Homework 1: Basic Rasterization Pipeline

By: Yuqi Meng and Junwen Yu Instructor: Prof. Jeong Joon Park

EECS 498-014 Graphics and Generative Models

2024 Fall

This homework is divided into three part, where we will gradually introduce the infrastructure and utilize the tools covered in lectures to build a simple rasterizer which can correctly handle occlusion and Blinn-Phong shading. You can retrieve the starter code here. In the codebase, for functions requiring implementation, an explanation of the parameters will be presented in this specification in the following section; and a brief explanation is also included in the declaration of the functions. A "`// TODO`" tag will be present before the implementation of the function. Some code necessary for logic or data retrieval will also be provided for your convenience. Note that **all your changes should be made to `rasterizer_impl.cpp`**. When grading your work, we will substitute all other files with the standard infrastructure, so the changes you've made elsewhere may get lost, and thereby influencing the outcome.

In each section, we will first give a walkthrough of the infrastructure needed for implementation; and then present the task that you are required to complete.

> **Note.** A "Note" box like this will be presented in each part, before the actual task is given. These are details prone to error, and may be helpful when debugging.

Some information are provided in the footnotes, which mainly constitutes reference materials, or elaborations on the subject. Refer to them if you stuck in understanding the content.

If you prefer looking into the code yourself and want to see the task directly, you are welcome to skip all previous parts. The requirements of the assignments are specified in the section starting with "*Task:*", appearing at the end of each section. Do look back to the note boxes if you encounter any problems.

Contents

1 Rendering 2D Triangle

1.1 Testfile: YAML and Wavefront Obj Format

In order to disentangle the tests from the renderer, we present testcases using YAML[1] for configuration, and Wavefront Obj[2] for actual mesh. Links to the tutorials are provided in the footnotes. Obj file is parsed as is; and the following introduces how YAML is used for testing, so that you can write your own testcases for examining how well your rasterizer is working if you wish. The YAML configuration will be gradually built up as more contents are introduced in the following sections. You can find the additional parts in subsections whose title starts with "*Testfile:*".

The default name of the YAML file is `config.yaml`. If you want to use a different name, you can specify it as an argument when running the program, for example `./rasterizer myconfig` for filename `myconfig.yaml`. Notice that the `.yaml` suffix is omitted in the argument.

> **Note.** YAML, like Python, uses indentation to indicate nesting. Therefore, if you encounter error messages starting with "missing tag" then it is very likely that there is something wrong with the indentation. All testcases existing in this document can be found under `sample-tests` directory. Do remember to change the file name to `config.yaml`, or add the argument `<filename>` when running the program, after copying the YAML files to the same directory as the source code files.

In all subsequent explanations of testfiles, the added parts will be highlighted with purple; and parts that have already been introduced will be highlighted in brown.

A typical YAML config for testcases of this part looks like this:

```yaml
task: triangle
antialias: SSAA
samples: 16
resolution:
    width: 800
    height: 800
obj: trig
output: output
```

The configurations are interpreted as follows:

- `task` must be set to `triangle`, distinguishing this from tasks in subsequent sections.

[1] A simple tutorial is learn yaml in y minutes. If you are interested, complete specifications can be found here. We are only using very basic functionalities.

[2] Wikipedia provides great explanation of the structure of Obj files. Similarly we are also using only very basic functionalities, with no materials involved. For self-testing, you can either create them by hand (which may be a bit tedious), or exporting them using modeling softwares (for example, refer to this article for details on exporting to Obj files in Blender).

- `antialias` must be either `none` or `SSAA`, specifying whether your rasterizer needs to perform Super Sampling Anti-aliasing.
- `samples` takes a positive integer value, specifying the number of samples you need to take for SSAA. If `antialias` is set to `none`, the value of `samples` is simply discarded.
- `resolution.width` and `resolution.height` specifies the width and height of the output file.
- `obj` specifies the Obj file from which the models (here information of triangles) are read. The `.obj` suffix is omitted.
- `output` specifies the name of the PNG file that is outputed. Similarly the `.png` suffix is omitted.

