# Computer Graphics Labs

Abel J. P. Gomes

## LAB. 4

Department of Computer Science and Engineering
University of Beira Interior
Portugal
2020

Copyright © 2009-2020 All rights reserved.

# LAB. 4

## WINDOWS AND VIEWPORTS

1. Learning goals
2. Getting started
3. 2D viewing and world-to-screen mapping
4. Example
5. Programming exercises
6. Interesting web links

# Lab. 4

# WINDOWS AND VIEWPORTS

This lecture continues with rendering of 2D scenes. But, now we use two or more viewports to render one or more scenes.

## 1. Learning Goals

At the end of this chapter **you should be able to**:

1. Explain how a scene within a domain window is mapped onto a screen viewport. This process involves 1 translation in the domain, 1 scaling between the domain window and the screen viewport, and 1 translation in the screen, a process known as window-viewport mapping.
2. Understand how this window-viewport mapping is done in an OpenGL program.
3. Write a graphics program to map a 2D scene onto two or more viewports.
4. Program a reshape callback to automatically keep the aspect ratios in the window-viewport mapping.

## 2. 2D Window-to-Viewport Mapping

Let us now see the maths behind the window-viewport mapping. Let us consider a domain window in $\mathbb{R}^2$ defined by $(x_{min}, y_{min}, x_{max}, y_{max})$ and a point $(x, y)$ such that $x \in [x_{min}, x_{max}]$ and $y \in [y_{min}, y_{max}]$. Let us also consider a viewport in a screen window given by $(X_{min}, Y_{min}, X_{max}, Y_{max})$.
Now, the problem is to determine the pixel $(X, Y)$ within the viewport that is located at the same relative position as $(x, y)$ within $(\boldsymbol{x_{min}}, \boldsymbol{y_{min}}, \boldsymbol{x_{max}}, \boldsymbol{y_{max}})$. That is, mapping a 2D point $(x, y)$ of the domain window onto a pixel $(X, Y)$ of a screen window (or viewport) requires to maintain the relative position of such a pixel within the viewport.
To make sure that the relative positions of $(x, y)$ and $(X, Y)$ are maintained, the following relationships must be satisfied:

$$\frac{x - \boldsymbol{x_{min}}}{\boldsymbol{x_{max}} - \boldsymbol{x_{min}}} = \frac{X - X_{min}}{X_{max} - X_{min}}$$

and

$$\frac{y - y_{min}}{y_{max} - y_{min}} = \frac{Y - Y_{min}}{Y_{max} - Y_{min}}$$

from where we can obtain the expressions of the pixel $(X, Y)$ from the point $(x, y)$ as follows:

$$X = X_{min} + \frac{X_{max} - X_{min}}{x_{max} - x_{min}}.(x - \boldsymbol{x}_{min})$$

and

$$Y = Y_{min} + \frac{Y_{max} - Y_{min}}{y_{max} - y_{min}}.(y - y_{min})$$

That is, the window-viewport mapping involves a translation $(-\boldsymbol{x}_{min}, -y_{min})$ in $\mathbb{R}^2$, a scaling $(S_x, S_y)$, and a translation $(X_{min}, Y_{min})$ in the screen. Thus, $S_x = (X_{max} - X_{min})/(x_{max} - x_{min})$ and $S_y = (Y_{max} - Y_{min})/(y_{max} - y_{min})$ are the scaling factors.

Note that keeping the relative positions does NOT mean to keep aspect ratios (or proportions). If the domain window and the viewport do not have identical aspect ratios, the image appears deformed on screen.

# 3. Windows and Viewports in OpenGL

As known, OpenGL (Open Graphics Library) is a cross-platform, hardware-accelerated, language-independent, standard API that we use to produce 2D and 3D scenes, i.e., OpenGL is the software interface to graphics hardware.

## 3.1. Graphics libraries and functions

We use the following libraries in our OpenGL programs:

1. **OpenGL (GL)**: consists of hundreds of functions, which start with the prefix "gl" (e.g., glDrawArrays, glBindBuffer, glGenBuffers, glDeleteBuffers, glViewport). It is this core that allows us to model objects and build up scenes using a set of geometric primitives like, for example, point (GL_POINTS), straight line segment (GL_LINES), and polygon (GL_POLYGON).
2. **The OpenGL Extension Wrangler Library (GLEW)**: is a cross-platform C/C++ library designed to query and load OpenGL extensions. That is, GLEW provides runtime mechanisms that determine which OpenGL extensions are supported on your computer. In a way, GLEW provides platform independency (e.g., Windows, Linux, and Mac OSX), as well as compatibility across OpenGL extensions.

