Tutorial 4: Implementing 2D Interactive Camera

Note to the reader

> Your objective is to write code so that your application behaves similar to the sample or better. The tutorial is verbose only because it assumes minimal previous experience in computer graphics and the OpenGL toolbox. If you're so inclined, you can skip this tutorial and instead play with the sample, and then read the submission guidelines and rubrics to ensure your submission is graded fairly and objectively. Or, you can pick-and-choose which portions to read and which to ignore with the disadvantage that continuity and comprehension might be lost. Or, just read the entire tutorial and it is possible you may learn one or two things.

Topics Covered

- Understand 2D affine transforms including scale, rotation, and translation transforms and 2D change-of-coordinate system transforms.
- Implement aspects of the 2D graphics pipe including model-to-world, world-to-view, and view-to-NDC transforms.
- Understand and implement an interactive camera for 2D applications.
- Implement a data-driven graphics application.

Prerequisites

- You must complete worksheets on 2D affine transformations.
- You must complete Tutorial 3 as a prerequisite for this project. Much of the code implemented in previous tutorials can be refactored into this tutorial.

First Steps

Overwrite the existing batch file csd2101.bat in csd2101-opengl-dev with a new version available on the assessment's web page. Execute the batch file.

1. Choose option **4 - Create Tutorial 4** to create a Visual Studio 2022 project tutorial-4.vcxproj in directory build with source in directory projects/tutorial-4 whose layout is shown below:

```
csd2101-opengl-dev/      # OpenGL sandbox directory
├─ build/                # Build files will exist here
├─ meshes/               # Mesh files (.msh, .obj) etc.
├─ scenes/               # Scene files (.scn)
├─ projects/             # Tutorials and assignments
|  └─ tutorial-4         # Tutorial 4 code exists here
|     └─ include         # Header files - *.hpp and *.h files
|     └─ src             # Source files - *.cpp and .c files
|     └─ shaders           # Shader files - .vert and .frag
|        └─ my-tutorial-4.vert # Vertex shader file
|        └─ my-tutorial-4.frag # Fragment shader file
└─ csd2101.bat                # Automation Script
```

Source and shader files are pulled into nested directory /projects/tutorial-4/src while header files are pulled into nested directory /projects/tutorial-4/include. This project makes use of external resource files for geometry and scene management that are pulled into directories /meshes and /scenes, respectively.

2. This tutorial requires the source and shader code that you implemented for Tutorial 3. Therefore, ***copy all files*** from /projects/tutorial-3/src and /projects/tutorial-3/include to /projects/tutorial-4/src and /projects/tutorial-4/include, respectively.

3. As the number of shaders involved in rendering a scene proliferate, it is not reasonable to define a global `std::string` object for each shader source [in glapp.cpp]. Rather than embedding shader source code as raw string literals, we prefer using the interface provided by class `GLSLShader` to directly read a pair of files [containing vertex and fragment shader source code], compile them separately, and link them together to create a shader program [that is referenced by a member of class `GLSLShader`]. Using shader source code embedded in text files removes potential errors caused by the proliferation of global objects. For easier management of source code, shader files for this and subsequent tutorials will be separated from C++ source code and placed in directory /projects/tutorial-<N>/shaders [where <N> is the tutorial number, e.g, for this tutorial, shader files are located in /projects/tutorial-4/shaders].

   Move the vertex and fragment shader source files from /projects/tutorial-4/src/my-tutorial-3.* to /projects/tutorial-4/shaders/my-tutorial-4.* and remove the C++ raw string literal notation `R"(...)"` from the moved shader sources. The vertex shader source in my-tutorial-4.vert will now look like this:

```
#version 450 core

layout (location=0) in vec2 aVertexPosition;
layout (location=1) in vec3 aVertexColor;
layout (location=0) out vec3 vColor;

uniform mat3 uModel_to_NDC;

void main() {
  gl_Position = vec4(vec2(uModel_to_NDC *
                          vec3(aVertexPosition, 1.f)), 0.0, 1.0);
  vColor = aVertexColor;
}
```

while the fragment shader source in my-tutorial-4.frag will now look like this:

```
#version 450 core

layout (location=0) in vec3 vInterpColor;
layout (location=0) out vec4 fFragColor;

void main () {
  fFragColor = vec4(vInterpColor, 1.0);
}
```

Next, remove references to all global `std::string` objects encapsulating the [now removed] raw string literals in file glapp.cpp:

```
// Remove all of the following string literals!!!
std::string const my_tutorial_3_vs {
  #include "my-tutorial-3.vert"
};
std::string const my_tutorial_3_fs {
  #include "my-tutorial-3.frag"
};
std::string const my_red_fs   {
  #include "my-red.frag"
};
std::string const my_green_fs {
  #include "my-green.frag"
};
std::string const my_blue_fs  {
  #include "my-blue.frag"
};
```

The use of `GLSLShader`'s interface to create a shader program from the vertex and fragment shader source code in files my-tutorial-4.vert and my-tutorial-4.frag is explained here.

Application Behavior

1. Begin by running the sample executable in directory ./sample+4/executable to understand the requirements, scope, and behavior of this tutorial.
2. At startup, the application displays a large 2D world consisting of objects that are instantiations of triangle, box, and circle geometrical models. Unlike previous tutorials, this application is data-driven. Scene layout information such as objects contained in the scene, their parameters, and name of model of which the object is an instantiation are parsed from metafile /scenes/tutorial-4.scn. Geometries of models are parsed from .msh mesh files located in directory /meshes.

```
csd2101-opengl-dev/      # Your solution folder $(SolutionDir)
├─ meshes/               # 3D mesh file likes .obj, .msh files
|  └─ circle.msh         # ◯ mesh
|  └─ square.msh         # □ mesh
|  └─ triangle.msh       # △ mesh
├─ scenes/               # Scene reside here .scn files
|  └─ tutorial-4.scn     # 🚗★✈ world objects
├─ projects/             # All tutorials and assignments
├─ test-submissions      # ⚠[IMPORTANT] verify submission
└─ csd2101.bat # 🔲 Automation Script
```

3. At startup, the world is rendered from an interactive camera's point of view taking into account only the camera's position but not its orientation. That is, the scene is rendered using a world-to-camera view transformation matrix that takes into account only the camera's position but not its orientation. Such a camera is called a *free camera*. The spearhead-like triangular object rendered in black color at the center of the viewport represents the camera object. Think of the camera object as an ordinary object with a camera embedded into it. According to the information printed on the display window's toolbar, the camera is located at position $(-19500, -19700)$ in the world. The direction in which the camera is looking is represented by the orientation of the spearhead's tip. Recall, from lectures that the camera's *view vector* or *up vector* is defined as the direction in which the camera is looking. The application specifies the camera's view vector using angular displacement with respect to default *up* direction of the scene which is the world coordinate system's $y-$axis. At startup, the camera's angular displacement is parsed from the scene file and is determined to be 30°. Using this angular displacement, the camera's view vector is computed as $\hat{v} = \langle -\sin 30^\circ, \cos 30^\circ \rangle = \left\langle \frac{-1}{2}, \frac{\sqrt{3}}{2} \right\rangle$.

