0 The CGV Framework

The CGV Framework is a collection of strongly coupled C++ libraries meant for rapid prototyping of performance-intensive applications in the fields of visualization and computer graphics research, using modern (version 3+) OpenGL. It implements common tasks such as window and graphics context creation, event handling, GUI creation and interaction, and many common computer graphics algorithms and data structures.

It provides a *viewer* application, which you write *plugins* for. This first exercise is meant for you to familiarize yourself with the plugin interface and its class hierarchy, as well as the most commonly used utilities provided by the Framework.

0.1 GUI, reflection and config files (2 points)

The file `cgv_demo.cxx` contains a simple example plugin. This plugin makes use of the most important concepts of the API. It defines a main class, `cgv_demo`, which implements several interfaces declared by the Framework. In implementing these interfaces, this main class exposes itself via *reflection*, defines a GUI, and interacts with the rendering system to draw its scene. At the end of the file, an instance of this class is registered with the Framework object system using the most common one of several available *registration helpers*.

Because the class supports reflection, its properties can be set externally. This enables the use of configuration files, where all exposed properties of all known classes can be set to arbitrary values at program startup without having to recompile anything. Exercise 1 comes with such a config file - it's called `config.def` and resides next to the other source files of this exercise.

Study both `cgv_demo.cxx` and `config.def` carefully. You will have to model your solution after the code provided to you in these two files.

You will notice that in the method `cgv_demo::init(context &)`, a large part of the code (starting at line 361) deals with creating a GPU-side vertex buffer and vertex array object (the Framework calls its abstraction for this an `attribute_array_binding`) containing the geometry of the unit square. This array is not currently being used by the `cgv_demo` class. Instead, one of the built-in tesselation methods provided by the Framework in its graphics `context` is being employed to render the quad. Your task is to add a toggle to `cgv_demo` that enables switching on and off the usage of this vertex array for rendering. For this, you will have to add a suitable class member (**0.5 points**).

The method `cgv_demo::draw_my_unit_square(context &)` (line 521) uses this vertex array to render the quad. You can call this method from within `cgv_demo::draw(context &)` when your toggle is enabled (**0.5 points**). To make your toggle fully functional, you will also have to add it to the properties that are accessible via reflection and provide a suitable GUI control (**0.5 points**). Finally, add an entry to the config file, setting the toggle to a different state than what you initializate it to during class instance construction (**0.5 points**).

0.2 Plugin creation (3 Points)

Your next task will be to create a plugin (almost) from scratch. This plugin will display a snowflake-like fractal of translated and scaled cubes, with some simple configuration options.

a) Adapting the code from `cgv_demo.cxx`, define a main class that supports reflection, provides a GUI and can interface with the rendering system. Also register it with the Framework object system so that the class gets created on plugin load (**0.5 point**).

b) Next, make use of the auxillary class `cubes_fractal` declared in `cubes_fractal.cxx` and `cubes_fractal.h` to render the cube structure. Note that `cubes_fractal` needs a different shader than the default shader used in `cgv_demo.cxx` for lighting to work. You can use `context::ref_surface_shader_progr` to obtain another default shader from the Framework that fits all requirements. Make the recursion depth and material color of the root cube configurable via GUI and config file (**1 point**).