To make the `.obj` file more readable in modelling software (e.g. Blender), the infrastructure makes a transformation on the input values. Treating position vectors as a 4-dimensional vector $(x, y, z, 1)^T$ (i.e. using homogeneous coordinates), for output file with width w and height h the positions given in the `.obj` file are transformed via

$$\begin{pmatrix} \frac{1}{2}w & 0 & 0 & \frac{1}{2}w \\ 0 & \frac{1}{2}h & 0 & \frac{1}{2}h \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} \frac{1}{2}w(1+x) \\ \frac{1}{2}h(1+y) \\ 0 \\ 1 \end{pmatrix} =: \begin{pmatrix} x' \\ y' \\ 0 \\ 1 \end{pmatrix} \tag{1}$$

where (x', y') is the 2D coordinate (in pixels) you will be handling in your work.

1.2 A Brief Walkthrough of the Codebase

Here we give a brief walkthrough of the codebase, in which you will implement the algorithms. The directory is structured as follows:

```
rasterizer
├── entities.hpp ..... Primitive infos: Triangle, MeshTransform, Light and Camera
├── image.hpp/cpp ..... Image I/O: Color and ImageBuffer<>, specialized to Image and ImageGrey
├── loader.hpp/cpp ..... Loading: Loader
├── rasterizer.hpp/cpp ..... Declaration of render processes
├── rasterizer_impl.cpp ..... Your implementation here
├── renderer.hpp/cpp ..... Organizing rendering process & logging
└── main.cpp ..... Entry Point
```

In a nutshell, `Render.Run()` creates a `Loader` to read the configurations specified in YAML, and then read the specified

`.obj` file to get the models. It will then call the function that you've implemented in `rasterizer_impl.cpp` where it should be used in the rendering process. Image I/Os will be handled automatically.

A feature that may be helpful during debugging, is that you can define a macro `PRINT_TRIG_DETAIL` by adding an argument `-DPRINT_TRIG_DETAIL` when compiling the code, which will print the coordinates of the triangle that is finally presented to the `rasterizer`. Being "final" means that, for the case of transforms, the coordinates printed out will be those after further undergoing screen space transformation from the canonical space. The `x` and `y` will be the pixel coordinates before truncating its fractional part, and `z` will be the depth value (which will be elaborated on in further sections). For the first task, `z` will always be 0, as specified in the matrix multiplication above (Eq. (1)).

Note (Coordinate System Convention).

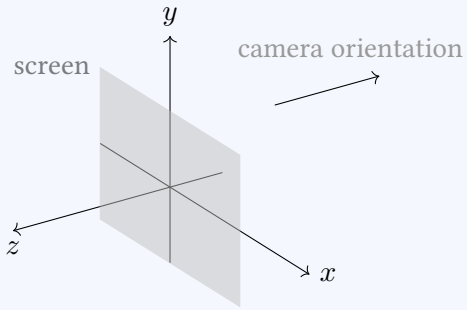


Figure 1: World Coordinates

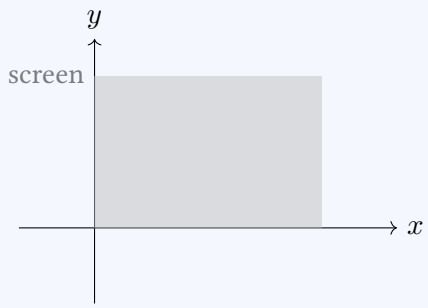


Figure 2: Screen Space Coordinates

It is worth mentioning that different graphics APIs adopt different conventions[a] , and it is important to clarify which convention one is using. Throughout this assignment, we are designating the positive x axis to be pointing to the right, and the positive y axis to be pointing upwards. We adopt the right-hand system, so the positive z axis must be pointing out of the screen (or paper, if you've printed it out). For the output image, the origin is set to the lower-left corner, with the positive x axis pointing to the right and the positive y axis pointing upwards.

