Technical University of Denmark

02561 Computer Graphics

Interactive 3D computer graphics

December 18, 2022

Period:	Autumn 2022

AUTHORS:

Eliise Kaha – s221889

Peter Russel – s221550

Contents

1 Introduction

This project's main goal is to design an interactive 3D arm that can grab another object and move it around. The first step to complete this project will be using blender to craft the arm. This will involve making a texture map for the hand as well as creating different frames to use for animation. These models will then be used to construct a scene within WebGL of the arm and another object. The arm will be controlled by the mouse movements and the ability to grab will be controlled using a mouse click. At the same time, the texture coordinates from the obj file will be used to map a texture to the hand object.

2 Methods

The first step of our project is to use Blender to model the hand. For animation, bones have to be added to the modeled hand. After that, all kind of different animations are possible, in our project we have decided to use a grabbing animation to move around the object.

As WebGL will be used to execute the main part of this project, the Blender modeled arm has to be read into WebGL files. For this reason we will export blender files as wavefront.obj and read them in using OBJParser.js . After that the hand can be rendered. To add texture on to the hand OBJParser.js has to be modified to additionally read in texture coordinates. With texture, Phong shading will be used. This will be with a set light source. Texture and Phong shading will be used to give our hand a realistic look.

To animate the hand in the browser we planned to use a series of obj files loaded into different buffers and then cycle through those different buffers whenever the animation is triggered. The plan is to have one model bound to the attribute variables in the shaders. Then create multiple obj_doc variables with different objects loaded into them. Then in the render function depending on whether the mouse is up or down choose which obj to use.

Hand-moving animation will be done by making a function that tracks mouse movements on the canvas and then uses the mouse coordinates to create a transformation that is sent to the shader before the hand is drawn and the transformation is changed before the second object is drawn depending on if it is being picked up or not. Since the object is rendered at (0.0, 0.0, 0.0) We can just add the mouseX position times the radius to the X coordinate of the hand and the mouseY position times the radius to the Y coordinate of the hand. Then we will determine if the other object is being picked up and apply the corresponding translation.

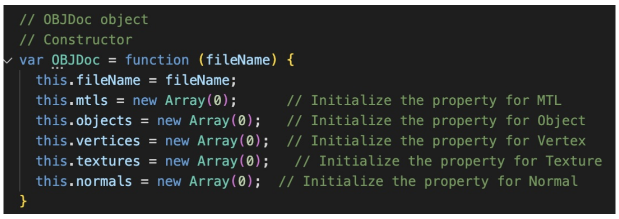
And with mouse input, we are chasing the position of the hand not translating the camera position, because while the hand is moving the movable object should be motionless. Unless it has been grabbed, in this case, it will move with the hand.

To grab the other object we need another function that registers mouse click. This function will make the object move with the hand.

3 Implementatsion

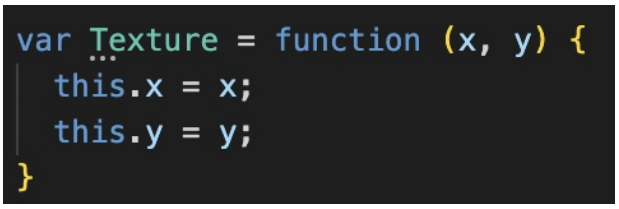
Before we could start working on our project in WebGL we had to have an object, which in this project will be a hand. We acquired it by using Blender. We started from Blender templett model with a cube. First we merged all the vertices to the center point, to have single starting vertex. Then we added a skin modifier and a subdivision surface modifier in this specific order, as the subdivision surface modifier cannot take effect unless there already exists skin. With creating the arm itself we stared with constructing the forearm, from that base we then made five digits. Which we modified until we accomplished our desired shape for the hand. After we applied our skin and subdivision surface modifier, which created a complete mesh. As we wanted our hand to look a bit more realistic we added another subdivision surface modifier to increase the number of polygons(shapes that are responsible for creating the mesh, in our case triangles).

To texture the hand we modified OBJParser.js (3.1) by adding in texture coordinate retrieval. First in OBJDoc constructor we initialized a new property for texture as an array (3.1a). Next we added a case 'vt' into OBJDoc.prototype.parse to read texture (3.1c). To seperate texture coordinates from one another we created OBJDoc.prototype.parseTexture (3.1d), which returns texture objects coordinates x and y. In OBJDoc.prototype.parseFace now in addition to previously existing: face normal indices and face vertex indices, exists face texture indices (3.1e). While dividing face into triangles we make new texture indices (3.1f), to correct the placement of texture coordinates. All previous changes done we wanted drawinfo in onReadComplete to additionally return texture with vertices, normals, colors, hence we created an texture object (3.1b) and copied texture info into an array and listed it in drawinfo object (3.1g ,3.1h).