   http://glew.sourceforge.net/

1. **Graphics Library Framework (GLFW)**: is a provides a lightweight utility library that enables us to create and manage windows and OpenGL contexts. It also allows for handling joystick, keyboard and mouse input. However, it does not allow us to create buttons and menus.

   https://www.glfw.org/

2. **OpenGL Mathematics** (**GLM**): is a header only C++ mathematics library for graphics programming. GLM provides a set of classes and functions designed and encoded using the same naming conventions and functionalities than GLSL (OpenGL Shading Language).

   https://glm.g-truc.net/0.9.9/index.html

   GLM manual:

   https://chromium.googlesource.com/external/github.com/g-truc/glm/+/refs/heads/master/manual.md

3. **OpenGL Shading Language (GLSL**): is a C/C++ similar high-level programming language to program different pipeline parts of the graphics card. GLSL programs are called *shaders*, which run on the GPU. There are types of shaders, namely: vertex shaders, fragment shaders, geometry shaders, and so forth.

   https://learnopengl.com/Getting-started/Shaders

   https://www.youtube.com/watch?v=uOErsQljpHs

   https://www.youtube.com/watch?v=oyFFTgFcfDo

## 3.2. OpenGL/GML orthographic projection function

GLM provides both orthographic and perspective projections. However, in this lab, we shall handle orthographic projections only, as needed for 2D scenes.

The 2D orthographic projection is defined by the following GLM function prototype:

```
glm::mat4 glm::ortho(
       float left, float right, float bottom, float top);
```

whose arguments represent the left, right, bottom, and top planes of the frustum. The viewer (or camera) is located at the infinite. By default, `left=-1.0`, `right=1.0`, `bottom=-1.0`, and `top=1.0`.

## 3.3. Window-viewport mapping in OpenGL

Let us recall that a viewport is defined as a region of the window that you wish to map the domain window defined by `glm:ortho`. Interestingly, we may define multiple viewports for a single domain window, so that we can display different parts of a scene into different viewports.

The OpenGL function prototype for creating a viewport is as follows:

```
glViewport(GLint x, GLint y, GLsizei width, GLsizei height );
```

where

1) `x`, `y`: denote the lower left corner of the viewport; the initial value is (0,0);
2) `width`, `height`: represent the width and height of the viewport in pixels.

To avoid distortion of an image when the window is resized, modify the aspect ratio of the projection to match the viewport. For example, assuming we want to maintain a 1:1 aspect ratio in our image, we have:

1) the viewport defined by `glViewport(0,0,400,400)` and the domain window defined by `glm::ortho(-1.0,1.0,-1.0,1.0)` hold the 1:1 ratio.
2) the viewport defined by `glViewport(0,0,400,200)` and the domain windows defined by `glm::ortho(-2.0,2.0,-1.0,1.0)` and `glm::ortho(-1.0,1.0,-0.5, 0.5)` also hold the 1:1 ratio.

## 3.4. Windowing functions in GLFW

We use GLFW **to create a window** as follows:

```
GLFWwindow* glfwCreateWindow (
       int                 width,
       int                 height,
       const char *        title,
       GLFWmonitor *       monitor,
       GLFWwindow *        share
       )
```

where

1) `width`, `height`: denote the width and height of the window;
2) `title`: is a string that appears as the title of the window;
3) `monitor`: the monitor to use for full screen mode, or `NULL` to use windowed mode;
4) `share`: the window whose context to share resources with, or `NULL` to not share resources.

**To set the size of the window**, we use the following GLFW function:

```
void glfwSetWindowSize (
      GLFWwindow * window,
      int *        width,
      int *        height
      )
```

where
1) `window`: the target window;
2) `width`, `height`: denote the desired width and height for target window.

**To get the size of the window**, we use the following GLFW function:

```
void glfwGetWindowSize (
      GLFWwindow * window,
      int *        width,
      int *        height
      )
```

where
1) `window`: the target window;
2) `width`, `height`: denote the width and height of the target window;

**To resize a window**, we use two GLFW functions. The first function carries out the registration of a callback that resizes the target window, and is as follows:

```
GLFWwindowsizefun glfwSetWindowSizeCallback (
      GLFWwindow *       window,
      GLFWwindowsizefun   cbfun
      )
```

where
1) `window`: the target window;
2) `cbfun`: denotes the callback function.

