Assignment 1: Line and Triangle Rasterization

Note to the reader

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

Topics Covered

- Model and viewport transforms
- Rasterization of line segments using Bresenham algorithm
- Rasterization of triangles using edge equations
- Barycentric interpolation of attributes (colors)

Prerequisites

- Correct implementation of Tutorial 6.

Getting Started

Overwrite the existing batch file csd2101.bat in csd2101-opengl-dev with a new version available on the assessment's web page. Execute the batch file. Choose option **7 - Create Assignment 1** to create a Visual Studio 2022 project assignment-1.vcxproj in directory ./build with source in directory ./projects/assignment-1 whose layout is shown below:

```
📁 csd2101-opengl-dev/      # 📦 OpenGL sandbox directory
├─ 📁 build/                  # Build files will exist here
├─ 📁 projects/             # Tutorials and assignments
|   └─📁 assignment-1         # ➕ Assignment 1 code exists here
|     └ 📁 include          # 🖨 Header files - *.hpp and *.h sfiles
|         └ 📄 glhelper.h
|         └ 📄 glpbo.h
|         └ 📄 glslshader.h
|     └ 📁 src              # 🌟 Source files - *.cpp and .c files
|         └ 📄 glhelper.cpp
|         └ 📄 glpbo.cpp
|         └ 📄 glslshader.cpp
|         └ 📄 main-pbo.cpp
└─ 📄 csd2101.bat           # 📖 Automation Script
```

This assignment will reuse your implementation of `GLPbo` from Tutorial 6. Therefore, you must overwrite the default header file glpbo.h and source file glpbo.cpp in Assignment 1 with the corresponding files from Tutorial 6. You can safely remove key events code in these files since Assignment 1 will not use key events from Tutorial 6.

Loading OBJ Files

1. This submission will use $\textbf{DigiPen Mesh Library}$ [or DPML] to load 3D models stored in OBJ format. Directory ./lib/dpml contains the two types of files necessary for clients to use this library. Recall that to use a static library, C/C++ requires header file(s) with suffix .h declaring the interface and library file(s) with suffix .lib [in Windows] providing the implementation. DPML's interface is declared in $\text{./lib/dpml/include/dpml.h}$ [this file must be `#include`ed by any source file calling DPML functions]. The debug and release versions of the static library are named dpml.lib and are located in directories $\text{./lib/dpml/lib/Debug}$ and $\text{./lib/dpml/lib/Release}$, respectively. Script csd2101.bat automatically configures the libraries for both debug and release builds. The only thing you need to do is include `#include <dpml.h>` in your implementation for the necessary DPML declarations.
2. OBJ files that specify geometry information for 3D models are stored in directory ./meshes.
3. This assignment defines a scene file ass-1.scn that is stored in directory ./scenes. This file contains the list of OBJ models that your submission must be able to display. Each of these models is stored in a corresponding file [located in ./meshes] having the model name as the prefix and an obj suffix.
4. Augment function `GLPbo::init` with a call `DPML::parse_obj_mesh` to parse geometric data for 3D models. This function [declared and documented in dpml.h] does the work of loading geometric information related to a 3D model from an OBJ file. The parser is guaranteed to correctly parse position coordinates, per-vertex normal coordinates, and triangle edge indices from OBJ files listed in $\text{./scenes/ass-1.scn}$. Check the return value from the parser to determine if the OBJ file was successfully parsed. If the OBJ file doesn't contain per-vertex normal coordinates, the parser will compute these coordinates. In either case, the parser will normalize each per-vertex normal (n_x, n_y, n_z) so that $n_x \in [-1, 1]$, $n_y \in [-1, 1]$, $n_z \in [-1, 1]$ and $\sqrt{n_x^2 + n_y^2 + n_z^2} = 1$. In this assignment, you'll not be using per-vertex normals to implement real-time lighting calculations. Instead, after calling the parser, you must map the n_x, n_y, and n_z values from range $[-1, 1]$ to range $[0, 1]$. This hack will allow the triangle rasterizer to treat per-vertex normal coordinates as RGB color coordinates. By interpolating these RGB color coordinates across the triangle surface, the rasterizer can determine the RGB color at a fragment. Assume OBJ files listed in ass-1.scn have been curated to ensure position coordinates are in normalized device context coordinate system. That is, position coordinates (x, y, z) are guaranteed to be in range $[-1, 1]$. As long as scale and translate transformations are avoided, you should be able to apply a viewport transform to map the coordinates to window [or screen or device or viewport] coordinates. After this mapping, you can rasterize the triangle primitives to generate wireframe or filled images.
5. The geometry data returned by function `DPML::parse_obj_mesh` can be encapsulated in an object of type `GLPbo::Model`. Here's a sample declaration of type `GLPbo::Model`:

```cpp
struct GLPbo {
// other stuff here

// Data structure to store vertex position and triangle index arrays
// returned by DPML::parse_obj_mesh()
  struct Model {
// The vertex position array pm contains vertex position coordinates
// returned by DPML::parse_obj_mesh(). These coordinates are similar to
// the data sitting in a vertex buffer and must therefore be invariant.
// For this submission, position coordinates in pm are assumed to be in
// NDC coordinate system. Every frame, these position coordinates
// must be transformed by viewport transformation matrix to window
// (or viewport or device or screen) coordinates pd.
    std::vector<glm::vec3>      pm;  // invariant
// DPML::parse_obj_mesh() will return per-vertex normal coordinates
// that you must map to color coordinates in range [0, 1]
    std::vector<glm::vec3>      nml; // will contain per-vertex normal
                                     // coordinates which must then be
                                     // converted by you to RGB values
    std::vector<glm::vec2>      tex; // not used in this submission
    std::vector<unsigned short> tri; // triangle indices

// window coordinates in array pd are obtained after NDC coordinates in
// array pm are transformed by rotation transform followed by
// viewport transformation matrix
    std::vector<glm::vec3>      pd;
  };
};
```

