

Computer Graphics (MIEIC)

Practical Topic 1

Introduction to WebGL and Project Configuration

Objectives

- Install, explore and learn to use the basic libraries and examples for the practical classes, as well as the steps for submitting results
- Learning how to create and use an interface (GUI) to control aspects of the scene and its objects.
- Learning how to create simple objects

Introduction

Currently, it is possible to generate interactive 3D graphics in web browsers, using **WebGL** technology and **JavaScript** language.

This method of 3D development has the advantages of not requiring the installation of libraries or the compilation of applications, while also being able to easily make applications available on different operating systems and devices (including mobile devices). However, for more demanding applications, it is recommended to use more efficient languages and libraries, such as **C ++** and **OpenGL**. However, given that the **WebGL** API is based on **OpenGL** (more specifically **OpenGL ES 2.0 / 3.0**), there are a number of shared concepts, so **WebGL** may be a good entry platform for current versions of **OpenGL**. In the context of **CGRA**, we will then use **WebGL** and **JavaScript** to create small graphical applications that illustrate the basic concepts of Computer Graphics, and allow for practical experimentation, which can be run in the most recent *web browsers (that support WebGL)*.

To support the learning process, the library **WebCGF (Web Computer Graphics @ FEUP)** was developed by the teachers of the curricular unit and by some students, specifically for the classes of **CGRA** and **LAIG**, in order to abstract some of the complexity of initialization and creation of support functionalities, allowing students to focus on the components relevant to the concepts of Computer Graphics.

Preparation of the development environment

An important part of this first practical work is the preparation of the development environment. To do this, ensure that you have configured the main necessary components, described below, and ensure that you can open the provided example application in the browser.

Necessary components

- **Web Browser with WebGL 2.0 support:**
 - The application will be effectively executed through the browser. An updated list of browsers that support WebGL 2.0 can be found at <http://caniuse.com/#feat=webgl2>. Currently, Firefox, Google Chrome and Opera browsers support this version of WebGL, both the desktop version and the Android version (although in some cases not all devices that can run these browsers have graphical capabilities to run WebGL applications).
- **The basic structure of the project:**
 - Includes the WebCGF library and other associated libraries, as well as the basic folder structure for the project and the HTML file that serves as the base / entry point of the application.
 - A .zip file with all the necessary files, including the base code for this practical lesson is available in Moodle.
 - This file must be unzipped to its own folder, in order to contain the **lib** (the library) and **ex1** (the base code for the exercise) folders.
 - For issues related to browser security restrictions, the main folder where the file is unzipped must be made available by a web / HTTP server (see next point).
- **HTTP Server:**
 - Since applications are accessible via the browser, and given their security restrictions that prevent access to scripts through the file system on the local disk, applications must be made available via an HTTP web server. In the context of CGRA, there will be no dynamic page generation (the dynamic components are run in the browser's Javascript interpreter), so any HTTP server that provides static content will do. There are several possible solutions to this requirement, including:
 - **Using a web server on the computer itself:** In some cases, students already have a web server (eg Apache, Node.js) running to support other projects. The same can be used for this purpose, as long as the server makes the folder with the application available through a URL accessible by HTTP. If you don't have a server, you can run a mini-server using one of the following options:
 - **Web server for Chrome:** A mini-web server that runs on Google Chrome itself:
<https://chrome.google.com/webstore/detail/web-server-for-chrome/ofhbbkphhbklhfoeikjpcbhemlocgigb>
 - **Python:** If Python is installed, a simple HTTP server can be created by running, in the folder to be shared via the web, the following command (depending on the version of Python):
 - `python -m SimpleHTTPServer 8080` (for versions 2.x)
 - `python -m http.server 8080` (for versions 3.x).

