Tutorial 2: Viewports and OpenGL Primitives

> Your objective is to write code to render an image *similar as the sample or better*. The tutorial is verbose only because it assumes little or no previous experience in computer graphics and the OpenGL toolbox. If you're so inclined, you can skip this tutorial and begin implementing code that satisfies the rubrics. Make sure to carefully read the submission guidelines and rubrics. As long as your submission's output matches the sample. your submission will be graded fairly and objectively.

Things to be covered in this tutorial:

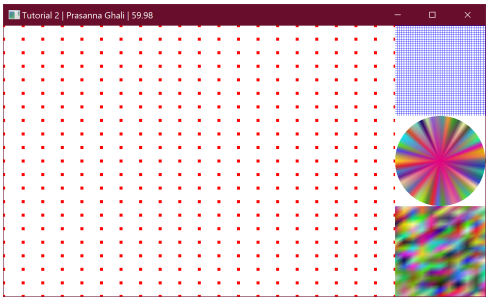
- Broaden understanding of 2D graphics pipe and OpenGL toolbox. Specific concepts include viewports, windows, Normalized Device Context (NDC) coordinate system, viewport transforms, and GLSL `uniform` variables.
- Render geometrical objects comprised of points, lines, and triangles using OpenGL primitives `GL_POINTS`, `GL_LINES`, `GL_TRIANGLE_FAN`, and `GL_TRIANGLE_STRIP`.
- Procedurally model basic geometrical objects such as circular disks and rectangular meshes.

Prerequisites

- Introductory understanding of the various coordinate systems in the OpenGL graphics pipe introduced in weeks 1 and 2.
- Handout viewport+xform+derivation.pdf must be read to get an understanding of viewports, windows, NDC, and the mathematics of viewport transform.
- Do not begin this assessment without completing $\text{Tutorial } 1$!!!

Application behavior

Begin by running the sample to understand the requirements and scope of this tutorial. The application creates four viewports with each viewport displaying distinct geometry.



In OpenGL, a geometrical object is made up of primitives such as point, line segment, line strip, line loop, triangle, triangle fan, triangle strip. Each of these primitives is made up of one or more vertices.

The left viewport displays a geometrical model containing 441 points that are rendered as `GL_POINTS` primitives. ***The point positions are procedurally computed by subdividing the*** 2×2 ***NDC box into a*** 21×21 ***rectangular pattern.***

The top-right viewport displays a geometrical model containing 41 horizontal and 41 vertical line segments rendered as `GL_LINES` primitives. ***The start and end points of these 82 line segments are procedurally computed by subdividing the 2×2 NDC box into a 40×40 grid pattern.***

The center-right viewport displays a circle subdivided into 30 triangle slices that are rendered as `GL_TRIANGLE_FAN` primitives. The triangles' vertex position coordinates are procedurally computed by slicing the circle's circumference using the circle parametric equation while vertex color attributes are randomly generated. The circle has unit radius and is centered at $(0, 0)$ in the 2×2 NDC box.

The bottom-right viewport subdivides the 2×2 NDC box into a 5×10 grid of rectangles comprising of 100 triangles that are rendered using `GL_TRIANGLE_STRIP` primitive. The triangles' vertex position coordinates and index buffer are procedurally computed while vertex color attributes are randomly generated.

Finally, the statistics used to describe the contents of the four viewports are written to the display window's title bar.

🎯Task 0: Render to multiple viewports

Overwrite the existing batch file csd2101.bat in csd2101-opengl-dev [where you completed Tutorial 1 with a new version available on the assessment's web page. Execute the batch file.

1. Choose option **2 - Create Tutorial 2** to create a Visual Studio 2022 project tutorial-2.vcxproj in directory build with source in directory projects/tutorial-2 whose layout is shown below:

```
csd2101-opengl-dev/      # Sandbox directory for all assessments
├─ build/                # Build files will exist here
├─ projects/             # Tutorials and assignments must exist here
|  └─ tutorial-2         # Tutorial 2 code exists here
|     └─ include         # Header files - *.hpp and *.h files
|     └─ src             # Source files - *.cpp and .c files
|        └─ my-tutorial-2.vert   # Vertex shader file
|        └─ my-tutorial-2.frag   # Fragment shader file
└─ csd2101.bat           # Automation Script
```

   Source and shader files are pulled into nested directory /projects/tutorial-2/src while header files are pulled into nested directory /projects/tutorial-2/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 1. Therefore, ***copy all source, shader, and header files*** from /projects/tutorial-1/src and /projects/tutorial-1/include to /projects/tutorial-2/src and /projects/tutorial-2/include, respectively.

3. In /projects/tutorial-2/src, delete shader files my-tutorial-2.vert and my-tutorial-2.frag and then rename vertex shader my-tutorial-1.vert to my-tutorial-2.vert and fragment shader my-tutorial-1.frag to my-tutorial-2.frag.

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

```
std::string const my_tutorial_1_vs = {
  #include "my-tutorial-1.vert"
};

std::string const my_tutorial_1_fs = {
  #include "my-tutorial-1.frag"
};
```

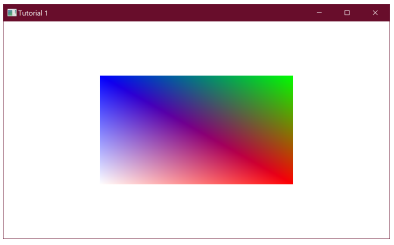
with this code:

```
std::string const my_tutorial_1_vs = {
  #include "my-tutorial-2.vert"
};

std::string const my_tutorial_1_fs = {
  #include "my-tutorial-2.frag"
};
```

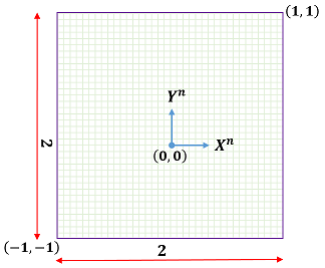
Further, in function `GLApp::GLModel::setup_shdrpgm` [also in file glapp.cpp], we replace references to objects `my_tutorial_1_vs` and `my_tutorial_1_fs` with `my_tutorial_2_vs` and `my_tutorial_2_fs`, respectively:

```
if (!shdr_pgm.CompileShaderFromString(GL_VERTEX_SHADER,
                                      my_tutorial_2_vs)) {
  // as before
}
if (!shdr_pgm.CompileShaderFromString(GL_FRAGMENT_SHADER,
                                      my_tutorial_2_fs)) {
  // as before
}
```

5. Edit my-tutorial-2.vert so that input vertex attribute variables `aVertexPosition` and `aVertexColor` are assigned generic vertex attribute indices `0` and `1`, respectively. Other than these minor edits, you don't have to amend the vertex and fragment shader code any further.
6. Changing the generic vertex attribute indices in the vertex shader will necessitate appropriate changes to the generic vertex attribute indices referenced by VAO handle `vaoid`. This is done by amending function `GLApp::GLModel::setup_vao` [in file glapp.cpp].
7. Make `GLApp::update` [in glapp.cpp] an empty function - we'll add new update logic as we progress through this assessment.
8. Ensure that you're able to compile and link your code and execute the application.
9. This tutorial requires the color buffer to have a background of white color. Which function was used in Tutorial 1 to set the RGBA values used to clear the color buffer at the top of each frame? Hint: Read this reference page. Your display should look like this:



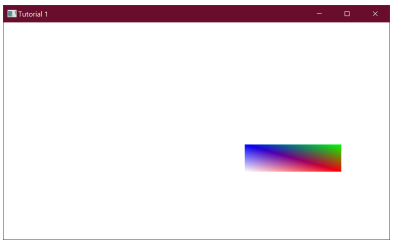
10. Recall that NDC coordinate system, shown in the figure below, is a square box of size 2×2 units with $x^n \in [-1, 1]$ and $y^n \in [-1, 1]$.



A viewport transform maps points from NDC to the window coordinate system. Applications call OpenGL command `glViewport` to apply a viewport transform to map NDC points to specific locations in the display window. In $\text{Tutorial } 1$, you rendered the rectangle into a viewport that had the same dimensions as the display window [recall that the display window has the exact shape and dimensions as the color buffer]. In your source code, locate the call to command `glViewport`. Establish your understanding of viewport transforms by changing the four arguments supplied to `glViewport`. For example, what happens if you replace the call to `glViewport` with the following:

```
GLint w {GLHelper::width}, h {GLHelper::height};
glViewport(w/2, h/4, w/2, h/4);
```

Does your display look like this?



