Assignment 2: Basic illumination, shading, and texture mapping

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 transforms
- View transform
- Orthographic projection transform
- Viewport transform
- Triangle Rasterization based on Edge Equations
- Point Sampling using Incremental Calculations
- Culling Back-Facing Triangles
- Basic illumination and shading
- Texture mapping

Prerequisites

- Correct implementation of previous assignments.
- Triangle rasterization handout and presentation deck used in class lectures

Information

- Handout on triangle rasterization
- Handout on barycentric coordinates
- Handout on orthographic projection transform
- Handout on describing matrices in GLM and GLSL
- Presentation decks from class lectures

Getting Started

Download the new version of the batch file, csd2101.bat, from the assessment's web page and overwrite the existing one in the csd2101-opengl-dev directory. Execute the batch file and select option **8 - Create Assignment 2**. This will create a Visual Studio 2022 project named assignment-2.vcxproj in the ./build directory, with source files located in the ./projects/assignment-2 directory. This assignment will use new mesh models for the cube and

ogre, which include an additional normal attribute. These models can be found in the $./\text{meshes}$ directory as `cube_ptn.obj` and `ogre_ptn.obj`. The texture for these models is located in the `ogre.tex` file in the $./\text{images}$ directory.

While setting up Assignment 2, the project may prompt you to overwrite the DPML library. It is advisable to overwrite it with the newer version.

The directory structure for $\text{Assignment } 2$ is shown below:

```
csd2101-opengl-dev/      # OpenGL sandbox directory
├─ build/                  # Build files will exist here
├─ images/                 # Texture files
|   └─ ogre.tex
├─ meshes/                 # 3d Mesh models
|   └─ cube.obj
|   └─ ogre.obj
├─ projects/              # Tutorials and assignments
|  └─assignment-2          # Assignment 2 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
```

Transforms

Model Transform

A *model transform* instantiates an object in world frame from a geometric model described in its own [model] frame. The sample uniformly scales the model by 2 units along the coordinate axes. followed by a rotation about vertical $y-$axis followed by translation of $(0, 0, 0)$ [that is, there is no translation].

View Transform

A *view transform* is a change-of-frame transform that maps points from world frame to a viewing frame; the viewing frame is defined by a camera's position and orientation in the world frame. The sample executable positions and orients a camera in the world frame using parameters $eye = (0, 0, 10)$, $tgt = (0, 0, 0)$, and $\overrightarrow{up} = \langle 0, 1, 0 \rangle$. The view transform matrix can be constructed using the techniques described in class or using external function `glm::lookAt`.

Orthographic Projection Transform

An orthographic transform implements a parallel projection of a 3D object onto a view plane analogous to placing the viewer at an infinite distance with a telescope. Unlike perspective projection, orthographic projection don't cause closer objects to appear larger and distant objects to appear smaller. Since relative sizes and angles of objects are preserved, orthographic projection is preferred for architectural and engineering applications.

In the view frame, the viewer [or camera] is at origin and looking down $-z-$axis. In this frame, an orthographic projection is specified by giving six axis-aligned *clipping planes* which form a rectangular parallelepiped. These clipping planes are specified by six values l, r, b, t, n, and f which are mnemonics for *left*, *right*, *bottom*, *top*, *near*, and *far*, respectively. The plane located at $(0, 0, -n)$ that is defined by outward normal $\langle 0, 0, 1 \rangle$ and plane equation $z^v + n = 0$ is called the *near clipping plane*. The plane located at $(0, 0, -f)$ that is defined by outward normal $\langle 0, 0, -1 \rangle$ and plane equation $-z^v - f = 0$ is called the *far clipping plane*. Points with $z^v > -n$ and $z^v < -f$ are *culled*. The usual OpenGL convention is to have programmers specify n and f as ***positive*** values. The remaining four values l, r, b, t define a rectangular two-dimensional *viewfinder* [or a "window to the world"] on the near plane that determines the portion of the scene visible to the camera along x^v- and y^v-axes . The bottom-left and top-right corners of the viewfinder have coordinates (l, b) and (r, t), respectively. Points with $x^v < l$ and $x^v > r$ are culled. Likewise, points with $y^v < b$ and $y^v > t$ are also culled. Primitives that straddle the rectangular parallelepiped's six planes are clipped. After culling and clipping, the rectangular parallelepiped will then consist of points (x^v, y^v, z^v) such that

$$\begin{aligned} x^v &\in [l, r] \\ y^v &\in [b, t] \\ z^v &\in [-n, -f] \end{aligned} \tag{1}$$

The result of applying the orthographic projection on points inside this rectangular parallelepiped is to map them into the left-handed normalized device context (NDC) frame. NDC defines a $2 \times 2 \times 2$ cube centered at origin and containing points (x^n, y^n, z^n) such that

$$\begin{aligned} x^n &\in [-1, 1] \\ y^n &\in [-1, 1] \\ z^n &\in [-1, 1] \end{aligned} \tag{2}$$

The mapping of view frame point (x^v, y^v, z^v) to corresponding NDC point (x^n, y^n, z^n) is computed from Equations (1) and (2) and is implemented by 4×4 orthographic transform matrix:

$$\mathrm{M}_{ortho} = \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & -\frac{2}{f-n} & -\frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3}$$

The orthographic projection transform can be constructed using Equation (3) or by external function `glm::ortho`.

