Assignment 3 – Graphics Project

UPR Mayagüez - COMP 4046 - Computer Graphics

Abstract: Implement shaded and textured scenes in WebGL with interactive features

Due date: May 5.

Instructions

- Hand in by using the Ecourse system exclusively, https://ecourses.uprm.edu/course/view.php?id=533 i(tem Assignment 3)
- One ZIP file contains all code and data necessary to run your assignment. It is organized as follows
- A main index.html HTML file is present at top level. It provides links to all results with a short explanation.
- If several results are asked in an exercise, make sure the result of each question is visible or accessible. File organization (filenames, flat or hierarchical subfolders…) is up to you, but should be clear and meaningful.
- The example should run automatically by simply following the links, when the code is run in the Firefox browser through a webserver as seen in class. If any additional action is required to run the results, they must be documented in the main page.
- Pay special attention to providing appropriate documentation of the use of your code, especially concerning interactivity: please display in the HTML page the keys used for the keyboard, and how to use the mouse.

Objective: Develop a maze game including complex model, textures, lighting and interativity

Note: The evaluation of this project is based on the fitness of the final result to the objectives and specifications of Part 1, 2 and 3. The hints are provided to help you, but feel free to adapt and improve on them, as long as the specifications are met. You should remove the "debugging" wireframes and temporary objects before submitting.

The page `maze/index.html` provides a startup template.

Useful documentation and sample code:
http://threejs.org/docs/
https://stemkoski.github.io/Three.js/
http://threejs.org/examples/

[10] General presentation and quality of the submission

Part 0: Analysis of the provided template

The template code is organized as follows:

`index.html`	Main code
`js/`	Javascript libraries and utility code
`js/three.min.js`	THREE core framework
`js/loaders`	THREE extensions to load object files, not included in core framework
`js/maze-utils.js`	you can add your own utility functions there
`textures/`	image files used for texturing the tiles
`objects/`	pre-made 3d models with complex meshes

Initialization is done in `webGLStart()`

- Create WebGLRenderer
- Create Scene
- Create cameras: cameras[0] is outside view, cameras[1] is hero view
- Create some helpers for debugging (can be hidden later on)

    - CameraHelper: shows the view frustum of the corresponding camera
    - AxisHelper: show a set of XYZ axes (X in red, Y in green, Z in blue)
    - PlaneGeometry: shows the plane where the maze is supposed to be built
- Create one tile of the maze with plain color material
- Create a hero with simple geometry and material (box + sphere)
- Create lights
- Create controllers (for mouse, keyboard, GUI)

Control uses a similar approach as seen in Assignment 1 and 2:

- the keyboard controller created in `initKeyboard` triggers the following callbacks:
    - `handleKeyDown`, `handleKeyUp`: update a structure `currentlyPressedKeys` that records what are the currently pressed keys (this structure is used in `handleKeys`)
    - `handleKeyDown` also processes keys that are supposed to react only once (key 'v' changes the current camera for instance)
    - `handleKeys` processes keys that are supposed to react as long as they remain down. This function is called repeatedly by `tick()` each time before rendering
- the mouse controller created in `initMouse` calls the following callbacks:
    - `handleMouseDown`
    - `handleMouseMove`
- The GUI controller created in `initGUI` changes the global parameters `params`

The View update is done in `animate()`
Following the Model-View-Controller principle of separation of concerns, the controllers should in principle not update the View directly, but first change the Model (represented here by global parameters: `params, posx, posy, az,` …), which is then used in `animate()` to update the View (the objects and the cameras in the scene)

Finally, the infinite loop for rendering is handled by `tick()`, which calls `renderer.render(scene, camera)` once the scene has been updated.

Before starting the project, read the template code and make sure you understand all steps.
For each line of code, identify all parameters used in the functions. In case of doubt, look at THREE.js documentation or corresponding examples.
In particular, pay specific attention to the following:

- Basic elements required to create a scene (Renderer, Scene, Object3D, Light…)
- How to create a Mesh object: geometry + material
- Transformations. Be careful that two different ways to move objects around are used:
    - The effect of `geometry.applyMatrix` is to **modify the vertices positions** in a 3D geometry. It uses 4x4 transformation matrices created using `THREE.Matrix4()`, is a similar way as when we used `gl_matrix` in Assigment 2. This approach is also used when merging two sets of vertices using the function `geometry.merge(geometry2, matrix, 0)`
    - The effect of `object3d.position.set(x,y,z)` and `object3d.rotation.set(phi,theta.psi,'XYZ')`, where `object3d` can be a `mesh` or a `camera` is to **change the Model Matrix** (for a mesh), or the View Matrix (for a camera). Be careful that after changing the position of a object this way, it may sometimes be necessary to call `object3d.updateMatrixWorld()` to refresh the scene (for instance for the `CameraHelper` to follow properly its camera)
- How a new Javascript object is created using the keyword **`new`** each time a new data structure is needed. Be very careful that the syntax "`object2 = object`" does not copy the data structure `object`, it just provides a new pointer `object2` that points to the same existing one. The `object2` still exist for some time, we just lost access to it. You do not need to deallocate (free)

the objects, as this is done automatically by the Javascript runtime. When the content of a data structure needs to be updated, try to change it in place:

- For instance, for Vector3, method set:

```
var vec = new Vector3(3,4,5)
vec.set(vec.x + 1, vec.y, vec.z)
object.position.set(vec.x, vec.y, vec.z)  or object.position.copy(vec)
```

The template comes with predefined cameras. Functions that manipulate the cameras include: `webGLStart` (create cameras), `handleKeys` (modify the parameters), `animation` (modify the camera according to parameters):
- `cameras[0]` is an elevation/azimuth/distance camera to give an overview of the scene
Its parameters are stored in the global variable `params` which are controlled by the dat.GUI controller and by the keyboard (Keys IJKL for az/el and UO for distance to target).
By default, the camera is looking at target=(0,0,0).
- `cameras[1]` is the subjective view of the hero. It is positioned at `(posx, posy, 0.85)`, where `(posx,posy)` is the position of the hero on the ground, and z=0.85 is the approximate height at which the head is assumed to be. The hero is looking in a direction given by angle `az`. The hero is controlled by the arrows keys.
- `cameras[2]` is a top view using an orthographic camera.

Part 1: The Maze [40]

Objective:
Convert the maze data structure into meshes composed of 1x1 textured square tiles and include them in the scene for display. Some tiles of the maze should use a shinny material with normal mapping to make the walls more realistic.

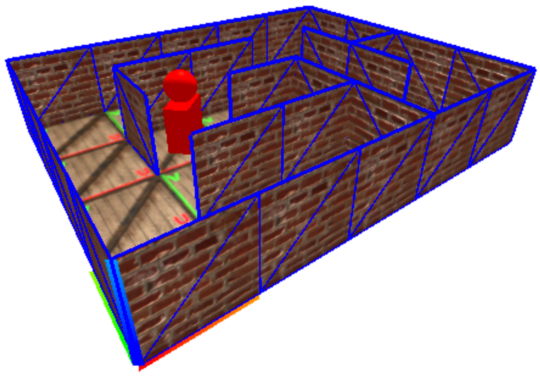
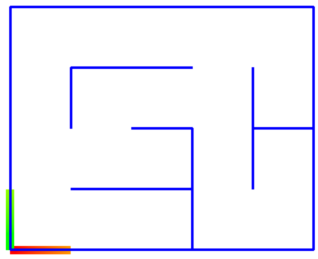


Figure 1. Illustration of maze textured mesh

Specifications:
The maze data structure is a Javascript object (container) with the following fields:
w (resp. h): number of tiles in the x (resp. y) direction
floor: w×h matrix representing the tile material for floor tiles.
xzwall: **w×(h+1)** matrix representing the tile material for wall tiles parallel to XZ plane
yzwall: **(w+1)×h** matrix representing the tile material for wall tiles parallel to YZ plane
The matrices are actually all stored as arrays in row-major order with the following convention:
- columns are stored in direct order: column i is for tile starting at x=i
- lines are stored in reverse order: line j is for tile starting at y=h-j (for floor and xzwall), and at y=h+1-j (for yzwall). This convention is chosen to ensure that the visual content of the matrix matches the visual presentation of the maze, as the (column,row) frame is reversed compared to the (x,y) frame when seen from positive Z (Y points up in the cameras[3] view instead of down).