Notice that this convention is different from how blender is interpreting the `.obj` file, where positive x is still rightward; but positive y is pointing into the screen, and positive z is pointing upwards to maintain a right-hand system. Therefore, do not panic if you see the axis in your own testcase and the axis in your output mismatch – it probably results from the difference in conventions.

[a] There is no "correct" convention to adopt in this scenario – the important thing is to stick to one convention to avoid confusion (and resulting inconsistencies in the outcome). We are using the same convention as OpenGL, but for example DirectX (a graphics API exclusive for Windows) adopts a completely different convention where the top-right corner of the screen is the origin, and a left-hand system in the world space is used instead.

It is also worth mentioning that OpenGL switches to a left-handed system in the screen space, as it changes the sign of the z axis in orthographic transformation. Details can be found in this article.

For simplicity we are using several external libraries (e.g. `fkyaml` and `tinyobjloader` for YAML parsing and Obj file reading, respectively). Most of them are irrelevant for implementation, but since you are implementing math-related algorithms, the one library that you will use intensively is `glm` (OpenGL Mathematics). It is then important to clarify

how data is organized in it.

Note (Row/Column-Major). To store a matrix one can either designate that elements in the same row should be adjacent in the memory; or that elements in the same column should be adjacent. Double arrays are **row-major**, as elements in the same row are stored consecutively. In opposition, `glm` matrices (e.g. `glm::mat4`) are **column-major**. Therefore, the format in which a `glm::mat4` is initialized corresponds to the transpose of the actual matrix. For example we have the correspondence:

```
glm::mat4 mat{
    1, 0, 0, 0,
    2, 1, 0, 0,
    3, 2, 1, 0,
    4, 3, 2, 1
};
```

$$\Longleftrightarrow \qquad \begin{pmatrix} 1 & 2 & 3 & 4 \\ 0 & 1 & 2 & 3 \\ 0 & 0 & 1 & 2 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

1.3 Review: Homogeneous Coordinates

Here we review the concept of **homogeneous coordinates**, and look a bit deeper into how it extends the possible transformations that can be expressed by linear transformations (matrix multiplications) on it. The homogeneous coordinate for $\mathbb{R}^d$ (for $d = 2, 3$) is defined as $\mathbb{R}^{d+1}$, with the correspondence with the coordinates in the Euclidean space

$$(x_1, x_2, \ldots, x_d) \mapsto (x_1, x_2, \ldots, x_d, 1) \quad \text{and} \quad (x_1/w, x_2/w, \ldots, x_d/w) \mapsfrom (x_1, x_2, \ldots, x_d, w)$$

It is required that the last field of every homogeneous vector is nonzero.

As expected, a "linear" transformation applied to the homogeneous coordinates is described by a $(d+1) \times (d+1)$ matrix. For our purpose, the following discussion restricts to the case of $d = 3$, where the transformations are 4-by-4 matrices. First recall that a matrix-vector multiplication can be viewed as a linear combination of the column vectors of the matrix. For example in $\mathbb{R}^3$ this is

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = x_1 \begin{pmatrix} a_{11} \\ a_{21} \\ a_{31} \end{pmatrix} + x_2 \begin{pmatrix} a_{12} \\ a_{22} \\ a_{32} \end{pmatrix} + x_3 \begin{pmatrix} a_{13} \\ a_{23} \\ a_{33} \end{pmatrix} \tag{$*$}$$

Split the transformation matrix into four parts:

$$T = \left(\begin{array}{c|c} R_{3\times 3} & T_{3\times 1} \\ \hline S_{1\times 3} & s \end{array}\right)$$

and consider how this acts upon a normalized[3] homogeneous vector $(x, y, z, w[=1])^T$:

- $R_{3\times 3}$ gives a transformation of coordinate system. The canonical basis for $\mathbb{R}^3$ is given by the basis $(1,0,0)^T$, $(0,1,0)^T$ and $(0,0,1)^T$. Applying the transform, and explicitly write the matrix-basis multiplication in the form of Eq. (∗), we have for the first basis vector