- **Node.js:** <https://www.npmjs.com/package/http-server>
- **Other alternative servers:**
 - <https://cesanta.com/binary.html>
 - <http://nginx.org/en/download.html>
 - <https://www.ritleabs.com/en/products/tinyweb/>
 - <https://caddyserver.com/download>
 - <https://www.vercot.com/~serva/>
 - <https://aprelum.com/abyssws/>
- **Use the FEUP student web site:** place the project in a folder within the public_html folder of the student's account, and access it through the public address **<http://paginas.fe.up.pt/~login/mytest>** (login will be the student code, upXXXX). In this case, by default it will be accessible to everyone (which can be bypassed, e.g., with an access control file *.htaccess*). It also has the disadvantage of involving the editing / updating of files on the FEUP server, and requiring a connection to the FEUP network in order to edit / load the application
- **A text editor or IDE:** The code that makes up the applications will be written in JavaScript, and stored in text files. For your edition, there are several alternatives too:
 - **Google Chrome** provides in its **"Developer Tools" (Ctrl-Shift-I)** a JavaScript debugger, which allows you to perform step-by-step execution, variable analysis, console query, and others, regarding the code that is being run in the browser, and also allows for mapping between the files accessible by HTTP and the original files stored on disk. It can therefore be used as an editor and debugger, and is currently **the recommended solution**.
 - Any text editor can be used to edit the files. However, we strongly suggest a text editor that supports a project structure with navigation in a file tree, to allow you to easily switch between the different files that will make up the project (Ex: Visual Studio Code, WebStorm, Brackets, Sublime, Atom , Notepad ++,...).

Testing the development environment

In this phase, you should already have a folder with the files of the base code provided in Moodle, and web server successfully running in that folder. You must therefore also have a URL address through which the folder is accessible (**NOTE: avoid folders with spaces and accents!**). Open the browser and direct it to the referred URL, and after a few seconds the sample application should appear, where you can manipulate the point of view with the mouse, using the **left button to rotate the scene**, the **right button to move it laterally**, and **pressing the central button (or Ctrl + left button) to zoom in / out**.

Available Resources

The 'WebCGF' Library

Documentation available at <https://paginas.fe.up.pt/~ruirodrig/pub/sw/webcgf/docs/>

Structure

The **WebCGF** library (Web Computer Graphics @ FEUP) - has the following main classes:

- **CGFapplication (+)** - Manages generic functions for initialization of the application and support libraries, and interconnects the other components.
- **CGFscene (*)** - It is responsible for the initialization, management and drawing of the scene
- **CGFinterface (*)** - It is used to build and manage the user interface; you can access the internal state of the scene to, for example, activate or deactivate features (e.g., lights, animations). At its base, the **dat.GUI** is being used:

<http://workshop.chromeexperiments.com/examples/gui>

The WebCGF library also includes the following classes that represent entities that can integrate a scene (non-exhaustive list):

- **CGFobject (*)** - Represents a generic object, which must implement the **display()** method; the objects to be created in the practical classes must be sub-classes of **CGFObject** class
- **CGFlight (+)** - Stores some information associated with a light source (can be extended by sub-classes to implement additional features)
- **CGFcamera (+)** - Stores information associated with a camera

For the correct development of the different practical exercises, it is expected that **the classes marked with (*)**, in order to implement the scenes, interface and objects required in each of exercise, as exemplified in the next section.

The classes marked with **(+)** are **example utility classes, not exhaustive**, to be instantiated to facilitate the management and storage of associated entities (however, they can be extended by sub-classes, if they wish to add functionality). The library also includes some pre-defined objects, such as axes (**CGFaxis**), and some auxiliary classes, although these will not be necessary for the first practical CGRA work.

Basic interaction

In terms of interaction, by default it is possible to manipulate the view using the mouse as follows:

- **Left button** - rotate the scene around the origin
- **Central button** (wheel pressed) - zoom in / out; alternatively, it can be used **CTRL + Left**
- **button Right button** - “slide” the camera laterally

Exercise's base code

The base code provided for the the first practical class' exercise extends the classes referred to in the previous section, in order to implement the drawing of a very simple scene.

The **MyScene** class extends **CGFscene**, and implements the **init()** and **display()** methods:

- **init()**: Contains code that is executed only once at the beginning, after the application's initialization. This is where variables are typically initialized, objects are created, or intensive calculations are made, the results of which can be stored for later reference.
- **display()**: Contains the code that effectively draws the scene repeatedly. This method will be the focus of this first work.

Read carefully the comments available in the code for these two methods, as they provide important information about their operation and use.

In particular, the sample code contained in the **display()** method is divided into three sections:

- Initialization of the background, camera and axis
- Geometric transformations
- Design of primitives


The **MyInterface** class extends **CGFInterface**, and implements the **init()** method:


- **init ()**: In this function, the graphical interface is initialized. It is possible to add text input, checkboxes, sliders, dropdown menus, among other elements that can be used to interact with the created scene. More information on its use is available at:

<http://workshop.chromeexperiments.com/examples/gui>.

