Tutorial 3: Geometric Instancing Using Model Transforms

> 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 and implement concept of geometric instancing.
- Implement aspects of 2D graphics pipe including model [or model-to-world] transforms and world-to-NDC transforms.
- Research specific OpenGL functions that facilitate different polygon rasterization modes.
- Understand and implement interactive graphics applications by setting up keyword and mouse controls and responding to these controls.
- Research and explore GLSL to extend functionality of vertex and fragment shaders.
- Implement a simple shader store to render instances of similar models with different shader programs.

Prerequisites

- You must complete Tutorial 2 as a prerequisite for this project. Much of the code implemented in Tutorial 2 will be refactored into this tutorial.
- You must understand viewport and affine transformations in a 2D graphics pipeline.

First Steps

Overwrite the existing batch file csd2101.bat in csd2101-opengl-dev [where you completed Tutorial 2] with a new version available on the assessment's web page. Execute the batch file.

1. Choose option **3 - Create Tutorial 3** to create a Visual Studio 2022 project tutorial-3 with associated project file tutorial-3.vcxproj in directory build and source in directory projects/tutorial-3 whose layout is shown below:

```
csd2101-opengl-dev/      # Sandbox directory for all assessments
├─ build/                # Build files will exist here
├─ projects/             # Tutorials and assignments reside here
|  └─ tutorial-3         # Tutorial 3 code exists here
|     └─ include         # Header files - *.hpp and *.h files
|     └─ src             # Source files - *.cpp and .c files
|        └─ my-tutorial-3.vert   # Vertex shader file
|        └─ my-tutorial-3.frag   # Fragment shader file
└─ csd2101.bat          # Automation Script
```

This option will also update the Visual Studio 2022 solution file opengl-dev.sln [also in directory build] to add newly created project tutorial-3. If you're currently in Solution Explorer, a dialog box will be displayed requesting an update. Press Reload button to refresh the Solution Explorer with the updated solution file that includes project tutorial-3.

Source and shader files are pulled into directory /projects/tutorial-3/src while header files are pulled into directory /projects/tutorial-3/include. Certain modifications must be made to these pulled files and these modifications are detailed below.

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

3. In /projects/tutorial-3/src, delete shader files my-tutorial-3.vert and my-tutorial-3.frag and then rename vertex shader my-tutorial-2.vert to my-tutorial-3.vert and fragment shader my-tutorial-2.frag to my-tutorial-3.frag.

4. In file glapp.cpp, we must replace references to shaders my-tutorial-2.vert and my-tutorial-2.frag with references to shaders my-tutorial-3.vert and my-tutorial-3.frag , respectively. We do this by replacing the following code

```
std::string const my_tutorial_2_vs {
  #include "my-tutorial-2.vert"
};
std::string const my_tutorial_2_fs {
  #include "my-tutorial-2.frag"
};
```

with this code:

```
std::string const my_tutorial_3_vs {
  #include "my-tutorial-3.vert"
};
std::string const my_tutorial_3_fs {
  #include "my-tutorial-3.frag"
};
```

Further, in function `GLApp::init` [also in file glapp.cpp], we replace references to objects `my_tutorial_2_vs` and `my_tutorial_2_fs` with objects `my_tutorial_3_vs` and `my_tutorial_3_fs`, respectively. We do this by replacing the following code:

```
// Part 3: set up model's VAO and create model's shader program
GLApp::models.emplace_back(GLApp::points_model(20, 20,
                                               my_tutorial_2_vs,
                                               my_tutorial_2_fs));
GLApp::models.emplace_back(GLApp::lines_model(40, 40,
                                              my_tutorial_2_vs,
                                              my_tutorial_2_fs));
GLApp::models.emplace_back(GLApp::trifans_model(50,
                                                my_tutorial_2_vs,
                                                my_tutorial_2_fs));
GLApp::models.emplace_back(GLApp::tristrip_model(10, 15,
                                                 my_tutorial_2_vs,
                                                 my_tutorial_2_fs));
```

with this code:

```
// Part 3: set up model's VAO and create model's shader program
GLApp::models.emplace_back(GLApp::points_model(20, 20,
                                               my_tutorial_3_vs,
                                               my_tutorial_3_fs));
GLApp::models.emplace_back(GLApp::lines_model(40, 40,
                                              my_tutorial_3_vs,
                                              my_tutorial_3_fs));
GLApp::models.emplace_back(GLApp::trifans_model(50,
                                                my_tutorial_3_vs,
                                                my_tutorial_3_fs));
GLApp::models.emplace_back(GLApp::tristrip_model(10, 15,
                                                 my_tutorial_3_vs,
                                                 my_tutorial_3_fs));
```

5. Now, you've a replica of project tutorial-2 in project tutorial-3. Right-click on project tutorial-3 in the Solution Explorer pane and select option Set as Startup Project. Build the project [using Ctrl + B] and then execute the application [using Ctrl+5] to obtain Tutorial 2 output.

Application Behavior

Now, let's shift our focus to the expected behavior of Tutorial 3. Begin by running the sample executable to understand the requirements and scope of this tutorial.

The application's behavior is [sort of] similar to a game's particle system but much more simpler. A game particle system will render particles using point, or line, or triangle primitives and its behavior is focused and nuanced to represent simulation of natural phenomena such as fire, sparks, water, and so on. The application uses 2D boxes composed from two triangle primitives and a star-like shape consisting of eight triangles. Particle behavior is simpler - once spawned at random positions with randomly specified scale factors, particles will neither change their sizes nor their positions. Instead, only the particle's orientation is animated. While a practical particle system's behavior is scripted, this project will rely on feedback from users to spawn new particles and destroy older particles.

Spawning objects

The application does nothing at startup until the left mouse button is pressed. A single (1) object [that could be the instantiation of either box or star model] having randomly specified scale, rotation, and position parameters is spawned. Every left mouse button click will double the number of objects by spawning the same number of objects as the count of displayed objects. Whether a particular object is an instantiation of the box or star model is randomly determined at runtime. In addition to the randomly chosen model type, every newly spawned object will have randomly specified scale, rotation, and position parameters. So, the next click would spawn one (1) additional object resulting in a total of two ($1 + 1 = 2$) objects being displayed. The third click will spawn two (2) additional objects resulting in the application now displaying a total of four ($2 + 2 = 4$) objects. This doubling of displayed objects with each click continues until a maximum of $32{,}768$ objects are displayed. Once this maximum number of displayed objects is reached, the spawning process is reversed into a de-spawning process with every click deleting half the displayed objects until only a single object is alive. The deleted objects are those with the oldest lifetime. So, in the reverse process, the first click will delete $(32{,}768 \div 2 = 16{,}384)$ objects having the oldest lifetime leaving $(32{,}768 \div 2 = 16{,}384)$ displayed objects. The next click will delete $(16{,}384 \div 2 = 8{,}192)$ objects with the oldest lifetime leaving $(16{,}384 \div 2 = 8{,}192)$ objects displayed. This process continues until a single (1) object is left. The next left mouse click will then begin the spawning process anew.

Changing render mode using `glPolygonMode`

Your implementation must provide functionality that allows users to change the render mode used by OpenGL to render triangles. The default render mode draws filled triangles. When keyboard button P is pressed, the render mode is altered to drawing triangle outlines. A further action on button P will change the polygon render mode to drawing triangle vertices as points. The render mode of triangles is an OpenGL context state setting that is controlled by the application [and not by shaders]. Research OpenGL API command `glPolygonMode` to investigate the various render modes available to programmers.

Changing shaders at runtime

As with earlier tutorials, the sample uses vertex color attributes associated with vertex position attributes to render the corresponding primitives with vertex color attribute interpolation. This means a single shader program is sufficient to render every object in the scene. It also means that the fragment shader from earlier tutorials will suffice for this tutorial. However, the vertex shader will be different and more details are provided here.