To generate symmetrical images, the sample uses b and t values of -1.5 and 1.5 values. To further ensure ratios of objects' widths and heights are equivalent to ratios of their projected images [that is, WYSIWYG)] the viewfinder's aspect ratio $\frac{r-l}{t-b}$ must be equivalent to viewport window's aspect ratio $ar = \frac{W}{H}$ where W and H are equivalent to `GLPbo::width` and `GLPbo::height`, respectively. To ensure this behavior, the sample uses $l = b \times ar$ and $r = t \times ar$ values, respectively.

The axis-aligned bounding box enclosing the model has approximate dimensions of $1.2 \times 1.2 \times 1.2$. The model's volume is amplified by incorporating an uniform scale of 2 units. To ensure projected $z-$values use the entire range of $32-$bit single-precision floating point values, the near and far planes are pushed as close as possible to the object. The sample uses n and f values of 8 and 12, respectively.

Viewport Transform

Viewport transform maps NDC point (x^n, y^n, z^n) to corresponding point (x^d, y^d, z^d) in window [or viewport] coordinates as preparation for rendering primitives into rectangular array of pixels. The x^d- and y^d-coordinates of a vertex determine its position in the window. The z^d-coordinate specifies a relative depth or distance value from the camera. The OpenGL pipeline uses command `glViewport` to map NDC x^n- and y^n-coordinates to window x^d- and y^d-coordinates, respectively. At the same time, the OpenGL pipeline maps NDC depth values $z^n \in [-1, 1]$ to range $z^d \in [0, 1]$. Programmers can choose other values for the interval using command `glDepthRange`. You'll then need to deduce a 4×4 viewport transform matrix that implements the following mapping of NDC frame points (x^n, y^n, z^n) to corresponding window frame points (x^d, y^d, z^d):

$$\begin{aligned} x^n \in [-1, 1] \quad &\mapsto \quad x^d \in [0, W] \\ y^n \in [-1, 1] \quad &\mapsto \quad y^d \in [0, H] \\ z^n \in [-1, 1] \quad &\mapsto \quad z^d \in [0, 1] \end{aligned} \tag{4}$$

where W and H are equivalent to `GLPbo::width` and `GLPbo::height`, respectively.

Chaining Transformations

View, orthographic projection, and viewport transform matrices must be computed *once for the entire scene*. Therefore, it is most efficient to concatenate these three matrices before processing individual objects within the scene. Meanwhile, a model is instantiated into a scene object using an unique model transform matrix per instantiation. The previously computed concatenation of view, orthographic projection, and viewport transform matrices is then concatenated with each object's individual model matrix to compute a single 4×4 matrix that directly transforms points from model frame to window frame.

Each vertex will have additional attributes such as normals [for evaluating lighting equations], texture coordinates [for indexing texture maps], and RGBA color values [from lighting equations evaluated by modeling packages]. Since the previously computed transformation matrix [that is a concatenation of model, view, orthographic project, and viewport transforms] has a certain geometrical meaning applicable only to a vertex's position coordinates. Hence, texture coordinates and RGBA color values must not be transformed by this matrix. Normals coordinates at each vertex are a different matter. When a vertex is transformed by model transformation matrix $\mathbf{M}$, the corresponding vertex normal must be transformed by a variation of matrix $\mathbf{M}$. This makes geometrical sense since the transformation of a triangle's vertices will change the orientation of the triangle's plane equation and therefore the normal vector defined at each triangle vertex. The specific variation of matrix $\mathbf{M}$ that must be applied to a vertex normal will be explained in the section on shading.

Things to Implement

The assignment requires six rasterization techniques to be implemented. You must provide functionality that allows the user to iterate through these techniques using keyboard button $\mathbf{W}$.

Keyboard button R should toggle the model's rotational transform about $y-$axis. When the model is rotating about $y-$axis [because button R was pressed], button X should change the rotation axis to $\langle 1, 1, 0 \rangle$ while button Z should change the rotation axis to $\langle 1, 1, 1 \rangle$.

The behavior of keyboard button L is described here.

Wireframe Images

Rasterize outlines of front-facing triangles using Bresenham line algorithm.

Depth Buffer Algorithm

Rasterizers render objects and their primitives in the order they're encountered. This means the rasterizer has no concept of a scene or even an object within a scene - it is only knows about the current primitive being rasterized. Therefore, it is possible for the rasterizer to overwrite primitives closer to the viewpoint with occluded or hidden primitives farther away. OpenGL solves this *hidden surface removal* or *visible surface determination* problem with a ***depth buffer*** that holds the distance or depth value [from the view point] for each pixel. The depth buffer lets OpenGL do hidden surface computations by the simple expedient of drawing a fragment (x^d, y^d) into a color buffer pixel only if the fragment's depth z^d is *less than* the depth of the closest pixel encountered so far. Since a rasterizer only has knowledge of the current primitive being rendered, it uses the depth buffer data structure to keep track of the closest depths encountered by the rasterizer as it rasterizes the entire set of triangles in the scene.

Creating a Depth Buffer

The depth buffer is not activated by default. To enable the use of the depth buffer in an OpenGL application, you must have an OpenGL rendering context with a depth buffer. This is automatically done by GLFW when initializing your OpenGL graphics window and rendering context with the command `glfwCreateWindow`. Likewise, the software pipe must dynamically allocate a depth buffer in CPU memory with the same shape and size as the color buffer with each element of the buffer able to record a single-precision floating-point value.