11. This assessment requires the color buffer to be subdivided into four viewports with each viewport displaying a distinct rendered image. This is possible by amending function `GLApp::draw` to enable the appropriate viewport transform before rendering the geometry. The outline for function `GLApp::draw` would look like this:

```
void GLApp::draw() {
  // Part 1: clear color buffer using OpenGL command glClear

  // Part 2: Render distinct geometry to four viewports:
     // First, use OpenGL command glViewport to specify appropriate
     // viewport transform that will render to a specific portion of
     // display screen
     // Next, use member function GLModel::draw to render geometry
     // Repeat previous two steps for remaining three viewports
}
```

12. Specifying each viewport transform using `glViewport` requires four arguments. It would be convenient to encapsulate these values in a structure `GLApp::GLViewport`:

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

  struct GLViewport {
    GLint x, y;
    GLsizei width, height;
  };
  static std::vector<GLViewport> vps; // container for viewports
};
```

C++ requires static data members to be defined in a source file and static data member `GLApp::vps` is defined in file glapp.cpp:

```
std::vector<GLApp::GLViewport> GLApp::vps;
```

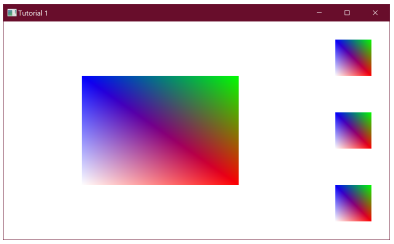
13. This assessment requires the four viewports to have the specific configuration illustrated below and must be specified in function `GLApp::init` [in glapp.cpp]:

```
void GLApp::init() {
  GLint w {GLHelper::width}, h {GLHelper::height};
  GLint hs {h/3}, ws {hs};

  /*
      +-------------------------+--------+
      |                         |        |
      |                         |  VP1   |
      |                         |        |
      |                         +--------+
      |                         |        |
      |           VP0           |  VP2   |
      |                         |        |
      |                         +--------+
      |                         |        |
      |                         |  VP3   |
      |                         |        |
      +-------------------------+--------+
  */

  // store the viewports VP0 thro' VP3 in container GLApp::vps
  // VP0 = { 0, 0, w - ws, h }
  // VP1 = { w - ws, 2 * hs, ws, hs }
  // VP2 = { w - ws, hs, ws, hs }
  // VP3 = { w - ws, 0, ws, hs }
}
```

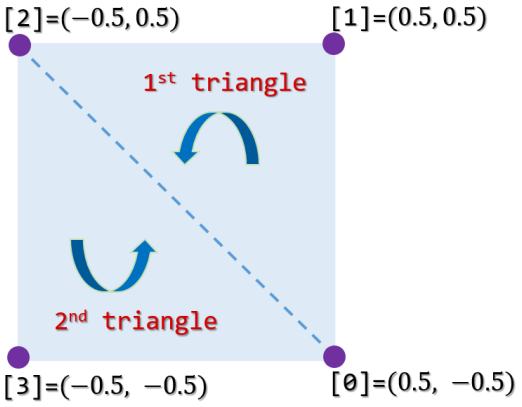
14. Finally, you'll have to amend function `GLApp::draw` to call command `glViewport` with arguments specified by each element of static container `GLApp::vps` so that the rectangle is rendered into each viewport:

```
void GLApp::draw() {
  // as before ...

  // for each element in vps ...
    // set viewport
    // render rectangle model in this viewport
}
```

You should now be able to obtain the output described below:



15. OpenGL drawing commands are roughly broken into two subsets: indexed and nonindexed draws. Tutorial 1 rendered a rectangle model consisting of two `GL_TRIANGLES` primitives with the following geometry:



using an *indexed draw command* `glDrawElements`. This draw command used an array of indices stored in a buffer object that is used to indirectly index into the enabled vertex attribute arrays. Let's review our understanding of this indexed draw command.

The model's four vertices were given storage in a per-vertex position and color coordinate arrays in client-side [that is, CPU] memory:

```
std::array<glm::vec2, 4> pos_vtx { // vertex position attributes
  glm::vec2(0.5f, -0.5f), glm::vec2(0.5f, 0.5f),
  glm::vec2(-0.5f, 0.5f), glm::vec2(-0.5f, -0.5f)
};

std::array<glm::vec3, 4> clr_vtx { // vertex color attributes
  glm::vec3(1.f, 0.f, 0.f), glm::vec3(0.f, 1.f, 0.f),
  glm::vec3(0.f, 0.f, 1.f), glm::vec3(1.f, 1.f, 1.f)
};
```

Further, the contents of these two arrays were transferred from client-side to a server-side buffer object:

```
glCreateBuffers(1, &vbo_hdl);
glNamedBufferStorage(vbo_hdl,
              4*static_cast<GLsizei>(sizeof(pos_vtx)+sizeof(clr_vtx)),
              nullptr, // nullptr means no data is transferred
              GL_DYNAMIC_STORAGE_BIT);
glNamedBufferSubData(vbo_hdl, 0, static_cast<GLsizei>(sizeof(pos_vtx)),
                     reinterpret_cast<GLvoid*>(pos_vtx.data()));
glNamedBufferSubData(vbo_hdl, sizeof(pos_vtx), sizeof(clr_vtx),
                     reinterpret_cast<GLvoid*>(clr_vtx.data()));
```

The topology of the rectangle model [consisting of two `GL_TRIANGLES` primitives] was described by a client-side index buffer consisting of indices [into the per-vertex position and color coordinate arrays]:

```
std::array<GLushort, 6> idx_vtx{ 0, 1, 2, 2, 3, 0 };
idx_elem_cnt = static_cast<GLuint>(idx_vtx.size());
```

The contents of this array was transferred from client-side to a server-side buffer object:

```
GLuint ebo_hdl;
glCreateBuffers(1, &ebo_hdl);
glNamedBufferStorage(ebo_hdl, sizeof(GLushort) * idx_elem_cnt,
                     reinterpret_cast<GLvoid*>(idx_vtx.data()),
                     GL_DYNAMIC_STORAGE_BIT);
glVertexArrayElementBuffer(vaoid, ebo_hdl);
```

Finally, the rectangle model was rendered by calling OpenGL indexed draw command `glDrawElements` in function `GLApp::GLModel::draw`:

```
glDrawElements(GL_TRIANGLES, idx_elem_cnt, GL_UNSIGNED_SHORT, NULL);
```

When `glDrawElements` is called, it uses the first three sequential elements 0, 1, and 2 from the buffer referenced by `ebo_hdl` to extract the position and color coordinates located at these indices from the buffer referenced by `vbo_hdl`. These coordinates are funneled to the vertex shader. Subsequently, the fragments within the interior of this triangle are identified by the fixed-stage rasterizer. Each such fragment is funneled to the fragment shader which will then compute the fragment's color.

16. In contrast to indexed draw commands, *nonindexed draw commands* do not use the array of indices at all, and simply read the vertex data sequentially. The most basic, nonindexed drawing command in OpenGL is `glDrawArrays`.

    To render the rectangle model with command `glDrawArrays`, the position and color arrays must now be defined like this:

```
std::array<glm::vec2, 6> pos_vtx{
  glm::vec2(0.5f, -0.5f),  glm::vec2(0.5f, 0.5f),
  glm::vec2(-0.5f, 0.5f),  glm::vec2(-0.5f, 0.5f),
  glm::vec2(-0.5f, -0.5f), glm::vec2(0.5f, -0.5f)
};

std::array<glm::vec3, 6> clr_vtx{
  glm::vec3(1.f, 0.f, 0.f), glm::vec3(0.f, 1.f, 0.f),
  glm::vec3(0.f, 0.f, 1.f), glm::vec3(0.f, 0.f, 1.f),
  glm::vec3(1.f, 1.f, 1.f), glm::vec3(1.f, 0.f, 0.f)
};
```

As usual, the contents of these two arrays must be transferred from client-side to a server-side buffer object:

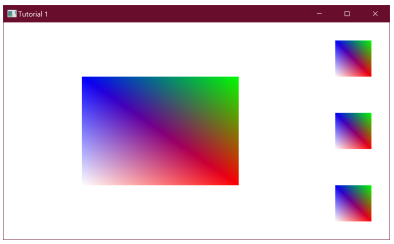
```
glCreateBuffers(1, &vbo_hdl);
glNamedBufferStorage(vbo_hdl,
              6*static_cast<GLsizei>(sizeof(pos_vtx)+sizeof(clr_vtx)),
              nullptr, // nullptr means no data is transferred
              GL_DYNAMIC_STORAGE_BIT);
glNamedBufferSubData(vbo_hdl, 0, static_cast<GLsizei>(sizeof(pos_vtx)),
                     reinterpret_cast<GLvoid*>(pos_vtx.data()));
glNamedBufferSubData(vbo_hdl, sizeof(pos_vtx),
                     static_cast<GLsizei>(sizeof(clr_vtx)),
                     reinterpret_cast<GLvoid*>(clr_vtx.data()));
```

Since command `glDrawArrays` doesn't use an array of indices, there's no need to define an index buffer. The rectangle model is rendered by replacing indexed draw command `glDrawElements` with nonindexed draw command `glDrawArrays`:

```
glDrawArrays(GL_TRIANGLES, 0, 6);
```

When `glDrawArrays` is called, it extracts the first three sequential values [located at indices 0, 1, and 2] from the position and color coordinate arrays and funnels these values to the vertex shader. These vertices define a triangle primitive. Subsequently, the fixed-stage rasterizer identifies the fragments within the interior of this triangle. Each such fragment is funneled to the fragment shader which will then compute the fragment's color.

Update your code to use the nonindexed draw command `glDrawArrays` and ensure your output is similar to the following picture:



17. The pros and cons of indexed and nonindexed draw commands will be identified in the remaining tasks of this assessment.

🎯Task 1: Render points

This outcome of this task is to render geometric shapes consisting of points using the OpenGL primitive type `GL_POINTS`. To reach this outcome, you'll have to make further changes to glapp.h and glapp.cpp.

Task 1a: Amend structure `GLApp::GLModel`

Consider the definition of structure `GLApp::GLModel` from Tutorial 1 that described a geometric model that requires an indexed draw call `gDrawElements` to be rendered:

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

  struct GLModel {
    GLenum     primitive_type; // which OpenGL primitive to be rendered?
    GLSLShader shdr_pgm;       // which shader program?
    GLuint     vaoid;          // handle to VAO
    GLuint     vbo_hdl;        // handle to VBO
    GLuint     idx_elem_cnt;   // how many elements of primitive_type
                               // are to be rendered?
    void setup_vao();
    void setup_shdrpgm();
    void draw();
  };
  static GLModel mdl;
};
```

This application must render models with point, line, and triangle primitives. These varied primitives have different needs. For example, models comprising points or unconnected lines don't require indices specifying topology of the underlying primitives. On the other hand, models comprising triangles will require indices to provide the topology of triangles from the arrays of per-vertex attributes. The single function `GLApp::GLModel::setup_vao` from Tutorial 1 cannot cater to the different needs of these varied primitives and therefore function `setup_vao` is deprecated. Instead, the responsibility for specifying a model's attributes is now dispersed among different functions. For example, static function `GLApp::points_model` will now return an instance of type `GLApp::GLModel` specifying a model comprising point primitives. Additional functions will be required to specify models consisting of lines, triangle fans, and triangle strips; these will be described in due course.