Even though the application can use a single shader program, this tutorial requires you to provide functionality that allows users to change the shader program that will render objects that will be spawned in the future. At startup, spawned objects are rendered with a shader program consisting of a vertex shader [encapsulated in file $/\mathrm{src}/\mathrm{my\text{-}tutorial\text{-}3.vert}$ and no different from the $\mathrm{Tutorial\ 2}$ vertex shader] and a fragment shader that assigns a linearly interpolated vertex color to each fragment [encapsulated in file $/\mathrm{src}/\mathrm{my\text{-}tutorial\text{-}3.frag}$ and no different from the $\mathrm{Tutorial\ 2}$ fragment shader]. When keyboard button S is pressed, a second shader program [having the same vertex shader but with a different fragment shader encapsulated in file $/\mathrm{src}/\mathrm{my\text{-}red.frag}$] is used to render objects that will be spawned in the future with color red. A further action on button S will cause a third shader program [having the same vertex shader but with a different fragment shader encapsulated in file $/\mathrm{src}/\mathrm{my\text{-}green.frag}$] to render objects that will be spawned in the future with color green. Another subsequent action on button S will cause a

fourth shader program [having the same vertex shader but with a different fragment shader encapsulated in file my-blue.frag] to render objects that will be spawned in the future with color blue. Notice that the new shader program will only affect objects that will be spawned in the future while previously spawned objects will continue to be rendered as before.

🎯Task 0: Understand geometric and shader program instancing

In computer graphics, the term *geometry instancing* implies the context of rendering multiple instances of the same model in a scene so that a small number of models can be used to define densely populated scenes. In graphics APIs, *instanced rendering* refers to a specific way of rendering the same geometry multiple times using a single command with each rendered image producing a slightly different result. This can be a very efficient method for rendering a large amount of geometry with very few API calls. This tutorial doesn't require you to replace command `glDrawElements` studied in earlier tutorials with its instanced version `glDrawElementsInstanced`. Instead, this tutorial is only concerned with introducing the idea of geometric instancing so that multiple instances of the same model can be rendered in a scene.

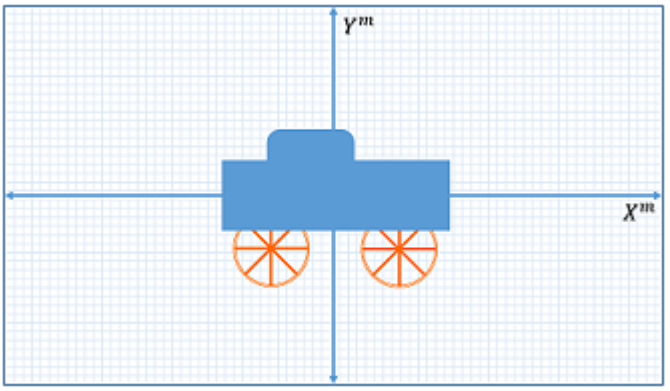

Figure 1: A car's geometric information is called car model

Figure 1 illustrates a complex geometrical car model that is defined in a convenient coordinate system called the *model coordinate system*. The coordinates we use to define the model are called *model coordinates*. To specify an instance of the car model in a game world, we need to apply a *model transformation matrix* [or *model-to-world transformation matrix*] to transform model coordinates of the model into the world coordinate system that defines the game world. An instance of a model is called an *object*. Figure 2 illustrates geometry instancing with the application of the model transformation M to create a car object from a car model.

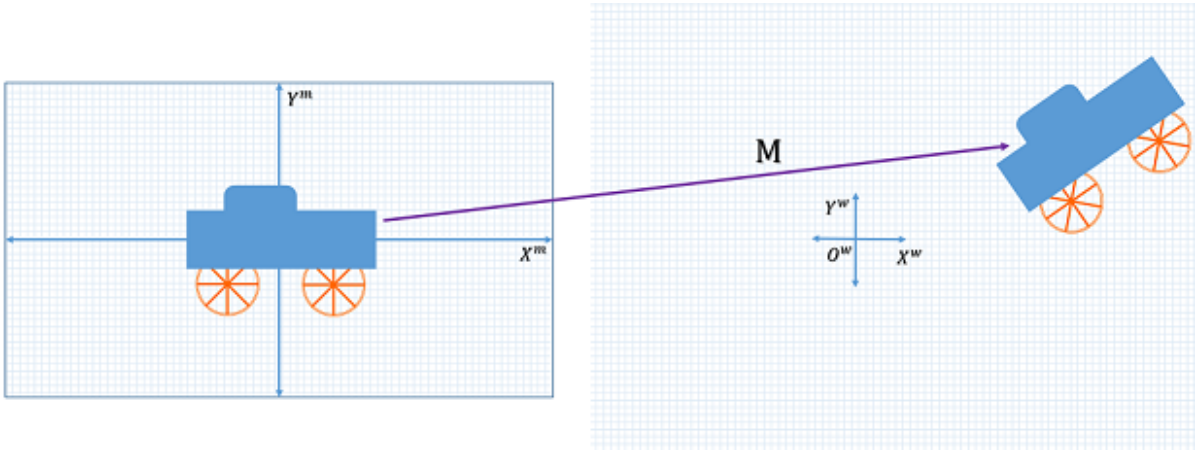

Figure 2: Model transformation matrix of car model into car object

If our game requires three car objects, we'll need to transform the car model using three different model transformations into the world coordinate system:

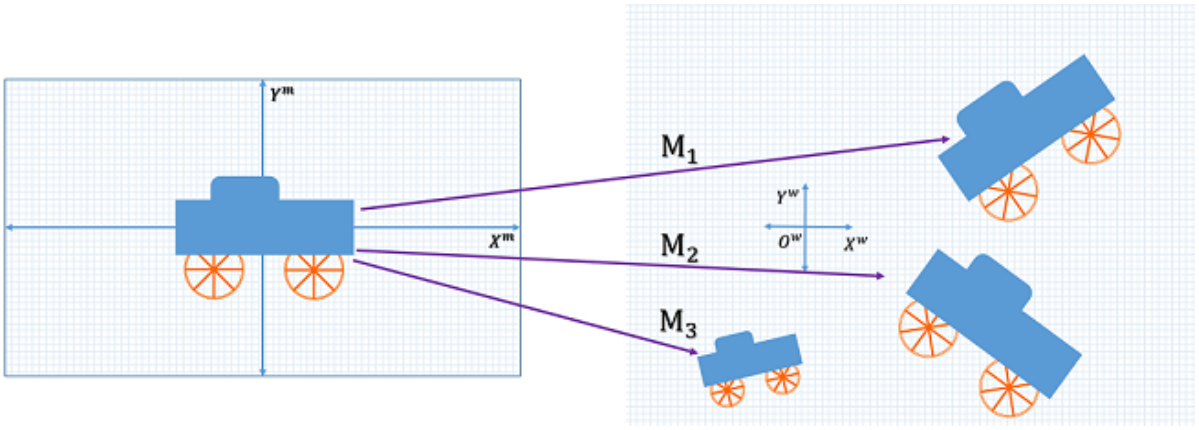

Figure 3: Many car object instantiations of a car model

So, the important idea here is that a game can define a single geometrical *model* in *model coordinates* and use a *model transformation matrix* to create an *object* [or, instance of the model] in the world. If the game requires n car objects in the world, then it will require n model transformation matrices - one matrix per car object - applied on a single car model. The advantage of geometric instancing is that every rendered object doesn't require storage for its own unique geometry. Instead, by using different model transformation matrices, the same geometry can be uniquely rendered as different objects. Without geometric instancing, n car objects in the world will require n model transformation matrices ***and*** n geometries.

Task 0a: Encoding Models

In Tutorial 2, a model's geometric details were abstracted in structure `GLApp::GLModel`:

```
1  struct GLApp {
2    // other stuff ...
3
4    struct GLModel {
5      GLenum     primitive_type;
6      GLuint     primitive_cnt;
7      GLuint     vaoid;
8      GLuint     draw_cnt;
9      GLSLShader shdr_pgm;
10
11     void setup_shdrpgm(std::string const& vtx_shdr,
12                        std::string const& frg_shdr);
13     void draw();
14   };
15   static std::vector<GLModel> models; // singleton
16 };
```

Two issues can be identified for improvement in structure `GLApp::GLModel`. The first obvious disadvantage is that the idea of geometrical instancing did not exist in previous tutorials since each model of type `GLApp::GLModel` was directly rendered. This means that to render thousands of instances of a particular shape, each rendered shape must individually and redundantly define the same geometry leading to wasted use of critical GPU buffer storage. The second issue is that structure `GLApp::GLModel` combines two separate concerns: geometry *representation* using a VAO handle and geometry *rendering* using a shader program. An obvious consequence of combining

these two separate concerns is the unnecessary duplication of the same shader program across many instances of the same shape. Not only does duplication of the same shader program use up critical GPU buffer storage but the applications startup time also increases because of the unnecessary repeated compilation, linking, and validation of the same shader program.