6. Vertex coordinates returned by `DPML::parse_obj_mesh` are assumed to be in NDC coordinate system. Declare and define a function `GLPbo::viewport_xform()` that applies a rotation transform about $z-$axis followed by a viewport transform to map these NDC coordinates (x^n, y^n, z^n) into window coordinates (x^d, y^d, z^d). The window's bottom-left corner is `(0, 0)` while the window's width and height are given by `GLHelper::width` and `GLHelper::height`, respectively. Since this submission is not concerned with mapping z^n coordinate, set z^d to 0.
7. Rendering a texture mapped full-screen quad is similar to $\text{Tutorial } 6$ and will not require alterations.

🎯Task 1: Wireframe images

1. In addition to clearing the PBO's buffer with a clear color, rasterization algorithms must be able to write a particular RGBA value at a specific location in the PBO buffer. Declare and define a function `GLPbo::set_pixel` to write a RGBA color at pixel with window coordinates (x^d, y^d). Ensure that this function implements scissoring with the entire window specified as the scissor rectangle. Otherwise, there could be a stray vertex in the OBJ file that may have position coordinates outside range $[-1, 1]$ causing writes to out-of-bounds PBO buffer memory.
2. Now it is time to update the PBO's buffer with a wireframe image. Iterate through each triangle. Cull back-facing triangles. Render each of the three edges of a front-facing triangle using function `GLPbo::render_linebresenham`:

```
struct GLPbo {
// other declarations here

// set all pixels with same color draw_clr on line segment starting
// at point P1(x1, y1) and ending at point P2(x2, y2)
// Note: points are in window coordinates
  static void render_linebresenham(GLint px0, GLint py0,
                          GLint px1, GLint py1, GLPbo::Color draw_clr);
};
```

that implements the Bresenham line algorithm described in class lectures. Recall from Tutorial 6 that writes to the PBO buffer must be fenced between calls to `glMapNamedBuffer()` and `glUnmapNamedBuffer()`.

3. The beauty of the Bresenham algorithm comes from its amazingly simple method to replace floating-point operations with integer-only arithmetic. To appreciate its simplicity and beauty, your implementation must not use ***any*** non-integral constants, variables, or expressions. The parameters are integer values and therefore the entire definition must be based on integer-only arithmetic. Use the class presentation and handout to understand and implement this seminal algorithm.

