Assignment 3 – 3D Graphics using Three.js

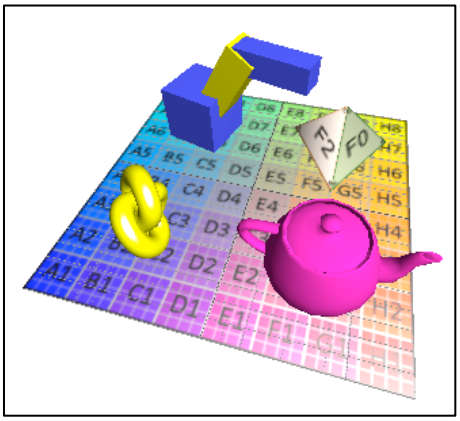
Abstract: Creation of a 3D scene in Three.js composed of objects created using a variety of approaches for the definition of their geometry, their shading and texture.

Guidelines for assignment hand in:
- Upload one ZIP file on Moodle, with extension .zip (please, no RAR or 7zip files)
- This ZIP file contains all code and data necessary to run your assignment. It is organized as follows
- Program should be using the 2D HTML5 canvas with Javascript.
- File organization (filenames, flat or hierarchical subfolders…) is up to you, but should be clear and meaningful.
- Make sure the example run by simply opening the HTML file, without having to modify the code. If any additional action is required to run the results, this must be explained in a clear way directly when opening the HTML page in the browser, with additional details in a README.txt if needed (for instance, if it needs to run through a webserver instead of directly from the file).

Objectives: Be able to…
- Create and render a simple 3D scene using ready-made geometry
- Change the object material, its shading parameters, and apply textures
- Load 3D models from file
- Create more complex scenes using scenegraph modeling, and add interactivity
- Create a mesh data structure from scratch and texture it by defining its UV-mapping

Baseline: you can use the code `cube_base.html` as base code to program your own scene.
This code provides: scene with a horizontal plane, a point light and rendered through a camera looking at the center of the scene located in (0,0,0). The position of the camera can be changed using the mouse, by using OrbitControls.

The final scene may look like the figure, and is composed of 5 objects on top of a texture plane:
- Object 1: a basic mesh that can change material
- Object 2: a textured plane
- Object 3: a ready-made 3D model loaded from a file
- Object 4: an articulated robot created using the scenegraph
- Object 5: a textured tetrahedron created from scratch.



Q0. Documentation [10]
- Your code should be well organized (pay attention to indenting) and commented (include comments that specify any non obvious convention or intent).
- The HTML page should provide below the canvas a short but clear explanation for the user to know how to access all the functionalities you developed.

OBJECT 1: Mesh and materials

Q1. Mesh Object [10] Create a scene with a curved geometric objects (e.g TorusKnot or Torus…) sitting on top of horizontal plane. Use a standard MeshLambertMaterial with uniform color.
Hints: Example of steps:
- create the geometry using ready made geometry such as

```
geometry = new THREE.TorusKnotGeometry(100, 20, 64, 16)
```

- to create a blue Lambertian material using:

```
material = new THREE.MeshLambertMaterial(
    { color: 0x0000ff, side: THREE.DoubleSide } )
```

- create the mesh by combining the two, and add it to the scene

```
mesh = new THREE.Mesh(geometry, material);
scene.add(mesh)
```

- scale, translate and/or rotate your mesh to put it where you want in the scene

```
mesh.position.set(px,py,pz) // tx,ty,tz is up to you
mesh.scale.set(sx,sy,sz)    // sx,sy,sz is up to you
mesh.rotation.set(rx,ry,rz) // rx,ry,rz is up to you
```

Q2. Materials [10] Let the user change the material of the mesh in Q1 by pressing specific keys:
- on 'L', use the Lambertian material of Q1
- on 'P', use a Phong Material with while specular reflections (shiny)
- on 'F', use a Phong material, but with flat shading, which renders the faces of the geometry as flat polygons instead of trying to smooth the appearance, unlike the default Phong Material
- on 'W', only draw the wireframe of the mesh

Make sure your HTML page provides explanation of the key shortcuts as reference for the user to use your system. Ensure also that your scene contains a point light in a position such that the difference in material is visible from the camera.
Hints:
For instance, when user presses 'P', replace the old material with the new one, and render again:

```
mesh.material = THREE.MeshPhongMaterial(
     { color: 0x00ff00, specular: 0xffffff, shininess: 10,
       side: THREE.DoubleSide } )
render()
```