Because models with points or line primitives don't need to specify array of indices, they can be rendered with lower memory requirements using nonindexed draw command `glDrawArrays`. On the other hand, it is more optimal if models with triangle primitives are rendered with indexed draw command `glDrawElements`. While command `glDrawElements` requires the number of indices in the index buffer to render the model's primitives, `glDrawArrays` requires the number of vertex attributes in the per-vertex data buffers. Therefore, data member `idx_elem_cnt` is deprecated and is replaced with data member `draw_cnt` that will refer to a count of vertices for nonindexed draw calls and a count of indices in an index buffer for indexed draw calls.

Data member `primitive_cnt` is added to conveniently access the number of OpenGL primitives contained in an instance of `GLApp::GLModel`.

Since this tutorial's application must manage multiple geometric shapes, static data member `GLApp::mdl` from Tutorial 1 is also deprecated. Instead, static data member `GLApp::models` of type `std::vector<GLApp::GLModel>` will be the repository of these multiple shapes and is declared in structure `GLApp`. C++ requires static data members to be defined in a source file and thus the definition of `GLApp::models` [in file glapp.cpp] would look like this:

```
std::vector<GLApp::GLModel> GLApp::models;
```

The amended definition of structure `GLApp::GLModel` would now look like this:

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

  struct GLModel {
    GLenum     primitive_type;  // same as Tutorial 1
    GLuint     primitive_cnt;   // added for Tutorial 2
    //GLuint     vbo_hdl;       // deprecated in Tutorial 2
    GLuint     vaoid;           // same as Tutorial 1

    //GLuint     idx_elem_cnt;  // deprecated in Tutorial 2
    GLuint     draw_cnt;        // added for Tutorial 2

    GLSLShader shdr_pgm;        // same as Tutorial 1

    //void setup_vao();          // deprecated in Tutorial 2
    //void setup_shdrpgm();      // deprecated in Tutorial 2

    // Tutorial 2's replacement for setup_shdrpgm()
    void setup_shdrpgm(std::string const& vtx_shdr,
                       std::string const& frag_shdr);
    void draw(); // same as Tutorial 1
  };

  //static GLModel mdl; // deprecated in Tutorial 2
  static std::vector<GLModel> models; // added for Tutorial 2

  // Tutorial 2's replacement for setup_vao for GL_POINT primitives
  static GLApp::GLModel points_model(std::string const& vtx_shdr,
                                     std::string const& frag_shdr);
};
```

Task 1b: Amend member function `GLModel::setup_shdrpgm`

In Tutorial 1, a single vertex shader and a single fragment shader were used to render a rectangle model. It is more likely that you'd be interested in rendering different models with different rendering effects. To satisfy this need, member function `GLModel::setup_shdrpgm` is augmented to take the names of strings containing vertex and fragment shader programs:

```
void GLApp::GLModel::setup_shdrpgm(std::string const& vtx_shdr,
                                   std::string const& frag_shdr) {
  if (!shdr_pgm.CompileShaderFromString(GL_VERTEX_SHADER, vtx_shdr)) {
```

```
    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, frag_shdr)) {
    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);
  }

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

Task 1c: Define static member function `GLApp::points_model`

Suppose you want to render the corners of a $\frac{1}{2} \times \frac{1}{2}$ square centered at $(0, 0)$ in NDC using `GL_POINTS` primitives. Begin the definition of static member function `GLApp::points_model` by specifying the squares' corners:

```
GLApp::GLModel GLApp::points_model(std::string const& vtx_shdr,
                                   std::string const& frag_shdr) {
  std::vector<glm::vec2> pos_vtx {
    glm::vec2(0.25f, 0.25f),   glm::vec2(-0.25f, 0.25f),
    glm::vec2(-0.25f, -0.25f), glm::vec2(0.25f, -0.25f)
  };
  // more to follow ...
```

Continue the definition of function `points_model` by adding the following code to define a VAO handle:

```
  GLuint vbo_hdl;
  glCreateBuffers(1, &vbo_hdl);
  glNamedBufferStorage(vbo_hdl, sizeof(glm::vec2) * pos_vtx.size(),
                       pos_vtx.data(), GL_DYNAMIC_STORAGE_BIT);
  GLApp::GLModel mdl;
  glCreateVertexArrays(1, &mdl.vaoid);
  glEnableVertexArrayAttrib(mdl.vaoid, 0);
  glVertexArrayVertexBuffer(mdl.vaoid, 0, vbo_hdl, 0, sizeof(glm::vec2));
  glVertexArrayAttribFormat(mdl.vaoid, 0, 2, GL_FLOAT, GL_FALSE, 0);
  glVertexArrayAttribBinding(mdl.vaoid, 0, 0);
  glBindVertexArray(0);
```

Since every point in this geometric shape is rendered using an uniform color, vertex color attributes are not required. Further, since a point is a discrete primitive unconnected to other points, indices are not required to define the shape's topology.

Next, data members of an instance of `GLApp::GLModel` are assigned appropriate attributes:

```
  mdl.primitive_type = GL_POINTS;
  mdl.setup_shdrpgm(vtx_shdr, frag_shdr);
  mdl.draw_cnt = static_cast<GLuint>(pos_vtx.size()); // number of vertices
  mdl.primitive_cnt = mdl.draw_cnt; // number of primitives
  return mdl;
} // end of function GLApp::points_model
```

Task 1d: Amend static member function `GLApp::init`

Function `GLApp::init` will invoke static function `GLApp::points_model` to return an instance of `GLApp::GLModel` that encapsulates a model consisting of 4 points representing the corners of a $\frac{1}{2} \times \frac{1}{2}$ square centered at $(0, 0)$ in NDC. Further, the returned instance of `GLApp::GLModel` is inserted into container `GLApp::models`:

```
void GLApp::init() {
  // Part 1: clear color buffer with white color ...

  // Part 2: insert 4 viewport specifications in GLApp::Vps ...

  // Part 3: create different geometries and insert them into
  // repository container GLApp::models ...
  GLApp::models.emplace_back(GLApp::points_model(my_tutorial_2_vs,
                                                 my_tutorial_2_fs));
}
```

Task 1e: Amend static member function `GLApp::draw`

The draw function `GLApp::draw` looks like this:

```
void GLApp::draw() {
  // clear back color buffer as before ...
  glClear(GL_COLOR_BUFFER_BIT);

  // render model with point primitives from NDC coordinates to viewport
  glViewport(vps[0].x, vps[0].y, vps[0].width, vps[0].height);
  models[0].draw();
}
```

Task 1f: `GLApp::update`

The update function `GLApp::update` updates the window title:

```
void GLApp::update() {
  // write window title with stuff similar to sample ... How?
  std::stringstream sstr;
  // collect everything you want written to title bar in sstr ...
  glfwSetWindowTitle(GLHelper::ptr_window, sstr.str().c_str());
}
```

Task 1g: Amend member function `GLModel::draw`

To render the geometrical shape as four isolated points, member function `GLModel::draw` is defined as:

```
void GLApp::GLModel::draw() {
  shdr_pgm.Use();
  glBindVertexArray(vaoid);
  if (primitive_type == GL_POINTS) {
    glVertexAttrib3f(1, 1.f, 0.0f, 0.f); // red color for points
    glDrawArrays(primitive_type, 0, draw_cnt);
  }
  glBindVertexArray(0);
  shdr_pgm.UnUse();
}
```

As seen in Tutorial 1, the call to `glBindVertexArray` uses attributes referenced by VAO handle `vaoid` to specify the OpenGL context on which subsequent OpenGL commands will operate. Before looking at the call to command `glVertexAttrib3f`, recall that function `GLApp::points_model` generates points having only vertex position attributes and no vertex color attributes. However, as seen in line 4, vertex shader my-tutorial-2.vert is programmed to receive a per-vertex color value in variable `aVertexColor`:

```
R"( #version 450 core

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

  layout (location=0) out vec3 vColor;

  void main() {
    gl_Position = vec4(aVertexPosition, 0.0, 1.0);
    vColor      = aVertexColor;
  }
)"
```

The situation where the geometry doesn't have vertex color attributes but the vertex shader expects these values is reconciled by command `glVertexAttrib3f`. The call to command `glVertexAttrib3f` sets a color value using three floating-point arguments for the color's red, green, and blue components. This color is a *generic vertex attribute* meaning that vertex shader variable `aVertexColor` is provided this value by the application once to render all four points instead of being stored in a VBO on a per-vertex basis. Since there are four vertices in this shape, the vertex shader will be invoked four times - once per vertex. The vertex shader will be given the color once by the application while it sequentially receives each individual vertex position attribute

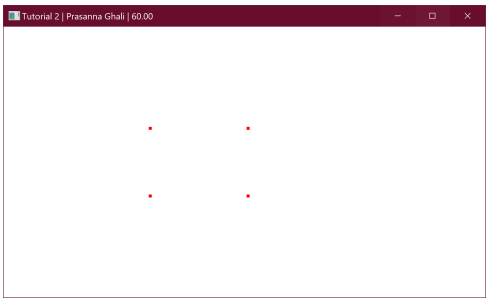
from the buffer containing these attributes. The first argument to `glVertexAttrib3f` is `1` since the vertex shader uses the `layout` qualifier on line 4 to specify slot `1` to access the color for rendering primitives.

The call to command `glDrawArrays` is the command that actually causes the vertices to be rendered. The first argument `GL_POINTS` tells OpenGL to draw the vertices as isolated points. The second argument `0` says to start with vertex number 0, that is, the first vertex. The third argument `draw_cnt` evaluates to 4 - the count of `GL_POINTS` primitives to be rendered.