🎯Task 2: Flat shaded triangle rasterizer

Implement a function `GLPbo::render_triangle` with declaration:

```
struct GLPbo {
// other declarations here ...

// Implements a flat shaded triangle rasterizer using edge equations
// and top-left tie-breaking rule.
// Whether fragment (x, y) is rasterized or not is based on the results of
// point sampling the fragment at its center (x+0.5, y+0.5).
// Top left tie-breaking rule must be implemented.
// Parameters p0, p1, and p2 represent vertices of triangle whose x- and y-
// values are specified in window coordinates.
// Front-facing triangles are assumed to have counterclockwise winding.
// Back-facing triangles must be culled.
// The clr parameter is specified by the caller with randomly generated
// color coordinates in range [0, 1].
// The function returns false if the triangle is back-facing; otherwise
// the function returns true.
  static bool render_triangle(glm::vec3 const& p0, glm::vec3 const& p1,
                              glm::vec3 const& p2, glm::vec3  const& clr);
};
```

to render flat shaded triangles.

🎯Task 3: Smooth shaded triangle rasterizer

The basic idea behind smooth shading is to compute a fragment's color by linearly interpolating the colors at each vertex. This results in smooth transitions between polygons of different colors. For this submission, you will render smooth shaded triangles by interpolating per-vertex normal coordinates [that you previously transformed from range $[-1, 1]$ to range $[0, 1]$]. To implement smooth shading, define an overloaded function `GLPbo::render_triangle`:

```cpp
struct GLPbo {
// other declarations here ...

// Implements a smooth shaded triangle rasterizer using edge equations,
// top-left tie-breaking rule, and Barycentric interpolation.
// Parameters p0, p1, and p2 represent vertices of triangle whose x- and y-
// values are specified in window coordinates.
// Front-facing triangles are assumed to have counterclockwise winding.
// Back-facing triangles must be culled.
// Parameters c0, c1, and c2 represent RGB color coordinates for vertices
// p0, p1, and p2, respectively. Color coordinates are specified in
// range [0, 1].
// The function returns false if the triangle is back-facing; otherwise
// the function returns true.
  static bool render_triangle(glm::vec3 const& p0, glm::vec3 const& p1,
                              glm::vec3 const& p2, glm::vec3 const& c0),
                              glm::vec3 const& c1, glm::vec3 const& c2);
};
```

Keyboard controls

Users must be able to iterate through each model [specified in ass-1.scn] using keyboard button M.

For each model, button W allows users to iterate through the following render modes:

- render wireframe image using black color
- render wireframe image with each triangle edge rendered using randomly generated color
- render flat shaded triangles using a randomly generated triangle color
- render smooth shaded triangles by interpolating per-vertex normal coordinates [that were previously transformed from range $[-1, 1]$ to range $[0, 1]$]

Button R allows users to rotate the models' 2D coordinates [with respect to $z-$axis].

🎯Task 4: Interactive painter application

Provide functionality that uses keyboard button P to toggle between *static* mode [consisting of Tasks 1, 2, and 3] and *painter* mode that allows users to draw line segments using the mouse. Define a static function `GLPbo::painter_mode` that implements the following *painter* features:

1. When mode is toggled from *static* to *painter*, the PBO buffer is filled with color yellow🟡.
2. Users *paint* by clicking on the left mouse button and then moving the mouse cursor. Every frame, the displaced mouse position is rendered as a line segment which is then rendered using color red🔴.
3. To ensure the *painting* doesn't disappear and *painted* lines persist forever [until the mode is toggled from *painter* to *static*], the PBO buffer should not be cleared.

Study the paint mode of the sample executable carefully to ensure an accurate reproduction of the sample's features in your submission.

🎯Task 5: Printing useful information

Add functionality to your submission to print the following information to the window title bar: a model's name, vertex count, triangle count, and the count of culled triangles. See the sample for guidance on the minimum functionality you must implement.

Submission

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

```
📘 foo-assignment-1       # 🖊 You're submitting Assignment 1
├─ 📁 include              # 🖨 Header files - *.hpp and *.h files
└─ 📁 src                  # 💥 Source files - *.cpp and .c files
```

2. It is ok to submit additional source and header files [excepting shader files] as long as they're clearly mentioned in pbo.cpp header declaration.
3. *Zip the directory* to create archive file foo-assignment-1.zip.

