Final project: Interactive graphics

Student: Cosmin Fane Cozma
Student n.: 1613880

1. Introduction

In this work we are going to implement a small library in order to demonstrate the comprehension of many of subjects analysed in class. We started from scratch including only the two previous homework projects and no other tools/libraries have been used. Also, the mathematical functions have been rewritten from scratch with the aim of giving a greater modularity to the code in terms of calls to methods and functions. It must be said, however, that inspiration has been drawn from the most popular libraries as far as the structuring of the project is concerned (folders, classes and separation of concepts on several classes as in the case of the code concerning animations).

Although the library offers many more features, in the *./examples* folder you can find some examples of the results that can be achieved. These examples are accessible through the *index.html* file (this file also contains a brief description for each example) present in the project's root directory.

Still in the root directory, there is a folder containing the files generated by TypeDoc[1] that describe the entire API of the library.

2. Development environment and tools

Despite the request to use only JavaScript in the development, we opted for a more typified language, TypeScript. TypeScript is a strongly typed superset of JavaScript, which means it adds some syntactical benefits to the language. The reason for our choice is that on large projects, despite later versions of the ES5 that offer an improved syntax for OOP, in JavaScript it is very difficult to find a logical error. Also, the best IDEs find it difficult to detect errors, sometimes even typing. Clearly typescript cannot be interpreted by browsers, so we need some tools for transpiling and dependency management. For the transpiling we use directly the typescript compiler provided by Microsoft and managed through npm (Node.js Package Manager). In the project folder there is a configuration file for the compiler with various compilation parameters, including that for transpiling the Typescript code in JavaScript version ES6 code. For the management of the dependencies and the generation of the distribution version of the library (an executable distribution on the browsers), the Rollup.js plugin was used, which takes the compiled code on more files as input and it supplies as output a single file containing all the code under the GLIGHT namespace. Rollup.js configurations are in the rollup.config.js file[2].

3. Shaders

Given the multiplicity of the properties of the meshes, materials, lights and textures, it was not possible to find a single couple vertex-fragment shaders that could be followed for any 3D object in the scene. Not even

[1] *TypeDoc* is a package which takes the *Doc* comments in the code and generate, base on specific meta tags, the code documentation

[2] To recompile the project just install node.js and npm. Then run *npm install -g typescript rollup* in shell, and then *npm run dist* in the project directory.

using indicator variables was an option. So, it was decided, to the detriment of the performances[3], to generate a couple vertex-fragment shaders contextualized for each 3D object in the scene. Each time an object is inserted[4] in the scene, the shaders are regenerated so that in the final rendering of the scene each object takes into account the other objects in the scene (a simple example: the insertion / deletion of a light in the scene changes the result of the rendering of an object). The generation of shaders for a mesh is made based on its geometry, which includes the attributes, and on its material, which also includes the textures if these are defined, in addition to the colours. As for the interpolation method, the Phong method is always used, while the reflection method is defined by the type of material (*LambertMaterial* or *PhongMaterial*) assigned to the mesh.

4. Project description

In order to describe the work done as much as possible, we will divide it by folders and then by file and describe each one of them.

./core: this folder contains the classes underlying the entire project structure. In particular with the *Base*, *Node* and *Object3D* classes, the concept of inheritance of the OOP is exploited to implement a hierarchical model of which every entity in the scene is an instance. Now let us look at the classes in more detail:

- ***Base.ts:*** Lights, cameras, materials and meshes inherit from this class three very important properties: *uuid*, *pickColor* and *type*. The first property uniquely identifies each instance object of the *Base* class or its subclass. The second property also is an identifier and serves to implement the Picking technique (select objects in the 3D scene). While the first two properties serve to identify an object, the third property serves to verify the belonging of the object to a specific group / category of objects. We will see how these properties are used.
- ***Node.ts***: This class extends the *Base* class. It represents a node in the hierarchical structure. *Parent* and *children* properties are defined to keep track of the tree structure, while *setParent()* and *unsetParent()* methods are offered to manage the structure[5]. Three other properties *isRoot*, *selected* and *hidden* have been defined. Each one indicates if a node is root, is selected (in which case its local axes are also drawn) and if it is hidden.
- ***Object3D.ts:*** Each entity in the scene (lights, cameras and meshes) inherits from this class. It is a subclass of the *Node* class and it is the epicentre of the transformations that can be performed on a generic 3D object in the scene. The properties *translate*, *rotate* and *scale* maintain the current values of the position, orientation and scale factors respectively. The *pivot* property maintains the current value of the pivot point position, the *target* and *targetDefined* properties are needed if you wanted the object 3D orienting itself according to the position of another object in the scene, the *forwardDirection*, *upDirection* and *rightDirection* properties maintain the directions of the local axes after the transformations and finally the *localMatrix* and *modelMatrix* properties are two matrices, where the first maintains the result of local transformations and the second is the result of the multiplication of the transformation matrix of the parent node and the local transformation matrix. It will be the *modelMatrix* matrix that will be sent to the GPU during rendering. A call to the *applyTransformations()* method makes the transformations

[3] In a more recent version of WebGL you can greatly improve the performance thanks to some features like UBO (Uniform Buffer Object).
[4] A better solution would be to use event-oriented programming.
[5] Note that methods like *addChild()* and *removeChild ()* would have been more logical. But this would have allowed us to create graphical structures since we can mistakenly add a child to several fathers causing errors in the application of transformations.

with the values of the properties listed above and sets the *modelMatrix* matrix with the final values to send to the GPU.

- ***Renderable.***ts: because of similarities of the representation of the lights, cameras and meshes in the scene, the *Renderable* class was created. It does all operations in common between the objects previously mentioned, for example the *render* method, which uses other private methods to setup uniforms variables and starts the drawing.
- ***Mesh.ts***: The *Mesh* class models an object in the scene. It extends the *Renderable* class and the parameters in the constructor are an instance of the *Geometry* class and an instance of the *Material* class, the latter being an optional parameter (default *LambertMaterial*). Another property of the class is *shaderProgram*, which keeps the reference to the program to be sent to the GPU when the mesh itself must be drawn. The two most important methods are *draw()* and *createShader()*. The first one sets the uniform locations and starts the processing in the GPU, the second, as mentioned, deals with the generation of the shader program based on the geometry, the material, and the lights present in the scene. There is another utility method, *offScreenDraw(),* used for the Picking.
- ***Sene.ts***: the *Scene* class mainly acts as a collector, so for this reason its methods are mainly query methods. Properties such as *children*, *viewer*, *activeObject*, *activeCamera* maintain the reference to objects in the scene. However, it must be emphasized that among the properties, there is one in particular, *lightsManager*. Since the lights need a different management from the other objects in the scene, it was decided to commission the management of the lights in the scene to a particular class, *LightManager*, whose job is to discriminate and extract the information necessary for the generation of the shaders and to return all the data to be loaded in the uniform locations in the rendering phase.
- ***Renderer.ts***: the main purpose of this class is to set the initial parameters of the context in which we are going to draw. The class takes as a parameter in the constructor an instance of the class *Scene*, and using the methods *draw()*, *drawLoop()* and *render()*, we can start drawing the scene. The class also offers other methods for setting the context such as *setViewport()*, *setClearColor()* and *setPixelRatio().*
- ***GL.ts:*** this script contains only some useful functions of various types, including those for initializing the context for access to the WebGL API.
- ***Constants.ts:*** a script containing the definition of all the constants used in the various classes for greater code readability (object3D types, interpolation types, easing types, etc…).
- ***Color.ts***: class for the purpose of collecting the colour management functions, as well as at the same time representing a method of managing the colours as if they were unary entities.
- ***ArrayBuffer.ts and ElementArrayBuffer.ts***: represents buffers in the GPU memory. The properties of the class maintain the information on the buffer (number of elements, size of each element, reference to the buffer, etc.) and methods to update and free the memory.

./geometry: this folder includes the classes representing the geometric figures. Each extends the Geometry parent class as well as the classes in the *./helpers* subfolder. Their main purpose is to generate the data for the geometric figure in question based on the parameters passed (for example the radius and the number of divisions for the Sphere geometry class) and pass them to the parent class.