These disadvantages can be alleviated by using both geometric instancing and shader program instancing. With *geometric instancing*, the application will define a small number of discrete models and use a model container as a repository for these discrete models. Redundant use of geometry is avoided when the application defines a large set of game objects by having each object reference a specific model in the model container. Shader instancing is implemented using a similar approach. The application uses a shader program container as a repository for the discrete shader programs required by the application. Each instance of a model is then rendered using a reference to a specific shader in the shader program container. The combination of geometry instancing and shader instancing is not only memory efficient but also flexible. The application now has the ability to populate the game world efficiently with instantiations of a variety of models that can also be rendered differently using a variety of previously compiled and linked shader programs that are encapsulated in a shader program container.

Begin by amending structure `GLApp::GLModel` [in glapp.h] to *only* encapsulate a model's geometry and to *exclude* the shader program that would render the geometry:

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

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

    // no longer required in tutorial 3
    //GLSLShader shdr_pgm;
    //void setup_shdrpgm(std::string vtx_shdr, std::string frg_shdr);
    //void draw();
  };
  static std::vector<GLApp::GLModel> models; // same as tutorial 2
};
```

Task 0b: Initialize container of models

Recall that Tutorial 2 used static functions `GLApp::points_model`, `GLApp::lines_model`, `GLApp::tristrips_model`, and `GLApp::trifans_model` to instantiate a geometric shape of type `GLApp::GLModel`. This tutorial will require two models. The first model is the familiar box centered at $(0, 0)$ with size 1×1. The second model is a *mystery* geometric shape that must be constructed from triangles and can neither be a box nor a circle [from the previous tutorial] nor the star [from this tutorial's sample]. This is an opportunity to use your imagination and creativity - see here for some ideas on constructing this mystery shape. Both models consist of both vertex position and color attributes.

Begin by providing the definitions of the following functions that construct a box and a mystery model, respectively:

```
struct GLApp {
  // other stuff ...
  static GLApp::GLModel box_model();
  static GLApp::GLModel mystery_model();
};
```

Next, models used by the application must be inserted into container `GLApp::models`. Declare a static function `GLApp::init_models_cont` [in glapp.h]:

```
struct GLApp {
  // other stuff ...
  static void init_models_cont();
};
```

and define the function [in glapp.cpp]:

```
void GLApp::init_models_cont() {
  GLApp::models.emplace_back(GLApp::box_model());
  GLApp::models.emplace_back(GLApp::mystery_model());
}
```

The final step is to setup a single viewport that encompasses the entire display window and invokes static function `GLApp::init_models_cont` from function `GLApp::init` [in glapp.cpp]:

```
void GLApp::init() {
  // other stuff ...

  // no longer required in tutorial 3
  //int hs = h / 3;
  //int ws = hs;
  //GLApp::vps.push_back({ 0, 0, w - ws, h });
  //GLApp::vps.push_back({ w - ws, 2 * hs, ws, hs });
  //GLApp::vps.push_back({ w - ws, hs, ws, hs });
  //GLApp::vps.push_back({ w - ws, 0, ws, hs });
  GLApp::vps.push_back({ 0, 0, GLHelper::width, GLHelper::height });

  // no longer required for tutorial 3
  // make sure to remove definitions of these functions too ...
  //GLApp::models.emplace_back(GLApp::points_model(20, 20,
  //                     my_tutorial_3_vs, my_tutorial_3_fs));
  //GLApp::models.emplace_back(GLApp::lines_model(40, 40,
  //                     my_tutorial_3_vs, my_tutorial_3_fs));
  //GLApp::models.emplace_back(GLApp::trifans_model(50,
  //                     my_tutorial_3_vs, my_tutorial_3_fs));
  //GLApp::models.emplace_back(GLApp::tristrip_model(10, 15,
  //                     my_tutorial_3_vs, my_tutorial_3_fs));
  GLApp::init_models_cont();
}
```

Task 0c: Initializing container of shader programs

Recall that a shader program consists of a vertex shader and a fragment shader that are individually compiled and then linked together into a single program. Such a shader program can be used to render different compatible models. It is more convenient to group all the shader programs used by the application into a container, say `GLApp::shdrpgms` that is declared as a static data member [in glapp.h]:

```
struct GLApp {
  // other stuff ...
  static std::vector<GLSLShader> shdrpgms; // singleton in tutorial 3
};
```

Recall that C++ requires a static data member of a structure or class to be defined in a source file. The definition of static data member `GLApp::shdrpgms` [in source file glapp.cpp] looks like this:

```
std::vector<GLSLShader> GLApp::shdrpgms;
```

How are discrete shader programs inserted into container `GLApp::shdrpgms`? The deprecated member function `GLModel::setup_shdrpgm` is now refactored as static member function `GLApp::init_shdrpgms_cont` [the change in the function's name is to avoid confusion] that is declared in glapp.h:

```
struct GLApp {
  // other stuff ...
  using VPSS = std::vector<std::pair<std::string, std::string>>;
  static void init_shdrpgms_cont(GLApp::VPSS const&);
};
```

and is defined in glapp.cpp:

```
// create a shader program from each pair of shader strings in vector
// vpss and then insert that shader program into container
// std::vector<GLSLShader> GLApp::shdrpgms
void GLApp::init_shdrpgms_cont (GLApp::VPSS const& vpss) {
  for (auto const& x : vpss) {
    GLSLShader shdr_pgm;
    if (!shdr_pgm.CompileShaderFromString(GL_VERTEX_SHADER, x.first)) {
      std::cout << "Vertex shader failed to compile: ";
      std::cout << shdr_pgm.GetLog() << std::endl;
      std::exit(EXIT_FAILURE);
    }
    if (!shdr_pgm.CompileShaderFromString(GL_FRAGMENT_SHADER, x.second)) {
      std::cout << "Fragment shader failed to compile: ";
      std::cout << shdr_pgm.GetLog() << std::endl;
      std::exit(EXIT_FAILURE);
    }

    if (!shdr_pgm.Link()) {
      std::cout << "Shader program failed to link!" << std::endl;
      std::exit(EXIT_FAILURE);
    }
```

```
22
23      if (!shdr_pgm.Validate()) {
24        std::cout << "Shader program failed to validate!" << std::endl;
25        std::exit(EXIT_FAILURE);
26      }
27
28      // insert shader program into container
29      GLApp::shdrpgms.emplace_back(shdr_pgm);
30    }
31  }
```

This tutorial requires a default shader program consisting of a vertex shader [in file my-tutorial-3.vert] and a fragment shader [in file my-tutorial-3.frag]. Function `GLApp::init_shdrpgms_cont` is invoked from `GLApp::init` [defined in glapp.cpp] to insert the shader program obtained after compiling and linking the `std::string`s encapsulated in these two shader files like this:

```
1  void GLApp::init() {
2    // other stuff ...
3
4    // container of pairs of std::strings with each pair encapsulating
5    // a vertex shader and a fragment shader
6    GLApp::VPSS shdr_strs {
7      std::make_pair(my_tutorial_3_vs, my_tutorial_3_fs)
8    };
9
10   // create a shader program from each pair of shader files in container
11   // shdr_strs and insert that shader program into container
12   // std::vector<GLSLShader> GLApp::shdrpgms:
13   GLApp::init_shdrpgms_cont(shdr_strs);
14
15   // more other stuff ...
16 }
```

Task 0d: Encoding objects as instances of `GLApp::GLModel`

The next step is to define structure `GLApp::GLObject` [in glapp.h] that encapsulates the state necessary to define a object. Such state includes a reference to a specific model in the model container `GLApp::models`, a reference to a specific shader in the shader program container `GLApp::shdrpgms` that will then be used to render the referenced model, and attributes necessary to compute a model transformation matrix that specifies the transformation of a model [in its model coordinate system] to an object [in the world coordinate system]. Further, a static data member `GLApp::objects` must be defined as a container of objects of type `GLApp::GLObject`:

```
1  struct GLApp {
2    // other stuff ...
3
4    // Tutorial 3: encapsulate state required to update
5    // and render an instance of a model
6    struct GLObject {
7      // These two variables keep track of the current orientation
8      // of the object:
9      // angle_speed: rate of change of rotation angle per second
```

```
    // angle_disp: current absolute orientation angle
    // All angles refer to rotation about Z-axis and it is up to you
    // whether angles are tracked in degrees or radians
    GLfloat angle_speed, angle_disp;

    glm::vec2 scaling; // scaling parameters
    glm::vec2 position; // translation vector coordinates

    // Compute objet's model transform matrix using scaling,
    // rotation, and translation attributes ...
    // Make sure to understand why the CPU must compute this matrix
    // and not the GPU
    glm::mat3 mdl_to_ndc_xform;

    // Which model is this object an instance of?
    // Since models are contained in a vector, we keep track of the
    // specific model that was instanced by index into vector container
    GLuint mdl_ref;

    // How to draw this instanced model?
    // Since shader programs are stroed in a container, we keep track of
    // the specific shader program using an index into that container
    GLuint shd_ref;

    // Following member functions must be defined in glapp.cpp:
    void init(); // Function to initialize object's state

    // Function to render object's model (specified by index mdl_ref)
    // using shader program specified by index shd_ref ...
    void draw() const;

    // Function to update the object's model transformation matrix
    void update(GLdouble delta_time);
  };
  static std::list<GLApp::GLObject> objects; // singleton
};
```

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

Begin by adding code in function `GLApp::init` [in glapp.cpp] to incorporate parts 1 through 4. Most of this code has been explained or implemented in the earlier section and previous tutorials.