Using the Depth Buffer

Just as the color buffer is cleared to the background color at the top of each frame, the depth buffer must be cleared to the largest possible depth value of 1.0. Why 1.0? See the discussion on viewport transform. As each triangle primitive is rasterized, the depth value, z^d, at each fragment (x^d, y^d) is computed through barycentric interpolation. If z^d is less than the depth value stored at the corresponding depth buffer location, it means the current fragment is the closest point in the scene encountered so far. In this case, the fragment is converted into a pixel by writing the fragment's color into the corresponding color buffer location and by writing the fragment's depth into the corresponding depth buffer location. On the other hand, if z^d is greater than the depth value stored at the corresponding depth buffer location, the current fragment is being occluded by another previously rendered fragment. In this case, the current fragment is invisible and therefore discarded from further consideration.

Note that depth buffering is not required when rendering a stand-alone solid object such as a platonic solid. However, rendering a stand-alone object such as the ogre [that has been rendered in previous assignments] will require depth buffer.

Generating Depth [or Shadow] Maps

Triangles map to triangles under orthographic projection. Use barycentric coordinates to interpolate depth coordinates, $z_i^d, i = 0, 1, 2$, at triangle vertices across the triangle surface. If fragment (x^d, y^d) is visible, use depth value, $z^d \in [0, 1]$, at this fragment to create RGB grayscale color (z^d, z^d, z^d) to write to the corresponding colorbuffer location.

Faceted Shading

The Lighting Equation

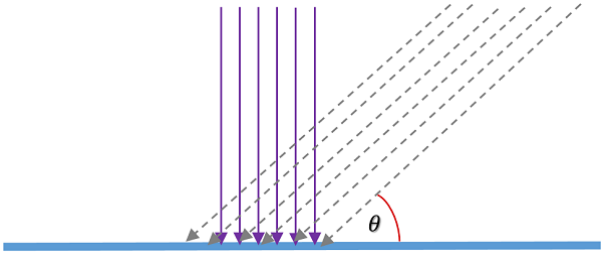
For a surface to be visible to the eye, there must be light that is reflected from, emitted from, or transmitted through that surface. In other words, the visible qualities of a surface can be defined solely by the way that the surface interacts with or emits light. The contribution from the light that goes directly from a light source and is reflected from the surface is called a *local illumination model*. The terms *reflection model*, *reflectance model*, and *lighting model* are commonly used substitutes for the term illumination model. Using the simplifying assumption that surfaces don't emit light, the visual appearance of any opaque surface can be described by a greatly simplified illumination model as

$$\text{Outgoing light} = \text{Reflectance Function} \times \text{Incoming Light} \tag{5}$$

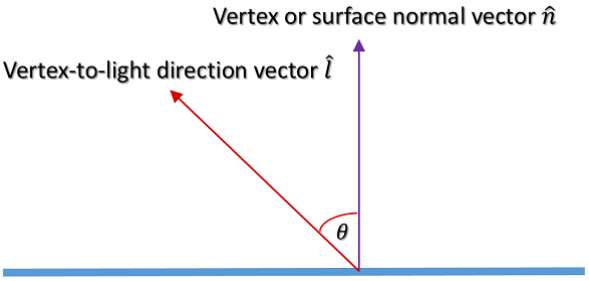
Equation (5) shows that the amount of light reflected from a surface is a function of the amount of incoming light energy incident upon the surface. The purpose of $\text{Reflectance Function}$ in Equation (5) is to modulate the incoming light in a manner that reproduces the surface's visual qualities. This reproduction can be based heavily on realistic physics [for example, bidirectional reflectance distribution function] or on ad hoc techniques that are physically motivated but not physically accurate [such as Phong reflection model]. We'll be using a highly simplified ad hoc technique based on Lambert's law for illumination which is also known as Lambert's cosine law.

Lambert's Law for Illumination

Incoming light energy is modeled on Lambert's cosine law and is based on the idea that light striking a surface point head-on illuminates the surface more and reflects more light compared to light striking at a glancing angle. In the following figure, we have two bundles of light rays coming to the surface, one bundle perpendicular to the surface [shaded dark] and one in an angle θ [shaded gray]. Assume the amount of energy in both bundles remains constant [we shade these bundles with dark and gray colors only to distinguish them in the picture]. Denoting a unit area by the width of the dark ray bundle, the dark bundle illuminates the unit area. However, the gray bundle comes in at an angle θ, and thus illuminates a larger surface area [larger by factor $\frac{1}{\cos\theta}$]. By increasing the affected area, the gray bundle decreases the amount of energy per unit of surface area. The law of conservation of energy says this must be the case. When angle θ decreases to 0°, the light energy in the bundle would travel past the surface, never hitting it. Of course, this is only the case for an ideal surface. A non-ideal surface has a lot of variations, so small details will inevitably receive and reflect a portion of the light energy.



The amount of incoming light energy that each unit area receives follows from the geometry shown in the following figure: the amount of incoming light energy that each unit area receives is a function of the incoming energy and the cosine of the angle between the surface normal vector and the vector from the surface point to the light source. This is referred to as ***Lambert's cosine law***.