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \begin{pmatrix} x_1[=1] \\ x_2[=0] \\ x_3[=0] \end{pmatrix} = 1 \cdot \begin{pmatrix} a_{11} \\ a_{21} \\ a_{31} \end{pmatrix} + 0 \cdot \begin{pmatrix} a_{12} \\ a_{22} \\ a_{32} \end{pmatrix} + 0 \cdot \begin{pmatrix} a_{13} \\ a_{23} \\ a_{33} \end{pmatrix} = \begin{pmatrix} a_{11} \\ a_{21} \\ a_{31} \end{pmatrix}$$

  It is then clear that the first basis vector is mapped to the first column vector in the matrix. Evaluating Eq. (∗) using other two basis vectors give similar results. Therefore, a left multiplication of a 3-by-3 matrix essentially maps the canonical basis vectors to the column vectors of the matrix, which is equivalent to a transformation of a coordinate system. This can be used to represent scaling along the axes and rotation (which are essentially non-uniform scaling of basis vectors, and uniform rotation applied on basis vectors), as they are in essence the corresponding transformations applied to the basis vectors.

- $T_{3\times 1}$ is multiplied only by the w-coordinate which is 1 here, and then added to the previous terms. This gives a translation.

- s is also multiplied only by the w-coordinate, which controls the uniform scaling of the object.

- $S_{1\times 3}$ is multiplied by the coordinates, and then added to the previous terms. This term specifies how coordinates control uniform scaling of the object.

The idea of introducing homogeneous coordinates is that this allows transforming affine equations (linear transformations with translation) into linear equations. The converse is, however, not true – the transform is affine only when $S_{1\times 3}$ is a zero vector, and s is nonzero. An example in graphics will be presented in projection transformation.

1.4 Review: Triangle Rasterization & Anti-aliasing

Recall that in the lecture we have introduced a simple algorithm for judging whether a point is inside the triangle, if it lies on the same plane as that triangle. Let the vertices of the triangle be $A_1 = (x_1, y_1)$, $A_2 = (x_2, y_2)$, and

[3] Do not confuse this with the "normalized vector" in Euclidean space, which refers to the norm being 1. A **normalized homogeneous coordinate** refers to the last field $w = 1$. It is "normal" in the sense that the last field is 1 (a canonical choice), and has a one-to-one correspondence with vectors in the Euclidean Space $\mathbb{R}^3$.

$A_3 = (x_3, y_3)$, and the point be $X = (x, y)$. Embed them in the 3-dimensional space. Then X is inside the triangle if and only if the three cross products have the same sign:

$$S_1 = (X - A_1) \times (A_2 - A_1), \quad S_2 = (X - A_2) \times (A_3 - A_2), \quad S_3 = (X - A_3) \times (A_1 - A_3)$$

where $\times$ denotes the cross product. A simple visualization of S_is is presented below, where the areas colored in green, blue and purple correspond to the area where S_1, S_2 and S_3 point out of the screen (again, or figure) for X in that area, respectively. The point X is inside the triangle if and only if it lies in the intersection of these three areas, which is exactly the triangle.

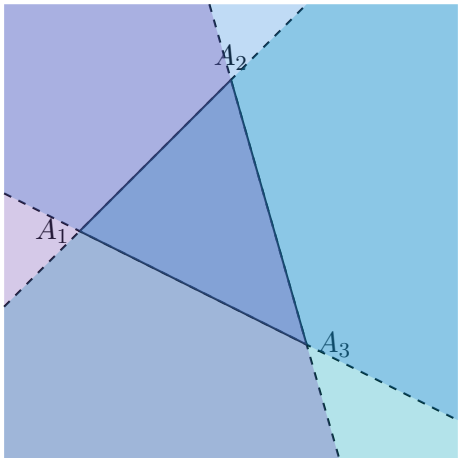


Figure 3: Sign of Cross Products S_i for X in Different Areas

The most straightforward antialias algorithm, super sampling anti-aliasing (SSAA) is achieved via simply increasing the sample frequency, i.e. taking multiple sample points in each pixel[4], and conduct the same test as above, and then average the results. This modification should be present when `config` is set to `AntiAliasConfig::SSAA`.