Practical work

The following points describe the topics covered during this practical class, as well as the tasks to be performed.

Some of the tasks are noted with the icon  (image capture). At these points, you should capture and save an image of the application in the browser (e.g., using Alt-PrtScr on Windows or Cmd-Shift-3 on Mac OS X to capture to the clipboard and then save to file in an image management application of your choice). At the end of each class, students must rename the images to the format "**ex1-t<class>g<group>-n.png**", where **class** corresponds to the class number (e.g., 2MIEIC01 is class '**01**'), and **group** corresponds to student group number (defined in the TP group file in Moodle), and **n** corresponds to the number provided in the exercise (eg "**ex1-t1g01-1.png**").

In tasks marked with the icon  (code), students must create a **.zip file from the folder that contains your code (typically in the 'ex1' folder, if you have code being used in other folders, include it too)**, and name it "**ex1-t<class>g<group>-n.zip**", (with class, group and provided number identified as described above "**ex1-t1g01-1.zip**").

At the end (or throughout the work), one of the elements must submit the files via Moodle, through the link provided for that purpose. Only one member of the group will need to submit the work.

1. Use of the Graphical Interface (GUI)

The graphical interface (GUI) allows you to interact with the created scene, providing different types of elements to control certain aspects of the said scene.

The GUI appears as a collapsible menu on the scene page, in the upper right corner. In the base code provided with the practical work, there is a **checkbox** controller which controls the visibility of the coordinate axis.

For this purpose, the scene was prepared to show the axes conditionally in the **MyScene** class, by:

- Initializing a variable in the scene - *showAxis* - in the **init()** function with a boolean value,
- Implementing the functionality that changes the scene in the **display()** function according to the variable's current value.

The graphical interface is defined in the **MyInterface** class (sub-class of *CGFInterface*), as follows:

- Initializes the GUI object (class **dat.GUI** (<http://workshop.chromeexperiments.com/examples/gui>)) in the **init()** function,
- Adds a controller to the GUI, associated to the scene and its variable *showAxis*. Since it was initialized with a Boolean value, the GUI infers that it must be represented by a **checkbox**.

2. Basic geometry and structuring

The design of objects in **WebGL** and in modern versions of **OpenGL** is typically based on the definition of a set of triangles with a set of associated characteristics.

These triangles are defined by a set of vertices (and possibly some characteristics associated with each vertex), and the way in which the vertices connect to form the triangles.

In the base code provided with the practical work, the code necessary to create a **rhombus (MyDiamond.js)** is provided, and included in the scene so that it can be visible.

Consider that this diamond is defined by the vertices A, B, C and D, which form two vertex triangles ABC and DCB (Figure 1).

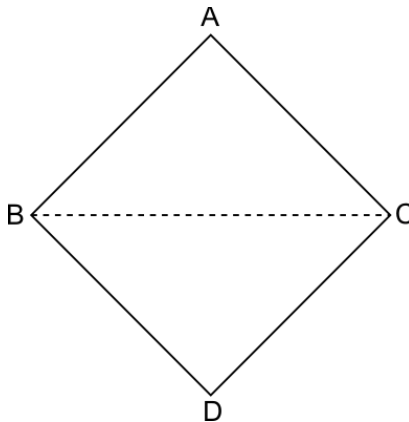


Figure 1: Example geometry (quadrilateral).

To create this geometry, we first have to create an *array* of vertices with the coordinates of the four corners.

```

vertices = [
    xA, yA, zA,
    xB, yB, zB,
    xC, yC, zC,
    xD, yD, zD
];

```

Next, we must indicate how these vertices will be connected together to form triangles.

For this, we created a new *array* that indicates, through the indices that refer to the order of the vertices, how to group them three to three. In this case, considering that the triangles are defined by the order ABC and DCB, we will have:

```

indices = [
    0, 1, 2,
    3, 2, 1
];

```

The use of indices allows to reduce the amount of information needed to define the geometry. Instead of repeating 3 coordinates in the list of vertices when the same vertex is used more than once, we just repeat its index in the list of indices.

The greater the geometry and the number of shared vertices (something quite common in 3D models composed of a mesh of triangles), the more benefit there is in using the indices to represent connectivity.

Taking this into consideration, the actual process of drawing the geometry implies passing this defined information (vertices, indices, ...) declared in **JavaScript** to **WebGL** buffers (already allocated in the graphics memory), and instructing it to draw the geometry considering their connectivity as sequences of triangles.