4. Keyboard buttons $\mathbf{H}$ and $\mathbf{K}$ update the interactive camera's orientation by increasing and decreasing its angular displacement by 2°, respectively. This effectively allows the user to interactively control the direction in which the camera is looking by changing the camera's view vector. The angular speed of 2° per frame is a per-object value that is specified in the scene file.

5. Keyboard button $\mathbf{U}$ displaces the camera in the direction it is looking, that is, the camera is displaced along its view vector. The camera's linear speed is defined as part of the camera structure. The sample application uses a value of 2 units per frame as the linear speed.

6. Recall from class lectures that a camera is modeled as the center of an oriented rectangle in the world. The rectangle represents a *window to the world* or a *view finder*, that is, the rectangle's dimensions determine how much of the world is visible from the camera. The interactive camera has a *zoom* feature. Use keyboard button $\mathbf{Z}$ to *zoom out* [by increasing the window's dimensions thereby allowing the user to see more of the world] or *zoom in* [by decreasing the window's dimensions thereby allowing the user to see less of the world].

7. The interactive camera described so far represents a *free camera*, that is, the *camera-to-world* [or *view* or *eye*] transformation is computed using only the camera's position but not its orientation. That is, even as the camera's view vector is arbitrarily changed [using buttons $\mathbf{H}$ and $\mathbf{K}$], the camera's window [to the world] will be an axis-aligned rectangle. Button $\mathbf{V}$ toggles the interactive camera's behavior from a *free* to *first-person camera*. Unlike a free camera, a first-person camera's window is an oriented rectangle. Such a camera renders the scene using a world-to-camera view transformation matrix that is computed using both the camera's position and its orientation.

8. Notice that the sample program's toolbar displays information related to the camera: current position in the world, angular displacement with respect to the world's *up* vector [that is, the $y-$axis], and window height.

🎯Task 0: Understanding input files

The application is data-driven and this section describes certain details necessary for initializing the application.

Understanding scene file

Recall from Tutorial 3 that a scene is composed of objects. Further, each object encapsulates state that specifies the model instanced by the object, the shader to render the model, and parameters to compute the model transformation matrix such as scale factors, orientation parameters, and position coordinates. The run-time application begins by parsing a textual metafile /scenes/tutorial-4.scn that contains information necessary to describe a scene. The first line of this file describes the number of objects in the scene.

The second and subsequent lines of the file describe individual objects and their initial states. Per-object initial states consist of six parameters with each parameter described on a new line. A typical fragment from the file describing an object's parameters looks like this:

```
1 square          # name of model instanced by object
2 Object1         # name of current object
3 tutorial4-shdrpgm ../projects/tutorial-4/shaders/my-tutorial-4.vert
  ../projects/tutorial-4/shaders/my-tutorial-4.frag # shader program
4 1.0 0.0 0.0     # RGB color for rendering object
5 200.0 150.0     # scale factors
6 0.0 0.6         # angular displacement and angular speed both in degrees
7 -19800 -20000   # initial position of object
```

1. Line 1 contains a string indicating the name of the model instanced by the object. The model's name provides a reference to the file containing the model's geometry. If a model with this name already exists in the container of models, this string is ignored. Otherwise, a model is instantiated from the corresponding mesh file [located in directory /meshes] and inserted into the singleton container used by the application as a repository for models. In either case, the object being instantiated will keep a reference to this model in the container for later use in rendering the model's geometry. In this example, since the line 1 contains string `"square"` and line 2 contains string `"Object1"`, the current object is named Object1 and is declared as an instantiation of a model named square [whose geometry is described in file /meshes/square.msh].
2. Comments can begin at any column in a line and are prefixed with character `#`.
3. Line 2 contains a string indicating the object's name. The ability to identify individual objects using unique names comes in handy in certain scenarios. For instance, a camera can be embedded into an object by assigning the special name Camera to the object.
4. Line 3 [including the wrapped text on the next line] contains three strings. The first string `"tutorial4-shdrpgm"` indicates the name of a shader program that will be used by the application program to render the geometry associated with the model instanced by the object. The second string [`"../shaders/my-tutorial-4.vert"`] and third string [`"../shaders/my-tutorial-4.frag"`] provide relative paths to vertex and fragment shader files, respectively. The contents of the vertex and fragment shader files specified by these

paths are used to compile, link, validate a shader program with the name specified by the first string. If a shader program with this name already exists in the singleton container of shader programs, these three strings are ignored. Otherwise, a shader program is compiled, linked, validated, and then inserted into the container. In either case, the object being instantiated will make a reference to this shader program in the container of shader programs for later use in rendering the object. The advantage of this system is that $m \times n$ shader programs could be created - in theory - from m vertex shaders and n fragment shaders. In this example, shader program tutorial4-shdrpgm is compiled, linked, validated from vertex shader code in file $/\text{tutorial-4/shaders/my-tutorial-4.vert}$ and fragment shader in $/\text{tutorial-4/shaders/my-tutorial-4.frag}$, and then inserted into a singleton container. Next, the object being instantiated will keep a reference to this shader program that will be later used by the application to render the model named square.

5. Line 4 contains a $3-$tuple specifying RGB components of the color used to render the object. The three color components are specified as floating-point values in range $[0.0, 1.0]$.
6. Line 5 contains a $2-$tuple specifying floating-point scale factors along $x-$ and $y-$axes. These scale factors will be used to size the instance of the model cited on line 1.
7. Line 6 contains a $2-$tuple that describes the object's orientation: the first floating-point number specifies the object's initial angular displacement with respect to $x-$axis in *degrees*; the second floating-point number specifies the object's angular speed in *degrees per frame*. An object having angular speed of 0.0 will therefore be invariant to rotation.
8. Finally, line 7 contains a $2-$tuple that specifies the object's world position coordinates.

Understanding mesh files

A model's geometry is stored in a mesh file located in directory $/\text{meshes}$. For example, if the current object's parameters specify a model name square, then the model's geometry data is in file $/\text{meshes/square.msh}$. The contents of this file look like this:

```
1 n square    # prefix n indicates name
2 v -0.5 -0.5 # prefix v indicates vertex position attribute
3 v 0.5 -0.5
4 v 0.5 0.5
5 v -0.5 0.5
6 t 0 1 2  # prefix t means primitives are rendered as GL_TRIANGLES
7 t 2 3 0  # prefix f means primitives are rendered as GL_TRIANGLE_FAN
```

1. Every line in a $.\text{msh}$ file begins with a prefix string. The first line contains prefix string `"n"` indicating the model's name. Lines beginning with prefix string `"v"` indicate that these lines contain 2D vertex position attributes. Lines beginning with prefix string `"t"` describe index information for rendering `GL_TRIANGLES` primitives.
2. In addition to prefix strings `"n"` and `"v"`, mesh file $/\text{meshes/circle.msh}$ contains lines with prefix string `"f"` to indicate index information for rendering `GL_TRIANGLE_FAN` primitives. In this file, the first occurrence of prefix string `"f"` is on line 723 [of the file and not below]:

```
1 f 0 1 2 # the prefix f here indicates that the triangles must be
2 f 3     # rendered as GL_TRIANGLE_FAN with 0 as the pivot point.
3 f 4     # The triangles in this model are then rendered as a fan
4 f 5     # with the first triangle specified as 0, 1, 2;
5 f 6     # the second rendered as 0, 2, 3, and so on.
```

3. Line 723 [of the file] contains three integral indices describing the first triangle of the fan with the first index 0 indicating the fan's pivot vertex. Each successive triangle in this fan requires only an additional vertex. Lines 724 onward provides the additional vertex per triangle with each line preceded by string `"f"`.