- ***Geometry.ts***: this script contains the Geometry class, whose task is to collect the data, create buffers and load data into the buffers. This is done using the utility class *Buffer*. It was decided to include buffer management operations in this class, due to the efficiency of memory management. The basic idea is to avoid creating many buffers with the same data. Starting from a generic conception of objects in the three-dimensional scene, we then try to eliminate the duplications, sending to the GPU memory only once the data that are equal to all the objects in the scene.
  The constructor of this class takes as input the data relating to the positions of the vertices, and other optional data such as texture coordinates, normals and indexes.

For each of these data categories, which can also be set at a later time, a buffer is created in the GPU memory. To lighten the Mesh class, a *startDraw()* method has been created which, in addition to binding the buffers to the attributes, starts the execution of the shader program on the GPU.
The class also calculates the tangents and bitangents if the indices, normals and texture coordinates have been defined.

./cameras: folder containing all classes related to the viewer. The classes *PerspectiveCamera* and *OrthographicCamera* are subclasses of the *Camera* class and their only purpose is to be specialized in the control of some specific parameters of the viewing volume.

- ***Camera.ts:*** the *Camera* class is a class very similar to the *Mesh* class, which also extends the *Renderable* class. Although the viewer is always only one, more cameras can be present in the scene simultaneously. This implies that the active camera, the one from which the viewer is looking at the scene, is not visible in the scene but others are, and therefore must be treated as any other mesh in the scene. To implement this behaviour it was therefore necessary to override the *applyTransformations()* method inherited from the class Objec3D, which based on the value of the *active* property of the class itself, as if the local transformation matrix were a *viewMatrix* or a *modelMatrix*. Also in this case, the *offScreenDraw()* method was defined for the Picking technique.

./helpers: directory containing two types of particular mesh that require different management than the others. As can be understood from the names, the Grid class takes care of constructing the floor grid based on some parameters (size and number of divisions). The Axis class instead deals with the creation of the axes. In this case the class includes three meshes, one for each axis, which are handled as a single object in the scene.

./input: within this folder two classes are defined, *Controls* and *Picker*. The first, which takes an instance of the *Scene* class as a parameter in the constructor, has the task of adding handlers and handling mouse and keyboard related events. The handlers act on the scene instance, or rather on the current selected object or/and on the currently selected camera. The second class, *Picker*, implements the unique colour selection technique. Using the *offScreenDraw()* methods defined in the *Lights*, *Cameras* and *Meshes* classes, a copy of the scene is drawn in an off-screen framebuffer. Each object is drawn using the *pickColor* property that the *Base* class provided. Thus, when the click is made the handler corresponding to the event calls the *find()* method of the *Picker* class with the pointer coordinates. This method reads the colour of the pixel corresponding to the coordinates of the pointer from the framebuffer and looks in the scene for the object corresponding to that colour. If an object is found, then it is assigned to the currently selected object in the scene.

./lights: five classes are defined here. *AmbientLight*, *PointLight* at *DirectionalLight* extend the *Light* class (look at the following paragraph) by adding new properties to the mother class based on the type of light. The *LightsManager* class, as mentioned above, is a utilitarian class through which the lights in the scene are managed.

- ***Light.ts***: as in the case of the Camera class, the Light class extends the Object3D class and is very similar to the Mesh class. This is because we want to give a visual representation of the lights present in the scene, so the Light class before representing a light, represents a Mesh.

./materials: this folder contains a lot of classes but the two main classes are Material and TextureImage and the other classes are extensions of those two. Here we make a distinction between PhongMaterial and LambertMaterial which is useful in the generation of the shaders since their types indicates the shading method to be implemented. Furthermore, at the base of each texture management there are some common operations that are included in *TextureImage* class.

- **Material.ts**: the properties of this class are the usual: ambient color, diffuse color and specular color. Other three corresponding attribute for intensity were included. The texture management was conceived as part of the material and for this reason the Material class has a property called texture intended to contain a reference to an instance of the *TextureImage* class.
- **TextureImage**.ts: To have a common starting base for each type of Texture, the Texture image class was created which is the mother of Texture and Texture Cube. It achieves the methods to modify the properties of a generic texture such as wrapping, magnification and minification filters.