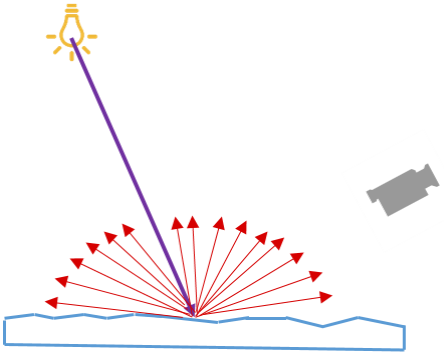
By denoting the surface normal vector by $\hat{n}$ and the direction vector to light source by $\hat{l}$, we see that $cos\theta = \hat{n} \bullet \hat{l}$. The above figure implies that the amount of incoming illumination must be clamped to positive values between 0 and 1. When the cosine of the angle between the two vectors is less than 0, the light source is behind the point on the surface, so the amount of light must be 0. If you forget to clamp the value, lights positioned behind a surface point will effectively remove energy from the surface. This is obviously incorrect. If the intensity or energy of the light source L is I_L, then the energy of the incoming light is

$$\text{Incoming Light } = \max(0, \hat{n} \bullet \hat{l}) I_L \tag{6}$$

The light source energy or intensity, I_L, in the equation is a color value that is encoded as three normalized RGB values, each value between 0 and 1.

Diffuse Reflectance

Diffuse reflectance is used to model fine-scale graininess in surfaces in which small, rough surface features scatter light evenly in all directions. Light rays from a light source travel toward a surface and then are scattered equally in all directions by the diffuse reflectance from the surface. A surface that is a perfectly diffuse reflector is called a ***Lambertian reflector***. The following figure provides an example for a Lambertian reflector that scatters light equally in all directions.



For example, a material such as matte paint will appear equally bright no matter what angle you view it from. The cosine law says that the total amount of illumination is dependent on the angle between the light direction and the surface normal, but for matte paint, the amount of reflected light the viewer sees is nearly the same from any angle. This is because the paint is a very rough surface. When light hits the paint, it is scattered in many directions. If you change your viewpoint, you won't see much variation in brightness. On average, it appears as though the same amount of light is scattered in every direction.

The color we perceive from objects is mostly diffuse reflection - it is diffusely scattered light that forms an object's image in an observer's eye. The reflectance function for a diffuse material then is simply a diffuse color D that determines the color of a surface perceived by an observer:

$$\text{Reflectance Function } = D = (D_{red}, D_{green}, D_{blue}) \tag{7}$$

For example, by setting diffuse color D to $(1, 0, 0)$, only red component of the light will be reflected so that the surface is perceived to have color red.

Calculating Outgoing Light

All things considered, combining Equations (6) and (7) into Equation (5), the reflected light for a Lambertian surface is simply the illumination value given by the cosine law multiplied by the diffuse color:

$$\text{Outgoing Light } = D \otimes I_L \max(0, \hat{n} \bullet \hat{l}) \tag{8}$$

In Equation (8), expression $D \otimes I_L$ means component-wise multiplication of colors in D and I_L. Suppose a light source L emits light with intensity $I_L = (0.3, 1.0, 1.0)$ and a point on a surface has diffuse color $D = (0.8, 1.0, 0.8)$. If the angle between surface normal $\hat{n}$ and point-to-light source vector $\hat{l}$ is 60°, the intensity of light reflected from the point is $(0.3 \times 0.8, 1.0 \times 1.0, 1.0 \times 0.8)0.5 = (0.12, 0.5, 0.4)$.

Point Lights

Legacy OpenGL defined several simple light sources: the global ambient light, point lights, directional lights, and spot lights. Lights have intensity values in the form of RGB values with each component specifying the energy radiated by the light source at that particular wavelength. OpenGL also stores alpha component values for reflectances and intensities, though they aren't really used in the lighting computation. They are stored largely to keep the application programming interface simple and regular, and perhaps so they are available in case there's a use for them in the future.

A point light source is a single point in space that radiates energy uniformly in all directions. Because a point light is located in the scene at a certain location and has an intensity specified by RGB color, it can be represented using the following structure:

```
struct PointLight {
  glm::vec3 intensity; // we choose to not store the alpha component
  glm::vec3 position;
};
```

The sample uses a point light located at $eye = (0, 0, 10)$ with intensity $(1, 1, 1)$. Keyboard button L should toggle rotational transform of the point light source about $y-$axis.

Faceted Shading

In the previous sections, we've used a local illumination model to compute the color of a point on a surface from light emitted by a point light source. ***Shading*** is the term used to describe the assignment of a color value to a pixel. The simplest and fastest approach to determining a triangle's color is to choose the coloring at one point on the triangle and use it for the entire triangle; this is called ***faceted shading***. Other terms used to describe faceted shading including *constant shading* or *flat shading*. To implement faceted shading, apply these steps for each triangle:

1. For front-facing triangles [why do unnecessary work for back-facing triangles that will be discarded?], compute the triangle's outward normal. You can perform this computation in either model coordinates [using the vertices buffered in GPU memory] or world coordinates [by applying the model transformation matrix on buffered vertices to transform them into world frame] or view coordinates [by applying the model-view transformation matrix on buffered vertices to transform them into view frame].
2. Compute shading color by evaluating the illumination model [see Equation (8)] at the triangle's centroid. Ensure vectors $\hat{n}$ and $\hat{l}$ in Equation (8) are defined in the same coordinate system. Evaluating the illumination model in model frame provides the greatest efficiency since only the point light's position must be [inverse] transformed [once per object] from world frame to model frame rather than transforming a large count of triangle vertices from model frame to either world frame or view frame.
3. Rasterize the triangle and assign each interior fragment the previously computed shading color.