4. While Tutorial 3 models had both vertex position and vertex color attributes, this tutorial does not require vertex color attributes. Instead, the .scn scene file specifies per-object RGB components of a color that will be used to uniformly render every primitive of the model instanced by the object.

Changes to structure `GLApp::GLModel`

1. You can continue to use the definition of `GLApp::GLModel` from Tutorial 3.

2. This tutorial uses a variety of models whose geometries must be parsed from .msh files located in /meshes. You will have to augment code from Tutorial 3 to parse geometry from a .msh file and set up a VAO to encapsulate vertex and element buffers. This can be done using member function(s) or any other way you wish.

3. Tutorial 3 used a `std::vector<GLApp::GLModel>` container as a repository for models. Since the first line of a .msh file contains prefix string `"n"` indicating the model's name, this tutorial will identify models through their names. Therefore, an associative array `std::map<std::string, GLApp::GLModel>` container will be used to store geometric models with the key identifying the model's name while the value is an instance of type `GLApp::GLModel`.

4. The updated `GLApp::GLModel` structure for this tutorial will look like this:

```
struct GLApp {
  // other stuff ...

  struct GLModel {
    GLenum     primitive_type;
    GLuint     primitive_cnt;
    GLuint     vaoid;
    GLuint     draw_cnt;

    // you could add member functions for convenience if you so wish ...
    void init(); // read mesh data from file ...
    void release(); // return buffers back to GPU ...
  };
  //static std::vector<GLApp::GLModel> models; // removed
  // added for tutorial 4: repository of models
  static std::map<std::string, GLModel> models; // singleton
};
```

Repository for shader programs

Different types of geometric models will require different types of shader programs. Tutorial 3 used a `std::vector<GLSLShader>` container as a repository for shader programs. As the number of shader programs increases, it is more convenient for designers, artists, and scripting tools to identify and access shader programs using names. Therefore, this tutorial will use a singleton container `GLApp::shdrpgms` of associative array type `std::map<std::string, GLSLShader>` as a repository for shader programs. The `GLApp` structure would now look like this:

```
struct GLApp {
  // other stuff ...

  //static void init_shdrpgms_cont(GLApp::VPSS const&); // removed
  // added for tutorial 4: repository of shader programs
  static std::map<std::string, GLSLShader> shdrpgms; // singleton
};
```

Changes to structure `GLApp::GLObject`

1. Since Tutorial 3 used `std::vector` containers for models and shader programs, an object of type `GLObject` referenced a specific model and a specific shader program using an index of type `GLuint`. Since this tutorial is using `std::map` containers as repositories for models and shader programs, the model reference must be updated to use an iterator of type `std::map<std::string, GLApp::GLModel>::iterator`. Likewise, the shader program reference must be updated to use an iterator of type `std::map<std::string, GLSLShader>::iterator`.
2. While the model geometry in Tutorial 3 contained both vertex position and vertex color attributes, this tutorial doesn't specify vertex color attributes. Instead, the .scn scene file defines RGB components of an object color. Since every triangle comprising this object's model will be uniformly rendered with this color, structure `GLApp::GLObject` will have to encapsulate the per-object color parsed from the `.scn` file.
3. One small matter relates to replacing data members `angle_disp` and `angle_speed` of type `GLfloat` from Tutorial 3 with a more convenient representation that uses data member `orientation` of type `glm::vec2`. In this tutorial, `orientation.x` and `orientation.y` will be the repositories of Tutorial 3 data members `angle_disp` and `angle_speed`.
4. Tutorial 3 required every object to compute a model-to-NDC transformation matrix. Recall that the purpose of this matrix is to map position coordinates of the model instanced by the object from model coordinate system to NDC system. To enable rendering of the referenced model, the matrix must be subsequently copied to the vertex shader. Tutorial 4 will continue the practice of computing the model-to-NDC transformation matrix. In addition, this tutorial will require every object to also compute and cache its model transformation matrix. An object's model [or model-to-world] transformation matrix maps coordinates of the model instanced by the object into the game world specified by the world coordinate system. Physics computations and interactions between objects such as collisions are conceptually simpler when all involved objects are described in a single and consistent coordinate system. The world coordinate system serves that purpose. Since the model transformation matrix is an important attribute for objects, structure `GLApp::GLObject` will have to encapsulate both model-to-NDC and model transformation matrices.
5. Recall that Tutorial 3 used a container of type `std::vector<GLApp::GLObject>` as a repository for models. Since the `.scn` scene file contains an entry for the object's name, it is more convenient for the application to identify objects through their names. Therefore, this tutorial will use associative array container `std::map<std::string, GLApp::GLObject>` as a repository for objects - the key is the object's name while the value is an instance of type `GLApp::GLObject`.
6. The updated `GLApp::GLObject` structure will now look like this:

```
struct GLApp {
```

```
  // other stuff ...

  struct GLObject {
    //GLfloat angle_disp, angle_speed; // removed in tutorial 4
    glm::vec2 orientation; // orientation.x is angle_disp and
                           // orientation.y is angle_speed
                           // both values specified in degrees

    // from tutorial 3
    glm::vec2 scaling;
    glm::vec2 position;
    glm::mat3 mdl_to_ndc_xform; // model-to-ndc transformation

    //GLuint mdl_ref; // removed in tutorial 4
    //GLuint shd_ref; // removed in tutorial 4
    // added in tutorial 4
    std::map<std::string, GLApp::GLModel>::iterator mdl_ref;
    std::map<std::string, GLSLShader>::iterator     shd_ref;

    // added to tutorial 4
    glm::vec3 color;
    glm::mat3 mdl_xform; // model transformation

    // you can implement them as in tutorial 3 ...
    void init();
    void draw() const;
    // update function could be
    void update();
    // or could be
    void update(GLdouble time_per_frame);
    // just make sure you have one update function
  };
  //static std::list<GLApp::GLObject> objects; // from tutorial 3
  // added for tutorial 4: repository of objects
  static std::map<std::string, GLObject> objects; // singleton
};
```

Updated version of structure `GLApp`

After incorporating the changes described earlier, the definition of structure `GLApp` will now look like this:

```
struct GLApp {
  // other stuff ...

  struct GLModel {
    GLenum     primitive_type;
    GLuint     primitive_cnt;
    GLuint     vaoid;
    GLuint     draw_cnt;
    // you could add member functions for convenience if you so wish ...
  };

  struct GLObject {
    glm::vec2 scaling;
```

```
    glm::vec2 orientation;
    glm::vec2 position;
    glm::vec3 color;
    glm::mat3 mdl_xform; // model (model-to-world) transform
    glm::mat3 mdl_to_ndc_xform; // model-to-NDC transform

    std::map<std::string, GLApp::GLModel>::iterator mdl_ref;
    std::map<std::string, GLSLShader>::iterator     shd_ref;

    // you can implement them as in Tutorial 3 ...
    void init();
    void draw() const;
    void update();
    //void update(GLdouble time_per_frame);
  };
  static std::map<std::string, GLSLShader> shdrpgms; // singleton
  static std::map<std::string, GLModel>    models;   // singleton
  static std::map<std::string, GLObject>   objects;  // singleton
};
```