⚠Before Submission: Verify and Test⚠

- Copy archive file foo-assignment-1.zip into directory test-submissions. Unzip the archive file to create project directory foo-assignment-1 by typing the following command in the command-line shell [which can be opened by typing cmd and pressing Enter in the Address Bar]:

```
powershell -Command "Expand-Archive -LiteralPath foo-assignment-1.zip -
DestinationPath ."
```

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

```
📁 csd2101-opengl-dev        # 📦 Sandbox directory for all assessments
├─ 📁 test-submissions       # ⚠Test submissions here before uploading
|   └─ 📘 foo-assignment-1  # 🖊 foo is submitting Assignment 1
|      └─ 📁 include         # 🖨 Header files - *.hpp and *.h files
|      └─ 📁 src             # 💥 Source files - *.cpp and .c files
└─ 📄 csd2101.bat            # 📖 Automation Script
```

- Delete the original copy of foo-assignment-1 from directory projects to prevent duplicate project names during reconfiguration.
- Run batch file csd2101.bat and select option R to reconfigure the Visual Studio 2022 solution with the new project foo-assignment-1.
- Build and execute project foo-assignment-1 by opening the Visual Studio 2022 solution in directory build.
- You can also verify the build using below command line option:

```
C:\csd2101-opengl-dev\build> Release\assignment-1.exe
```

- Use the following checklist to ***verify and submit*** your submission:

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

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 files submitted is correctly named.
    - Source code files are *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 or from a previous tutorial, the file header must be annotated to indicate your co-authorship and the changes made to the original or previous document. 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***] Wireframe images are correctly generated using Bresenham ***integer-only*** algorithm.
- [***core4***] Flat shaded triangle images are correctly generated using edge equations with top-left tie-breaking rule.

- [***core5***] Smooth shaded triangle images are correctly generated using edge equations with top-left tie-breaking rule and barycentric interpolation of vertex colors.
- [***core6***] An interactive simple painter application with Bresenham ***integer-only*** algorithm.
- [***core7***] Correct information printed to window title bar **plus** ability to rotate a model.

Mapping of Grading Rubrics to Letter Grades

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

Grading Rubric Assessment	**Letter Grade**
There is no submission.	F
core1 rubric is not satisfied. Submitted property page and/or source code doesn't build. Or, executable generated by build crashes or displays nothing.	F
If ***core2*** rubrics are not satisfied, final letter grade will be decreased by one. This means that if you had received a grade A and ***core2*** is not satisfied, your grade will be recorded as B, an $A-$ would be recorded as $B-$, and so on.	
core3 rubric is complete. Use of floating-point variables, or expressions, or arithmetic will result in zero grade for this rubric.	
core4 rubric is complete. Use of edge equations and top-left tie-breaking rule is mandatory. Dropouts and flickers will result in deductions.	
core5 rubric is complete. Use of edge equations, top-left tie-breaking rule, and barycentric interpolation is mandatory. Dropouts and flickers will result in deductions.	
core6 rubric is complete. Use creative thinking to implement real-time interactive painter application with line and possibly extend with triangles rasterization(in-future).	
One of ***core3***, ***core4***, ***core5***, ***core6*** rubrics. *If you don't provide functionality [similar to sample] to cycle thro' different rendering modes, then you understand that your submission satisfies only one of these rubrics. If you don't provide functionality [similar to sample] to cycle thro' different models [that is you display just a single model], then you understand that your submission satisfies only one of these rubrics.*	D
Three of ***core3***, ***core4***, ***core5***, ***core6*** rubrics.	C
Two of ***core3***, ***core4***, ***core5***, ***core6*** rubrics.	B
All of ***core3***, ***core4***, ***core5***, ***core6*** rubrics.	$A-$
core7 rubric is complete.	A

Grading Rubric Assessment	**Letter Grade**
Implementation of your own OBJ parser. Carefully read this link for information about OBJ files. Read this link for information on computing vertex normals if OBJ file doesn't contain vertex normal data. NOTE: Other OBJ files will be used to test your implementation - especially files that have a different number of position, normal, and texture coordinates. Therefore, submit an implementation that can switch between your parser and the one supplied to you - possibly thro' a keyboard button.	$A+$