Smooth [or Gouraud] Shading

The basic idea behind ***smooth shading*** is to interpolate the colors at each vertex to produce the fragment color. This results in smooth transitions between polygons of different colors. If vertex lighting is combined with smooth shading, then the triangles are ***Gouraud shaded***. Gouraud shading aims to eliminate the sharp discontinuities in color at triangle edges caused by faceted shading. The color of the reflected light at a triangle vertex is computed by Equation (8) and is therefore dependent on the normal $\hat{n}$ at that vertex. If this vertex normal attempts to approximate the normal that the untriangulated surface would have exhibited at that point, then it would produce a shading that is continuous in value. Function `dpml::parse_obj_mesh` returns normalized vertex normals that are obtained by averaging the normals to the triangles that share the vertex.

To implement smooth [or Gouraud] shading, apply these steps for each triangle:

1. For front-facing triangles, compute a shading color at each vertex by applying the illumination model at that triangle vertex. The illumination model can be evaluated in either model or world or viewing coordinates. Evaluating the illumination model in model frame provides the greatest efficiency since only the light's position must be [inverse] transformed [once per object] from world frame to model frame in contrast to transforming the large count of model vertices from model frame to either world frame or view frame. Further, when lighting is computed in world or viewing coordinates, vertex normals must be transformed from model frame using the *inverse-transpose* of the transformation matrix that is applied to vertex position coordinates to transform them from model frame to either world or view frame. More details are provided here.
2. Using barycentric interpolation on the shading colors computed at each vertex, determine the shading color at an interior fragment.

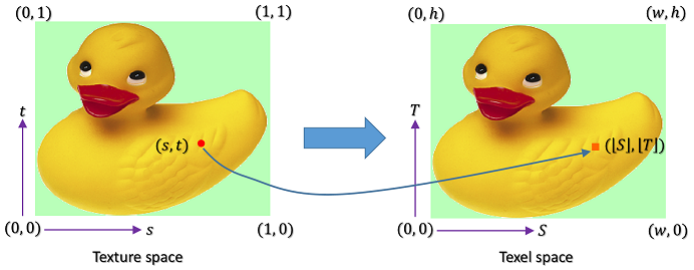
Texture Mapping

First, texture images stored in $./\text{images}$ must be loaded into client memory. These texture images have been parsed from their original image formats into the tex format that has a 12 byte header consisting of three `int` elements that provide the images' width, height, and bytes per texel, respectively. The rest of the file contains the image stored in OpenGL's texel space.

Texture coordinates returned by the DPML OBJ file parser are described in a normalized coordinate system called ***texture space***. It is these values that must be interpolated across a triangle's surface. The discrete color samples stored in a texture image are stored in a discrete coordinate system called ***texel space***. During initialization, the application must save the width, say w, and height, say h of the texture image that is to be mapped to the geometric object being rendered. After interpolating texture coordinates (s, t) at fragment P^d, the corresponding integral texel coordinates (S, T) are computed from floating-point texture coordinates as follows:

$$\begin{aligned} S &= \lfloor w \times s \rfloor \\ T &= \lfloor h \times t \rfloor \end{aligned} \tag{9}$$

The following figure illustrates the mapping of texture coordinates to texel coordinates. These texel coordinates are used to extract the color sample from the texture image. Since we use the `floor` function, the texel with coordinates containing the mapping of the fragment center is accessed even if the fragment center maps closer to other adjacent fragments. This can cause aliasing artifacts because we're using only a single sample point in the fragment [its center], rather than being representative of all the features within a fragment. However, point sampling requires less computation than using other techniques that require accessing multiple texels and then averaging their colors.



Encapsulating the above discussion, apply the following steps to incorporate texture mapping:

1. During application initialization, load texture images into CPU memory. Cache the width and height attributes of each texture image.
2. For each front-facing triangle, interpolate texture coordinates specified as per-vertex attributes.
3. At each fragment, map interpolated texture coordinates to texel coordinates and read the color from the associated texture image.
4. Use the texel color to update the colorbuffer.

Texture Mapping with Faceted Shading

At each fragment, colors from texturing and faceted shading are modulated [this means component-wise multiplication of colors].

Texture Mapping with Smooth Shading

At each fragment, colors from texturing and smooth shading are modulated.