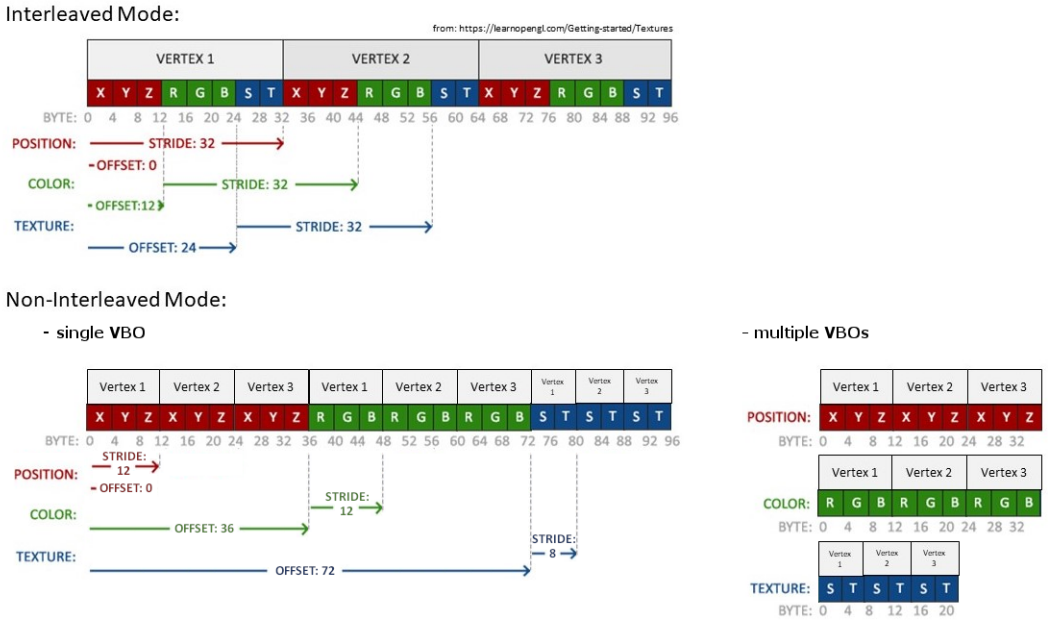

Figure 1: Interleaved vs. non-interleaved storage modes. You may decide which variant of non-interleaved storage you implement.

c) Finally, `cubes_fractal` supports using a vertex array (again, in the form of the CGV Framework abstraction `attribute_array_binding`) for rendering. The vertex array to be used can be set via `cubes_fractal::use_vertex_array`. If no vertex array is specified (the default), it makes use of the built-in cube tesselation provided by the graphics `context`. This built-in tesselation statically stores the geometry in main memory, which causes data upload overhead everytime a cube is drawn. Adapt the code in `cgv_demo.cxx` to create a vertex array object of a unit cube. The vertex buffer bound by the array does not need to contain texture coordinates this time, but it should contain vertex normals (**0.5 points**). The way the vertex buffer in `cgv_demo.cxx` is set up is called *interleaving*, since the vertex attributes are stored in a per-vertex manner (see Figure 1). Additionally provide an array that stores the attributes

in a non-interleaved manner, and make which of the three drawing modes to use (built-in, interleaved or non-interleaved) an option that can be set from the GUI or via config file (**1 point**).

0.3 Bonus tasks (max. 5 Points)

- **0.3.1** Recreate the cubes structure of task 0.2 without using the `cubes_fractal` auxillary class - write all geometry to a single vertex buffer to eliminate the overhead of the many draw calls and compare the performance (**3 points**).
- **0.3.2** Look at the default shaders in `framework/shader/cgv_gl/glsl`. The ones you use in tasks 0.1 and 0.2 are `textured_default.glpr` and `default_surface.glpr` respectively. Using those as a reference, implement a custom shader that supports both texturing and lighting, built and link it using Framework facilities (namely `cgv::render::shader_program`) and modify either of the two drawables of this exercise to make use of your shader (**3 points**).
- **0.3.3** As a more efficient variant of bonus task 0.3.1, implement instanced rendering for drawing the cubes fractal. You will need a custom shader program to accomplish this. For maximum efficiency, tessellate the actual cubes in a geometry shader (**4 points**).

Submission

Put all files in the folder `exercise0` that you modified or added in a Zip archive (named according to the pattern `[YourName]_ex0.zip`) and upload that archive to Opal. Make sure that your solution runs on the type of system used for evaluation (Windows, Visual Studio $\geq$ 2015). If you do not have access to such a system and are in doubt whether your solution will work in this configuration, contact your tutor or supervisor.