w: 5, h:4, floor: [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 1, 1, 1, 1, 1,],	yzwall: [2, 0, 0, 0, 0, 2, 2, 2, 0, 0, 2, 2, 2, 0, 0, 3, 2, 2, 2, 0, 0, 3, 0, 2,],	xzwall : [2, 2, 2, 2, 2, 0, 2, 2, 0, 0, 0, 0, 2, 0, 2, 0, 3, 3, 0, 0, 3, 3, 3, 2, 2,],	

Each tile is associated a materialIndex, that corresponds to a texture file.
These materialIndex codes are stored in the tile arrays as follows: 0 for no tile, 1 for texture #1 (floor image), 2 for texture #2 (wall image), 3 for shinny bump mapped texture…

Detailed hints to reach the objective:

H1) createXYTile()

The provided code creates the 4 vertices of a square [x, x+1]x[y,y+1], then create two triangle primitives to be added to the geometry, storing also its materialIndex (used later)

```
var geometry = new THREE.Geometry()

var nvertices = geometry.vertices.length
geometry.vertices.push(
  new THREE.Vector3( x  , y  , z ),
  new THREE.Vector3( x+1, y  , z ),
  new THREE.Vector3( x+1, y+1, z ),
  new THREE.Vector3( x  , y+1, z )
);

var nfaces = geometry.faces.length
geometry.faces.push(
new THREE.Face3( nvertices, nvertices+1, nvertices+2,
                 null, null, materialIndex),
new THREE.Face3( nvertices, nvertices+2, nvertices+3,
                 null, null, materialIndex)
)
```

Add the UV coordinates to each face as follows:

```
geometry.faceVertexUvs[0].push(
            [
                new THREE.Vector2( 0,0 ),
                new THREE.Vector2( 1,0 ),
                new THREE.Vector2( 1,1 )
            ], [
                new THREE.Vector2( 0,0 ),
                new THREE.Vector2( 1,1 ),
                new THREE.Vector2( 0,1 )
            ]
        );
```

In the webglstart function, replace the single color material of the test tile with a texture. We will use the `crateUV.jpg` texture, as it includes UV annotations to facilitate the debugging.

```
var tex = THREE.ImageUtils.loadTexture("textures/crateUV.jpg");
var material = new THREE.MeshPhongMaterial( {map: tex } );
var testtile = new THREE.Mesh( geometry, material );
scene.add( testtile );
```

Observe that the tile does not appear if you look at it from below, due to Hidden Surface Culling, based on the triangle orientation. Add the property

```
{map: tex, side: THREE.DoubleSide  }
```

to the material to make the tile visible from both sides.

H2) Create two other functions createXZTile and createYZTile that create tiles in XZ plane and YZ planes respectively, starting at position (x,y,z), and with the given material index.

```
createXZTile = function(x,y,z,materialIndex) { … }
createYZTile = function(x,y,z,materialIndex) { … }
```

Two approaches can be used: duplicate the code of createXYTile and change the vertices coordinates, or use geometry.applyMatrix on the geometry created by createXYTile to transform the horizontal tile into a vertical one.
Check the three tile creation functions in a configuration similar to the one shown in Figure 2 below. In particular, using the crateUV texture, make sure that the +V direction corresponds to +Z (the up direction of the images should be up in the scene).

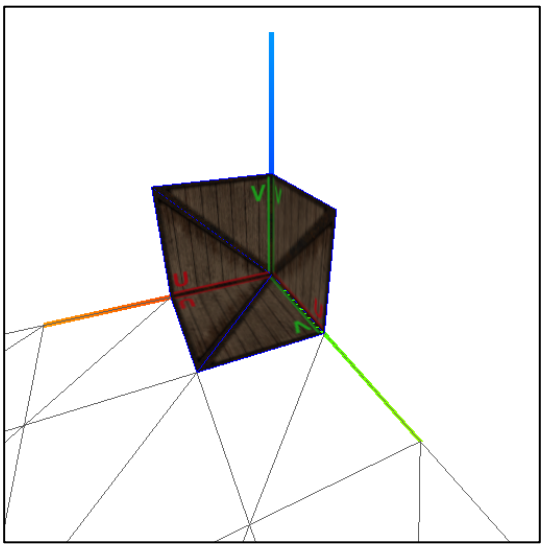


Figure 2: Illustration of crateUV texture on XY, XZ and YZ tiles

H3) Create the floor

Fill in the content of function

```
createFloorGeometry = function(maze) {…}
```

so that it create all floor tiles, merged into a single geometry object. Your code should go through maze.floor to create one by one each tile using createXYTile and then merging it into the geometry. A new tile can be added to an existing geometry as follows:

```
var tile = createXZTile( … )
geometry.merge(tile, new THREE.Matrix4(), 0)
```

The matrix argument may be used to translate the tile if needed, and the last argument, 0, states that the materialIndex of the tile should be left unchanged during the merging.
Use the function in webGLStart to create the floor, create a mesh with it, and add it to the scene.