```
// define singleton containers
std::vector<GLSLShader>      GLApp::shdrpgms;
std::vector<GLApp::GLModel> GLApp::models;
std::list<GLApp::GLObject>   GLApp::objects;

// define strings to contain shader code ...
std::string const my_tutorial_3_vs {
  #include "my-tutorial-3.vert"
};
std::string const my_tutorial_3_fs {
  #include "my-tutorial-3.frag"
};

```

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

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

  // Part 3: create as many shared shader programs as required and
  // store the handles to these shader programs in GLApp::shdrpgms

  // Part 4: initialize as many geometric models as required and
  // store these geometric models in GLApp::models

  // We don't need to add any objects to container GLApp::objects
  // since the simulation begins with no objects being displayed
}
```

🎯 Task 2: Updating function `GLApp::update`

Implement parts 1, 2, and 3 of `GLApp::update` [in file glapp.cpp] whose outline looks like this:

```
void GLApp::update() {
  // Part 0: As in Tutorial 2, adapt viewport to resized window

  // Part 1: Update polygon rasterization mode using glPolygonMode
  // Check if key 'P' is pressed
  //   If pressed, update polygon rasterization mode

  // Part 2: Spawn or kill objects ...
  // Check if left mouse button is pressed
  //   If maximum object limit is not reached, spawn new object(s)
  //   Otherwise, kill oldest objects

  // Part 3:
  // for each object in container GLApp::objects
  //   Update object's orientation
  //   A more elaborate implementation would animate the object's movement
  //   A much more elaborate implementation would animate the object's size
  //   Using updated attributes, compute model-to-ndc transformation matrix
}
```

The following sections will briefly describe the processing of keyboard and mouse events, initializing attributes of a spawned object, and computing its model-to-NDC transformation matrix.

Task 2a: Processing keyboard and mouse events

1. GLFW provides an extensive interface to process input events. First, a callback function is declared for every event to be processed. The starter code from Tutorial 1 contains declarations of callback functions for keyboard buttons, mouse buttons, mouse position, and mouse scrolls [in glhelper.h]:

```
struct GLHelper {
  // other stuff here ...

  // I/O callbacks ...
  static void key_cb(GLFWwindow*, int key, int scancode,
                     int action, int mod);
  static void mousebutton_cb(GLFWwindow*, int button,
                             int action, int mod);
  static void mousescroll_cb(GLFWwindow*,
                             double xoffset, double yoffset);
  static void mousepos_cb(GLFWwindow*, double xpos, double ypos);
};
```

File glhelper.cpp contains skeleton definitions of these functions.

In function `GLHelper::init` [in file glhelper.cpp], each of these callback functions is associated with the appropriate input event using GLFW interface functions:

```
bool GLHelper::init(GLint w, GLint h, std::string t) {
  // other stuff here ...

  glfwSetKeyCallback(GLHelper::ptr_window, GLHelper::key_cb);
  glfwSetMouseButtonCallback(GLHelper::ptr_window,
                             GLHelper::mousebutton_cb);
  glfwSetCursorPosCallback(GLHelper::ptr_window,
                           GLHelper::mousepos_cb);
  glfwSetScrollCallback(GLHelper::ptr_window,
                        GLHelper::mousescroll_cb);
}
```

For example, in line 4, GLFW interface function `glfwSetKeyCallback` is called to associate the user-defined callback function `GLHelper::key_cb` with keyboard button events.

Once every frame, `glfwPollEvents()` is called in function `update` [defined in main.cpp] to process events in the input event queue. Events are processed by calling the callbacks associated with these events. When you press a keyboard button, this keyboard event is placed in the input event queue. Once every frame, function `glfwPollEvents` is invoked to process all the events in the event queue. If a keyboard event has occurred [through the process of pressing a keyboard button], this event is processed by invoking function `key_cb` with the appropriate arguments.

2. Use GLFW documentation on keyboard input and mouse input to implement code that updates the polygon rendering mode for the application and to spawn or kill objects. An explanation of reading keyboard events with function `GLApp::key_cb` to update polygon rendering mode is described below. Updating the definition of `GLApp::mousebutton_cb` is left to the reader as an exercise.
3. At present, function `GLApp::key_cb` contains code to quit the application when Esc button is pressed:

```
void GLHelper::key_cb(GLFWwindow *pwin, int key, int scancode,
                      int action, int mod) {
  if (GLFW_KEY_ESCAPE == key && GLFW_PRESS == action) {
    glfwSetWindowShouldClose(pwin, GLFW_TRUE);
  }
}
```

The Command Pattern is commonly used in games and similar software to provide a separation of responsibilities between game logic and input functions. Instead of the Command pattern, a much simpler strategy involves declaring a static Boolean data member in structure `GLHelper` to record the current state of button P. The state of this flag can then be queried by a function defined in `GLApp` to update the polygon rendering mode. Start by declaring the Boolean static data member in structure `GLHelper` [in glhelper.hpp]:

```
struct GLHelper {
  // other stuff from before ...

  // this flag is true if button P was toggled from released
  // (or not pressed) position to pressed
  static GLboolean keystateP;
};
```

Define the static data member in file glhelper.cpp:

```
GLboolean GLHelper::keystateP = GL_FALSE;
```

Update callback function `GLHelper::key_cb` to record the current state of keyboard button P in `GLHelper::keystateP`:

```
void GLHelper::key_cb(GLFWwindow *pwin, int key, int scancode, int
action, int mod) {
  // key state changes from released to pressed
  if (GLFW_PRESS == action) {
    if (GLFW_KEY_ESCAPE == key) {
      glfwSetWindowShouldClose(pwin, GLFW_TRUE);
    }
    keystateP = (key == GLFW_KEY_P) ? GL_TRUE : GL_FALSE;
  } else if (GLFW_REPEAT == action) {
    // key state was and is being pressed
    keystateP = GL_FALSE;
  } else if (GLFW_RELEASE == action) {
    // key start changes from pressed to released
    keystateP = GL_FALSE;
  }
}
```

The state of `GLHelper::keystateP` can subsequently be accessed by other functions to determine the specific rendering mode to be set

Task 2b: Initializing objects of type `GLApp::GLObject`

1. Let's recap implementation details so far:
   - As described in Task 0a, container `GLApp::models` encapsulates two models. The first model is the familiar box centered at $(0, 0)$ with size 1×1 defined in NDC containing both vertex position and color attributes. The second model is a *mystery* geometric shape that must be constructed from triangles and can neither be a box nor a circle nor a star. Previous tutorials have described the process of setting up vertex buffer and array objects for these models.
   - As with $\text{Tutorial } 1$, assume all objects are uniformly rendered with vertex color attribute interpolation. This means that a single shader program is sufficient to render every object in the scene. It also means that the fragment shader from $\text{Tutorial } 1$ will suffice for this tutorial. However, the vertex shader will be different and more details are provided here. Even though the application currently requires only a single shader program, use a `std::vector` container to store GLSL shader programs of type `GLSLShader` [as described in Task0c]. Again, using this container framework will allow you to rapidly add other shader programs to the application.
2. In Task 0d, we suggested a new type `GLApp::GLObject` to encapsulate the information required to describe an instance of a model and a static data member `GLApp::objects` of type `std::list` as a container of objects of type `GLApp::GLObject`:

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

  // encapsulates rendered object
  struct GLObject {
    // angular speed and angular displacement are with respect to
    // X-axis and together represent the orientation of an object
    GLfloat angle_speed, angle_disp; // orientation
    glm::vec2 position, scaling; // translation and scaling
    // matrix to map geometry from model to world to NDC coordinates
    glm::mat3 mdl_to_ndc_xform;
    GLuint mdl_ref, shd_ref; // which model and which shader

    void init(); // function to initialize object's state

    // function to render object's model (specified by index mdl_ref)
    // uses model transformation matrix mdl_to_ndc_xform matrix
    // and shader program specified by index shd_ref ...
    void draw() const;

    // function to update angle_disp and then compute mdl_to_ndc_xform
    void update(GLdouble delta_time);
  };
  static std::list<GLApp::GLObject> objects; // singleton

  // other stuff ...
};
```

Since objects of type `GLApp::GLObject` come alive at runtime and those with the oldest lifetime die at runtime, you should think carefully about the container type [`std::vector` versus our suggestion of `std::list`] used to store such objects.

3. Every time a new object [of type `GLApp::GLObject`] is to be spawned, you must think how its data members are initialized by static member function `GLObject::init`.

> **Although `GLObject::init` is declared to take zero parameters in this document, you can modify the declaration of the function to suit your needs.**

4. First, you must decide which of the two models [unit box or mystery shape] is instantiated. The application must randomly choose between the two indices 0 and 1 to assign to data member `GLObject::mdl_ref`.

5. Assuming at present only a single shader program is contained in `GLApp::shdrpgms`, data member `GLObject::shd_ref` must be assigned value 0.

6. Next, the objects' world position parameters, scaling factors to represent object's size, initial angular displacement, and angular speed per frame must be initialized. Together angular speed and angular displacement represent the object's orientation and are defined with respect to the x—axis. Since the application will spawn large numbers of `GLApp::GLObject` objects at runtime, it will be extremely inefficient and tedious to hand-initialize multiple attributes for each spawned object. Using randomly generated values to initialize these attributes is a better alternative. If you're more familiar with the C standard library random number generator `rand` [from a previous tutorial], then watch this presentation to understand why you should stop using `rand` and instead switch to the C++ standard library. If you've not used the C++ random number generation library presented in `<random>`, this is a good opportunity to make the switch.

7. The C++ random number number generation library presented in `<random>` produces random numbers using combinations of *generators* and *distributions*.

   Generators are objects that produce uniformly distributed numbers. A uniform distribution allows any value in a certain range to have an equal opportunity to appear, thereby preventing the randomness to be skewed towards a set of values. All standard generators defined in the library are *random number engines*, which are types of generators that each use a particular algorithm to generate a series of pseudo-random numbers. These algorithms need a seed as a source of randomness. The standard library provides a generator class `std::default_random_engine` for generating a sequence of unsigned values that provides at least acceptable engine behavior for relatively casual, inexpert, and lightweight use. The following code snippet illustrates the use of this generator:

```
#include <iostream>
#include <random>

int main() {
  std::default_random_engine dre1;

  std::cout << dre1() << ", " << dre1() << ", " << dre1 << '\n';
  std::cout << "Range: [" << dre1.min() << ", " << dre1.max() << "]\n";

  // you can initialize the generator with a seed to produce a
  // different series of unsigned values ...
  std::default_random_engine dre2(12345678);
  std::cout << dre2() << ", " << dre2() << ", " << dre2 << '\n';
```

```
  std::cout << "Range: [" << dre2.min() << ", " << dre2.max() << "]\n";
}
```

The output produced by the above code fragment looks like this:

```
16807, 282475249, 1622650073
Range: [1, 2147483646]
1335380034, 380636641, 6240874
Range: [1, 2147483646]
```

The same output is produced every time you execute the program. To obtain a different series of values every time you run the program, you need to initialize the engine with a random seed. You can use the computer's internal clock or use an object of type `std::random_device` that uses information about your computer's hardware to spit out randomly generated seeds:

```
std::random_device rd;

// initialize a default_random_engine with a randomly generated seed
std::default_random_engine dre3(rd());
// generate series of unsigned values that are different each
// time the program is executed
std::cout << dre3() << " " << '\n';
std::cout << dre3() << " " << '\n';
std::cout << dre3() << " " << '\n';
```

Distributions are objects that transform sequences of numbers generated by a generator into sequences of numbers that follow a specific random variable distribution, such as Uniform distribution, Normal distribution, or Binomial distribution. The following code snippet illustrates the generation of values between $[-100, 100)$ that each have equal probability of appearing:

```
#include <iostream>
#include <random>

int main() {
  std::random_device rd;
  std::default_random_engine dre4(rd());
  // initialize an object of type uniform_int_distribution
  // to transform values generated by engine dre4 which are
  // in range [1, 2147483646] (on my machine) to a range [-100,100)
  // so that each value has equal probability of appearing:
  std::uniform_int_distribution<int> uid(-10, 10);
  // generate a few values:
  for (int i{}; i < 5; ++i) {
    std::cout << uid(dre4) << ", ";
  }
  std::cout << '\n';
}
```

8. Since different data members of spawned objects will require different ranges of randomly generated values, it is simplest to use `<random>` to generate random floating-point values in range $[-1.0, 1.0]$ which can then be easily mapped to any desired range. This is possible by defining a default-initialized object of type `std::default_random_engine`. The random unsigned values generated by this engine are uniformly distributed in range $[-1.0, 1.0]$ by a distribution object of type `uniform_real_distribution<GLfloat>`. It is important to note that the range provided to `uniform_real_distribution<GLfloat>` is half-open on the right. You'll have to do some research to specify an inclusive, closed range $[-1.0, 1.0]$.

9. Now that you're able to generate random values in closed range $[-1.0, 1.0]$, let's look at how such values can be used to initialize the world position of an object of type `GLApp::GLObject`. Suppose the game world for this tutorial is a 2D square box centered at $(0, 0)$ with dimensions $10{,}000 \times 10{,}000$ so that x^w and y^w coordinates [superscript w indicates $world$ coordinates] of any point inside this game world will have range $[-5000, 5000]$. Also, suppose the previously defined geometric model of type `GLApp::GLModel` encapsulates a 2D unit square box with dimensions 1×1 centered at $(0, 0)$. To randomly position this 2D unit box in the game world, the object's x^w coordinate is computed by mapping a random value in range $[-1.0, 1.0]$ to range $[-5000, 5000]$. The objects' y^w coordinate is computed similarly by mapping another random value in range $[-1.0, 1.0]$ to range $[-5000, 5000]$. The following code fragment illustrates this mapping:

```
// create default engine as source of randomness
std::default_random_engine dre;
// get numbers in range [-1, 1) - notice that interval is half-
// open on the right. When you submit the tutorial, specify a
// closed range [-1, 1]
std::uniform_real_distribution<GLfloat> urdf(-1.0, 1.0);

GLfloat const world_half_extent {5000.0f};

GLApp::GLObject go;
// position object in game world such that its x and y coordinates
// are in range [-5,000, +5,000) ...
go.position = glm::vec2(urdf(dre) * world_half_extent,
                        urdf(dre) * world_half_extent);
// a call urdf(dre) gives a new random value that is uniformly
// distributed in range [-1.0, 1.0)
```

10. Suppose you want objects to be displayed as *rectangular* boxes whose widths and heights range from 50 to 400 units. Since the model of type `GLApp::GLModel` encapsulates a unit *square* box with dimensions 1×1, each object must be initialized with non-uniform scaling factors. That is, each object must have different x and y scaling factors. Given a random value in range $[-1.0, 1.0]$, how will you map this value to the range $[50.0, 400.0]$? Once you know how to compute this mapping, you can then randomly compute the x and y scaling factors for an object such that their values are in range $[50.0, 400.0]$.

11. The initial angular displacement and angular speed [per frame] represent the final set of attributes that must be randomly initialized for each spawned object. The initial angular displacement simply denotes the initial orientation [with respect to the $x-$axis] of the object when it is spawned - this could be any value in range $[-360^\circ, 360^\circ]$. Limit the angular speed to values in range $[-30^\circ, 30^\circ]$ so that objects don't rotate too fast. The following code fragment illustrates the initialization of these two object attributes:

```
// initialize position of object in game world ...
// initialize non-uniform scaling factors represent object's size
// in game world

// initialize initial angular displacement and angular speed of object
go.angle_disp  = urdf(dre)*360.f; // current orientation
go.angle_speed = urdf(dre)*30.f; // degrees per frame
```

Task 2c: Updating objects of type `GLApp::GLObject`

The task of function `GLObject::update` is to compute an object's model-to-world-to-NDC transformation matrix. 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 rotation matrix R, the current orientation must be computed:

$$angle\_disp \quad += angle\_speed$$

Since $angle\_disp$ 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:

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

The 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:

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

M_{model} maps incoming vertex coordinates to the game world whose extents were specified earlier in this document as $x^w \in [-5000, 5000]$ and $y^w \in [-5000, 5000]$. However, the vertex shader has the responsibility to map incoming vertex coordinates to NDC coordinates. Recall that the OpenGL specification defines the 2D NDC box extents as $x^n \in [-1, 1]$ and $y^n \in [-1, 1]$. Therefore, an additional matrix is required to map $x^w \in [-5000, 5000]$ to $x^n \in [-1, 1]$ and $y^w \in [-5000, 5000]$ to $y^n \in [-1, 1]$. A scaling matrix that engineers this mapping can be written as:

$$\mathrm{H}_{\frac{1}{5000}} = \begin{bmatrix} \frac{1}{5000} & 0 & 0 \\ 0 & \frac{1}{5000} & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

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

$$\begin{aligned} \mathrm{M}_{model-to-NDC} &= \mathrm{H}_{\frac{1}{5000}} \circ \mathrm{M}_{model} \\ &= \mathrm{H}_{\frac{1}{5000}} \circ \mathrm{T} \circ \mathrm{R} \circ \mathrm{H} \end{aligned}$$

Obviously, if you've chosen different game world extents, the translation and scaling matrices would be different than the ones illustrated above.

> **Although `GLObject::update` is declared to take a single parameter in this document, you can add additional parameters to suit your needs.**

🎯Task 3: Drawing objects of type `GLApp::GLObject`

Your task is to implement parts 1 through 4 of `GLApp::draw` [in file glapp.cpp] whose outline looks like this:

```
void GLApp::draw() {
  // Part 1: Write window title with exact same information as sample
  //   Print DEFAULT when using the usual shader program that interpolates
  //   vertex color attributes
  //   Print RED or GREEN or BLUE when using the shader program that uses
  //   red, green, or blue color for fragments

  // Part 2: Clear back buffer of color buffer

  // Part 3: Special rendering modes
  // Use status of flag GLHelper::keystateP to set appropriate polygon
  // rasterization mode using glPolygonMode
  // if rendering GL_POINT, control diameter of rasterized points
  // using glPointSize
  // if rendering GL_LINE, control width of rasterized lines
  // using glLineWidth

  // Part 4: Render each object in container GLApp::objects
  for (auto const& obj : GLApp::objects) {
    obj.draw(); // call member function GLObject::draw()
  }
}
```

The rest of this section deals with the implementation of part 4. In function `GLApp::draw`, the work of drawing an object is handled by member function `GLObject::draw`. An overview of this member function looks like this:

```
void GLApp::GLObject::draw() const {
  // Part 1: Install the shader program used by this object to
  // render its model using GLSLShader::Use()

  // Part 2: Bind object's VAO handle using glBindVertexArray

  // Part 3: Copy object's 3x3 model-to-NDC matrix to vertex shader

  // Part 4: Render using glDrawElements or glDrawArrays

  // Part 5: Clean up
  // Breaking the binding set up in Part 2: glBindVertexArray(0);
  // Deinstall the shader program installed in Part 1 using
  // GLSLShader::UnUse()
```

```
15 }
```

Task 3a: Updating the vertex shader

Your vertex shader [in my-tutorial-3.vert] contains the following shader source or something similar:

```
1  R"( #version 450 core
2
3  layout (location=0) in  vec2 aVertexPosition;
4  layout (location=1) in  vec3 aVertexColor;
5
6  layout (location=0) out vec3 vColor;
7
8  void main() {
9    gl_Position = vec4(aVertexPosition, 0.0, 1.0);
10   vColor      = aVertexColor;
11 }
12 )"
```

Shader `in`/`out` variables

Recall that a *vertex shader* is an algorithm expressed in GLSL that executes on a *vertex processor* to produce output values based on the provided input values. A *vertex attribute* is simply the information provided to a vertex shader, on a per-vertex basis. Examples of vertex attributes include per-vertex values of model coordinates, color, normal, normal matrix, or texture coordinates. Vertex shaders can access vertex attributes through user-defined variables. Whether a variable name represents input or output values is determined by `in` and `out` type qualifiers.

To extract maximum performance, applications commonly specify vertices in vertex arrays allowing you to store vertex data in vertex buffer objects and set offsets to those buffers. When a draw call such as `glDrawElements` is issued while an appropriate vertex array object is bound, the vertex puller will stream vertex data from associated vertex buffers, one vertex at a time. Each vertex attribute from the fetched data is copied to an `in` variable in the vertex shader.

From lines 3 and 4 of the vertex shader, the vertex shader is expecting the vertex puller to copy (x, y) model coordinates and (r, g, b) color coordinates of each vertex into `in` variable `aVertexPosition` and `in` variable `aVertexColor`, respectively.

The purpose of the vertex shader is to take the inputs in variables `aVertexPosition` and `aVertexColor` and transform them into outputs using the algorithm specified in the `main` function.

Built-in variable `gl_Position` on line 9 represents a special output variable that writes the vertex position in clipping coordinates after it has been computed in a vertex shader. Every vertex shader must *minimally* generate a result in variable `gl_Position` that has type `vec4`. Otherwise, results of primitive assembly and rasterization are undefined if a vertex shader is executed and it does not store a value into `gl_Position`. `gl_Position` is a key variable because the clipping coordinates assigned to it provide necessary information to the fixed functionality stages between vertex processing and fragment processing, namely, primitive assembly, clipping, culling, and rasterization.

The `out` variable `vColor` on line 6 represents an output value supplied by the vertex shader to the fragment shader. In this case, on line 10, the vertex shader just passes through the input vertex color attribute in `aVertexColor` as output to `out` variable `vColor` without affecting any transformations. Output values such as `vColor` are subject to subsequent processing by fixed stages such as clipping and interpolation.

`layout` qualifier

For vertex attribute variables, the application can specify - before linking occurs - the attribute *slot* or *index* or *location* through which the variables are initialized. For example, the following code fragment containing [a possible] VAO setup for both position and color attributes is reproduced from Tutorial 2:

```
1  glCreateVertexArrays(1, &vaoid);
2  glEnableVertexArrayAttrib(vaoid, 0);
3  glVertexArrayVertexBuffer(vaoid, 0, vbo_hdl, 0, sizeof(glm::vec2));
4  glVertexArrayAttribFormat(vaoid, 0, 2, GL_FLOAT, GL_FALSE, 0);
5  glVertexArrayAttribBinding(vaoid, 0, 0);
6  glEnableVertexArrayAttrib(vaoid, 1);
7  glVertexArrayVertexBuffer(vaoid, 1, vbo_hdl,
8        sizeof(glm::vec2)*pos_vtx.size(), sizeof(glm::vec3));
9  glVertexArrayAttribFormat(vaoid, 1, 3, GL_FLOAT, GL_FALSE, 0);
10 glVertexArrayAttribBinding(vaoid, 1, 1);
```

On lines 2 and 6, VAO with handle `vaoid` is updated - through calls to OpenGL command `glEnableVertexArrayAttrib` - with information that slots 0 and 1 are associated with vertex position and color attributes, respectively. While OpenGL code refers to vertex attributes using indices, we can explicitly specify the relationship between these slots and attributes in a shader using a `layout` qualifier. For example, in the following code fragment from a vertex shader:

```
1  #version 450 core
2
3  layout (location = 0) in vec2 aVertexPosition;
4  layout (location = 1) in vec3 aVertexColor;
```

we use the `layout` qualifier to assign data read from attribute slot 0 to user-defined `in` variable `aVertexPosition` and to assign data read from attribute slot 1 to user-defined `in` variable `aVertexColor`.

`uniform` variables

In Task 2c, a 3×3 model-to-NDC transformation matrix was computed to specify the size, orientation, and position of every rendered object. This matrix is stored in data member `mdl_to_ndc_xform` of an instance of type `GLApp::GLObject`. The values in matrix `mdl_to_ndc_xform` change each time function `GLApp::GLObject::display` is called and thus `mdl_to_ndc_xform` is different for each frame. Even though it is the OpenGL application that generates this matrix every frame for each object to be displayed, it is most efficient for the vertex shader to apply this matrix on the position coordinates of each models' vertices specified in `in` variable `aVertexPosition`. This is only possible if the vertex shader can obtain access to the values in data member `mdl_to_ndc_xform` of each object.

Thus, in addition to requiring per-vertex attributes such as position, color, and texture coordinates stored in vertex buffer objects, vertex shaders require a second method of accessing data that is specified for all vertices of a model [such as the elements of matrix `mdl_to_ndc_xform`]. The mechanism that performs this copy of the matrix `mdl_to_ndc_xform` from the OpenGL application to the vertex shader uses the concept of a *uniform variable*. The term *uniform* probably comes from the idea that this data remains the same for multiple executions of a shader while the per-vertex attribute data is *varying* data because it is unique for each execution of the vertex shader. A parameter that is passed from the application to either or both vertex and fragment shaders is called a *uniform variable*. A uniform variable is read-only and is global to the shader in which it is defined.

Uniform variables are defined in a shader using keyword `uniform`. The following example, which defines a variable to hold the 3×3 model-to-NDC transformation matrix, will be suitable for this tutorial:

```
R"(  #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;
}
)"
```

Keyword `mat3` indicates that `uModel_to_NDC` is defined as a 3×3 matrix. Keyword `uniform` indicates that values in variable `uModel_to_NDC` are read-only until they are changed by the OpenGL application. This matrix is used by the vertex shader to transform model coordinates into NDC coordinates and write the resultant NDC coordinates to built-in output variable `gl_Position`.

To satisfy laws of matrix multiplication and prevent the GLSL compiler from emitting errors, 3×3 matrix `uModel_to_NDC` must transform a 3×1 matrix. Therefore, constructor `vec3(aVertexPosition, 1.f)` will instantiate from 2×1 matrix $[\mathrm{aVertexPosition.x}\ \mathrm{aVertexPosition.y}]^{\mathrm{T}}$ a 3×1 matrix $[\mathrm{aVertexPosition.x}\ \mathrm{aVertexPosition.y}\ 1.0]^{\mathrm{T}}$ which is then pre-multiplied by 3×3 matrix `uModel_to_NDC`. The resultant 3×1 matrix representing transformed `vec3` NDC coordinates are converted to `vec2` NDC coordinates by constructor `vec2(uModel_to_NDC * vec3(vVertexPosition, 1.f))`. The subsequent `vec4` constructor will augment the `vec2` NDC coordinates to `vec4` clip coordinates with z coordinate set to 0 and homogeneous coordinate set to 1. The resultant `vec4` clip coordinates are assigned to built-in output variable `gl_Position`. Remember, every vertex shader must *minimally* generate a result in built-in `vec4` variable `gl_Position`. Otherwise, the results of primitive assembly and rasterization are undefined.

As with the previous version of the vertex shader, contents of `in` variable `aVertexColor` are *passed through* to `out` variable `vColor` without any transformation.

Note on naming shader variables

It is useful to adopt a convention for variable names that are passed between the OpenGL application and the vertex and fragment shaders. A convention is useful during writing and debugging shaders because it helps in keeping track of the source of the variables that come into and leave a shader. The convention is described in the following table:

Prefix	Stage that wrote it	Example
`a`	Attribute (from application)	`aVertexColor`
`u`	Uniform (from application)	`uModel_to_NDC`
`v`	Vertex shader	`vColor`
`f`	Fragment shader	`fFragColor`

Task 3b: Fragment shader

No changes are required to the fragment shader [in my-tutorial-3.frag] which is inherited from Tutorial 2:

```
R"( #version 450 core

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

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

Each color associated with a vertex is passed through by the vertex shader [in my-tutorial-3.vert] to the rasterizer. The rasterizer interpolates the colors at the vertices of the triangle primitive to associate a color with each fragment. This is the color provided to the fragment shader through input variable `vInterpColor`. In the above fragment shader, the color from input variable `vInterpColor` is passed through to the output variable `fFragColor`. Because the output variable has an associated layout qualifier `0`, the value in output variable `fFragColor` is written to the corresponding location of the back color buffer.

Task 3c: Application setup for `uniform` variables

The mechanics of copying each particles' `mdl_to_ndc_xform` data member to the `uniform` variable `uModel_to_NDC` in the shader program are discussed. For the sake of continuity, the incomplete `GLApp::GLObject::draw` function is reproduced below:

```
void GLApp::GLObject::draw() const {
  // Part 1: Use GLSLShader::Use() to install the shader program
  // used by this object to render its model

  // Part 2: Bind object's VAO handle using glBindVertexArray

  // Part 3: Copy object's 3x3 model-to-NDC matrix to vertex shader

  // Part 4: Render using glDrawElements or glDrawArrays

  // Part 5: Clean up
}
```

Before any operations dealing with `uniform` variables, the shader program must be enabled by installing it. Likewise, the appropriate VAO handle must be enabled. So parts 1 and 2 would look like this:

```
GLApp::shdrpgms[shd_ref].Use(); // Part 1
glBindVertexArray(GLApp::models[mdl_ref].vaoid); // Part 2
```

Now, each particles' `mdl_to_ndc_xform` data member contents must be copied to the `uniform` variable `uModel_to_NDC` in the shader program. The basic model for copying uniform variables is different from copying vertex data from buffers to vertex attribute variables. As discussed earlier, for attribute variables, the OpenGL application can specify the attribute location *before* linking occurs using the `layout` qualifier. In contrast, the locations or offsets of uniform variables cannot be specified by OpenGL 4.5 shaders using the `layout` qualifier. Instead, they've to be determined by OpenGL at link time. As a result, the following steps are required to copy data from an OpenGL application to a `uniform` variable:

- acquire a reference to the `uniform` variable defined in the shader, and
- associate a pointer to the desired values in the application with the acquired reference.

> Since OpenGL 4.6, it is possible to use `layout` qualifiers for `uniform` variables.

To acquire a reference to `uniform` variable `uModel_to_NDC`, call OpenGL command `glGetUniformLocation` with a handle to the shader program and a C-string specifying the uniform variable's name:

```
#include <glm/gtc/type_ptr.hpp> // for glm::value_ptr

glUniformMatrix3fv(shader.GetUniformLocation("uModel_to_NDC"), 1, GL_FALSE,
                   glm::value_ptr(mdl_to_ndc_xform));
```

The `GLSLShader::GetUniformLocation` is already implemented in file glslshader.cpp as part of the interface of class `GLSLShader`. This member function returns -1 if the C-string argument doesn't correspond to an active uniform variable in the shader program object. If successful, the call will return an integer representing the location of the queried uniform variable in the shader program object and the desired values in the application are copied to the uniform variable using a pointer to the values:

```
GLint uniform_var_loc1 = glGetUniformLocation(
             GLApp::shdrpgms[shd_ref].GetHandle(), "uModel_to_NDC");
if (uniform_var_loc1 < 0) {
  std::cout << "Uniform variable doesn't exist!!!\n";
  std::exit(EXIT_FAILURE);
}
```

This function returns -1 if the C-string argument doesn't correspond to an active uniform variable in the shader program object. If successful, the call will return an integer representing the location of the queried uniform variable in the shader program object and the desired values in the application are copied to the uniform variable using a pointer to the values:

```
#include <glm/gtc/type_ptr.hpp> // for glm::value_ptr

GLint uniform_var_loc1 =
  glGetUniformLocation(GLApp::shdrpgms[shd_ref].GetHandle(),
                       "uModel_to_NDC");
if (uniform_var_loc1 >= 0) {
  glUniformMatrix3fv(uniform_var_loc1, 1, GL_FALSE,
                     glm::value_ptr(GLApp::GLObject::mdl_to_ndc_xform));
} else {
  std::cout << "Uniform variable doesn't exist!!!\n";
  std::exit(EXIT_FAILURE);
}
```

The above code fragment assumes that GLM library was used to define the type of `GLApp::GLObject::mdl_to_ndc_xform` as `glm::mat3`. GLM function `glm::value_ptr` returns a pointer to the first element of the matrix data, which is then used by OpenGL command `glUniformMatrix3fv` to transfer matrix elements' values to uniform variable `uModel_to_NDC`.