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-2 named <login>-<assignment-2>. That is, if your Moodle student login is foo, then the directory should be named foo-assignment-2. Ensure that directory foo-assignment-2 has the following layout:

```
foo-assignment-2        # You're submitting Assignment 2
├─ 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-2.zip.

⚠ Before Submission: Verify and Test ⚠

- Copy archive file foo-assignment-2.zip into directory test-submissions. Unzip the archive file to create project directory foo-assignment-2 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-2.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-2  # foo is submitting Assignment 2
|     └─ 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-2 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-2.
- Build and execute project foo-assignment-2 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\foo-assignment-2.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	☐

Things to test before submission	Status
Executable generated and successfully launched in Debug and Release mode	☐
Directory is zipped using the naming conventions outlined in submission guidelines	☐
Zipped file is uploaded to assessment submission page	☐

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

Grading Rubrics

The core competencies assessed for this assessment are:

- [***core1***] Submitted code must build an executable file with ***zero*** warnings. This rubric is not satisfied if the generated executable crashes or displays nothing. In other words, if your source code doesn't compile nor link nor execute, there is nothing to grade.
- [***core2***] This competency indicates whether you're striving to be a professional software engineer, that is, are you implementing software that satisfies project requirements [are you submitting the required files? are they properly named? are they properly archived? are you submitting unnecessary files?], that is maintainable and easy to debug by you and others [are file and function headers properly annotated?]. Submitted source code must satisfy ***all*** requirements listed below. Any missing requirement will decrease your grade by one letter grade.
    - Source code must compile with ***zero*** warnings. Pay attention to all warnings generated by the compiler and fix them.
    - Source code 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***] Depth mapped triangle images are correctly generated by interpolating vertex depth values across triangle surface.
- [***core5***] Faceted triangle images are correctly generated.
- [***core6***] Smooth shaded triangle images are correctly generated.

- [***core7***] Texture mapped triangles images are correctly generated.
- [***core8***] Color from texture map and color computed by evaluating illumination model at triangle centroid are modulated to compute color at each pixel.
- [***core9***] Color from texture map and smooth shaded color are modulated to compute color at each pixel.
- [***core10***] After file documentation block in glpbo.cpp, ***define and initialize*** `const` variables in a namespace `CORE10` that will allow the grader to change the following parameters: ***model file name*** [that will be located in ./meshes]; ***texture file name*** [that will be located in ./images]; ***camera position and target*** [both must be in world frame]; ***six clipping plane*** values l, r, b, t, n, and f that specify orthographic projection matrix; ***light position*** [must be in world frame] and ***light intensity***.

  > ***To make `const` variables described in this rubric be easily identifiable and to make it easier for the grader to alter values of these parameters, they must be defined and initialized in namespace `CORE10` which should be located right after the file header!!!***
  >
  > ***If your submission doesn't provide the information exactly in the manner described above, your submission will be heavily penalized [see below]. You're forewarned!!!***

- [***core11a***] Keyboard button M should toggle between the ogre and cube models.
- [***core11b***] Keyboard button W allows iteration through wireframe, depth shading, faceted shading, smooth shading, texture mapped, texture mapped with faceted shading, and texture mapped with smooth shading rendering modes. Keyboard button R should toggle rotational transform of the model about $y-$axis.
- [***core11c***] Keyboard button L should toggle rotational transform of the light source about $y-$axis.
- [***core11d***] When the model is rotating about $y-$axis [because button R was pressed], button X should change the rotation axis to $\langle 1, 1, 0 \rangle$ while button Z should change the rotation axis to $\langle 1, 1, 1 \rangle$.
- [***core11e***] Information should be printed to title bar - see the sample for specific information and order in which this information must be presented. main-pbo.cpp will not make any calls to print this information to the title bar. Instead, your code should ensure that this rubric is satisfied.

  > ***BONUS points will not be awarded unless the core rubrics are successfully implemented. Don't disregard this notice and implement the bonus before successfully implementing the basic rubrics. You're forewarned!!!***

- **BONUS** After completing previous rubrics, can you implement the depth buffer algorithm to solve the hidden surface problem for multiple objects? Your submission should illustrate the utility of the algorithm to generate correct images in a scene consisting of multiple objects, irrespective of the objects' distances from viewer and the objects' draw orders. This is how you should implement this rubric: Define a scene with two *different* objects labeled *obj1* and *obj2* such that *obj1* is located between the camera and *obj2*. If *obj1* is rendered after *obj2*, a hidden surface algorithm is unnecessary to generate the correct image [because the rasterizer will overwrite *obj2*'s colors in the colorbuffer with *obj1*'s colors]. If *obj2* is rendered after *obj1*, the algorithm will prevent points on *obj2* occluded by *obj1* from overwriting corresponding locations in the colorbuffer with *obj2* colors. In addition to changing draw orders of the two objects, the grader must be able to change the relative positions of *obj1* and *obj2* with respect to the camera. Thus, in source file glpbo.cpp you must declare a