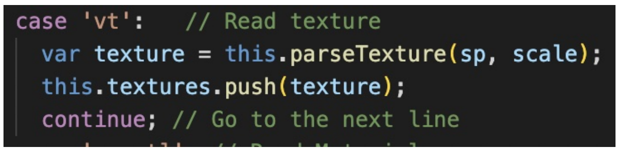
```
// OBJDoc object
// Constructor
var OBJDoc = function (fileName) {
  this.fileName = fileName;
  this.mtls = new Array(0);      // Initialize the property for MTL
  this.objects = new Array(0);   // Initialize the property for Object
  this.vertices = new Array(0);  // Initialize the property for Vertex
  this.textures = new Array(0);   // Initialize the property for Texture
  this.normals = new Array(0);  // Initialize the property for Normal
}
```

(a) Texture array



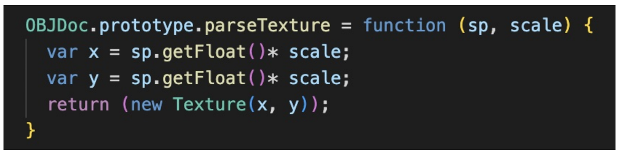
```
var Texture = function (x, y) {
  this.x = x;
  this.y = y;
}
```

(b) Texture object



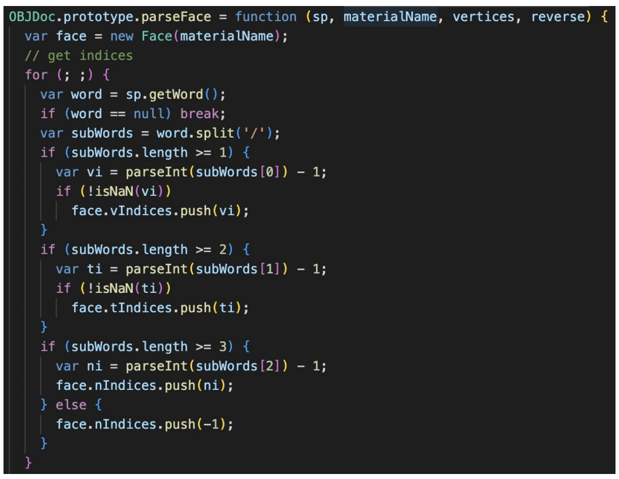
```
case 'vt':   // Read texture
  var texture = this.parseTexture(sp, scale);
  this.textures.push(texture);
  continue; // Go to the next line
```

(c) Case 'vt'



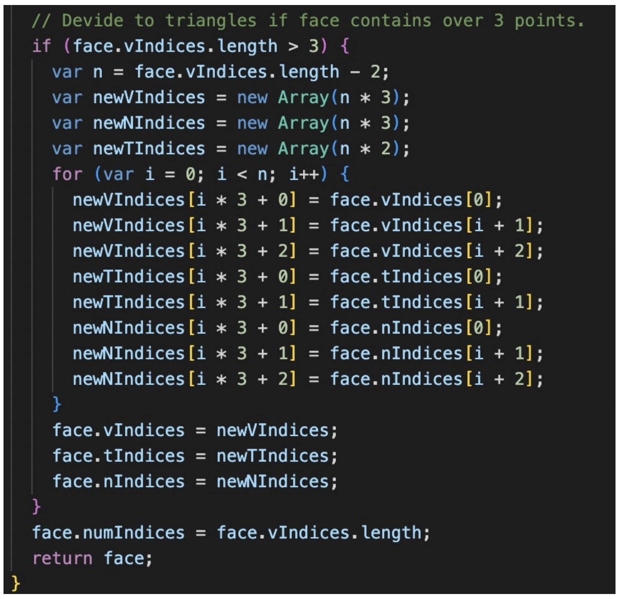
```
OBJDoc.prototype.parseTexture = function (sp, scale) {
  var x = sp.getFloat()* scale;
  var y = sp.getFloat()* scale;
  return (new Texture(x, y));
}
```