Flat shading is obtained with parameter `shading: THREE.FlatShading`. Phong default is to use `THREE.SmoothShading`. Wireframe can be obtained using the following material:

```
THREE.MeshBasicMaterial({ color: 0xff0000, wireframe: true })
```

Read Three.js documentation for precise use of each material parameter: https://threejs.org/docs

The lights are already provided in the template code. This code is reproduced here for completeness:

```
var ambientLight = new THREE.AmbientLight(0x303030);
scene.add(ambientLight)
var pointLight = new THREE.PointLight(0xffffff)
pointLight.position.set(150, 250, 125)
scene.add(pointLight)
```

These lights are not visible in the scene. For this reason, a sphere is also added in the template code, that you can see when rotating around the scene.

OBJECT 2: Textured plane

Q3. Loading and applying a texture [10]
Change the material on the horizontal plane to be textured using an image.
Hint:
THREE provides a way to attach the object "texture" to a material without waiting for the image loading, as follows:

```
textureLoader = new THREE.TextureLoader()
textureLoader.crossOrigin = '';
texture = textureLoader.load('moretextures/uv-grid.png')
plane.material = new THREE.MeshPhongMaterial(
            { map: texture, side: THREE.DoubleSide,
              specular: '#ffffff', shininess: 10 } )
```

Note on asynchronous loading: Be conscious, that in the same way as when loading images for 2D canvas, the texture is usable only after is has been loaded. Some examples in Three.js documentation use a second parameter of the load function to define an asynchronous callback that receives the texture object and assign the material only at that moment. In the proposed approach, the texture can be used before that, but it will remain completely black until the image has been received. Next time the scene is rendered, the loaded image will then be used as the texture. Try to move the scene with the mouse to force rendering if your scene shows a black texture.

Note on cross-origin policy: On Chrome, if the texture is not loaded due to an error such as "Access to Image at 'file:///…' from origin 'null' has been blocked by CORS policy", you need to serve your HTML page through a webserver. A simple way to do it is using python:

1. Open a terminal and set current directory to where your code is
2. For Python 2, type: `python -m SimpleHTTPServer`
   For Python 3, type: `python3 -m http.server`
   Python should answer: `Serving HTTP on 0.0.0.0 port 8000 ...`
3. Use your web browser to access your page at `http://localhost:8000`

Another workaround is to disable cross-origin policies in Chrome, but is *not advised* due to security issues. See: http://stackoverflow.com/questions/3102819/disable-same-origin-policy-in-chrome

Note on Firefox bug: In Firefox, you can ignore the error "THREE.WebGLShader: gl.getShaderInfoLog() vertex WARNING: 0:1: extension 'GL_ARB_gpu_shader5' is not supported" that prints a lot of code on the console.

OBJECT 2: 3D Model from file

Q4. Loading ready-made 3D models [10] Load a 3D model from a file and display it with the other objects on top of the plane. Adjust the position, scale and rotation of the model so that it stands on the plane, has dimensions similar to the other objects, and is rotated appropriately.
Hint: The following code loads `teapot-claraio.json`, scale it and add it to the scene. It uses THREE.ObjectLoader to load the object stored in .json file.

```
var loader = new THREE.ObjectLoader();
loader.load(
    "models/json/teapot-claraio.json",
    // pass the loaded data to the onLoad function.
    // Here it is assumed to be an object
    function (object) {
        //add the loaded object to the scene
        scene.add(object);
         // Be careful to adjust position/scale/rotation
        obj.scale.set(100,100,100);
        // To make sure all model is visible
        obj.material.side=THREE.DoubleSide
    }
);
```

Note on cross-origin policy: the same comments on cross-origin policy applies as in Q3.
Warning on the multiplicity of 3d model formats: The provided code works only if the model is a Three.js compatible JSON model. Other models may be stored in various format (some of them also called .json or js), that would need to be loaded with a different loader: THREE.JSONLoader, THREE.OBJLoader, etc... that depends on the actual format of the model file. For instance, to load the other provided model male02, you can use

```
var loader = new THREE.JSONLoader();
loader.load(
    "models/obj/male02/Male02_slim.js",
    function ( geometry, materials ) {
        var material = new THREE.MeshMultimaterials( materials );
        var object = new THREE.Mesh( geometry, material );
        scene.add(object);
    }
);
```

to load the JSON object, or

```
var loader = new THREE.OBJLoader();
loader.load( 'models/obj/male02/male02.obj', function ( object ) {
        scene.add( object );
    }
)
```