If you run the application with default OpenGL settings, you will discover that `GL_POINTS` primitives are drawn as very small, single pixel points — perhaps so small as to be almost invisible. To display points as large, square dots, `GLModel::draw` changes the default size from 1 to 10 using command `glPointSize`. After the draw call, it is good practice to reset the size back to 1:

```
1  if (primitive_type == GL_POINTS) {
2    glPointSize(10.f);
3    glVertexAttrib3f(1, 1.f, 0.0f, 0.f); // red color for points
4    glDrawArrays(primitive_type, 0, draw_cnt);
5    glPointSize(1.f);
6  }
```

Executing these lines will cause the program to draw nice round dots for points. However, the effect of these commands varies with different implementations of OpenGL, so you may see square dots instead of round dots, or even no change at all. How well, and whether, the point sizing works will depend on the implementation of your OpenGL driver. At this point, the image displayed by your application should look like this [look closely - there are red dots in the picture]:



Task 1h: Render grid of points

Suppose you'd like to determine the x values that would slice range $[x_{left}, x_{right}]$ into 5 equivalent subintervals. If $x_{left} = -1$ and $x_{right} = 1$, the length l of each subinterval is:

$$l = \frac{(x_{right} - x_{left})}{5} = \frac{2}{5} = 0.4$$

Using $l = 0.4$, the first subinterval is $[-1, -1 + 0.4 = -0.6]$ and the second subinterval is $[-0.6, -0.6 + 0.4 = -0.2]$. Continuing this subdivision, the fifth and final subinterval is $[0.6, 1]$. The enumerated sequence of 6 values $\{-1, -0.6, -0.2, 0.2, 0.6, 1.0\}$ defines these 5 subintervals. These values can be computed as:

$$\begin{array}{l} \textbf{Algorithm create-xvals} \\ \quad \text{vector } x(slices + 1) \\ \quad \text{for } (i \ : \ slices + 1) \\ \quad\quad x[\text{i}] = (l \times i) - 1.0 \end{array}$$

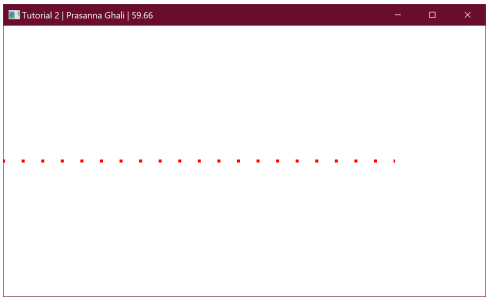
Using the above algorithm and given parameter `slices`, refactor function `GLApp::points_model` to procedurally generate `slices+1` count of points that slice the horizontal span between points $(-1, 0)$ and $(1, 0)$ into `slices` count of equivalent spans. Store the computed points in a `vector` container. The declaration and definition of static function `GLApp::points_model` must be augmented with an additional parameter `slices`. And, using the logic from algorithm **create-xvals**, the definition of function `GLApp::points_model` looks like this:

```
// This function generates (slices+1) number of points for the
// horizontal span between points (-1, 0) and (1, 0)
GLApp::GLModel GLApp::points_model(int slices, std::string const& vtx_shdr,
                                   std::string const& frag_shdr) {
  // store (slices+1) count of points in a vector container
  // rest remains same as before ...
}
```

The call to `GLApp::points_model` in `GLApp::init` must be updated with additional argument specifying `slices`:

```
void GLApp::init() {
  // same as before ...

  // Part 3: initialize VAO for different geometries ...
  GLApp::models.emplace_back(GLApp::points_model(20, my_tutorial_2_vs,
                                                 my_tutorial_2_fs));
  // rest as before ...
}
```

The image displayed by your application should now look like this:



To generate a grid of points that uniformly fill the viewport, the sliced set of points simply need to be stacked one above the other starting from $y = -1$ and ending at $y = 1$. The refactored function `GLApp::points_model` that takes an additional parameter `stacks` will look like this:

```
GLApp::GLModel GLApp::points_model(int slices, int stacks,
                                   std::string const& vtx_shdr,
                                   std::string const& frag_shdr) {
  // store the (slices+1)*(stacks+1) number of points in a vector
  // rest remains same as before ...
}
```

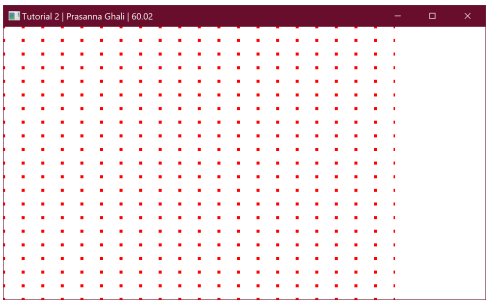
Provide a declaration for this final updated version of `GLApp::points_model` in structure `GLApp`:

```
struct GLApp {
  // same as before ...
  static GLApp::GLModel points_model(int slices, int stacks,
                                     std::string const& vtx_shdr,
                                     std::string const& frag_shdr);
};
```

The new call to function `GLApp::points_model` in `GLApp::init` would look like this:

```
void GLApp::init() {
  // same as before ...

  // Part 3: initialize VAO for different geometries ...
  GLApp::models.emplace_back(GLApp::points_model(20, 20,
                             my_tutorial_2_vs, my_tutorial_2_fs)
                            );
  // rest as before ...
}
```

No other changes need to be made and the image displayed by your application should now look like this:



🎯Task 2: Render lines

This task introduces OpenGL primitive `GL_LINES` that will be used by the application to render geometric shapes consisting of line segments in the top-right viewport. Line segments are useful in certain scenarios including particle systems.

Task 2a: Declare static member function `GLApp::lines_model`

The idea is to borrow the idea of *slices* and *stacks* from the previous task to render a geometric shape consisting of a grid of line segments. Each line segment is rendered using OpenGL primitive `GL_LINES`. Declare static function `GLApp::lines_model` to create such a geometrical shape:

```
struct GLApp {
  // same as before ...
  static GLApp::GLModel lines_model(int slices, int stacks,
                                    std::string const& vtx_shdr,
                                    std::string const& frag_shdr);
};
```

Task 2b: Define static member function `GLApp::lines_model`

We've previously seen that $(slices + 1)$ points are required to subdivide $x \in [-1, 1]$ into $slices$ count of subsegments. Suppose we wish to render $(slices + 1)$ number of vertical line segments from start (bottom) point $(x, -1)$ to end (top) point $(x, 1)$ where $x \in [-1, 1]$. Since two end points are required to define a line segment, $2 \times (slices + 1)$ points are required to define $(slices + 1)$ vertical lines. Provide a definition for static function `GLApp::lines_model` [in file glapp.cpp] that will use parameter `slices` to generate the appropriate set of vertical lines:

```
// ignore parameter stacks for now ...
GLApp::GLModel GLApp::lines_model(GLint slices, GLint stacks,
                                  std::string const& vtx_shdr,
                                  std::string const& frag_shdr) {
  // compute and store endpoints for (slices+1) set of vertical lines
  // for each x from -1 to 1
  //   start endpoint is (x, -1) and end endpoint is (x, 1)
  std::vector<glm::vec2> pos_vtx;
  pos_vtx.reserve((slices + 1) * 2);
  GLfloat const u{ 2.f / static_cast<GLfloat>(slices) };
  for (GLint col{ 0 }; col <= slices; ++col) {
    GLfloat x{ u * static_cast<GLfloat>(col) - 1.f };
    pos_vtx.emplace_back(glm::vec2(x, -1.f)); // bottom end point of line
    pos_vtx.emplace_back(glm::vec2(x, 1.f));  // top end point of line
  }

  GLApp::GLModel mdl;
  // set up VAO as in GLApp::points_model
  // and then set up the rest of object mdl ...
  mdl.primitive_type = GL_LINES;
  mdl.setup_shdrpgm(vtx_shdr, frag_shdr);
  mdl.draw_cnt = 2*(slices+1);        // number of vertices
  mdl.primitive_cnt = mdl.draw_cnt/2; // number of primitives
  return mdl;
}
```

Task 2c: Amend static member function `GLApp::init`

In function `GLApp::init`, static member function `GLApp::lines_model` is invoked from function `GLApp::init` with first argument of 40 to instantiate a geometric shape consisting of 41 vertical line segments spanning the 2×2 NDC square centered at $(0, 0)$:

```
void GLApp::init() {
  // same as before ...
    GLApp::models.emplace_back(GLApp::lines_model(40, 40,
                                                  my_tutorial_2_vs,
                                                  my_tutorial_2_fs));
  // rest as before ...
}
```

Task 2d: Amend static member function `GLApp::draw`

In addition to rendering the geometric shape consisting of points in the top-right viewport, function `GLApp::draw` is updated to also render the geometric shape consisting of vertical lines:

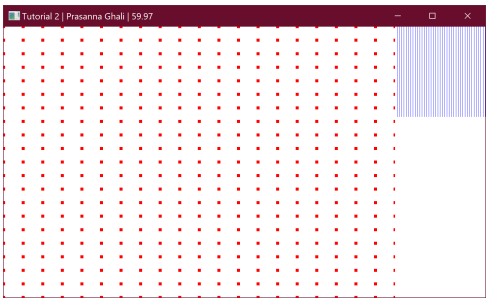
```
void GLApp::draw() {
  // write window title with stuff similar to sample ...

  // clear back buffer as before

  // render points in top-left viewport
  glViewport(vps[0].x, vps[0].y, vps[0].width, vps[0].height);
  GLApp::models[0].draw();

  // render line segments in top-right viewport
  glViewport(vps[1].x, vps[1].y, vps[1].width, vps[1].height);
  GLApp::models[1].draw();
}
```

Task 2e: Amend member function `GLModel::draw`

To render the geometrical shape consisting of line segments, member function `GLModel::draw` is defined as:

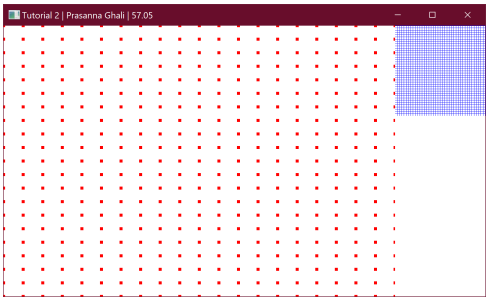
```
void GLApp::GLModel::draw() {
  shdr_pgm.Use();
  glBindVertexArray(vaoid);
  switch (primitive_type) {
  case GL_POINTS:
    // as before ...
    break;
  case GL_LINES:
    glLineWidth(3.f);
    glVertexAttrib3f(1, 0.f, 0.f, 1.f); // blue color for lines
    glDrawArrays(primitive_type, 0, draw_cnt);
    glLineWidth(1.f);
    break;
  }
  glBindVertexArray(0);
  shdr_pgm.UnUse();
}
```

The default OpenGL setting is to rasterize `GL_LINES` primitives with a width of 1 pixel. To rasterize thicker line segments, `GLModel::draw` changes the default size from 1 to 3 using command `glLineWidth`. After the draw call, it is good practice to reset line size back to its original value. No other changes need to be made and the image displayed by your application should now look like this:



Task 2f: Complete static member function `GLApp::lines_model`

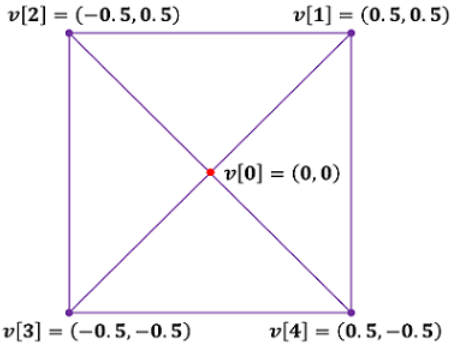
You'll now need to augment function `GLApp::lines_model` so that it creates a geometrical shape consisting of both vertical and horizontal lines. To render $(stacks + 1)$ number of horizontal lines, $2 \times (stacks + 1)$ additional points are required. You will only be updating the previous definition of function `GLApp::lines_model` and the image displayed by your application at the conclusion of this task should look like this:



🎯Task 3: Render triangle fans

Task 3a: Understand triangle fans

Suppose we wish to render the following triangle mesh consisting of 5 vertices and 4 triangles using OpenGL primitive `GL_TRIANGLES`.



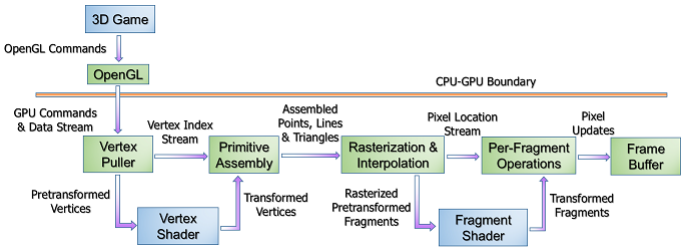
To render these triangles efficiently using a single call to OpenGL command `glDrawElements`, the shape's vertices must be stored in a GPU buffer that for simplicity is assumed to be represented by the following construct:

```
std::vector<glm::vec2> v {
  glm::vec2(0.f, 0.f),     // v0
  glm::vec2(0.5f, 0.5f),   // v1
  glm::vec2(-0.5f, 0.5f),  // v2
  glm::vec2(-0.5f, -0.5f), // v3
  glm::vec2(0.5f, -0.5f)   // v4
};
```

In addition, indices which specify the topology of the shape must also be stored in an index buffer that again for simplicity is assumed to be represented by the following construct:

```
std::vector<GLushort> idx {
  0, 1, 2,
  0, 2, 3,
  0, 3, 4,
  0, 4, 1
};
```

When the primitive type is `GL_TRIANGLES`, the OpenGL pipeline treats each triangle as an independent unit unrelated to other triangles in the geometric shape.



The vertex puller uses a set of three indices from the index buffer `idx` to determine which vertices must be consumed by the vertex shader. Since the first three elements of index buffer `idx` are indices 0, 1, and 2, the vertex puller will provide the vertex shader the vertex contained at index 0 in vertex buffer `v` followed by vertex at index 1, and then followed by the vertex at index 2. Vertices subsequently supplied to the vertex shader are determined by the next set of three indices in index buffer `idx` which are 0, 2, and 3.

Based on Euler's characteristic, large triangular meshes with m triangles and n vertices typically have about twice as more triangles than vertices, that is $m \approx 2n$. Why? For a closed mesh with no holes, Euler's characteristic formula asserts

$$n - e + m = 2 \tag{1}$$

where n, e, m are the number of vertices, edges, and polygonal faces of the mesh. In a closed triangle mesh, every edge is shared by two faces, and every triangular face has three edges. It is useful to think of the mesh as made up of half-edges where each half-edge is a pair of an edge and the face it borders. Since each edge is shared by two faces, there are two half-edges for each edge for a total of $2e$ half-edges. Since each face has 3 half-edges, there are a total of $3m$ half-edges. Therefore, we've:

$$2e = 3m \tag{2}$$

As an example, consider a tetrahedron, where $e = 6$ and $m = 4$. When e is replaced in Equation (1) by $\frac{3}{2}m$ derived from Equation (2), we obtain the following:

$$\begin{aligned} n - \frac{3}{2}m + m &= 2 \\ n - \frac{1}{2}m &= 2 \\ \frac{1}{2}m &= n - 2 \\ \implies m &= 2n - 4 \end{aligned}$$

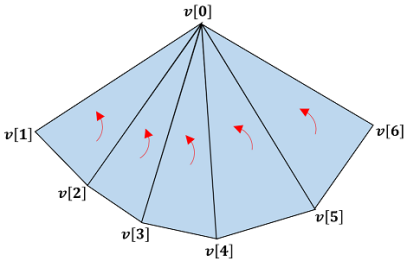
As the mesh size increases, the number of triangles m converges to twice the number of vertices n:

$$\boxed{m \approx 2n}$$

Since each triangle must be specified with 3 indices, the index buffer for this large mesh would require storage for about $3m$ or $6n$ indices. Since 3 vertices per triangle must be transformed by the vertex shader, the mesh will require about $3m$ or $6n$ vertex transformations. The storage requirements for triangle indices and transformations performed by vertex shaders are of the order of about $6n$ for `GL_TRIANGLES` primitives.

One of the simplest ways to speed up an OpenGL program while simultaneously saving storage space is to convert independent triangles into triangle fans or triangle strips. The idea here is that rather than having OpenGL treat each triangle primitive of type `GL_TRIANGLES` as being independent of other triangles, the application can provide information to OpenGL about relationships between successive triangles using primitive types `GL_TRIANGLE_FAN` or `GL_TRIANGLE_STRIP`.

A triangle fan begins with a pivot vertex, and uses the next two vertices to create the first triangle. The OpenGL pipe will cache the pivot vertex and the third vertex of the previous triangle. Each subsequent vertex will create another triangle, fanning around the original pivot vertex. The pivot vertex is the first vertex of the new triangle, the third vertex of the previous triangle that was cached will now become the second vertex of the new triangle, with the new vertex completing the fan as the third vertex of the new triangle. Since the application is using counter-clockwise winding with `GL_TRIANGLES` primitives, triangle fans must be started with counter-clockwise winding, with all subsequent triangles assumed by the OpenGL pipe to be wound in the same direction.



Assuming the vertices of the above triangle mesh are stored in vertex buffer `v`, the indices of the five triangles are stored in index buffer `idx` like this:

```
std::vector<GLushort> idx {
  0,    // pivot vertex
  1, 2, // remaining vertices for 1st triangle: v0 -> v1 -> v2
  3,    // new vertex for 2nd triangle: v0 -> v2 -> v3
  4,    // new vertex for 3rd triangle: v0 -> v3 -> v4
  5,    // new vertex for 4th triangle: v0 -> v4 -> v5
  6     // new vertex for 5th triangle: v0 -> v5 -> v6
};
```

Since each triangle [other than the first triangle] requires a single additional vertex (or an index to the vertex), large geometric meshes would require storage for about m or $2n$ indices. Since only a single vertex per triangle (other than the first triangle) must be transformed by the vertex shader, the geometric mesh will require about m or $2n$ vertex transformations to render the mesh. The storage requirements for index buffer for meshes rendered using `GL_TRIANGLE_FAN` is about three times smaller compared to equivalent meshes rendered using primitive `GL_TRIANGLES`. Unless the GPU contains hardware for post-transform vertex caches, meshes rendered using `GL_TRIANGLE_FAN` require three times less vertex shader transforms compared to meshes rendered using `GL_TRIANGLES`. Although the common assumption on the web is that GPUs

contain post-transform vertex cache, there is no documentation from vendors of modern GPUs explicitly confirming the presence of post-transform vertex caches.

Task 3b: Declare static member function `GLApp::trifans_model`

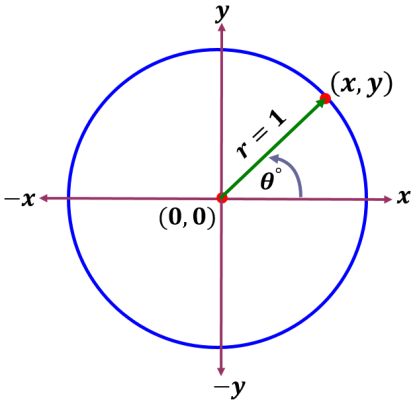
Begin by declaring static member function `trifans_model` in structure `GLApp`:

```
struct GLApp {
  // same as before ...
  static GLApp::GLModel trifans_model(int slices,
                                      std::string const& vtx_shdr,
                                      std::string const& frag_shdr);
};
```

Since circular disks can be divided into triangular pieces using just the `slices` parameter, this function doesn't declare a parameter `stacks`.