> **Note.** The following questions may be worth thinking about in your implementation:
>
> - For each pixel, what is the exact position in the coordinate system for which you need to perform the inside/outside judgement? How this changes under the case of anti-aliasing?
> - What are the boundary cases? For example, how do you treat the case where one sample point lies exactly on the edge of the triangle? Is your implementation consistent over all of them?

1.5 Task: Implement `DrawPixel`

For this part of the assignment, your task is to implement the function `DrawPixel`. Its signature is:

```
1   void Rasterizer::DrawPixel(uint32_t x, uint32_t y, Triangle trig, AntiAliasConfig config,
    uint32_t spp, Image& image, Color color)
```

[4]Refer to the lecture notes on how to actually do the sampling. If you would like to try on some other methods, Wikipedia gives a good introduction on this.

The variables are interpreted as follows:

- `x` and `y` are the pixel coordinates in the image.
- `trig` is the triangle to be rasterized, presented in pixel coordinates.
- `config` is the anti-aliasing configuration, being either `AntiAliasConfig::None` or `AntiAliasConfig::SSAA`.
- `spp` specifies the samples per pixel for SSAA. If `config` is set to `AntiAliasConfig::None`, the value of `spp` is simply discarded.
- `image` is the image buffer to which the pixel is written.
- `color` is the color of the pixel if it is completely in the triangle.

The following pseudocode gives a brief overview of the algorithm. The parts that you need to implement are highlighted in green.

Algorithm 1: Single Triangle Rasterization with Optional Anti-aliasing

```
1  foreach pixel possibly inside the triangle do
2      if anti-aliasing is off then
3          if the pixel is inside the triangle then
4              Write to the corresponding pixel;
5          end
6      else
7          foreach pixel in the image do
8              Generate sample points based on spp;
9              count = 0;
10             foreach sample point in the pixel do
11                 foreach triangle in the model do
12                     if the sample point is inside the triangle then ++count;
13                 end
14             end
15             Write to the pixel, with value based on count and spp;
16         end
17     end
18 end
```

The reference output generated by instructor's solution is the following:

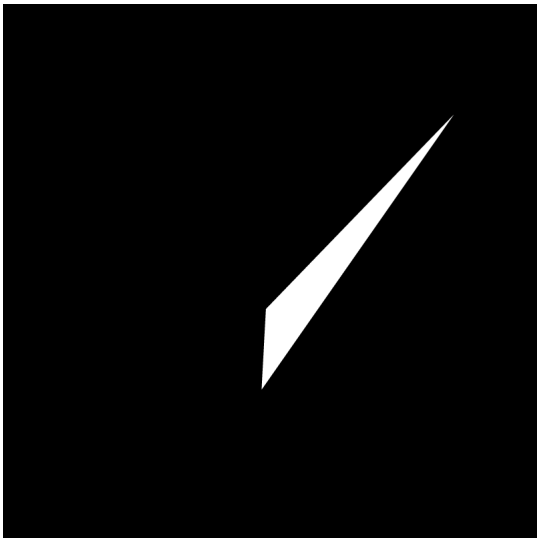


Figure 4: Reference Result for Triangle Rasterization (`test-triangle.yaml`)

2 MVP Transformation

2.1 Testfile: Additional YAML Configuration

2.1.1 Full Test

To capture the transformations of the objects, more information needs to be provided about the scenario. The YAML configuration for this part is extended as follows:

```yaml
task: transform
antialias: SSAA
samples: 16
resolution:
    width: 800
    height: 800
obj: trig
output: output
camera:
    pos: [0.0, 1.0, 2.0]
    lookAt: [0.0, 0.0, 0.0]
    up: [0.0, 2.0, -1.0]
    width: 0.2
    height: 0.2
    nearClip: 0.1
    farClip: 100.0
transforms:
    -
        rotation: [0.886, 0.0897, 0.3455, 0.2958]
        translation: [0.0, 0.0, 0.0]
        scale: [1.0, 1.0, 1.0]
    -
        rotation: [-0.374, 0.843, 0.094, 0.375]
        translation: [2.0, 0.0, 0.0]
        scale: [1.0, 1.0, 1.0]
```