The second function, the callback `cbfun`, is called whenever the `window` is resized by the user interactively. This callback function is a C function, much like we do in the legacy OpenGL. A typical callback prototype is:

```
// function to be called
void myWindowsResizeCallback (
      GLFWwindow *       window,
      GLint              width;
      GLint              height;
      )
```

For more details about GLFW window handling, the reader is referred to:

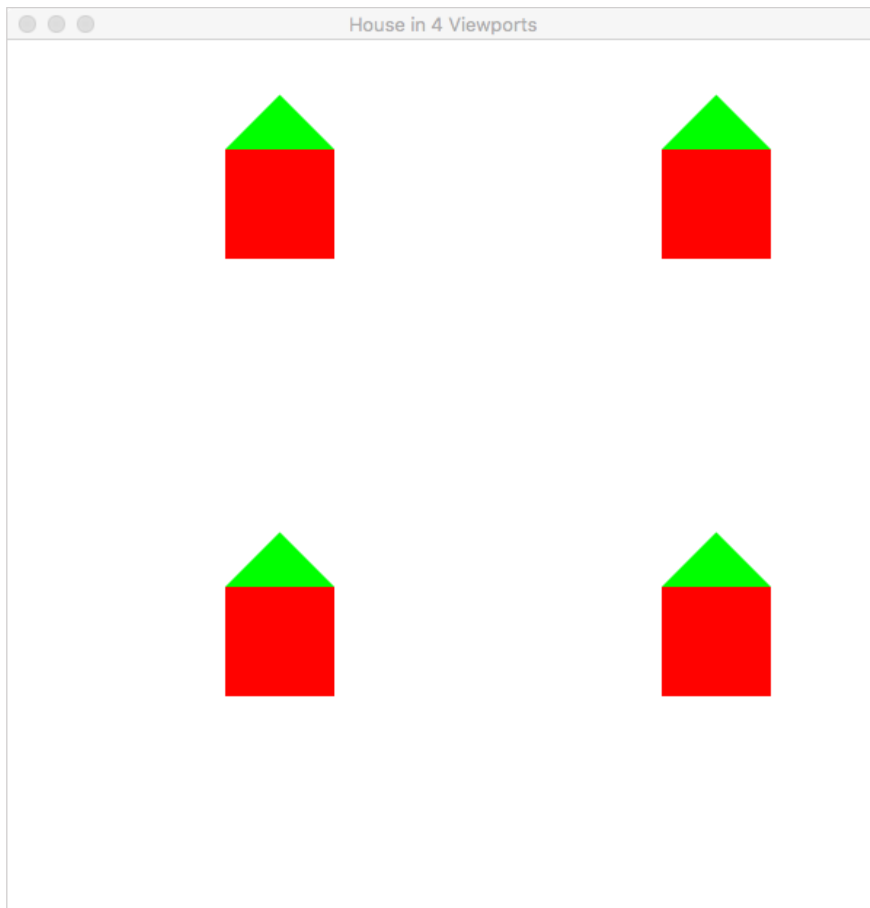
https://www.glfw.org/docs/3.0/group__window.html#gaa40cd24840daa8c62f36cafc847c72b6

## 4. Example

A program to draw the same house in four viewports is available at:

http://www.di.ubi.pt/~agomes/cg/praticas/house4viewports.zip

which produces the following output:



## 5. Programming Exercises

1. Write a program to draw the *cos* function on a viewport and the *sin* function on another viewport.
2. Re-write the previous program to add a third viewport displaying a bouncing ball.
3. Re-write the house-building program (with a single viewport) implemented in the previous classes to simulate the panning operation through the arrow keys (GLFW_KEY_UP, GLFW_KEY_DOWN, GLFW_KEY_LEFT, and GLFW_KEY_RIGHT) of the keyboard. For that purpose, we should move the domain window around the house. The movement step in each direction is 5 percent of the domain window's width. See the following pages to learn about keyboard input events and callbacks:
   https://www.glfw.org/docs/3.0/group__keys.html#gaac6596c350b635c245113b81c2123b93
   and
   https://www.glfw.org/docs/3.0/quick.html#quick_key_input
   As for any GLFW window, you need to implement your keyboard callback, and then to register it with a specific GLFW registration function. This registration function is named `glfwSetKeyCallback`, as described at:
   https://www.glfw.org/docs/3.0/group__input.html#ga7e496507126f35ea72f01b2e6ef6d155
4. Re-write the house-building program above to add the zooming operation. The zoom-in operation should be controlled by the '+' key, while the zoom-out operation should use the '-' key.
5. Add the reshaping facility to the previous program.
6. Add the full-screen facility to the previous program.