H4) Create the walls

Fill in the content of function

```
createWallGeometry = function(maze) {…}
```

so that it creates all wall tiles, merged into a single geometry object.
Use the function in webGLStart to create the walls, create a mesh with it, and add it to the scene.

H5) Add multiple materials

Up to now, the tile display may all have the same material (crateUV).
Let's create a multimaterial, which will be rendered by picking up for each face the material corresponding to materialIndex. Put several materials in an array, and create a MeshFaceMaterial:

```
tex0 = THREE.ImageUtils.loadTexture( "textures/crateUV.jpg" );
tex1 = THREE.ImageUtils.loadTexture( "textures/floor1.png" );
tex2 = THREE.ImageUtils.loadTexture( "textures/brick-c.jpg " );
…
materials[0]=new THREE.MeshPhongMaterial({map:tex0,…} );
materials[1]=new THREE.MeshPhongMaterial({map:tex1,…} );
materials[2]=new THREE.MeshPhongMaterial({map:tex2,…} );
…
multimaterial = new THREE.MeshFaceMaterial( materials );
```

Use multimaterial to create the meshes instead of material

```
floorMesh = new THREE.Mesh( geometry, multimaterial );
```

H6) Add shiny bump mapped tiles

The pair of images image-c.jpg / image-n.jpg correspond to a pair color/normalMap.

```
tex = THREE.ImageUtils.loadTexture( "textures/brick-c.jpg" );
tex_n = THREE.ImageUtils.loadTexture( "textures/brick-n.jpg" );
materials[2]=new THREE.MeshPhongMaterial(
                        {map:tex, normalMap: tex_n, …} );
```

adapt the various Phong parameters (`specular`, `shininess`, `normalScale`…) to give a shiny appearance to these tiles.

Part 2: The Hero [40]

Objective: Improve the appearance of the hero, and make sure he cannot go through the walls anymore. Also fix the lighting so that the maze is darker and requires the hero to bring a spotlight.

Specifications:

- Load a textured model (person or other) from a file to be used for the hero, and:
  - Scale and rotate it correctly to be standing, and fitting comfortably in one tile of the maze.
  - The "feets" of the hero correspond to the (posx, posy) coordinates, and its "head" faces front (same direction as when pressing the up arrow).
- Prevent the hero to cross through the walls
- Darken the global lights, and add a portable spotlight to the hero, that shines on its hand at approximately z=0.5, slightly to the right of the hero, and brings a yellowish light in a limited angle towards the front.

Hints to reach the objective

H1) Load a model from file

Insipred from the http://threejs.org/examples/ - webgl_loader_json_objconverter demo:

```
THREE.Loader.Handlers.add( /\.dds$/i, new THREE.DDSLoader() );
var loader = new THREE.JSONLoader();
var onJSONLoaded = function ( geometry, materials ) {
  hero = new THREE.Mesh( geometry,
            new THREE.MeshFaceMaterial( materials ) );
  scene.add( hero );
}
loader.load( 'obj/male02/Male02_dds.js', onJSONLoaded);
```

This should load the textured model and display it, but the scale and rotation are wrong for our maze (unzoom using camera0 to see this very large object).

H2) Fix the scale and rotation

Use

```
hero.geometry.applyMatrix( … )
```

before adding the model to the scene, in order to scale, translate and rotate the model to meet the specifications.