> ***Class `GLSLShader` provides member functions that encapsulate the code fragment described above.***

🎯Task 4: Switching shader at run time

As indicated in Task 0, there should be loose coupling between models and shader programs so that shader programs can be easily switched at runtime to render instances of models with different visual effects. Consider a spaceship model in a game. Spaceship objects must be rendered without any special effects using a basic or default shader to provide a primary color, a color from real-time lighting computations, and a color from texture mapping. In certain scenarios, another shader program might be applied to the spaceship object to create a glowing effect. During the course of the game, the spaceship might be damaged and a third shader program is required to render the spaceship object with the visual effects indicating damage. In this task, we will learn how to render different instances of a model with different shaders.

Task 4a: Defining new shaders

To illustrate the mechanics of rendering different instances of a model with different shaders, you must author three new shaders: a Red fragment shader🔴 [in my-red.frag], a Green fragment shader🟢 [in my-green.frag], and a Blue fragment shader🔵 [in my-blue.frag]. All three files should be located in directory projects/tutorial-3/src. As their names imply, each fragment

shader must write the corresponding color as its output and their implementations is left as an exercise for the reader.

We have previously created a shader program by combining the vertex shader [authored in my-tutorial-3.vert] with the fragment shader [authored in my-tutorial-3.frag]. We need to create three additional shader programs by combining the vertex shader with each of the newly authored fragment shaders. This is done by adding to part 3 of function `GLApp::init` [in glapp.cpp] which currently should look like this:

```
// objects with file scope ...
std::string const my_tutorial_3_vs {
  #include "my-tutorial-3.vert"
};
std::string const my_tutorial_3_fs {
  #include "my-tutorial-3.frag"
};

// excerpt from GLApp::init() ...

// collection of pairs of vertex & fragment shader strings
GLApp::VPSS shdr_strs {
  std::make_pair(my_tutorial_3_vs, my_tutorial_3_fs)
};

// create shader program from each pair of vertex and fragment
// shader strings in container shdr_strs and then insert each
// shader program into container GLApp::shdrpgms:
GLApp::init_shdrpgms_cont(shdr_strs);
```

The updated code would look like this:

```
// objects with file scope ...
std::string const my_tutorial_3_vs {
  #include "my-tutorial-3.vert"
};
std::string const my_tutorial_3_fs {
  #include "my-tutorial-3.frag"
};
// add three new objects with file scope ...
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"
};

// excerpt from GLApp::init() ...

// collection of pairs of vertex & fragment shader strings
GLApp::VPSS shdr_strs {
  std::make_pair(my_tutorial_3_vs, my_tutorial_3_fs),
```

```
  std::make_pair(my_tutorial_3_vs, my_red_fs),
  std::make_pair(my_tutorial_3_vs, my_green_fs),
  std::make_pair(my_tutorial_3_vs, my_blue_fs)
};
GLApp::init_shdrpgms_cont(shdr_strs);
```

Task 4b: Applying shaders at runtime

Testing the sample with keyboard button S will provide the details necessary to implement this task.

Recall that type `GLApp::GLObject` uses data member `shd_ref` to represent an index into the static container of shader programs `GLApp::shdrpgms`. So far, every object's `shd_ref` has been initialized to 0 since `GLApp::shdrpgms` contained only a single shader program. Now that `GLApp::shdrpgms` has been augmented with three additional shader programs, we'd like to use key event S to initialize data member `shd_ref` of all spawning objects to reference a different shader program and thereby alter the way these spawning objects are rendered compared to previously spawned objects.

At startup, the shader program located at index 0 of container `GLApp::shdrpgms` [obtained from vertex shader my-tutorial-3.vert and fragment shader my-tutorial-3.frag] is used for rendering every spawning object. When keyboard button S is pressed, all spawning objects are rendered using the shader program located at index 1 of container `GLApp::shdrpgms` [obtained from vertex shader my-tutorial-3.vert and fragment shader my-red.frag]. A further action on button S will cause all subsequently spawning objects to be rendered with the shader program located at index 2 of container `GLApp::shdrpgms` [obtained from vertex shader my-tutorial-3.vert and fragment shader my-green.frag]. A further action on button S will cause all subsequently spawning objects to be rendered with the shader program located at index 3 of container `GLApp::shdrpgms` [obtained from vertex shader my-tutorial-3.vert and fragment shader my-blue.frag]. A further action on button S will cycle the selected shader program back to index 0.

Submission

1. Source and header files for Tutorial 3 must be placed in a directory labeled as: <login>-<tutorial-3>. If your Moodle student login is foo, then the directory would be labeled as foo-tutorial-3 and would have the following structure and layout:

```
foo-tutorial-3          # You're submitting Tutorial 3
├─ include              # Header files - *.hpp and *.h files
└─ src                  # Source files - *.cpp and .c files
   └─ my-tutorial-3.vert  # Default vertex shader
   └─ my-tutorial-3.frag  # Default fragment shader
   └─ my-red.frag         # Red fragment shader
   └─ my-green.frag       # Green fragment shader
   └─ my-blue.frag        # Blue fragment shader
```

2. *Zip* the directory and upload the resultant file foo-tutorial-3.zip to the assessment's submission page.

⚠ Before Submission: Verify and Test it ⚠

1. Unzip your archive file foo-tutorial-3.zip in directory test-submissions directory. Your directory and file layout should look like this:

```
csd2101-opengl-dev    # Sandbox directory for all assessments
├─ test-submissions   # ⚠ Test submissions here before uploading
|  └─ foo-tutorial-3 # Tutorial 3 is submitted by foo
|     └─ include     # Header files - *.hpp and *.h files
|     └─ src         # Source files - *.cpp and .c files
|        └─ my-tutorial-1.vert  # Default vertex shader
|        └─ my-tutorial-1.frag  # Default fragment shader
|        └─ my-red.frag         # Red fragment shader
|        └─ my-green.frag       # Green fragment shader
|        └─ my-blue.frag        # Blue fragment shader
└─ csd2101.bat                # Automation Script
```

2. Run batch file csd2101.bat and select option R to reconfigure the solution with new project foo-tutorial-3.
3. Select option B to build the project. If there are no errors, an executable file foo-tutorial-3.exe will be created in directory build/Release. Alternatively, you can verify the project through the Visual Studio 2022 IDE.
4. Use the following checklist before uploading to the assessment's submission page:

Things to test before submission	Status
Assessment compiles without any errors	☐
All warnings resolved, zero warnings during compilation	☐
Executable generated and successfully launched in \text{Debug} and \text{Release} mode	☐
Directory is zipped, ensuring adherence to naming conventions as outlined in submission guidelines	☐
Upload zipped file to appropriate submission page	☐

ℹ 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. More and cooler stuff is ok. Less is not ok. What is minimally required?
    - Application must define two models: a box model and a mystery model. Note that the mystery model cannot be a box nor a circle [from $\text{Tutorial } 2$] nor a star [it is not a mystery model since it is already present in the sample executable].
    - Implementation of geometric instancing is required.
    - Implementation of shader instancing is required.
    - Using ***left mouse button***, users must be able to spawn and remove objects.
    - Your application must display at least $32{,}768$ spinning objects with each object's model being determined randomly at runtime.
    - At birth, each geometry instance must be initialized with randomly generated values for the following attributes: model reference, world position, scaling parameters, initial angular displacement, and angular speed [per frame].
    - In each iteration, the model-to-NDC matrix must be computed.
    - Contents of the matrix must be copied to a uniform variable in vertex shader $\text{my-tutorial-3.vert}$ that you've authored. In addition, this vertex shader must transform vertex position coordinates from model to NDC coordinates.
- [***core4***] Keyboard button P allows users to iterate the polygon rasterization mode between filled triangles, lines, and points.
- [***core5***] Keyboard button S allows users to switch different shader on existing models.
- [***core6***] Application prints useful information similar to the sample on the window toolbar - see sample executable for expected text.

Mapping of Grading Rubrics to Letter Grades

The core competencies listed in the grading rubrics will be mapped to letter grades using the following table:

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 rubrics are satisfied [see here for complete list]. If one or more items are missing, the maximum possible grade for this assignment will be C.	B
core4 rubric is satisfied.	$B+$
core5 rubric is satisfied.	A
core6 rubric is satisfied.	$A+$