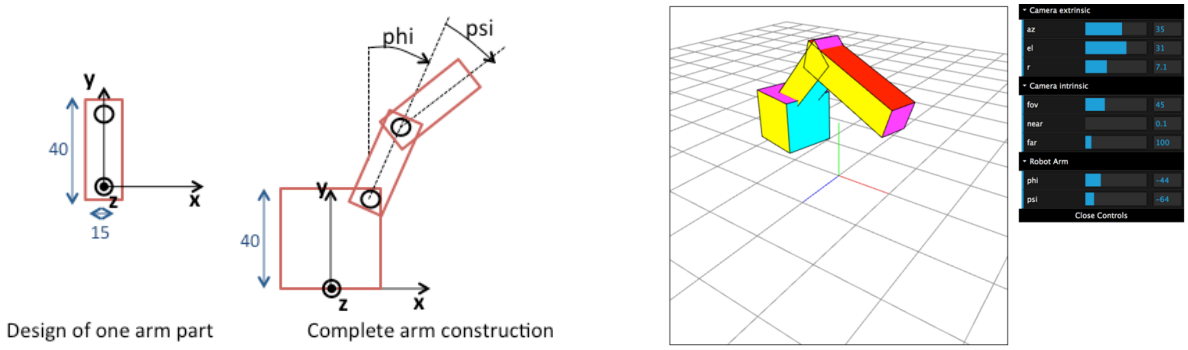
to load the OBJ file (this one does not have texture). In this case, OBJLoader is not included by default in Three.js, and you will need to include in your header one of the scripts provided in the subdirectory `js/loaders`, that belongs to the extended source distribution of Three.js:

```
<script src="js/loaders/OBJLoader.js"></script>
```

Other object formats follow a similar pattern, read the documentation if you want to load other formats: https://threejs.org/docs/index.html?q=load

OBJECT 4: Articulated robot

Q5. Scenegraph design [15] Using only simple boxes (BoxGeometry), create an articulated robot similar to the figure. Use params and dat.GUI to control the two angles phi and psi of the lower and upper arm. The robot is composed of two elongated box parts, as shown in the Figure. The user should control the phi and psi angles. The articulated arm should be attached to the cube and follow it when it moves.



Design of one arm part Complete arm construction

Figure. Schematics of the robot, and mockup of 3D Geometry display (the exact rendering may differ)

Hints:

- For the base cube, you can reuse existing cube
- The design of each arm part can be done using BoxGeometry, and translating the geometry such that the hinge is centered in (0,0,0). Using the units shown on the shematics, the box geometry created could have (width,height,depth)=(15,40,15):
  `lowerarmgeom = new THREE.BoxGeometry(15,40,15);`
  Since that box is created with the origin in the center of the box it must be translated up by some amount to have the new origin on the extremity of the box. Scaled up, this gives:
  `lowerarmgeom.applyMatrix(new THREE.Matrix4().makeTranslation(0,20-5,0));`
  The mesh can then be created using this translated geometry:
  `lowerarm = THREE.Mesh(lowerarmgeom, material)`
  Make diagrams of your own where you represent all parameters you need in the code.
- Each part can then be attached to its parent to create a scenegraph. In this case, the scenegraph is just a chain: scene→cube→lowerarm→upperarm. Each mesh is attached to its parent using the add function. For instance, to declare that `cube` is a child of `scene`:
  `scene.add(cube)`
  To declare that `lowerarm` is a child of `cube`:
  `cube.add(lowerarm)`
- When a part is a child of another part, the properties position, scale and rotation are defined relative to the parent. Therefore, in the following code
  `lowerarm.position.set(tx,ty,tz) // tx,ty,tz is up to you`
  `lowerarm.rotation.set(rx,ry,rz) // rx,ry,rz is up to you`
  - (tx,ty,tz) has the usual interpretation of the position of the *origin of the child coordinate frame, expressed in the parent coordinate frame*, i.e. if the hinge of lowearm is located at (0,0,0) in local coordinates, then its position in cube coordinates will be (tx,ty,tz).