🎯Task 1: Updating function `GLApp::init`

Since this tutorial reads model and object data from files, initialization function `GLApp::init` [in glapp.cpp] will now look like this:

```
// define singleton containers
std::map<std::string, GLSLShader>      GLApp::shdrpgms;
std::map<std::string, GLApp::GLModel>  GLApp::models;
std::map<std::string, GLApp::GLObject> GLApp::objects;

void GLApp::init() {
  // Part 1: Initialize OpenGL state ...

  // Part 2: Use the entire window as viewport ...

  // Part 3: parse scene file /scenes/tutorial-4.scn
  // and store repositories of models of type GLModel in container
  // GLApp::models, store shader programs of type GLSLShader in
  // container GLApp::shdrpgms, and store repositories of objects of
  // type GLObject in container GLApp::objects
  GLApp::init_scene("../scenes/tutorial-4.scn");

  // Part 4: initialize camera
  // explained in a later section ...
}
```

Code for $\text{Part } 1$ and $\text{Part } 2$ has been explained and implemented in previous tutorials. The focus here will be on $\text{Part } 3$. Begin by declaring the following functions in structure `GLApp` in file glapp.h:

```
struct GLApp {
  // other stuff ...

  // function to insert shader program into container GLApp::shdrpgms ...
  static void insert_shdrpgm(std::string, std::string, std::string);
  // function to parse scene file ...
  static void init_scene(std::string);
};
```

Task 1a: Shader initialization in `GLApp::insert_shdrpgm`

As indicated in the section on understanding scene files, every object's description consists of three strings providing the shader program's name, the file containing the vertex shader source code, and the file containing the fragment shader source code. If the shader program doesn't exist in singleton container `GLApp::shdrpgms`, the vertex and fragment shaders will be compiled, linked, and validated to create and insert the shader program into the container. Otherwise, the three strings are ignored. A static function `GLApp::insert_shdrpgm` is introduced in this tutorial to implement the actions of compiling, linking, validating, and inserting the shader program into container `GLApp::shdrpgms`. The function is declared in structure `GLApp` in file glapp.h and defined in file glapp.cpp:

```
void GLApp::insert_shdrpgm(std::string shdr_pgm_name,
                           std::string vtx_shdr,
                           std::string frg_shdr) {
  std::map<std::string, GLSLShader>::iterator it =
          GLApp::shdrpgms.find(shdr_pgm_name);
  if (it != GLApp::shdrpgms.end()) return;

  std::vector<std::pair<GLenum, std::string>> shdr_files {
    std::make_pair(GL_VERTEX_SHADER, vtx_shdr_name),
    std::make_pair(GL_FRAGMENT_SHADER, frg_shdr_name)
  };
  GLSLShader shdr_pgm;

  // Automation hook. [!WARNING!] Do not alter/remove this!
  AUTOMATION_HOOK_SHADER(shdr_pgm, shdr_files);

  shdr_pgm.CompileLinkValidate(shdr_files);
  if (GL_FALSE == shdr_pgm.IsLinked()) {
    std::cout << "Unable to compile/link/validate shader programs\n";
    std::cout << shdr_pgm.GetLog() << "\n";
    std::exit(EXIT_FAILURE);
  }
  // add compiled, linked, and validated shader program to
  // std::map container GLApp::shdrpgms
  GLApp::shdrpgms[shdr_pgm_name] = shdr_pgm;
}
```

> ***Ignore the line that makes a call to function `AUTOMATION_HOOK_SHADER`. This is a sandbox hook that provides testing and debugging functionalities.***

Task 1b: Parsing scene and mesh files in `GLApp::init_scene`

Function `GLApp::init_scene` will implement the following activities:

1. open scene file specified as its parameter,
2. determine count of objects described in the scene file,
3. instantiate each object of type `GLApp::GLObject` with parameters parsed from the file,
4. if the model listed in the scene file is not present in singleton container `GLApp::models`, the corresponding mesh file in $/\text{meshes}$ is parsed to instantiate a model of type `GLApp::GLModel` which is then inserted into container `GLApp::models`.
5. if the shader program listed in the scene file is not present in singleton container `GLApp::shdrpgms`, the corresponding vertex and fragment shaders are compiled, linked, and validated to create and insert the shader program into container `GLApp::shdrpgms`.
6. insert the newly instantiated object from step 3 into singleton container `GLApp::objects`. The instantiated object will obtain a reference to the model that it instantiates in container `GLApp::models` and a reference to the shader program it should use for rendering from container `GLApp::shdrpgms`.

Using C++ standard library classes `std::ifstream`, `std::string`, `std::istringstream`, `std::map`, and function `std::getline` presented in `<std::string>`, the outline of function `GLApp::init_scene` will look like this:

```
1  void GLApp::init_scene(std::string scene_filename) {
2    std::ifstream ifs{ scene_filename, std::ios::in };
3    if (!ifs) {
4      std::cout << "ERROR: Unable to open scene file: "
5                << scene_filename << "\n";
6      exit(EXIT_FAILURE);
7    }
8    ifs.seekg(0, std::ios::beg);
9
10   std::string line;
11   getline(ifs, line); // first line is count of objects in scene
12   std::istringstream line_sstm{ line };
13   int obj_cnt;
14   line_sstm >> obj_cnt; // read count of objects in scene
15   while (obj_cnt--) { // read each object's parameters
16     getline(ifs, line); // 1st parameter: model's name
17     std::istringstream line_modelname{ line };
18     std::string model_name;
19     line_modelname >> model_name;
20
21     /*
22     add code to do this:
23     if model with name model_name is not present in std::map container
24     called models, then add this model to the container
25     */
26
27     /*
28     add code to do this:
```

```
29      if shader program listed in the scene file is not present in
30      std::map container called shdrpgms, then add this shader to the
31      container
32      */
33
34      /*
35      add code to do this:
36      read remaining parameters of object from file:
37        object's name
38        RGB parameters for rendering object's model geometry
39        scaling factors to be applied on object's model
40        orientation factors: initial angular orientation and angular speed
41        object's position in game world
42
43      set data member GLApp::GLObject::mdl_ref to iterator that points to
44      model instantiated by this object
45
46      set data member GLApp::GLObject::shd_ref to iterator that points to
47      shader program used by this object
48
49      insert this object to std::map container objects
50      */
51    }
52  }
```

The scene file may describe multiple objects instantiated from the same model. Therefore, lines 21 through 25 ensure that a mesh file is parsed and inserted into the corresponding singleton container only if the corresponding model name is encountered the very first time. Similarly, lines 28 through 32 ensure that a shader program is compiled, linked, validated, and then inserted into the corresponding singleton container only if the corresponding shader program name is encountered the very first time.

🎯Task 2: Interactive camera

The application displays a large 2D world consisting of objects that are instantiations of triangle, box, and circle models. The world is modeled in the canonical coordinate system represented by the usual and familiar Cartesian coordinate system with $x-$axis oriented right in direction $\hat{i} = \langle 1, 0 \rangle$, $y-$axis oriented up in direction $\hat{j} = \langle 0, 1 \rangle$, and origin located at $(0, 0)$. Task 1 defines a scene with each object having specific scaling factors, angular displacement, angular speed, and a position in the world. These parameters can be used to compute a per-object 3×3 model transformation matrix that sizes, orients, and positions these objects in the world.