The configurations are interpreted as follows:

- For this part `task` is always set to `transform`.
- Similar to the previous part, `antialias` can be either `none` or `SSAA`, and `samples` specifies the number of samples if `antialias` is set to `SSAA`; otherwise the value is simply discarded.
- `resolution`, `obj`, and `output` are the same as in the previous part.
- `camera` specifies the camera parameters. The names of its sub-keys should be self explanatory. There are two implementation-specific points to note:

  - `pos`, `lookAt`, and `up` will be read into the infrastructure as `glm::vec3`s.
  - All the values in any of the fields is a `float`. Due to limitations of the library we are using, it strictly distinguishes between `float`s and `int`s, so make sure you have added a '.0' when writing the configurations, so as to avoid type errors.

- `transforms` specifies the transformations to be applied to the object. This field has value of a list of dictionaries:
  - `rotation` is a quaternion in the sequence of `w`, `x`, `y`, `z`; and `pos` is the translation vector. This is presented as a `MeshTransform struct` in the infrastructure. Since the conversion from a quaternion to a rotation matrix is a bit tedious, the rotation matrix will be presented.
  - Notice that there can be multiple objects in read from the `.obj` file, each one of which can have its own set of transformations. The infrastructure will give a warning when the number of objects in the `.obj` file does not match the number of transformations specified in the YAML file.

2.1.2 Validation Tests

Graphics applications are often hard to debug, as some errors will cause the resulting image drift completely out of the canvas, and thus yielding a completely black image from which no information can be extracted. In order to alleviate the difficulty of debugging, we provide a set of validation tests, which will provide the configurations, input position, and the expected position on canvas. The YAML configuration is correspondingly extended:

```yaml
task: transform-test
antialias: SSAA
samples: 16
resolution:
    width: 800
    height: 800
obj: trig
output: output
camera:
    pos: [0.0, 0.0, 5.0]
    lookAt: [0.0, 0.0, 0.0]
    up: [0.0, 1.0, 0.0]
    width: 100.0
    height: 100.0
    nearClip: 0.1
    farClip: 100.0
transforms:
    -
        rotation: [0.886, 0.0897, 0.3455, 0.2958]
        translation: [0.0, 0.0, 0.0]
        scale: [1.0, 1.0, 1.0]
input: [0.0, 0.0, 0.0]
expected: [400.0, 400.0, -0.962, 1.0]
```

where `input` is the position to be transformed; and the `expected` field gives the expected output from your program, after applying the matrix transformations.

Notice that this kind of testcases will not be large in number, and serve only as a guide for writing your own testcases. Although succeeding in all of these tests do suggest that your solution is largely correct, problems can occur when these data are actually rendered; and you are encouraged to see how your triangle rasterization algorithm in section 1 cooperates with the implementation of transforms.

2.2 Review: Model, View, Projection, and Screen Space Transformations

This part gives a brief explanation of how the transformations need to be constructed. You are welcome to refer to the lecture notes for more detailed information.

The overall goal of this transformation is to fit the whole scene into the *canonical space* $[-1, 1]^3$, where x and y presents the final position of objects, and z encodes the occlusion information. The transformation is split into four parts, each of which is elaborated in a subsection below.

2.2.1 Model Transformation

Oftentimes an object may have multiple instances with different postures; or some transformations need to be applied to the object, without changing the original model data of it. Then a model transformation can applied, transforming the object from the *model space* (where the model is defined) to the *world space*. In the codebase, it is encapsulated as a `struct`:

```
MeshTransform
├─ glm::quat rotation ..... Quaternion encoding of rotation (the conversion to matrix will be given)
├─ glm::vec3 pos ..... Translation (displacement) of center
└─ glm::vec3 scale ..... Scaling (can be anisotropic)
```

Scaling and translation can be easily convert to a matrix. For rotation, first notice that a normalized quaternion can indeed encode a rotation, as any rotation can be fixed via specifying the orientation of a specific point of the object. Converting a quaternion to a rotation matrix is a bit tedious, so you can `glm::toMat4` function for conversion. This is the *only* function you are allowed to use from the `glm` library: other matrices in subsequent chapters must be composed by hand.