In **WebCGF**, the complexity of this final drawing phase is encapsulated in the **CGFObject** class. Thus, to create a particular 3D object, we can simply:

- create a sub-class **CGFObject**, e.g., **MyObject**
- implement the **initBuffers()** method, where we:
 - declare the above-mentioned arrays,
 - invoke the **initGLBuffers()** method so that the information is passed to **WebGL**
- In our scene:
 - Create and initialize an instance of the new object in the scene's **init()** method
 - Invoke the **display()** method for this object in the scene's **display()** method

In the **MyDiamond** class, a complete example of this process is presented, which will serve as the basis for creating other geometries.

Exercises

Exercise 1

1. Create a new subclass of *CGFobject* named ***MyTriangle***, in a new file named ***MyTriangle.js***, which defines a right triangle in the XY plane, with a side length of two units, as shown in Figure 2, where the red point is the origin (0, 0, 0).
NOTE: Follow the naming guidelines of the classes indicated in this paragraph and in all others.
2. In the GUI defined in the ***MyInterface*** class, add two ***checkbox*** controllers, which should control the visibility of the rhombus already shown in the scene, as well as the triangle created in the previous paragraph.
3. Create a new *CGFobject* subclass called ***MyParallelogram***, which creates a parallelogram as shown in Figure 3, with the left-most vertex at the origin of the scene (0.0,0). Its height is 1 unit, and the total width is 3 units. It should be *double-sided*, that is, visible on both sides (suggestion: repetition and order of indices should be explored).
4. Add another *checkbox* to control the visibility of the parallelogram. (📷 1) (📄 JS 1)

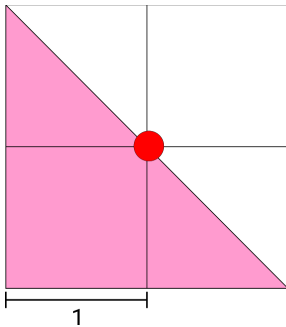


Figure 2: Right triangle
(***MyTriangle***)

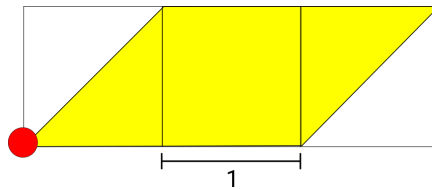


Figure 3: Parallelogram
Invertible(MyParallelogram)

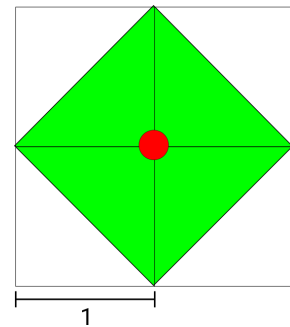


Figure 4: Provided rhombus
(***MyDiamond***)

Exercise 2

Create the additional pieces shown in the figures 5 and 6.

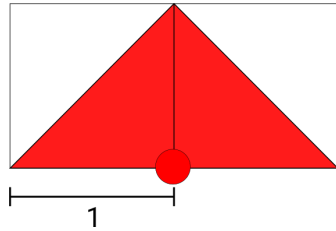


Figure 5: Small right triangle
(MyTriangleSmall)

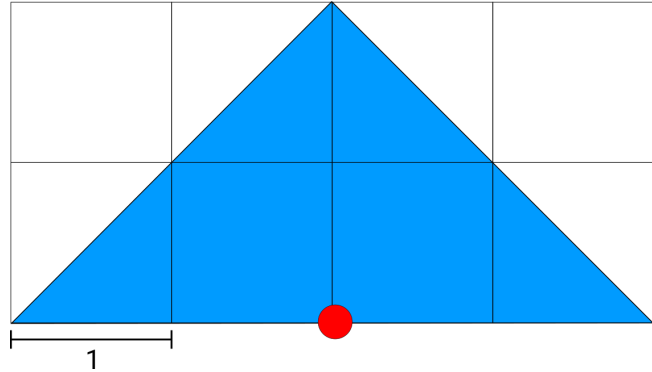




Figure 6: Large right triangle
(MyTriangleBig)

Checklist

Until the end of the work you should save and later submit the following images and versions of the code via Moodle, **strictly respecting the naming rule**:

-  Images (1): 1 (named as "ex1-t <class>g<group>-n.png")
-  Code in zip file (1): 1 (named as "ex1-t <class>g<group>-n.zip")