionLoc);
    gl.bindBuffer(gl.ARRAY_BUFFER, lineBuffer);
    gl.vertexAttribPointer(positionLoc, 3, gl.FLOAT, false, 0, 0);

    gl.enable(gl.DEPTH_TEST);

    // Draw the connected line
    gl.drawArrays(gl.LINE_STRIP, 0, linePositions.length / 3);
}
```

The renderLine function draws the Path of the last 10 second of the camera.
Therefor it pushes the location of the camera into an array every time it is rendered.
Then after 10 seconds of the total time we start removing first insterted position from the array, so that only the last 10 seconds are shown.

Then we need to bind the Buffers every time it is rendered.
Normaly this it down once at initialisation but because we don't have fixed point, but an expanding array this has to be done every time.

Then we change the program to lineDrawProgram which uses static color shader for drawing.
The color of the fragments is set to red. The fragments are in our case lines.
Then we set the Position of the vertices according to the views.

To draw it we need to bind the buffer and point to the position of the vertices.
Now we can finally draw the Line from the linePositions that are bound in the ARRAY_BUFFER.
Here we need to specifie the starting and end of the lines.
We start form 0 till the last point which is linePosition.length/3, because each point has a x, y, z parameter obviously.

Transformations

To be able to do Time-Based Transformation in the Movie, we introduced the Class MovingNode and MovingPoint.
MovingNode is only a wraper of MovingPoint that extends SceneGraphNode. So we can append other Nodes to this node and vice versa.
MovingPoint implements the Movement.
With this Point it is posible to set a movement with setSpeed() that goes on till you stop it manually.
Or with moveTo(), which specifies the position we want to go and when we want to be at that position.
So moveTo() implements the key-frame technique with linearly interpolating between the current position and the position we want the point to move.

```
/**
 * Move to global position within the given amount of time (in milliseconds)
 */
moveTo(position, timeToGetThereInMilliseconds) {
   var difference = vec3.subtract(vec3.create(), position, this.getPosition());
   var speed= vec3.scale(vec3.create(),difference,1/
(timeToGetThereInMilliseconds*slowDownFactor));
   this.timeToStopMoving = projectTimeInMilliSeconds + timeToGetThereInMilliseconds;
   this.isMovingToSpecificPosition = true;
   this.setSpeed(speed);
}
```

Camera Animation

To animated camera flight is based on the three scenes. Everytime a scene ends/starts the camera position changes.

```
var eyePoint;
var centerPoint;
function calculateViewMatrix() {
    switch (sceneIndex) {
        case 1:
            eyePoint.setPosition([8, 1.8, 1.3]);
            centerPoint.setPosition([6.7, 1.6, 1.05]);
            break;
        case 2:
            eyePoint.moveTo(vec3.add(vec3.create(), tram.getPosition(), [2, 0.1, 0.05]),
2000);
            centerPoint.moveTo(vec3.add(vec3.create(), tram.getPosition(), [3, 0.1, 0.05]),
2000);
            break;
        case 3:
            eyePoint.setPosition([40, 2, 0]);
            centerPoint.setPosition([39, 1.9, 0]);
            break;
    }

    ....
```

In the first scene the camera statically looks at the Main Station with eye and center statically defined. Then the eye and the center are moving closer and closer to a point that is in the tram.

Note: Because we render more often that every 2000 milliseconds the moveTo is always overwritten with the always chaning position of the tram. So we converge to the position but never get to actual position we put in here as parameter.

What we now set where only points but not the actual eye and center, so we set those if the animated flight is activated.

Note: we calling setCenterPosition sets the centers as normalized direction. This is needed so that the jump to the user Camer works

```
//compute the camera's matrix
viewMatrix = mat4.create();
if (userCamera) {
    ...
}
else if (tramFrontCamera) {
    ...
}
else {
    //direction have to be normalized so that the jump to the user camera works
    eye = eyePoint.getPosition();
    setCenterPosition(centerPoint.getPosition());
    up = [0, 1, 0];
    jumpToUserCamera = true;
}
viewMatrix = mat4.lookAt(viewMatrix, eye, center, up);
return viewMatrix;
```

Triggering Animation

Our triggering Animation consist of the Person Objects.
These Person Objects are only rendered iff the eye is in a certain radius.

```
function getDistance(position) {
    let vector = vec3.sub(vec3.create(), eye, position);
    return Math.sqrt(vector[0] * vector[0] + vector[1] * vector[1] + vector[2] * vector[2]);
}
```

```
function setRenderPerson() {
    persons.forEach(function(person) {
        if(getDistance(person.getPosition()) > 10) {
            rootNode.remove(person);
        }
        else {
            let inList = false;
            rootNode.children.forEach(function (child) {
               if(child == person) inList = true;
            });
            if(!inList) {
                //remove tram to see persons through windows
                rootNode.remove(tramNode);
                rootNode.append(person);
                //append tram again
                rootNode.append(tramNode);
            }
        }
    });
}
```