> **Note.** Although the constructions of the three matrices are straightforward, the order of multiplication is important, as rotations and translations in general do not commute; and the rotation specified here should not be affected by translation and scaling. The issue of ordering matrix multiplications will appear in all subsequent transformations.

2.2.2 View Transformation

Recall that the final objective is to render the scene onto a 2D plane, where the dimension corresponding to depth value should not affect the output (although it determines the occlusion relation). The ideal case would be that the camera is such aligned so that one dimension can be simply "trimmed off" when we are finally drawing onto the canvas. View transformation prepares the exactly for that.

View transformation places the camera at the origin, and orients it towards negative z-axis. It is also required that the transformation should be affine, i.e. preserving the relative position between objects. This breaks down to a translation and a rotation. The former can be constructed similarly as in the model transformation, and the latter is explained a bit more in detail.

"Orienting" the camera can be viewed as a coordinate transformation, where the `lookAt` and `up` (guaranteed to be orthogonal) corresponds exactly to the negative z- and positive y-direction. The third axis (often called binormal) is thus fixed by conducting a cross product. Denote the `lookAt` and `up` by $\boldsymbol{g}$ and $\boldsymbol{h}$, correspondingly. Denote the binormal axis by $\boldsymbol{f}$. The following figure illustrates the orientation process, where correspondence of axes are identified through color:

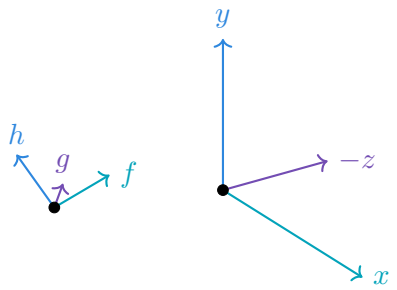


Figure 5: View Transformation as Orientation of Coordinate System

In the review on homogeneous coordinates, we mentioned how a transformation of coordinate system from the canonical basis can be constructed. The problem we have here is to orient an arbitrary orthogonal basis to a canonical basis, which is exactly the inverse process. Although this is not that straightforward, by the fact that camera `lookAt`, `up` and the binormal vector ($\boldsymbol{g}$, $\boldsymbol{h}$ and $\boldsymbol{f}$), it is easy to figure out its inverse. Denoting the i-th component of $\boldsymbol{g}$ to be g_i and similar for $\boldsymbol{h}$ and $\boldsymbol{f}$, notice that

$$\begin{pmatrix} g_1 & g_2 & g_3 \\ h_1 & h_2 & h_3 \\ f_1 & f_2 & f_3 \end{pmatrix} \begin{pmatrix} g_1 & h_1 & f_1 \\ g_2 & h_2 & f_2 \\ g_3 & h_3 & f_3 \end{pmatrix} = \begin{pmatrix} \langle \boldsymbol{g}, \boldsymbol{g} \rangle & \langle \boldsymbol{g}, \boldsymbol{h} \rangle & \langle \boldsymbol{g}, \boldsymbol{f} \rangle \\ \langle \boldsymbol{h}, \boldsymbol{g} \rangle & \langle \boldsymbol{h}, \boldsymbol{h} \rangle & \langle \boldsymbol{h}, \boldsymbol{f} \rangle \\ \langle \boldsymbol{f}, \boldsymbol{g} \rangle & \langle \boldsymbol{f}, \boldsymbol{h} \rangle & \langle \boldsymbol{f}, \boldsymbol{f} \rangle \end{pmatrix}$$

where $\langle a, b \rangle = a \cdot b$ is the inner product in the Euclidean space $\mathbb{R}^3$, which is symmetric. Orthogonality implies that all off-diagonal elements are zero; and the diagonal elements are the norm of the corresponding vector, which are all 1 if $\boldsymbol{g}$, $\boldsymbol{h}$ and $\boldsymbol{f}$ are all normalized (i.e. they are *orthonormal*, being orthogonal and normalized). This implies that *transpose of transformation matrix between orthonormal basis is its inverse.* Combining this result with the construction of matrices

representing coordinate system transformation in review on homogeneous coordinates we have the rotation part of the view transformation matrix.