- rx=ry=0 for rotation around Z axis. All these values can be directly obtained by studying the robot schematics and drawing each part in its own coordinate frame with its direct child.

Q6. Interaction [10] Let the user move the robot around the plane with the keyboard as follows:
- move forward or backward (using the arrows UP and DOWN).
- rotate around its vertical axis (using LEFT and RIGHT)

Make sure to explain to the user which controls your system uses on the HTML page below the canvas. The cube should slide on the plane, and rotate around its center.

Hints:

- Note that forward and backward are defined with respect to the X axis of the *cube*, not the *scene*. When the cube is rotated, this is not the same as changing `params.x` only. Both `params.x` and `params.z` needs to be changed, depending on `params.angle` and a bit of trigonometry. This is very similar to the asteroid game, with the small changes that translation occurs in the XZ plane, and rotation is defined around the Y axis.
- You may choose to use either discrete movements (processed directly in `onKeyDown`), or continuous fixed velocity (processed in the animation callback), as you prefer.
- to help with the debugging, you may want to update dat.GUI so that it reflects the current position and rotation of the cube, by calling the provided function `updateGUI()` after changing the content of `params`.

OBJECT 5: Textured tetrahedron from scratch

Q7. Manual Geometry [15] Instead of using ready-made objects, we now design a (simple) object from scratch. Create a tetrahedron (pyramid with 4 vertices and 4 faces) and add it to the object on the plane. Rearrange the objects on the plane to avoid overlapping. **Constraint:** you must create this object "from scratch", i.e. by specifying the vertices and the faces explicitly one by one, instead of using a ready-made geometry.
Hints: First start with a diagram of the geometry you want to create, in front view (XY) and top view (XZ). Give explicit names to each of the 4 vertices (P0..P3), and identify their 3D coordinates. Give explicit names to each of the 4 faces (F0..F3), and identify for each of them its 3 vertices.
For the geometry part, create an empty Geometry

```
tgeom = new THREE.Geometry()
```

then push each of the 4 vertices using

```
tgeom.vertices.push(new THREE.Vector3( x,y,z )) // x,y,z are up to you
```

then push each of the 4 faces using

```
tgeom.faces.push(new THREE.Face3( a,b,c ))      // a,b,c are up to you
```

The face normal can be computed automatically from the vertices using

```
tgeom.computeFaceNormals()
```

The object can then be created as a mesh in the same way as in Q1:

```
tmesh = new THREE.Mesh(tgeom, material)
```

Don't forget to add it to the scene. More examples can be found here:
`http://www.johannes-raida.de/tutorials/three.js/tutorial02/tutorial02.htm`
To help you debug, don't hesitate to use the Javascript console to display the content of the variables `tgeom.vertices` or `tgeom.faces`… and make sure they have the correct content.

Q8. Manual UV mapping [10] We now add texture to the mesh of Q7. Using an image for the texture (you may use the file `moretextures/triangle_texture.png`), define the corresponding UV-mapping manually. (see figure hereafter)
Hints:

- Select or create the texture image and identify the texture (UV) coordinates for the vertex in each face. For instance, if you want all faces to look the same and use the provided image "moretextures/triangle_texture.png" shown in the figure below, the UV mapping would be for each face [A,B,C]→[(0,0), (1,0), (0.5,1)]. The UV coordinates are normalized between 0 and 1, and have their origin in the bottom-left corner of the texture image.