namespace `BONUS` in which you must provide definitions of static variables that allow the grader to change positions and draw orders of *obj1* and *obj2*. Further, your submission must use W keyboard button to cycle thro' to this final rendering mode and provide appropriate title bar text indicating implementation of bonus task.

Namespace `BONUS` must immediately follow namespace `CORE10`!!! Recall that namespace `CORE10` must immediately follow the file header. If the grader is unable to change both the draw orders and positions of the two objects, it will not be possible to assign grades for this rubric.

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 letter. This means grade A will be recorded as B, an $A-$ would be recorded as $B-$, and so on.	
core3 rubric is complete.	D
core4 rubric is complete. Use of edge equations and top-left tie-breaking rule is mandatory. Dropouts and flickers will result in deductions.	$D+$
core5 rubric is complete.	C
core6 rubric is complete.	$C+$
core7 rubric is complete.	B
core8 rubric is complete.	$B+$
core9 rubric is complete.	A
If ***core10*** rubric is not satisfied, final grade will be capped to C.	
If ***core11b*** or ***core11c*** or ***core11e*** rubrics are not satisfied, it is not possible to evaluate your submission.	F
If ***core11a*** or ***core11d*** rubric not satisfied, grade will be decreased by one letter.	
BONUS task is complete.	$A+$

Inverse-Transpose Matrix for Planes and Normals

Why should you transform a model's vertex normals with a matrix that is the inverse-transpose of the matrix applied to vertex positions? We provide an answer to this question by answering a related question: if a triangle's vertices are transformed by a 4×4 affine transformation matrix M to generate a new triangle, what transformation matrix must be applied on the plane equation of the original triangle to compute the new triangle's plane equation?

Consider a triangle $\mathcal{T}$ with counter-clockwise oriented vertices $P_i, i = 0, 1, 2$. Outward normal $\hat{n}$ to the plane spanned by $\mathcal{T}$ is computed like this:

$$\begin{aligned}\overrightarrow{u} &= P_1 - P_0 = \langle u_x, u_y, u_z \rangle \\ \overrightarrow{v} &= P_2 - P_0 = \langle v_x, v_y, v_z \rangle \\ \hat{n} &= \frac{\hat{u} \times \hat{v}}{\left\| \hat{u} \times \hat{v} \right\|} = \langle n_x, n_y, n_z \rangle\end{aligned}$$

Suppose X is an arbitrary point on the plane spanned by $\mathcal{T}$. By definition, vector $X - P_0$ is orthogonal to $\mathcal{T}$'s outward normal $\hat{n}$:

$$\hat{n} \cdot (X - P_0) = 0$$

$$\implies \hat{n} \cdot X - \hat{n} \cdot P_0 = 0 \tag{1}$$

If $\hat{n} = \langle n_x, n_y, n_z \rangle$ and $X = (x, y, z)$, dot product $\hat{n} \cdot X$ in equation (1) can be expanded as:

$$n_x x + n_y y + n_z z - \hat{n} \cdot P_0 = 0 \tag{2}$$

Constants n_x, n_y, n_z, and $-\hat{n} \cdot P_0$ of plane equation $\mathcal{L}$ can be expressed using a 4×1 column matrix

$$\mathcal{L} = \begin{bmatrix} n_x \\ n_y \\ n_z \\ -\hat{n} \cdot P_0 \end{bmatrix}$$

and equation (2) can be compactly written as a dot product or as a matrix multiplication operation:

$$\mathcal{L} \cdot X = 0$$

$$\implies \mathcal{L}^{\mathrm{T}} \circ X = 0$$

Since $\mathcal{L}$ is the plane equation for triangle $\mathcal{T}$ [which is defined by vertices P_0, P_1, and P_2]:

$$\begin{gathered}\mathcal{L} \cdot P_i = 0 \ \ \forall \ \ i = 0, 1, 2 \\ \implies \mathcal{L}^{\mathrm{T}} \circ P_i = 0 \ \ \forall \ \ i = 0, 1, 2\end{gathered} \tag{3}$$

Suppose 4×4 affine transformation matrix M transforms vertices $P_i, i = 0, 1, 2$ of $\mathcal{T}$ to vertices $P_i^{'}, i = 0, 1, 2$ that define a new triangle $\mathcal{T}^{'}$:

$$\mathrm{M} \circ P_i = P_i^{'} \ \ \forall \ \ i = 0, 1, 2$$

Similar to the explicit computation of plane equation $\mathcal{L}$ using equation (1), plane equation $\mathcal{L}^{'}$ can be explicitly computed for triangle $\mathcal{T}^{'}$, so that

$$\mathcal{L}^{'} \cdot P_i^{'} = 0 \ \ \forall \ \ i = 0, 1, 2 \tag{4}$$

Rather than reevaluating plane equation $\mathcal{L}^{'}$ from transformed vertices $P_i^{'}, i = 0, 1, 2$, we'd like to compute $\mathcal{L}^{'}$ by applying [as yet unknown] 4×4 transformation matrix D on $\mathcal{L}$:

$$\mathrm{D} \circ \mathcal{L} = \mathcal{L}^{'}$$

Rewriting equation (4) in terms of the matrix transforms applied on $\mathcal{L}$ and P_i:

$$\mathcal{L}^{'} \cdot P_i^{'} = 0 \quad \forall \ \ i = 0, 1, 2$$

$$\implies (\mathrm{D} \circ \mathcal{L}) \cdot (\mathrm{M} \circ P_i) = 0$$

Rewriting dot product expression as matrix multiplication

$$(\mathrm{D} \circ \mathcal{L}) \cdot (\mathrm{M} \circ P_i) = 0$$

$$\implies (\mathrm{D} \circ \mathcal{L})^{\mathrm{T}} \circ (\mathrm{M} \circ P_i) = 0$$

$$\implies \left(\mathcal{L}^{\mathrm{T}} \circ \mathrm{D}^{\mathrm{T}}\right) \circ (\mathrm{M} \circ P_i) = 0$$

$$\implies \left(\mathcal{L}^{\mathrm{T}} \circ \mathrm{D}^{\mathrm{T}} \circ \mathrm{M}\right) \circ P_i = 0 \tag{5}$$

Comparing equations (3) and (5), we conclude:

$$\mathrm{D}^{\mathrm{T}} \circ \mathrm{M} = \mathrm{I}_{4 \times 4} \qquad [\mathrm{I}_{4 \times 4} \text{ is identity matrix}]$$

$$\implies \mathrm{D}^{\mathrm{T}} = \mathrm{M}^{-1}$$

$$\implies \mathrm{D} = \left(\mathrm{M}^{-1}\right)^{\mathrm{T}} \tag{6}$$