(d) OBJDoc.prototype.parseTexture



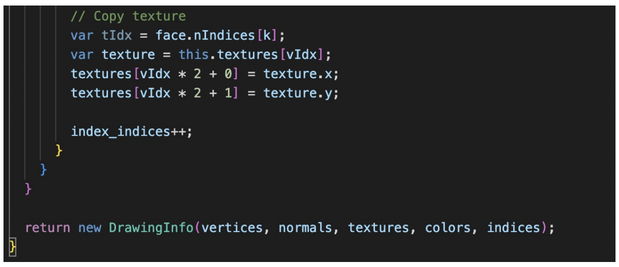
```
OBJDoc.prototype.parseFace = function (sp, materialName, vertices, reverse) {
  var face = new Face(materialName);
  // get indices
  for (; ;) {
    var word = sp.getWord();
    if (word == null) break;
    var subWords = word.split('/');
    if (subWords.length >= 1) {
      var vi = parseInt(subWords[0]) - 1;
      if (!isNaN(vi))
        face.vIndices.push(vi);
    }
    if (subWords.length >= 2) {
      var ti = parseInt(subWords[1]) - 1;
      if (!isNaN(ti))
        face.tIndices.push(ti);
    }
    if (subWords.length >= 3) {
      var ni = parseInt(subWords[2]) - 1;
      face.nIndices.push(ni);
    } else {
      face.nIndices.push(-1);
    }
  }
```

(e) OBJDoc.prototype.parseFace



```
  // Devide to triangles if face contains over 3 points.
  if (face.vIndices.length > 3) {
    var n = face.vIndices.length - 2;
    var newVIndices = new Array(n * 3);
    var newNIndices = new Array(n * 3);
    var newTIndices = new Array(n * 2);
    for (var i = 0; i < n; i++) {
      newVIndices[i * 3 + 0] = face.vIndices[0];
      newVIndices[i * 3 + 1] = face.vIndices[i + 1];
      newVIndices[i * 3 + 2] = face.vIndices[i + 2];
      newTIndices[i * 3 + 0] = face.tIndices[0];
      newTIndices[i * 3 + 1] = face.tIndices[i + 1];
      newNIndices[i * 3 + 0] = face.nIndices[0];
      newNIndices[i * 3 + 1] = face.nIndices[i + 1];
      newNIndices[i * 3 + 2] = face.nIndices[i + 2];
    }
    face.vIndices = newVIndices;
    face.tIndices = newTIndices;
    face.nIndices = newNIndices;
  }
  face.numIndices = face.vIndices.length;
  return face;
}
```

(f) Dividing face into triangles

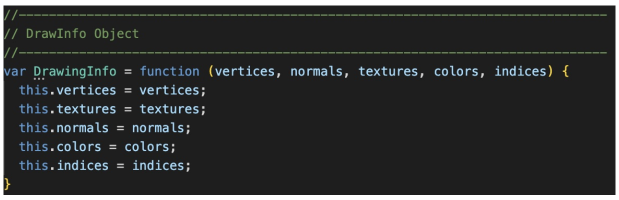


```
        // Copy texture
        var tIdx = face.nIndices[k];
        var texture = this.textures[vIdx];
        textures[vIdx * 2 + 0] = texture.x;
        textures[vIdx * 2 + 1] = texture.y;

        index_indices++;
      }
    }
  }

  return new DrawingInfo(vertices, normals, textures, colors, indices);
}
```

(g) Drawing info addition



```
//------------------------------------------------------------------------------
// DrawInfo Object
//------------------------------------------------------------------------------
var DrawingInfo = function (vertices, normals, textures, colors, indices) {
  this.vertices = vertices;
  this.textures = textures;
  this.normals = normals;
  this.colors = colors;
  this.indices = indices;
}
```

(h) Drawing info object

Figure 3.1: Code changes in OBJParser.js

Texure coordinates retrived by OBJParser.js are parametric coordinates u and v mapped in blender. We influence the color of a fragment with two-dimensional texture mapping using a picture of earth. For texture sampeling we used a mipmap, spesifically we used point sampling with the best mipmap (gl.NEAREST_MIPMAP_NEAREST). For the color we used Phong shading(with white light), which we crated ourselves in HTML. Phong shading is the hardest form of shading requiring the lighting model be applied to each fragment, however it gives a most realistic look and that was what we amid for.