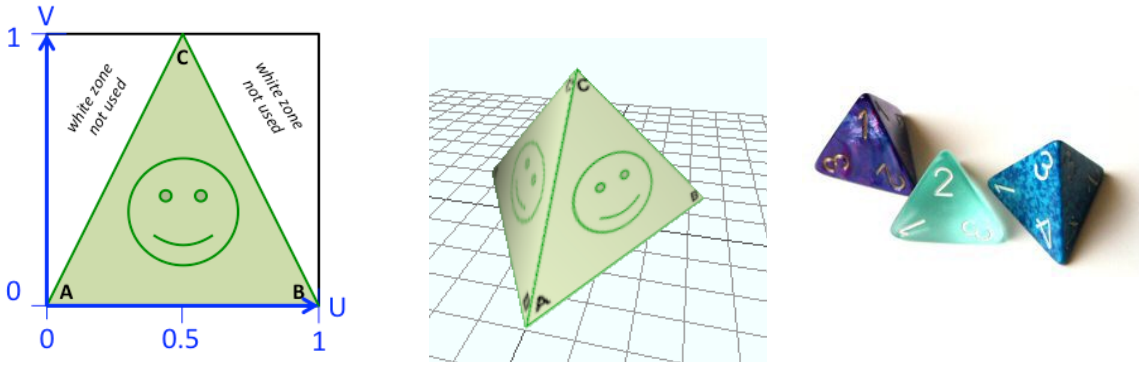


Figure. *(left) Texture for one face displayed in UV coordinates. (middle) Textured tetrahedron using the texture image on the left, with the same UV mapping for all faces. (right) Example of real life tetrahedron.*

- For each Face in your geometry, define the UV coordinates associated to its vertices. E.g for the face created as the triangle with vertices (a,b,c):
  `tgeom.faces.push(new THREE.Face3( a,b,c ))`
  the UV mapping [a,b,c]→[(au,av),(bu,bv),(cu,cv)] is created as:

```
tgeom.faceVertexUvs[0].push([
      new THREE.Vector2(au, av),
      new THREE.Vector2(bu, bv),
      new THREE.Vector2(cu, cv)]);
```

where (au,av) represent the UV coordinates of vertex #a in the face, and similarly for b and c. (the index [0] is necessary, as a geometry may have several sets of UV mappings to allow for sprite animations, we only use one UV mapping here). Once defined, the UV coordinates of vertex k=0..2 within triangular face n=0..N-1 are accessed as

`tgeom.faceVertexUvs[0][n][k].x` for U and

`tgeom.faceVertexUvs[0][n][k].y` for V.

- After creating/modifying the UV mapping, you need to inform Three.js that internal buffers need to be updated, else the rendering will use an old UV mapping or no UV mapping at all:
  `tgeom.uvsNeedUpdate = true;`
- Load the texture image and assign it to the mesh as in Q3

```
ttexture = textureLoader.load('moretextures/triangle_texture.png')
tmesh.material = new THREE.MeshLambertMaterial(
            { map: ttexture, side: THREE.DoubleSide} )
```

As in Q3, the texture is loaded asynchronously, make sure you refresh the rendering if the tetrahedron appears completely black.

Bonus [max 10], for instance:

Create a new camera that follows the robot and let the user switch between robot and Orbit view using the keyboard [10]

Let the robot arm launch projectiles that bounce on the floor [10]

Texture the robot with custom textures on the various sides, so that it looks really like a robot [10]

Texture the tetrahedron differently on each face [5]

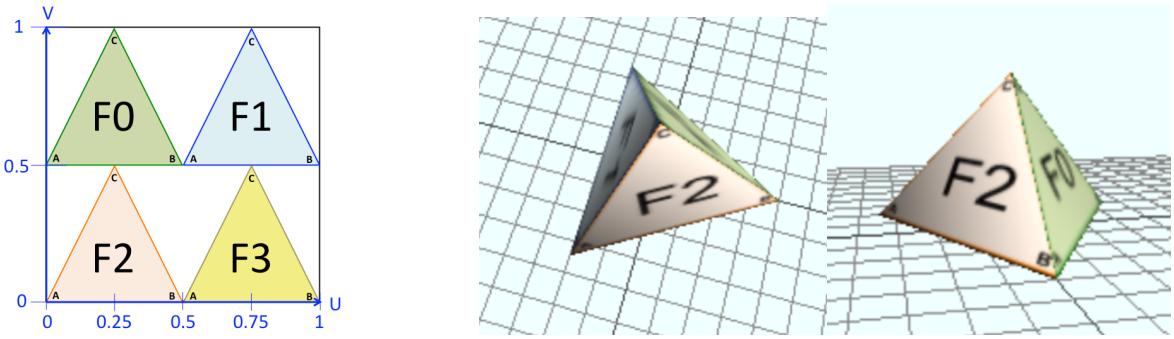


Figure: (left) Example of image "tetra-texture.png" (right) result of using it to texture each face of the tetrahedron with a different part of the image