Equation (6) informs us that if a triangle's vertices are transformed by 4×4 affine transformation matrix M to generate a new triangle, then the original triangle's plane equation can be transformed by 4×4 matrix $\mathrm{D} = \left(\mathrm{M}^{-1}\right)^{\mathrm{T}}$ to compute the new triangle's plane equation. However, note that if the original plane equation specifies a normal vector with unit magnitude, it is not guaranteed that the transformed plane equation will specify a normal vector that also has unit magnitude.

For lighting computations at a vertex, we're concerned only with the vertex normal vector rather than the plane equation defined by the vertex position and normal vector. How can we use the result from equation (6) to transform vertex normal vectors? Let M denote a matrix transforming a model, that is, all of the model's vertex positions are transformed by M.

If M is a translation transform, should vertex normals be transformed? The answer is no since vectors are defined as being independent of position [as they only define a direction] and so only translating vertex positions should not affect the normals at these positions. Therefore, the lighting equation can directly use the original, buffered vertex normals. This can be shown formally for a vector $\vec{w}$ from point P to point Q:

$$
\begin{aligned}
\mathrm{M}\left(\vec{w}\right) &= \mathrm{M}\left(Q - P\right) && [\text{ by definition of } \vec{w}] \\
&= \mathrm{M}\left(Q\right) - \mathrm{M}\left(P\right) && [\text{ since M is an affine transformation}] \\
&= \left(P + \vec{w}\right) - P && [\text{ applying definition of translation}] \\
&= \left(P - P\right) + \vec{w} && [\text{ rearranging terms}] \\
&= \vec{0} + \vec{w} && [\text{ because } P = P + \vec{0}] \\
&= \vec{w} && [\text{ because } \vec{0} \text{ is additive identity for vectors}]
\end{aligned}
$$

If M is a rotation transform, it is obvious that vertex normals should be transformed. Since matrix M applied on vertex positions is an orthogonal matrix, it can be directly applied on vertex normals since $\mathrm{D} = \left(\mathrm{M}^{-1}\right)^{\mathrm{T}} = \left(\mathrm{M}^{\mathrm{T}}\right)^{\mathrm{T}} = \mathrm{M}$. Further, since determinants of orthogonal matrices evaluate to 1, that is, $||\mathrm{M}|| = 1$, if the original, buffered vertex normals are normalized, then their transformed counterparts will remain normalized. However, if buffered vertex normals are not normalized, their transformed counterparts will have to be renormalized using expensive square root evaluations.

If M is an uniform scale transform, should vertex normals be transformed? If the individual coordinates of a vertex normal $\hat{n} = \langle n_x, n_y, n_z \rangle$ are equally scaled by uniform scale factor s to $s\hat{n} = \langle sn_x, sn_y, sn_z \rangle$, normalizing vector $s\hat{n}$ produces the original normalized vector $\hat{n}$. This means that if it is known that M is only a uniform scale transform, then we can optimize away the transformation of vertex normals followed by their normalization. Instead, the lighting equation can directly use buffered vertex normals.

If M is a non-uniform scale transform, vertex normals should be transformed. Since scale matrices are diagonal matrices, $\mathrm{D} = \left(\mathrm{M}^{-1}\right)^{\mathrm{T}} = \mathrm{M}^{-1}$, only the inverse of M is computed and applied on vertex normals. However, since individual coordinates of a vertex normal $\hat{n} = \langle n_x, n_y, n_z \rangle$ are transformed by non-uniform scale factors $\langle s_x, s_y, s_z \rangle$ to $\overrightarrow{n'} = \langle s_x n_x, s_y n_y, s_z n_z \rangle$, the transformed vertex normals must be renormalized using expensive square root calculations.

What is to be done when the programmer is unaware or is unable to determine the specific nature of transform [translation or rotation or uniform scale or non-uniform scale] embedded in arbitrary 4×4 matrix M? Since translation transforms don't affect vertex normals, the starting 4×4 matrix

$$
\mathrm{M} = \begin{bmatrix}
m_{00} & m_{01} & m_{02} & m_{03} \\
m_{10} & m_{11} & m_{12} & m_{13} \\
m_{20} & m_{21} & m_{22} & m_{23} \\
m_{30} & m_{31} & m_{32} & m_{33}
\end{bmatrix}
$$

must be reduced to a 3×3 matrix consisting of only the linear part of the affine transform. This requires the extraction of the upper-left 3×3 sub-matrix from M:

$$
\mathrm{N} = \begin{bmatrix}
m_{00} & m_{01} & m_{02} \\
m_{10} & m_{11} & m_{12} \\
m_{20} & m_{21} & m_{22}
\end{bmatrix}
$$

Next, the inverse of this 3×3 matrix is computed:

$$
\mathrm{N}^{-1} = \begin{bmatrix}
m_{00}^{'} & m_{01}^{'} & m_{02}^{'} \\
m_{10}^{'} & m_{11}^{'} & m_{12}^{'} \\
m_{20}^{'} & m_{21}^{'} & m_{22}^{'}
\end{bmatrix}
$$

Finally, the transpose of this inverse matrix is computed:

$$\mathrm{D} = \left(\mathrm{N}^{-1}\right)^{\mathrm{T}} = \begin{bmatrix} m_{00}^{'} & m_{10}^{'} & m_{20}^{'} \\ m_{01}^{'} & m_{11}^{'} & m_{21}^{'} \\ m_{02}^{'} & m_{12}^{'} & m_{22}^{'} \end{bmatrix}$$

The matrix D is then used to transform each vertex normal $\hat{n}$ to arbitrary vector $\overrightarrow{n^{'}}$

$$\mathrm{D}\ \hat{n} = \overrightarrow{n^{'}}$$

which must then be normalized to have unit magnitude

$$\hat{n^{'}} = \frac{\overrightarrow{n^{'}}}{\|\vec{n^{'}}\|}$$

The transformed and renormalized vector $\hat{n^{'}}$ is then used in the lighting equation.

A simple way to minimize the work of transforming vertex normals and potentially renormalizing them using expensive square root instructions is to normalize vertex normals offline so that buffered vertex normals are always normalized. Next, the light's position is transformed from world frame to the object's model frame by applying the inverse of the object's model-to-world transformation matrix. The illumination model is evaluated in the model frame and the computed color values can then be directly used by the rasterizer.