Task 3c: Procedurally generating a mesh for a disk

Consider a circle with radius $r = 1$ centered at origin $(0, 0)$. As illustrated in the following picture, any point (x, y) on the circle's circumference can be computed in terms of an angular parameter θ° spanning vectors $\hat{i} = (1, 0)$ and (x, y).

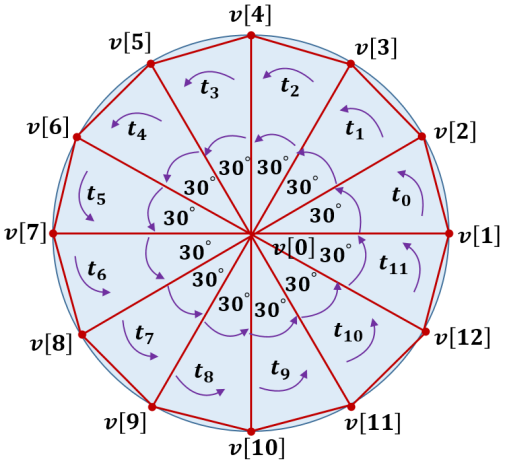


The point's coordinates (x, y) are computed in terms of parameter θ° as:

$$x = r\cos\theta^\circ = \cos\theta^\circ$$
$$y = r\sin\theta^\circ = \sin\theta^\circ$$

Given input parameter $slices$, a set of $slices$ count of points can be parameterized on the circle's circumference using angular parameter $\theta^\circ = \frac{360^\circ}{slices}$. The first parameterized point will have coordinates $(\cos 0^\circ, \sin 0^\circ)$, the second parameterized point will have coordinates $(\cos\theta^\circ, \sin\theta^\circ)$, the third parameterized point will have coordinates $(\cos 2\theta^\circ, \sin 2\theta^\circ)$, and so on.

The following picture illustrates parameterization of the circle's circumference for $slices = 12$. The circle's center and $slices$ count of parameterized points can be used to divide the circle into a set of $slices$ number of equivalent triangular slices. Since triangular slices share the circle's center, it is more efficient to render the set of $slices$ triangles as a triangle fan rather than as individual triangles.



We wish to render these triangle-fan primitives using `glDrawArrays` without the need to store indices [similar to how the other geometrical shapes with `GL_POINTS` and `GL_LINES` primitives are being rendered]. We will use the above picture where $slices = 12$ to determine the data store for the VBO. To render the first triangle t_0, three vertices $v[0]$, $v[1]$, and $v[2]$ are required. To render the next ten triangles (ranging from t_1 through t_{10}), a single vertex is required. So far, 13 vertices are required to render $(slices - 1)$ triangles. To render the final triangle t_{11}, we need an additional vertex $v[1]$. This means that $slices$ triangles can be rendered as a triangle fan using $(slices + 2)$ vertices.

Task 3d: Define static member function `GLApp::trifans_model`

The outline of function `GLApp::trifans_model` will look like this and is left to the reader as an exercise:

```
GLApp::GLModel GLApp::trifans_model(int slices, std::string const& vtx_shdr,
                                    std::string const& frag_shdr) {
  // Step 1: Generate the (slices+2) count of vertices required to
  // render a triangle fan parameterization of a circle with unit
  // radius and centered at (0, 0)

  // Step 2: In addition to vertex position coordinates, compute
  // (slices+2) count of vertex color coordinates.
  // Each RGB component must be randomly computed.

  // Step 3: Generate a VAO handle to encapsulate the VBO(s) and
  // state of this triangle mesh

  // Step 4: Return an appropriately initialized instance of GLApp::GLModel
}
```

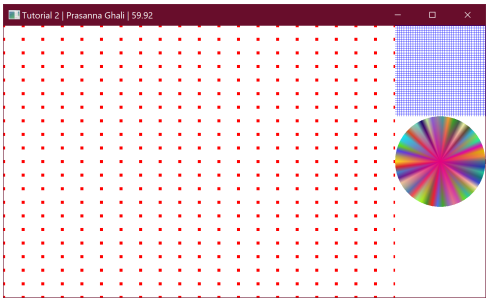
We'll postpone the discussion of C++ random number library to a future tutorial. This section describes the generation of pseudo-random numbers using the C standard library.

Task 3e: Render circle geometry shape

Complete the remaining tasks required to render the parameterized circle. You'll need to amend `GLApp::init` to invoke static member function `GLApp::trifans_model` and insert the returned instance of type `GLApp::GLModel` to container `GLApp::models`. The sample calls function `GLApp::trifans_model` like this:

```
GLApp::models.emplace_back(GLApp::trifans_model(50,
                                                my_tutorial_2_vs,
                                                my_tutorial_2_fs));
```

Function `GLApp::draw` must be amended to set the bottom-left viewport as the current rendering viewport and invoke member function `GLModel::draw` to render the triangle fan to the appropriate viewport using a draw call to `glDrawArrays`. The image displayed by your application at the conclusion of this task should look like this.

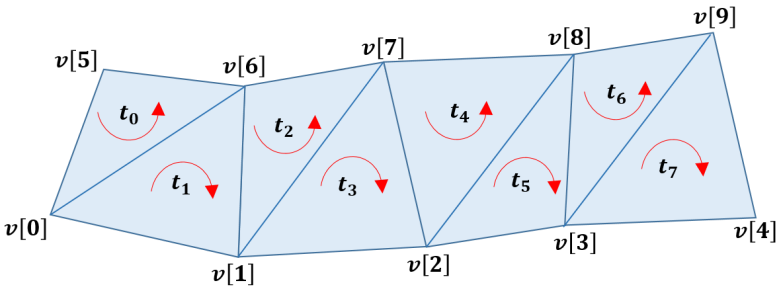


The circle disk is rendered as an ellipse because the mesh is defined in NDC system which is a square box centered at $(0, 0)$ having size 2×2 while the image is displayed in the viewport which has an aspect ratio of $\frac{16}{9}$ in the sample.

🎯Task 4: Render triangle strips

Task 4a: Understand triangle strips

Another interesting representation of a triangle mesh is a triangle strip. As shown in the following figure, a triangle strip describes the mesh as a sequence of adjacent triangles.



Suppose the index buffer for this triangle strip is defined like this:

```
std::vector<GLushort> vtx_idx { 5, 0, 6, 1, 7, 2, 8, 3, 9, 4 };
```

The first triangle t_0 is composed by fetching three vertices in vertex buffer pointed to by the first three indices from index buffer `vtx_idx`. Thus, triangle t_0 is then rendered by processing vertices $v[5]$, $v[0]$, and $v[6]$. The last two vertices $v[0]$ and $v[6]$ are cached while the first vertex $v[5]$ is discarded. To render the second triangle t_1, cached vertices $v[0]$ and $v[6]$ already exist. Hence, only a single index with value 1 must be fetched from the index buffer and the corresponding vertex $v[1]$ is fetched from the vertex buffer and processed. Again, the most recent two vertices $v[6]$ and $v[1]$ are cached. Similarly, to render the third triangle t_2, the cached vertices are simply fetched from the cache. Again, only a single index with value 7 is fetched from the index buffer and the corresponding vertex $v[7]$ is fetched from the vertex buffer and processed. The remaining triangles are processed in the same manner: reuse the two cached vertices which are the final two vertices of the most recent rendered triangle and fetch a new index from the index buffer and then fetch the corresponding vertex from the vertex buffer.

Decide whether the first triangle should have a clockwise or counterclockwise winding, then ensure all subsequent triangles in the triangle sequence have alternate windings. In the example, since the first triangle t_0 has counter-clockwise winding, the subsequent triangles must have their vertices ordered so that they follow alternate clockwise followed by counter-clockwise windings.

All the advantages enumerated for triangle fans accrue for triangle strips. Since OpenGL does not have a way to specify generalized triangle strips, the user must choose between `GL_TRIANGLE_STRIP` and `GL_TRIANGLE_FAN`. In general, the triangle strip is the more versatile primitive. While triangle fans are ideal for large convex polygons that need to be converted to triangles or for triangulating geometry that is cone-shaped, most other cases are best converted to triangle strips. For this reason, Direct3D 10 or later don't support triangle fans.

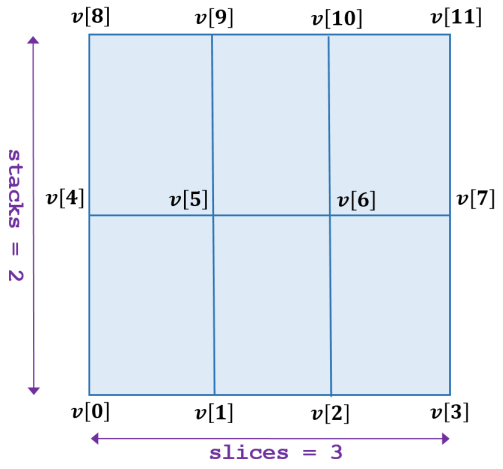
Task 4b: Subdividing NDC box into triangle strip

Assuming `slices` and `stacks` are input parameters, this task is concerned with subdividing the 2×2 NDC box centered at $(0, 0)$ into $(2 \times slices \times stacks)$ number of triangles that can be rendered using OpenGL primitive `GL_TRIANGLE_STRIP`. A two-pass algorithm is required to perform this task. In the first pass, using `slices` and `stacks` as input parameters, the vertex position and color attributes are computed. In the second task, an index buffer that defines a triangle strip topology is generated.