For the hero position, just make sure that the feet are centered around (0,0,0), as the model matrix of the hero is simply defined inside animate() by

```
hero.position.set(posx, posy, 0)
hero.rotation.set(0,0,degToRad(az),"ZXY")
```

H3) Add collision detection with the walls

The animate() function is responsible to move the hero from (posx, posy) to (posx+dx, posy+dy), as determined inside the handleKeys() controller. If a wall is present in front of the hero in the (dx,dy) direction, the motion should be canceled (dx,dy)←(0,0)

The following code uses the object Raycaster to check if the ray [(posx,posy), (posx+dx, posy+dy)) intersect the walls within a max distance of 1 unit:

```
var pos = new THREE.Vector3(posx,posy,0.5)
var dir = new THREE.Vector3(dx,dy,0)
var raycaster = new THREE.Raycaster(pos, dir, 0, 1)
var intersects = raycaster.intersectObject( wallMesh )
if (intersects.length > 0) {
  // intersect[0].distance to the closest wall
  // in the moving direction. Use it to decide
  // if we should cancel the motion : dx=0, dy=0
}
```

H4) Improve the lights

Reduce the ambient and directional global light by changing their color.

Create a new SpotLight in startWebGL

```
spotLight = new THREE.SpotLight( 0xffff00, 2.0, 3.0,
Math.PI/180*30 );
scene.add( spotLight );
```

And update its position and direction of lighting as follows:

```
var handPos = new THREE.Vector3( posx+ … , posy+ …,    0.5)
var spotDir = new THREE.Vector3(      … ,      …,   -0.5)
spotLight.position.copy(handPos)
spotLight.target.position.copy(spotDir)
spotLight.target.position.add(handPos)
spotLight.target.updateMatrixWorld()
```

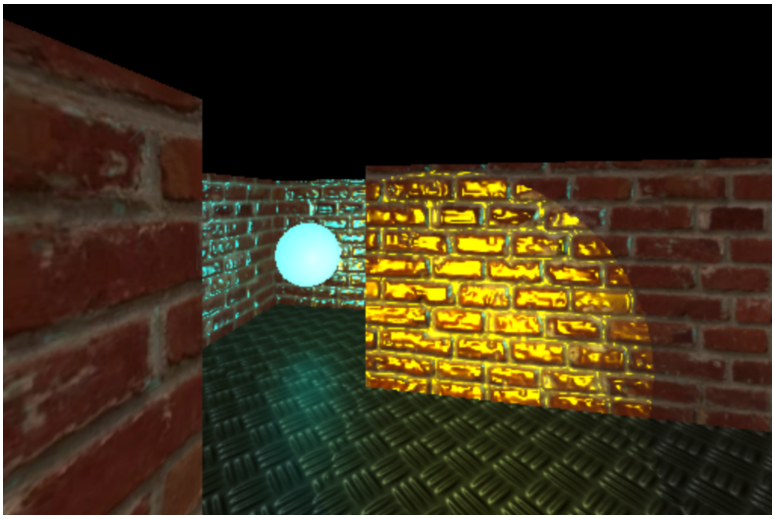


Figure 3: Example of shiny surface with specularities and normal mapping, reflecting a yellow spotlight. The blue orb of Part 3 is shown in the background.

Part 3: The Game [10]

Objective: Add a goal to the maze, as a rotating orb on one of the tiles and declare victory when it is reached

Specification:

- The orb is a sphere that emits a bright blue light around.
- The position of the orb in the maze is given by the field `goal: {'x': 4, 'y': 3}` in the maze structure, which defines the tile where the orb should appear, floating in the middle of the tile, at a height of 0.5.
- The game should declare victory when the hero is standing on the correct tile and presses the key 'c' to collect the orb

Detailed hints to reach the objective

H1) Adding the orb and light emission

The orb is a simple Sphere object (use `ShereGeometry`), and use the field `emissive` when constructing the Material. Its position is computed from `maze.goal.x` and `maze.goal.y`.
Note that emissive material only concern the appearance of the orb itself, it does not influence what is around. It is necessary to also add a `PointLight` in the center of the orb that will emit the light reflected by the tiles around.

H2) Victory

When the 'c' key is pressed, use the current `(posx, posy)` to compute the corresponding tile using `Math.floor`.

EXTRA:

[max 20 points] Originality and extended features

If you completed the project, you may consider extending it with more ideas.
Extra score depends on difficulty and quality of the additional features. For instance: using the mouse for looking around [10], texture and make the orb rotate as some planet [10], improved collision to really avoid going into the walls, even with the shoulders [15], launch projectiles that rebound on the walls and the floor [20]…
RPG style: collect and drop objects in the maze [15], alternate motion for the hero that is constrained to stop only on the middle of tiles, but moves smoothly between tiles and when rotating [20]