To navigate this large 2D world, the application must implement an interactive camera. Most applications use an existing game object as the camera and it is from this object's point of view that the surrounding world is viewed. In scene file $\text{scenes/tutorial-4.scn}$, an object with special name `"Camera"` has a camera embedded into it. When the scene file is parsed and objects are inserted into a `std::map` container, the camera object is subsequently identified using this special name `"Camera"`.

Task 2a: Defining camera coordinate system

1. From lectures, we know that an interactive camera must specify a position in the world [where is the camera located?], an orientation angle and vectors [where is the camera looking?], and a window to the world [how much of the world can be seen from the camera's viewfinder?]. The parameters specifying the camera's position, orientation, and world-to-camera view transformation matrix are abstracted into structure `GLApp::Camera2D`:

```
struct GLApp {
  // other stuff ...

  struct Camera2D {
    GLObject *pgo; // pointer to game object that embeds camera
    glm::vec2 right, up;
    glm::mat3 view_xform;

    // you can implement these functions as you wish ...
    void init(GLFWwindow*, GLObject* ptr);
    void update(GLFWwindow*);
  };

  // define object of type Camera2D ...
  static Camera2D camera2d;
};
```

2. Data member `Camera2D::pgo` points to an object of type `GLApp::GLObject` with name `"Camera"` in container `GLApp::objects` of type `std::map<std::string, GLApp::GLObject>`. This allows the static camera object `GLApp::camera2d` to specify the camera's position and orientation using the corresponding parameters of the object into which it is embedded. The object's position is specified by data member `GLObject::position` of type `glm::vec2`. In this tutorial, the object's orientation parameters are specified by data member `GLObject::orientation` with `orientation::x` representing the object's angular displacement in degrees with respect to x—axis while `orientation::y` represents angular speed in degrees.

3. Class discussions on modeling 2D cameras and world-to-camera transformations specified the camera's orientation in world coordinate system using a *right* vector $\hat{u}$ representing the camera's x—axis and an *up* vector $\hat{v}$ representing the camera's y—axis. Further, an object's angular displacement is defined with respect to the world coordinate system's x—axis. One method to compute the camera's right and up vectors consists of rotating world coordinate system's x—axis by the camera's angular displacement to compute right vector $\hat{u}$ which is then rotated through angle 90° to compute up vector $\hat{v}$. This is so because just as with the canonical coordinate system, the camera's right and up vectors must be orthonormal vectors. The second method consists of rotating world coordinate system's y—axis by the camera's angular displacement to compute up vector $\hat{v}$ which is then rotated through angle -90° to compute right vector $\hat{u}$. The second method will be discussed below.

4. Suppose the interactive camera object is initialized from scene file scenes/tutorial-4.scn to have an angular displacement of θ°. The 3×3 rotation matrix to rotate through an angle of θ° is:

$$\mathrm{R}_{\theta^\circ} = \begin{bmatrix} \cos\theta^\circ & -\sin\theta^\circ & 0 \\ \sin\theta^\circ & \cos\theta^\circ & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

The camera's up vector in the world coordinate system is obtained by applying $\mathrm{R}_{\theta^\circ}$ on the world coordinate system's $y-$axis [or $\hat{j} = \langle 0, 1 \rangle$]:

$$\hat{v} = \mathrm{R}_{\theta^\circ}\hat{j} = \begin{bmatrix} \cos\theta^\circ & -\sin\theta^\circ & 0 \\ \sin\theta^\circ & \cos\theta^\circ & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} -\sin\theta^\circ \\ \cos\theta^\circ \\ 0 \end{bmatrix}$$
$$\implies \hat{v} = \langle -\sin\theta^\circ, \cos\theta^\circ \rangle$$

Since the camera's up vector $\hat{v}$ represents its $y-$axis, the camera's right vector $\hat{u}$ representing its $x-$axis is easily obtained by rotating up vector $\hat{v}$ through an angle of -90°:

$$\hat{u} = \mathrm{R}_{-90^\circ}\hat{v} = \begin{bmatrix} \cos(-90^\circ) & -\sin(-90^\circ) \\ \sin(-90^\circ) & \cos(-90^\circ) \end{bmatrix} \begin{bmatrix} v_{\hat{i}} \\ v_{\hat{j}} \end{bmatrix}$$
$$= \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix} \begin{bmatrix} v_{\hat{i}} \\ v_{\hat{j}} \end{bmatrix} = \begin{bmatrix} v_{\hat{j}} \\ -v_{\hat{i}} \end{bmatrix}$$
$$\implies \hat{u} = \left\langle v_{\hat{j}}, -v_{\hat{i}} \right\rangle = \langle \cos\theta^\circ, \sin\theta^\circ \rangle$$

5. Suppose the camera's world position is $C = \left(C_{\hat{i}}, C_{\hat{j}}\right)$. In the previous step, the camera's right vector $\hat{u}$ and up vector $\hat{v}$ were computed. These three pieces of information make it possible to compute the *world-to-camera* [or *view* or *camera* or *eye*] transformation matrix. The purpose of this transformation is to map points in world coordinate system to a coordinate system whose $x-$ and $y-$axes are camera orientation vectors $\hat{u}$ and $\hat{v}$, respectively and whose origin is camera's position C in the world. This transform is what enables the rendering of only those portions of the world visible from the camera's point of view. From class discussions, the world-to-camera transformation matrix is computed as:

$$\mathrm{M}_{view} = \begin{bmatrix} u_{\hat{i}} & u_{\hat{j}} & -\hat{u} \cdot C \\ v_{\hat{i}} & v_{\hat{j}} & -\hat{v} \cdot C \\ 0 & 0 & 1 \end{bmatrix}$$

Matrix M_{view} represents the world-to-camera transformation matrix required for a *first-person camera*. A *free camera* specifies a world-to-camera transformation matrix that maps points in world coordinate system to a coordinate system whose $x-$ and $y-$axes are similar to the world coordinate system but whose origin is camera's position in the world. That is, a free camera's world-to-camera transformation matrix requires only the camera's position but not its orientation:

$$\mathrm{M}_{view} = \begin{bmatrix} 1 & 0 & -C_{\hat{i}} \\ 0 & 1 & -C_{\hat{j}} \\ 0 & 0 & 1 \end{bmatrix}$$

6. A final detail is to add to function `GLApp::init` the invocation of the camera initialization function:

```
void GLApp::init() {
  // Part 1: Initialize OpenGL state ...
  // Part 2: use the entire window as viewport ...
  // Part 3: call GLApp::init_scene() ...

  // Part 4: initialize camera
  GLApp::camera2d.init(GLHelper::ptr_window,
                       &GLApp::objects.at("Camera"));

  // Part 5: Print OpenGL context and GPU specs
}
```

Function `GLApp::Camera2D::init` will use the first argument `GLHelper::ptr_window` to compute the viewport's aspect ratio [more on this later]. The second argument is a pointer of type `GLObject*` that points to the object in `std::map` container `objects` with name `"Camera"`.