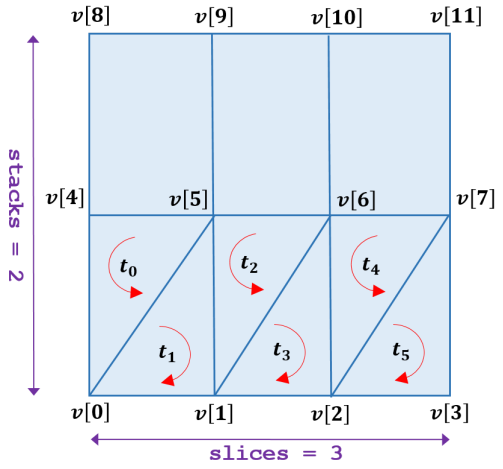
Luckily, the first pass is straightforward; you used the *slices* and *stacks* method in $\mathrm{Task}\ 1$ to generate a grid of points in the NDC box. The previously defined function `GLApp::points_model` from $\mathrm{Task}\ 1\mathrm{h}$ is repeated here:

```
GLApp::GLModel GLApp::tristrip_model(int slices, int stacks,
                                     std::string const& vtx_shdr,
                                     std::string const& frag_shdr) {
  // Compute and store a "stack" of uniformly "sliced" points
  // such that there are (stacks+1) number of point sequences
  // with each point sequence consisting of (slices+1) number of
  // points ranging from (-1, y) to (1, y).
  // The 1st point sequence is at y = -1 and the last sequence
  // is at y = 1
  // Store the (slices+1)*(stacks+1) number of points in a vector
  // Randomly generate r, g, b values for each point and store
  // them in a separate vector container
}
```

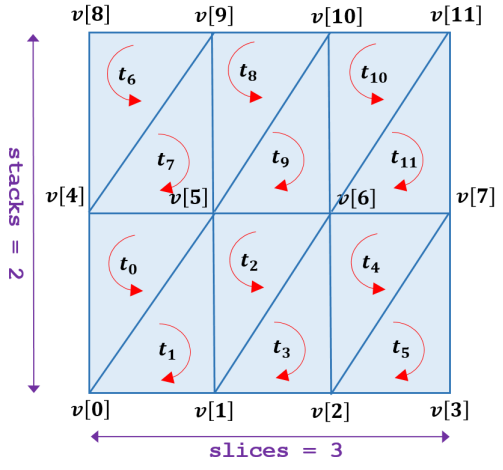
Assuming parameter `slices` is 3 and parameter `stacks` is 2, the first pass will generate a set of $(3 + 1) \times (2 + 1) = 12$ vertices. The 4×3 grid of vertices is illustrated in the following picture. The 4 vertices in the first (bottom) row are inserted first into the vertex buffer `v`, the 4 vertices in the second (middle) row are inserted next, followed by the insertion of the 4 vertices in the third (top) row.



Generating the index buffer describing a triangle strip requires a bit more work. Based on the earlier example, the process begins with the leftmost vertex in the second row labeled $v[4]$ and the leftmost vertex in the first row labeled $v[0]$. To generate the first triangle t_0 that is counter-clockwise oriented, indices 4, 0, and 5 are inserted into the index buffer. The second triangle t_1 that is clockwise oriented is generated by inserting index 1 into the index buffer. Subsequent triangles in the bottom stack are generated by inserting indices 6, 2, 7, followed by index 3 into the index buffer. The 6 triangles in the triangle strip generated by an index buffer consisting of indices $\{4, 0, 5, 1, 6, 2, 7, 3\}$ is illustrated below:



Now consider the triangle strip for the top row generated by an index buffer consisting of indices $\{8, 4, 9, 5, 10, 6, 11, 7\}$. This top triangle strip is illustrated below:



Is it possible to connect the bottom discrete triangle strip specified by index list $\{4, 0, 5, 1, 6, 2, 7, 3\}$ and the top discrete triangle strip specified by index list $\{8, 4, 9, 5, 10, 6, 11, 7\}$ into a single triangle strip? The short answer is yes and the solution relies on the OpenGL specification indicating that a triangle with zero area is considered *degenerate* and is discarded. A triangle's area is zero if two or more triangle edges are collinear or have zero length because the cross product of the triangle's edges will result in a $\vec{0}$. To connect the bottom triangle strip with index list $\{4, 0, 5, 1, 6, 2, 7, 3\}$ with the top triangle strip with index list $\{8, 4, 9, 5, 10, 6, 11, 7\}$, do the following:

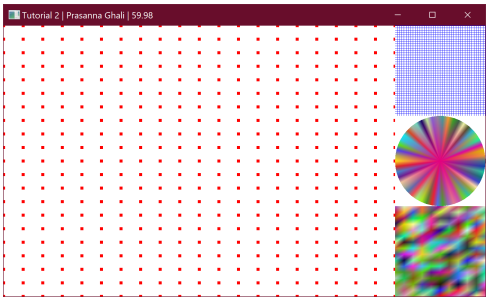
1. Amend the bottom strip's index list by inserting the final index to the back of the index list; the amended index list for the bottom strip is: $\{4, 0, 5, 1, 6, 2, 7, 3, 3\}$
2. Amend the top strip's index list by inserting its first index to the front of the index list; the amended index list for the top strip is: $\{8, 8, 4, 9, 5, 10, 6, 11, 7\}$
3. Merge the two index lists by appending the top list to the back of the bottom list; the merged index list looks like this: $\{4, 0, 5, 1, 6, 2, 7, 3, 3, 8, 8, 4, 9, 5, 10, 6, 11, 7\}$

The addition of indices 3 and 8 causes four degenerate triangles: the first is specified by indices 7, 3, 3; the second is specified by indices 3, 3, 8; the third is specified by indices 3, 8, 8; and the fourth and final degenerate triangle is specified by indices 8, 8, 4.

You must implement a static member function `GLApp::tristrip_model` that is declared as:

```
struct GLApp {
  // stuff from before ...
  static GLApp::GLModel tristrip_model(int slices, int stacks,
                                       std::string const& vtx_shdr,
                                       std::string const& frag_shdr);
};
```

In addition, you must complete the rest of the program to generate the triangle strip and render it to the bottom-right viewport. The image generated by your program should match [or exceed] the image generated by the sample. The sample calls function `GLApp::tristrips_model` with `slices` equal to 10 and `stacks` equal to 15.



🎯Task 5: Printing to window title bar

The sample prints the assignment's name followed by the author's name followed by information relevant to the primitives rendered in each viewport followed by the frames per second count with each field separated by a vertical bar:

> Tutorial 2 | Parminder Singh | POINTS : 441, 441 | LINES : 82, 164 | FAN : 50, 52 | STRIP : 300, 358 | 60.0118

Begin by adding your name to the text printed to the title bar. The top-left image is generated with `slices` and `stacks` equivalent to 20. The image is generated by buffering 441 `GL_POINTS` primitives [$(20+1) \times (20+1) = 441$] and setting `count` parameter of draw command `glDrawArrays` to 441. The top-right image is generated with `slices` and `stacks` equivalent to 40. It requires 82 ($2 \times (40+1) = 82$) `GL_LINES` primitives to be buffered and rendered by setting `count` parameter of draw command `glDrawArrays` to 82. The bottom-right image is created with `slices` set to 50. The information printed for this image indicates there are 50 triangular slices rendered with `count` parameter of draw command `glDrawArrays` for `GL_TRIANGLE_FAN` primitive set to 52. The bottom-right image is rendered with `slices` and `stacks` set to 10 and 15, respectively. The displayed value 300 indicates the actual number of triangles that must be rendered ($10 \times 15 \times 2 = 300$) while 358 indicates the argument to the `count` parameter of draw command `glDrawElements` for `GL_TRIANGLE_STRIP` primitive.

🎯Task 6: Adapting viewport for resizing window

When you resize the window, you'll notice that the viewports don't adapt to the new window size. This happens because viewports are set in function `GLApp::init` and are not changed when the display window's dimensions change. Viewport settings are mapped to updated window dimensions in function `GLApp::update`:

```
void GLApp::update() {
  // previous code

  GLint w{GLHelper::width}, h{GLHelper::height};
  static GLint old_w{}, old_h{};
  // update viewport settings in vps only if window's dimension change
  if (w != old_w || h != old_h) {
    GLint hs{h/3}, ws{hs};
    GLApp::vps[0] = { 0, 0, w - ws, h };
    GLApp::vps[1] = { w - ws, 2 * hs, ws, hs };
    GLApp::vps[2] = { w - ws, hs, ws, hs };
    GLApp::vps[3] = { w - ws, 0, ws, hs };
    old_w = w;
    old_h = h;
  }
}
```

This code guarantees that viewport settings adapt to resized window dimensions, ensuring accurate rendering and display across various window sizes. Static variables `old_w` and `old_h` optimize the process by verifying the display window's dimensions have been updated since the previous frame. Upon detection of a change, the `GLApp::vps` are appropriately updated to reflect the new viewport configuration.

🎯Task 7: Dynamically Switching Viewports

Introduce key event 1, 2, 3, and 4 to render geometric shapes with `GL_POINTS`, `GL_LINES`, `GL_TRIANGLE_FAN`, and `GL_TRIANGLE_STRIP` primitives, respectively on the left [that is, larger] viewport. Declare a static variable `showModelID` of type `GLint` in class `GLHelper`. This variable is updated with the index of the appropriate model to be displayed in the left viewport when the key event occurs. Study the sample's behavior when these key events occurs to assist you in implementing this task.

Bonus Task

From the information printed to the window title bar, it is clear that a large number of degenerate triangles are being created to render the rectangular mesh [in bottom-right viewport] as a single triangle strip. In contrast to the degenerate triangle method, OpenGL provides the *primitive restart* method via command `glPrimitiveRestartIndex` to render a single triangle strip. In this bonus task render the bottom-right image using command `glPrimitiveRestartIndex` to gain valuable insight(s) into comparing the pros and cons of one method versus the other. Using the same count of `slices` and `stacks`, replace 358 [from degenerate method implemented by the sample] with your implementation's argument to the `count` parameter of draw command `glDrawElements`. Clearly indicate in the file header of glapp.cpp the specific function and the specific line numbers where the bonus task is being implemented.