./math: as we said, the mathematical functions have also been rewritten. The reason is that we wanted to give greater modularity to the code. The Matrix4 class represents a matrix and its methods always return the object itself giving the possibility to carry out concatenated and compact operations. Within the _Math class are defined all utilities functions like identifier generator, linear interpolation, Bezier curves, and other mathematical constants. The BSpline class computes the curve given the point and the degree of interpolation.

./shaders: there are two main classes in this folder; the first, *rProgram*, represents a program that can be run on the GPU shaders and mainly deals, with the help of the functions present in the *ShadersUtils* script, with creating, linking and compiling a shader program. It also takes care of retrieving the variable locations in the shader program. The second one, Parser, has the task of creating the source of shaders programs based on the parameters passed: mesh material, mesh geometry, and the lights present in the scene. In fact, the static method *createProgram(),* which take as input in instance of *Material*, an instance of *Geometry* and an instance of *LightsManager*, make use of some methods in order to generate shaders source codes.

./animations: this folder contains all the classes necessary for creating animations. Four of them are similar (*PositionsTrack*, *RotationsTrack*, *ScalesTrack* and *PivotTrack*) so we focus only on one example, say *PositionsTrack* class.

- ***Kayframe.ts:*** it represents the elementary object on which animations are based. It simply encapsulates the properties of a keyframe and is therefore made up of a variable indicating the time, a variable containing the values of the property of the object (position, orientation, scale, or pivot position) and the type of twining.
- **KeyframeTrack.ts:** an instance of this class represents a track in which keyframe can be added and removed. Within this class there are some algorithms for sorting and duplication management. Its main purpose is to maintain an ordered list of keyframes and to return, given a parameter t, the value of the property (position, orientation, dimensions) at time t using the interpolation method (linear or BSpline) and the type of indicated easing. In fact, its subclasses, PositionsTrack, RotationsTrack, ScalesTrack and PivotTrack are only concerned with indicating the property of the object in question to be saved in the keyframes.
- **AnimationClip.ts:** This class has the task, so to speak, of managing time. It takes an instance of the Renderer class as input into the constructor, that it uses to render the scene on every tick. A set of tracks form an animation and each track can be added to the animation through the *add()* method. It also contains the loop rendering in which they come. Basically, this class deals with managing the loop rendering. In fact, for each *KeyframeTrack* instance added to the animation, it calls the *update()* method with the *elapsedTime* parameter indicating the time passed from the beginning of the animation. This class also includes methods such as *start()*, *stop()* and *loop ()*.

To sum up what has been said so far, the library presents, under the namespace GLIGHT, the following classes and constants, which in principle also represent the API of the library itself:

- EASE
- INTERPOLATION

- Color
- Vector3
- Vector4
- Matrix4
- LineGeometry
- PlaneGeometry
- BoxGeometry
- SphereGeometry
- PerspectiveCamera
- OrthographicCamera
- Scene
- Mesh
- Group
- LambertMaterial
- PhongMaterial
- Texture
- TextureCube
- DirectionalLight
- PointLight
- Renderer
- AnimationClip
- PivotTrack
- PositionsTrack
- RotationsTrack
- ScalesTrack

5. Conclusion

There are many other things to do in order to consider the project complete. With this we refer to some features (marked with the *TODO:* comment) to be implemented like the *Manipulators* class, other kinds of keyframes interpolation, better management of textures and other features the we have thing to do.

Another thing that we want to underline is that for time reasons we didn't write test codes. This was difficult for the development because of bugs and because, in order to find them, we spent a lot of time. Actually, there are still some warnings to fix.

Particular difficulty was encountered in the generation of the source code of shaders, in fact there is also a lot of code written but never used in the project (for example the generation of tangents and bi-tangents for bump mapping). The difficulty comes from the fact that there is less evidence of errors due to the execution of shaders on the GPU.