7. The outline of function `GLApp::Camera2D::init` looks like this:

```
void GLApp::Camera2D::init(GLFWwindow* pWindow, GLApp::GLObject* ptr) {
  // assign address of object of type GLApp::GLObject with
  // name "Camera" in std::map container GLApp::objects ...
  pgo =

  // compute camera's up and right vectors ...
  up    =
  right =

  // at startup, the camera must be initialized to free camera ...
  view_xform =
}
```

Task 2b: Defining camera's window to world

1. Recall from class lectures that in addition to orientation and position in the world, a camera also requires a rectangular *window to the world* that describes the camera's viewfinder. While the camera's orientation and position are used to compute the world-to-camera transformation matrix $\mathbb{M}_{view}$, the camera's rectangular window determines which objects in the world are rendered to the viewport. Objects completely inside the rectangular window are rendered to the viewport while objects completely outside the rectangular window [that is, objects sufficiently far off from the camera] are culled by the application. Finally, objects straddling the window are clipped by the GPU to ensure that only portions of these straddling objects completely inside the window are rendered.

2. The camera's window parameters must be added to the previously defined structure `GLApp::Camera2D`:

```
struct GLApp {
  // other stuff ...

  struct Camera2D {
    GLObject *pgo; // pointer to game object that embeds camera
    glm::vec2 right, up;
    glm::mat3 view_xform;
```

```
    // additional parameters ...
    GLint height{ 1000 };
    GLfloat ar;
    glm::mat3 camwin_to_ndc_xform;
    glm::mat3 world_to_ndc_xform;
    // you can implement these functions as you wish ...
    void init(GLFWwindow*, GLObject* ptr);
    void update(GLFWwindow*);
  };
  // define object of type Camera2D ...
  static Camera2D camera2d;
};
```

3. To define the camera's rectangular window, both its width W and height H are required. To ensure WYSIWYG interface, the aspect ratio of the camera's window must be equivalent to the graphics pipeline's viewport which in turn must be equivalent to the color buffer's aspect ratio. During initialization, the camera's aspect ratio ar is initialized to be the same as the viewport, so that subsequently, given height H, the camera's width is computed at runtime as $W = ar \times H$.

4. If the camera's window has width W and height H, the following transformation matrix transforms points in camera coordinate system into NDC:

$$\mathrm{M}_{cam-win \to NDC} = \begin{bmatrix} \frac{2}{W} & 0 & 0 \\ 0 & \frac{2}{H} & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

5. Since matrices M_{view} and $\mathrm{M}_{cam-win \to NDC}$ are computed using camera-specific parameters, these two matrices are computed once per frame for the entire scene and can be further concatenated together:

$$\mathrm{M}_{world \to NDC} = \mathrm{M}_{cam-win \to NDC} \circ \mathrm{M}_{view}$$

   As will be shown later, each object computes a model transformation matrix M_{model} per frame which is pre-multiplied with $\mathrm{M}_{world \to NDC}$ to compute a $\mathrm{M}_{model \to NDC}$ matrix:

$$\mathrm{M}_{model \to NDC} = \mathrm{M}_{world \to NDC} \circ \mathrm{M}_{model}$$

6. The outline of function `GLApp::Camera2D::init` must now be extended to incorporate the computation of the aspect ratio and computation of $\mathrm{M}_{cam-win \to NDC}$ and $\mathrm{M}_{world \to NDC}$ matrices:

```
void GLApp::Camera2D::init(GLFWwindow* pWindow, GLApp::GLObject* ptr) {
  // assign address of object of type GLApp::GLObject with
  // name "Camera" in std::map container GLApp::objects ...
  pgo =

  // compute camera window's aspect ratio ...
  GLsizei fb_width, fb_height;
  glfwGetFramebufferSize(pWindow, &fb_width, &fb_height);
  ar = static_cast<GLfloat>(fb_width) / fb_height;

  // compute camera's up and right vectors ...
  up    =
  right =

```

```
  // at startup, the camera must be initialized to free camera ...
  view_xform =
  // compute other matrices ...
  camwin_to_ndc_xform =
  world_to_ndc_xform  =
}
```

Task 2c: Defining camera's interactive features

1. Both the free and first-person cameras are interactive. When keyboard button $\mathbf{H}$ is pressed, camera's angular displacement increases:

$$\begin{aligned}&\text{if (keyboard button } \mathbf{H} \text{ is pressed)}\\&\quad orientation.x \mathrel{+}= orientation.y\end{aligned}$$

   When button $\mathbf{K}$ is pressed, the angular displacement decreases:

$$\begin{aligned}&\text{if (keyboard button } \mathbf{K} \text{ is pressed)}\\&\quad orientation.x \mathrel{-}= orientation.y\end{aligned}$$

   Recall from the previous section that the camera's current angular displacement is used to specify its right vector $\hat{u}$ and up vector $\hat{v}$ which are then used to compute the first-person camera's world-to-camera view transformation matrix M_{view}. Modifying these vectors will therefore modify M_{view}. If θ° represents the camera's current angular displacement, the camera's up vector $\hat{u}$ and right vector $\hat{v}$ are computed as:

$$\begin{aligned}\hat{v} &= \left\langle v_{\hat{i}}, v_{\hat{j}} \right\rangle = \langle -\sin\theta^{\circ}, \cos\theta^{\circ} \rangle\\ \hat{u} &= \left\langle u_{\hat{i}}, u_{\hat{j}} \right\rangle = \left\langle v_{\hat{j}}, -v_{\hat{i}} \right\rangle = \langle \cos\theta^{\circ}, \sin\theta^{\circ} \rangle\end{aligned}$$

2. The application uses keyboard button $\mathbf{U}$ for users to displace the camera in the direction of its up vector $\hat{v}$. A common idiom for controlling interactive cameras in both $2\mathrm{D}$ and $3\mathrm{D}$ simulations is to first figure out where the camera is looking by computing its view vector, and then to displace the camera along this newly computed view vector [so that the camera will move in the direction it is looking]. For button $\mathbf{U}$ press, the camera's position is updated as:

$$C_{new} = C_{prev} + k\hat{v}$$

   where constant k represents the camera's linear speed. Data member `Camera2D::linear_speed` represents constant k in the `GLApp::Camera2D` structure below and the sample initializes this scalar value 2.