Submission

1. Source and header files for Tutorial 2 must be placed in a directory labeled as: <login>-<tutorial-2>. If your Moodle student login is foo, then the directory would be labeled as foo-tutorial-2 and would have the following structure and layout:

```
foo-tutorial-2          # You're submitting Tutorial 2
├─ include              # Header files - *.hpp and *.h files
└─ src                  # Source files - *.cpp and .c files
   └─ my-tutorial-2.vert # Vertex shader file
   └─ my-tutorial-2.frag # Fragment shader file
```

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

⚠ Before Submission: Verify and Test it ⚠

- Unzip your archive file foo-tutorial-2.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-2 # Tutorial 2 is submitted by foo
|     └─ include     # Header files - *.hpp and *.h files
|     └─ src         # Source files - *.cpp and .c files
|        └─ my-tutorial-2.vert  # Vertex shader file
|        └─ my-tutorial-2.frag  # Fragment shader file
└─ csd2101.bat               # Automation Script
```

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

Things to test before submission	Status
Assessment compiles without any errors	☐

Things to test before submission	Status
All warnings resolved, achieving zero warnings	☐
Executable generated and successfully launched in debug and release mode	☐
Directory is zipped, ensuring adherence to naming conventions as outlined in the submission guidelines	☐
Upload zipped file to assessment submission page on Moodle	☐

ℹ️ *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***] $\mathbf{Task\ 1}$ completion shows understanding of NDC, window coordinate system, viewports, viewport transforms, OpenGL viewport transform command `glViewport`, 2D space parameterization, and OpenGL primitive `GL_POINTS`.
- [***core4***] $\mathbf{Task\ 2}$ completion shows basic understanding of NDC, window coordinate system, viewports, viewport transforms, OpenGL viewport transform command `glViewport`, circle parameterization, and OpenGL primitive `GL_LINES`.

- [***core5***] $\mathbf{Task\ 3}$ completion shows basic understanding of NDC, window coordinate system, viewports, viewport transforms, OpenGL viewport transform command `glViewport`, 2D space parameterization, and OpenGL primitive `GL_TRIANGLE_FAN`.
- [***core6***] $\mathbf{Task\ 4}$ completion shows basic understanding of NDC, window coordinate system, viewports, viewport transforms, OpenGL viewport transform command `glViewport`, 2D space parameterization, and OpenGL primitive `GL_TRIANGLE_STRIP`.
- [***core7***] $\mathbf{Task\ 5}$ completion shows basic understanding of ability to follow instructions in specification.
- [***core8***] $\mathbf{Task\ 6}$ completion illustrates impact of resizing the window on the viewport and solution for correct rendering.
- [***core9***] $\mathbf{Task\ 7}$ completion shows basic understanding of dynamic interaction with viewports.

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**
core1 rubric is not satisfied. Submitted property page and/or source code doesn't build. Or, executable generated by build crashes or displays nothing. Or no submission.	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.	
Only ***one*** of ***core3*** through ***core9*** rubrics are satisfied.	D
Only ***two*** of ***core3*** through ***core9*** rubrics are satisfied.	$D+$
Only ***three*** of ***core3*** through ***core9*** rubrics are satisfied.	C
Only ***four*** of ***core3*** through ***core9*** rubrics are satisfied.	$C+$
Only ***five*** of ***core3*** through ***core9*** rubrics are satisfied.	B
Only ***six*** of ***core3*** through ***core9*** rubrics are satisfied.	$A-$
All ***seven*** of ***core3*** through ***core9*** rubrics are satisfied.	A
Bonus Task: Grade for successfully completing bonus task is possible only if you've completed all seven core rubrics satisfactorily.	$A+$

Random Numbers

A sequence of random numbers is not defined by an equation; instead, it has certain characteristics that define it such as maximum, minimum, and average values. These characteristics also indicate whether the possible values are equally likely to occur or whether some values are more likely to occur than others. Sequences of random numbers can be

generated from experiments, such as tossing a coin or rolling a die. Sequences of random numbers can also be generated using the computer.

Many engineering problems require the use of random numbers in the development of a solution. In some cases, the numbers are used to develop a simulation of a complicated problem. The simulation can be run over and over to analyze the results; each repetition represents a repetition of the experiment.

The C standard library contains a function `rand` that returns a random positive number of type `int` between `0` and `RAND_MAX`, where `RAND_MAX` is a system-dependent integer defined in `<cstdlib>`. Most implementations of the library provide a value of $2^{31} - 1$ for `RAND_MAX`. Function `rand` has no arguments and is referenced by the expression `rand()`. Thus, to generate and print a sequence of two random numbers, you could use this statement:

```
std::cout << "random numbers: " << rand() << " " << rand() << "\n";
```

Each time a program containing this statement is executed, the same two values are printed, because function `rand` generates integers in a specified sequence. Because this sequence eventually begins to repeat, it is sometimes called a *pseudo-random sequence* instead of a *random sequence*. However, if you generate additional random numbers in the same program, they will be different. Thus, this pair of statements generates four different random numbers:

```
std::cout << "random numbers: " << rand() << " " << rand() << "\n";
std::cout << "random numbers: " << rand() << " " << rand() << "\n";
```

Each time function `rand` is referenced in a program, it generates a new value; however, each time that the program is run, it generates the same sequence of values.

In order to cause a program to generate a new sequence of random values each time that it is executed, the random-number generator must be seeded with a value. The function `srand` declared in header file `<cstdlib>` specifies the seed for the random-number generator; for each seed value, a new sequence of random numbers is generated by function `rand`. The argument of function `srand` is an integer of type `unsigned int` that is used in computations in the library that initializes the sequence. Note that the seed value is not the first value in the sequence. If function `srand` is not called before function `rand` is called, the library assumes a value of `1` for the seed value. Therefore, if the following statement is added to a program

```
srand(1);
```

the program will generate the same sequence of values from function `rand` if function `srand` was not called.

In the following program, the user is asked to enter a seed value, and then the program generates 10 random numbers. Each time the user executes the program and enters the same seed, the same set of 10 random integers is generated. Each time a different seed is entered, a different set of 10 random integers is generated.

```
#include <iostream> // std::cout, std::cin
#include <cstdlib>  // std::srand, std::rand

int main() {
  std::cout << "Enter a seed: ";
```

```
  unsigned int seed;
  std::cin >> seed;
  std::srand(seed);

  // generate and print some random numbers ...
  std::cout << "Random numbers: ";
  for (int i {0}, N {10}; i < N; i++) {
    std::cout << rand() << (i!=N-1 ? ' ' : '\n');
  }
}
```

Experiment with the program: use the same seed to generate the same set of random numbers, and use different seeds to generate different sets of random numbers.

> ***Note that the values generated by function `rand` are system dependent; the same set of random numbers may not be generated by function `rand` from a different compiler.***

Generating random integers over a specified range is simple with function `rand`. Suppose you want to generate random integers between 0 and 9. The following statement first generates a random number that is between `0` and `RAND_MAX`; then it uses the modulus operator to compute the modulus of the random number and integer `10`:

```
int x = rand() % 10;
```

The result of the modulus operation is the remainder after `rand()` is divided by `10`, so the value of `x` can assume integer values between `0` and `9`.
Suppose you want to generate a random integer between -30 and 30. The total number of possible integers is 61, and a single random number in this range can be computed with this statement:

```
int y = rand()%61 - 30;
```

The expression `rand()%61` evaluates to a value between `0` and `60`. Subtracting `30` from this expression will yield a new value between `-30` and `30`.

How should integers be generated between two specified integers m and n? First, the count of all integers between m and n, inclusive is determined: $num = n - m + 1$. Second, the value returned by function `rand` and num are given as operands to the modulus operator to obtain values in $[0, n - m]$. Finally, lower limit m is added to values in $[0, n - m]$ to obtain values in range $[m, n]$. All three steps are combined in one expression in the `return` statement of function `rand_int`:

```
// return random integers in range [m, n]
int rand_int(int m, int n) {
  return rand()%(n-m+1) + m;
}
```

Function `rand_int` is illustrated in the following program that generates and prints 10 random integers between user-specified limits:

```
#include <iostream> // std::cout, std::cin
#include <cstdlib>  // srand, rand
```

```
int rand_int(int m, int n); // declaration of rand_int

int main() {
  // get seed value and interval limits
  std::cout << "Enter a positive integer seed value: ";
  unsigned int seed;
  std::cin >> seed;
  srand(seed);
  std::cout << "Enter integer limits m and n (m < n): ";
  int m, n;
  std::cin >> m >> n;

  // generate and print some random numbers ...
  std::cout << "Random Numbers: ";
  for (int i {0}, N {10}; i < N; i++) {
    std::cout << rand_int(m, n) << (i!=N-1 ? ' ' : '\n');
  }
}
```

In many engineering problems, it is necessary to generate random floating-point values in a specified interval $[m, n]$. The computation to convert an integer between `0` and `RAND_MAX` to a floating-point value between m and n has three steps. First, the value returned by function `rand` is divided by `RAND_MAX` to generate a floating-point value in range $[0, 1]$. Next, values in this range are scaled by $n - m$ to values in range $[0, n - m]$. Finally, the lower limit m is added to values in $[0, n - m]$ to obtain values in range $[m, n]$. All three steps are combined in one expression in the `return` statement of function `rand_dbl`:

```
// return random double-precision floating-point values in range [m, n]
double rand_dbl(double m, double n) {
  double rand_max = static_cast<double>(RAND_MAX);
  double rand_val = static_cast<double>(rand());
  return (rand_val/rand_max) * (n - m) + m;
}
```

The earlier program can be easily modified to generate and print double-precision floating-point values.