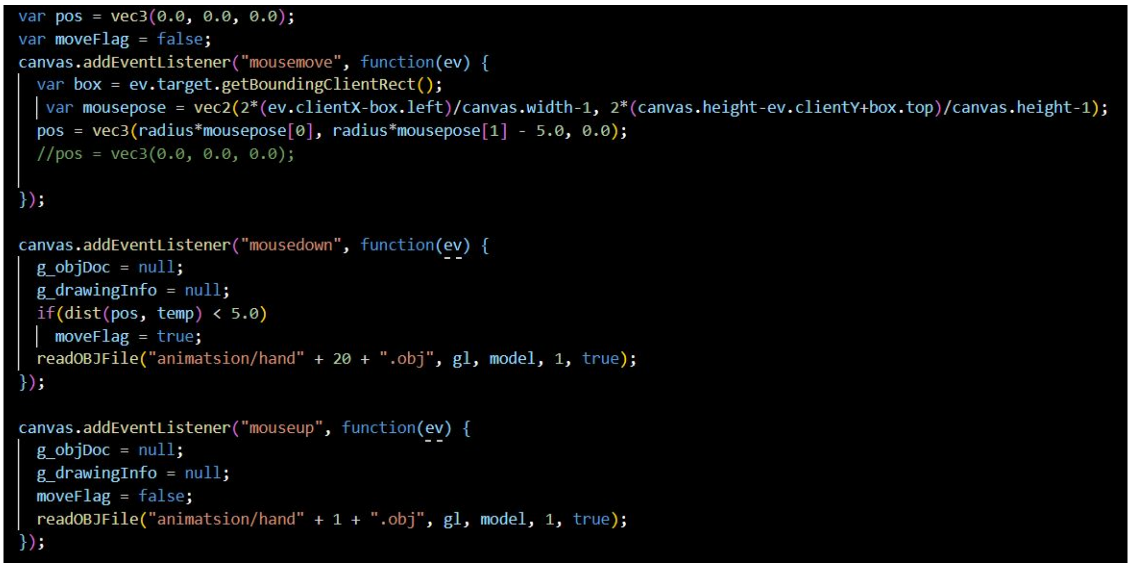
To use the arm for grabbing objects we had to use Blenders animation features. First we added an amateur to our model to enable pose mode, which allows users to animate their models. Then we created a structure of bones (imitating bones inside a real hand) inside the hand model. For easier manipulation of the fingers we connected the bones using copy rotation to link together parent and child bone, parent being a bone at the base of the finger, and children, being the bones on top of it. After all of this prep-work we could start animating. Grabbing motion is an x-axis based motion, meaning we could rotate the hand only on x-axis leaving other rotations constant. We animated the hand with 20 frames, adding checkpoints at every five for the animation to have our desired look. On the 20th frame the modeled hand had been rotated over x-axis enough to give it the appearance of holding something. We exported the animation in wavefront.obj, as it was something we had worked with before.

At first, we tried to use the method from our methods section and load multiple objects into different object files how this is difficult as we tried to change the onReadOBJFile function to return an obj_doc but since it is Asynchronous that is not really feasible. In the end, to animate the hand we ended up using only two frames to create the illusion of animation. We chose to have the start and end of the animation as those two frames. To do this we added listeners to mouseup and mousedown and then in those listeners we emptied g_objdoc and g_drawingInfo and called readOBJFile on the frame, open hand for mouseup and closed hand for mousedown.

For our hand to move with the mouse we created an event listener that reacts on mouse event("mousemove"). We are using clip coordinates for our applications, which range from (-1, 1). The function calculates mouse position using clientX and clietY, which give the position of the mouse in window coordinates, for this position to be useful in our application, we must transform these values to the same units as the application, hence we need canvas width and canvas height, as positions are measured in pixels with the origin at the upperleft. With these transformed coordinates we can multiply by the radius and then our hand will move at the same rate as the mouse horizontally and vertically across the canvas. There was one more problem we had to fix in that in that the mouse isnt really centered on the hand on the Y-axis. The simple fix was to add an offset to the Y mouse position[1].

There were a lot of issues when trying to add the second object. And as a result, we ended up using a second draw call to add a second hand to the canvas. We also use a second translation matrix which is discussed in the next paragraph. Our initial attempts to solve this problem are discussed in the results section.