3. At startup, the camera is configured to behave as a free camera. Subsequently, keyboard button $\mathbf{V}$ allows users to toggle between free and first-person implementations. If $\hat{u}$, $\hat{v}$, and C represent camera's right vector, up vector, and position, respectively, the world-to-camera view transformation matrix for free camera is:

$$\mathrm{M}_{view} = \begin{bmatrix} 1 & 0 & -C_{\hat{i}} \\ 0 & 1 & -C_{\hat{j}} \\ 0 & 0 & 1 \end{bmatrix}$$

   and for first-person camera is:

$$\mathrm{M}_{view} = \begin{bmatrix} u_{\hat{i}} & u_{\hat{j}} & -\hat{u} \cdot C \\ v_{\hat{i}} & v_{\hat{j}} & -\hat{v} \cdot C \\ 0 & 0 & 1 \end{bmatrix}$$

4. Users can zoom into and out from the scene using keyboard button $\mathbf{Z}$. Zooming *in* effect is caused when the camera's rectangular window shrinks causing smaller portions of the world to be visible which are then rendered to the fixed sized viewport and color buffers. Zooming *out* is the opposite effect caused by increasing the window's dimensions so that larger portions of the world are visible which are then rendered to the fixed sized viewport and color buffers. Recall that $\mathrm{M}_{cam-win \to NDC}$ is a scale matrix involving window's width W and height H that maps contents of camera's window to NDC:

$$\mathrm{M}_{cam-win \to NDC} = \begin{bmatrix} \frac{2}{W} & 0 & 0 \\ 0 & \frac{2}{H} & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

   This means that through button $\mathbf{Z}$, the user can interactively change height H of the camera's rectangular window. Changing height H will in turn change width W [because the aspect ratio of the window must be maintained equivalent to the viewport's aspect ratio], which in turn will change scale matrix $\mathrm{M}_{cam-win \to NDC}$ causing the appropriate zoom effect. The sample sets the window's default height to 1000 with interactive height limited to interval $[500, 2000]$.

5. The interactive parameters discussed above are added to the previously defined structure `GLApp::Camera2D`:

```
struct GLApp {
  // other stuff ...

  struct Camera2D {
    GLObject *pgo; // pointer to game object that embeds camera
    glm::vec2 right, up;
    glm::mat3 view_xform, camwin_to_ndc_xform, world_to_ndc_xform;

    // window parameters ...
    GLint height{ 1000 };
    GLfloat ar;

    // window change parameters ...
    GLint min_height{ 500 }, max_height{ 2000 };
     // height is increasing if 1 and decreasing if -1
    GLint height_chg_dir{ 1 };
    // increments by which window height is changed per Z key press
    GLint height_chg_val{ 5 };

    // camera's speed when button U is pressed
    GLfloat linear_speed{ 2.f };

    // keyboard button press flags
    GLboolean camtype_flag{ GL_FALSE };    // button V
    GLboolean zoom_flag{ GL_FALSE };       // button Z
    GLboolean left_turn_flag{ GL_FALSE };  // button H
    GLboolean right_turn_flag{ GL_FALSE }; // button K
    GLboolean move_flag{ GL_FALSE };       // button U

```

```
    // you can implement these functions as you wish ...
    void init(GLFWwindow*, GLObject* ptr);
    void update(GLFWwindow*);
  };

  // define object of type Camera2D ...
  static Camera2D camera2d;
};
```

6. The outline of the camera's update function `GLApp::Camera2D::update` looks like this:

```
void GLApp::Camera2D::update(GLFWwindow* pWindow) {
  // check keyboard button presses to enable camera interactivity

  // update camera aspect ratio - this must be done every frame
  // because it is possible for the user to change viewport
  // dimensions

  // update camera's orientation (if required)

  // update camera's up and right vectors (if required)

  // update camera's position (if required)

  // implement camera's zoom effect (if required)

  // compute appropriate world-to-camera view transformation matrix

  // compute window-to-NDC transformation matrix

  // compute world-to-NDC transformation matrix
}
```

🎯Task 3: `GLApp::update` and `GLApp::draw`

1. The outline of function `GLApp::update` looks like this:

```
void GLApp::update() {
  // first, update camera
  GLApp::camera2d.update(GLHelper::ptr_window);

  // next, iterate through each element of container objects
  // for each object of type GLObject in container objects
  //   call update function GLObject::update(delta_time) except on
  //   object which has camera embedded in it - this is because
  //   the camera has already updated the object's orientation
}
```

2. Every frame, each object's update function `GLObject::update` will compute a model transformation matrix M_{model} from scale factors, angular displacement with respect to $x-$ axis, and position coordinates. The scale matrix H is computed as:

$$\mathrm{H} = \begin{bmatrix} scaling.x & 0 & 0 \\ 0 & scaling.y & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

To compute the rotation matrix, the current orientation must first be updated:

$$orientation.x \quad += orientation.y$$

Since $orientation.x$ is expressed in degrees, it must be first converted to radians. Assuming θ represents the angular displacement in radians, rotation matrix R is computed as:

$$\mathrm{R} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Translation matrix T is computed as:

$$\mathrm{T} = \begin{bmatrix} 1 & 0 & position.x \\ 0 & 1 & position.y \\ 0 & 0 & 1 \end{bmatrix}$$

The model [or model-to-world] transformation matrix is then computed as:

$$\mathrm{M}_{model} = \mathrm{T} \circ \mathrm{R} \circ \mathrm{H}$$

Recall that the camera computed the world-to-NDC transformation matrix $\mathrm{M}_{world \to NDC}$. Every object must then pre-multiply its model transformation matrix M_{model} with $\mathrm{M}_{world \to NDC}$ matrix to compute a $\mathrm{M}_{model \to NDC}$ matrix:

$$\mathrm{M}_{model \to NDC} = \mathrm{M}_{world \to NDC} \circ \mathrm{M}_{model}$$

This $\mathrm{M}_{model \to NDC}$ matrix will transform vertex position coordinates of the model referenced by the object from model coordinate system to NDC and therefore must be copied to a vertex shader `uniform` variable. The viewport transform specified by the OpenGL command `glViewport` command will then further transform NDC coordinates to window coordinates.

3. Vertex shader file my-tutorial-4.vert must be authored in directory /projects/tutorial-4/shaders and would be a minimal version of your vertex shader from Tutorial 3. The geometry read from .msh files contains vertex position attributes but not vertex color attributes. Vertex shader my-tutorial-4.vert would look like this:

```
#version 450 core

layout (location = 0) in vec2 aVertexPosition;
//layout (location = 1) in vec3 aVertexColor;

//layout (location = 0) out vec3 vColor;

uniform mat3 uModelToNDC;

void main() {
  gl_Position = vec4(vec2(uModelToNDC * vec3(aVertexPosition, 1.f)),
                     0.0, 1.0);
  //vColor = aVertexColor;
}
```

4. Fragment shader file my-tutorial-4.frag must also be authored in directory /projects/tutorial-4/shaders and would be a minor variation of your shader from Tutorial 3. Since the vertex shader is not emitting vertex color attributes for consumption by GPU pipeline's rasterization stage, the rasterizer will not supply the fragment shader an interpolated color for each fragment [think of a fragment as a *would be* pixel]. However, the main objective of the fragment shader is to generate a color for each fragment. Each object has a per-object color attribute that was read from the scene file. This per-object color value must now be sent by the object's draw function `GLObject::draw` to a `uniform` variable defined in the fragment shader. The model referenced by an object may have many triangles and each rendered triangle would generate thousands or even millions of fragments. The fragment shader will retain the per-object color value copied to the shaders' uniform variable until it is updated by the next object's draw call. This enables the shader to write the same color in the color buffer for every fragment of a particular object. The amended fragment shader from Tutorial 3 would now look like this:

```
#version 450 core

/*
A per-fragment color attribute is no longer received from rasterizer.
Instead per-fragment color attribute is supplied by the application to
a uniform variable:

uniform vec3 uColor;

The uniform variable will retain the value for every invocation of the
fragment shader. That is why every fragment of the triangle primitive
rendered by an object has the same color
*/

//layout(location=0) in vec3 vColor;
uniform vec3 uColor;

layout (location=0) out vec4 fFragColor;

void main() {
  //fFragColor = vec4(vColor, 1.0);
  fFragColor = vec4(uColor, 1.0);
}
```

5. The outline of function `GLApp::draw` looks like this:

```
void GLApp::draw() {
  // write window title with appropriate information using
  // glfwSetWindowTitle() ...

  // clear color buffer

  // render camera afer everything else has been rendered
  // otherwise, the black triangle will be occluded by other objects
}
```

6. The outline of function `GLObject::draw` looks like this:

```
void GLApp::GLObject::draw() const {
  // load shader program in use by this object

  // bind VAO of this object's model

  // copy object's color to fragment shader uniform variable uColor

  // copy object's model-to-NDC matrix to vertex shader's
  // uniform variable uModelToNDC

  // call glDrawElements with appropriate arguments

  // unbind VAO and unload shader program
}
```

🎯Task 4: Adapting viewport for resizing window

As with Tutorial 2, your submission must guarantee that the graphics pipeline's viewport settings adapt to resized window dimensions, ensuring accurate rendering and display across various window sizes. See Tutorial 2 specs and this tutorial's sample for details.

Submission

1. Create a copy of project directory tutorial-4 named <login>-<tutorial-4>. That is, if your Moodle student login is foo, then the directory should be named foo-tutorial-4. Ensure that directory foo-tutorial-4 has the following layout:

```
foo-tutorial-4          # You're submitting Tutorial 4
├─ include              # Header files - *.hpp and *.h files
└─ src                  # Source files - *.cpp and .c files
└─ shaders              # Shader files - *.vert and .frag files
   └─ my-tutorial-4.vert # Vertex shader file
   └─ my-tutorial-4.frag # Fragment shader file
```

2. *Zip the directory* to create archive file foo-tutorial-4.zip.

⚠️Before Submission: Verify and Test⚠️

- Copy archive file foo-tutorial-4.zip into directory test-submissions. Unzip the archive file to create project directory foo-tutorial-4 by typing the following command in the command-line shell [which can be opened by typing cmd and pressing Enter in the Address Bar]:

```
powershell -Command "Expand-Archive -LiteralPath foo-tutorial-4.zip -
DestinationPath ."
```

- After executing the command, the layout of directory test-submissions will look like this:

```
csd2101-opengl-dev    # Sandbox directory for all assessments
├─ test-submissions   # Test submissions here before uploading
|  └─ foo-tutorial-4 # foo is submitting Tutorial 4
|     └─ include     # Header files - *.hpp and *.h files
|     └─ src         # Source files - *.cpp and .c files
|     └─ shaders     # Shader files - *.vert and .frag files
|        └─ my-tutorial-4.vert  # Vertex shader file
|        └─ my-tutorial-4.frag  # Fragment shader file
└─ csd2101.bat              # Automation Script
```

- Delete the original copy of foo-tutorial-4 from directory projects to prevent duplicate project names during reconfiguration.
- Run batch file csd2101.bat and select option R to reconfigure the Visual Studio 2022 solution with the new project foo-tutorial-4.
- Build and execute project foo-tutorial-4 by opening the Visual Studio 2022 solution in directory build.
- Use the following checklist to ***verify and submit*** your submission:

Things to test before submission	Status
Assessment compiles without any errors	☐
All compilation warnings are resolved; there are zero warnings	☐
Executable generated and successfully launched in Debug and Release mode	☐
Directory is zipped using the naming conventions outlined in submission guidelines	☐
Zipped file is uploaded to assessment submission page	☐

The purpose of this verification step is not to guarantee the correctness of your submission. Instead, it verifies whether your submission meets the most basic rubrics [it compiles; it doesn't generate warnings; it links; it executes] or not and thus helps you avoid major grade deductions. And, remember you'll need every single point from these programming assessments to ensure a noteworthy grade!!! Read the section on Grading Rubrics for information on how your submission will be assigned grades.

Grading Rubrics

The core competencies assessed for this assessment are:

- [***core1***] Submitted code must build an executable file with ***zero*** warnings. This rubric is not satisfied if the generated executable crashes or displays nothing. In other words, if your source code doesn't compile nor link nor execute, there is nothing to grade.
- [***core2***] This competency indicates whether you're striving to be a professional software engineer, that is, are you implementing software that satisfies project requirements [are you submitting the required files? are they properly named? are they properly archived? are you submitting unnecessary files?], that is maintainable and easy to debug by you and others [are file and function headers properly annotated?]. Submitted source code must satisfy ***all***

requirements listed below. Any missing requirement will decrease your grade by one letter grade.

  - Source code must compile with ***zero*** warnings. Pay attention to all warnings generated by the compiler and fix them.
  - Source code file submitted is correctly named.
  - Source code file is *reasonably* structured into functions and *reasonably* commented. See next two points for more details.
  - If you've created a new source code file, it must have file and function header comments.
  - If you've edited a source code file provided by the instructor, the file header must be annotated to indicate your co-authorship. Similarly, if you're amending a previously defined function, you must annotate the function header to document your amendments. You must add a function header to every new function that you've defined.
- [***core3***] Completed all necessary tasks to generate an executable similar to the sample. Less is not ok nor cool. More and cooler stuff is ok. What is minimally required?
  - Provide appropriate vertex and fragment shaders for rendering objects.
  - Parse scene file /scenes/tutorial-4.scn and mesh files in /meshes to instantiate models of type `GLApp::GLModel`, instantiate objects of type `GLApp::GLObject`, and create shader programs of type `GLSLShader`.
  - Use `std::map` containers as repositories for models, objects, and shader programs.
  - Implement geometric instancing of models.
  - Compute model-to-world transformation matrices for each object.
  - Define a camera that is embedded to the object in scene file with name `"Camera"` with corresponding shader applied.
  - Based on user input, compute a world-to-camera view transformation matrix for either a free or a first-person camera.
  - Compute camera-to-NDC transformation matrix to map camera window contents to NDC.
  - Compute world-to-NDC transformation matrix to map world coordinate system contents to NDC.
- [***core4***] Provide interactive properties for the camera [keyboard buttons $\mathbf{Z}$, $\mathbf{V}$, $\mathbf{H}$, $\mathbf{K}$, and $\mathbf{U}$] described in this document and exhibited in the sample.
- [***core5***] Print information in window toolbar similar to sample executable - more information is ok but not less.
- [***core6***] Dynamically adjust the rendering output when the window is resized.

Mapping of Grading Rubrics to Letter Grades

The following illustrates the mapping of core competencies listed in the grading rubrics to letter grades:

Grading Rubric Assessment	Letter Grade
There is no submission.	F
core1 rubric is not satisfied. Submitted property page and/or source code doesn't build. Or, executable generated by build crashes or displays nothing.	F
If ***core2*** rubrics are not satisfied, final letter grade will be decreased by one. This means that if you had received a grade A and ***core2*** is not satisfied, your grade will be recorded as B, an $A-$ would be recorded as $B-$, and so on.	
core3 rubric is satisfied.	B
core4 rubric is satisfied.	$B+$
core5 rubric is satisfied.	$A-$
core6 rubric is satisfied.	A
Outstanding work with extra material beyond those described in specs. The key word here is *outstanding*.	$A+$