To be able to grab the second arm we created another event listener, which also reacts to mouse event("mousedown"), with that we could pick up the object. but we also needed to be able to drop it. To do this we added event listeners to both mouseup and mousedown. the first thing that needed to happen was the hand needed to change to do this we clear g_drawingInfo and g_objDoc and then call readOBJFile on the file depending on which event is triggered. Opening the hand when the mouseup happens and closing the hand when mousedown happens. Next we had to be able to grab the other hand to do this we first wanted to determine if the mouse is over the second hand. This is done by finding the size of the hypothetical translation on the second hand when the mouse is pressed. If it is small enough then the translation is applied and the hand will continue to follow the mouse until the mouse is released. In order to not always move the second hand we will sometimes change the translation before that hand is sent to the shaders by the render function.



```
var pos = vec3(0.0, 0.0, 0.0);
var moveFlag = false;
canvas.addEventListener("mousemove", function(ev) {
  var box = ev.target.getBoundingClientRect();
   var mousepose = vec2(2*(ev.clientX-box.left)/canvas.width-1, 2*(canvas.height-ev.clientY+box.top)/canvas.height-1);
  pos = vec3(radius*mousepose[0], radius*mousepose[1] - 5.0, 0.0);
  //pos = vec3(0.0, 0.0, 0.0);

});

canvas.addEventListener("mousedown", function(ev) {
  g_objDoc = null;
  g_drawingInfo = null;
  if(dist(pos, temp) < 5.0)
    moveFlag = true;
  readOBJFile("animatsion/hand" + 20 + ".obj", gl, model, 1, true);
});

canvas.addEventListener("mouseup", function(ev) {
  g_objDoc = null;
  g_drawingInfo = null;
  moveFlag = false;
  readOBJFile("animatsion/hand" + 1 + ".obj", gl, model, 1, true);
});
```

Figure 3.2: Listeners

4 Results

The animation was given out in obj form with all 20 frames and looked good, but we soon realized this is not the way that animatsions are usually done in WebGL, so we ended up using only two of the frames first and the last. Which works as an animation but is not what we originally thought to do. This is due to the fact that it is expensive and unstable to load so many models into the canvas.

In this projects method part we describe using texture to give our hand a realistic look. This did no end up happening as we realised that not every picture is suitable to be used as a texture in WebGL. So in the end we used a texture image we had tested before (earth.jpg). It worked out and ended up looking like a tie-dye glove.

Initially, we had the translation being done by moving the camera however this only works when you care about one rendered object and it has the added problem of changing the angle at which you are looking which looks odd. This was switched to using a translation matrix which solves all of those problems.

This section is about what we attempted when trying to add the second object to the canvas. The main issues were that the readOBJFile is asynchronous and also creates buffers different from how they are done with the code for the spheres in part 7 that we were modeling after. As well as the fact that the buffers were not able to be resized after the obj file was read. We also tried to load a second obj file into the buffer however the same issue arose from not having a correctly sized buffer. In the end in order to test the movement and grabbing functionality we ended up redrawing a hand(second hand).

5 Discussion

Blender is a very versatile and visual application we could have definitely taken more advantage of it. We could have played around with the axis in order to have the hand in position as soon as the obj file has been read in. Alternatively we rotated the hand in java script after being read in. We could have set the light there and used it For Phong shading. That would have meant using the light components in hand.mlt file, instead we added the ambient light, diffuse, spectacular light, emission, and shininess manually into the HTML.

For animation, we could have used the full animation, which would have meant not downloading the animation as obj, probably in a video format then, and playing the video animation in between the two obj files. Another option could have been to use off-screen rendering to render all the frames out of view of the camera and then have them change places with the current hand whenever the mouse is pressed.

To improve our project a hand texture that would work with WebGL should be found. A regular picture from google will not work for this purpose, a better chance of finding a suitable image for this purpose will probably be from some website that specifically gives out texture images.

The single biggest improvement to our project would be to add different objects to the canvas. This would likely involve using a separate WebGL library like threejs to have helper functions to make handling multiple objects and their movements easier.

The idea behind our extensions to the project was that by adding interaction to the canvas you can create something between a game and basic WebGL. A large number of the problems we faced as well as our implementations of certain features are solved by pre-existing libraries, which does make a lot of the work that we did unnecessary in the future.

Contributsions

Both worked on almost every part.

Blender modelling and animation, OBJParser modificatsions and texturing were mainly done by Eliise Kaha(S221889).

WebGL animations, event listeners and rendering multible objects were mainly done by Peter Russel(S221550).

Report was written equally by both of us, each mostly condributed on writing about the parts they implemented.

References

[1] “Interactive computer graphics. a top-down approach with webgl-pearson.” "Edward Angel, Dave Shreiner-Interactive Computer Graphics. A Top-Down Approach with WebGL-Pearson (2014).pdf". "2014(accessed September, 2022)".