> **Note.** Same as before, the order of multiplication of rotation and translation matrix matters.
>
> Furthermore, since cross product is not commutative, the order of $\boldsymbol{g}$ and $\boldsymbol{h}$ being composed matters. Remember we need to maintain a right-handed system.

2.2.3 Projection Transformation

The objective of projection transformations are to map the scene to the canonical space $[-1, 1]^3$. It is split into the **orthographic transformation** and the **perspective transformation**. The former simply fits the scene to the canonical space; while the later mimics the perspective relation via "squishing" a frustum into a cube (which is not affine, as we will see later). The image below from stackoverflow gives a good visualization of the difference:

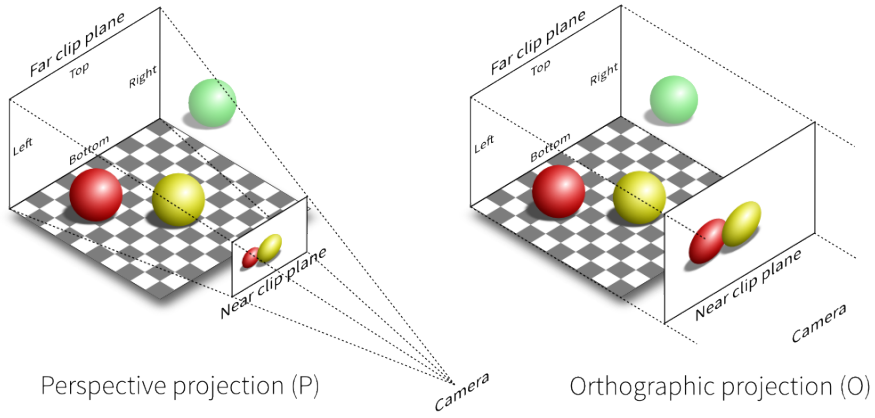


Figure 6: Perspective and Orthographic Transformations

Recall that the camera parameters contain the near clipping distance and far clipping distance. Ideally every object within the viewing frustum should be visible in the final output. However, since occlusion needs to be computed definitely and it must be encoded in finitely many bits, a far clipping plane is necessary for rendering correct occlusion relations.[5] This, together with the camera width and height, then fixes the region visible to the camera, be it a cube (orthographic) or frustum (perspective).

For orthographic projections, all objects are treated as of same size, regardless of their distance to the camera. Therefore, rescaling and translation are sufficient to achieve this.

Now consider perspective projections. The goal is to map the frustum to a cube, which should be ready for transformation by orthographic projection. This gives the following requirements:

1. The scaling coefficient should be proportional to its z-value.

[5] A talk in GDC 2007 suggests taking the limit when far clipping plane is at the infinity on the projection matrix for rendering "background" objects, for example the sky box. This, however, could cause problems with occlusion (e.g. floating point precision issues) if not treated with great care. This is considered a hack of, instead of an improvement on, the standard method based on canonical space.

2. All points on the near clipping plane should be fixed.

3. The intersection of the z-axis and the far clipping plane should be fixed.

> **Note.** In the deduction we are assuming that both n and f are strictly positive; but in our setup the camera points towards the *negative* z-axis, so they are actually negative. Think about the adjustments needed in the implementation.
>
> In the codebase, both n and f provided will be strictly positive, i.e. the distance from the origin to the clipping plane.

The first requirement can be achieved using the method introduced in review of homogeneous coordinates. Denote the near and far clipping distance to be n and f, strictly positive. Both x and y are scaled by a coefficient of n/z, fixes three rows of the transformation matrix (where a_is are unknowns):

$$T = \begin{pmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ a_1 & a_2 & a_3 & a_4 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

The second requirements says that for $z = n$, T fixes the position. This yields

$$\begin{pmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ a_1 & a_2 & a_3 & a_4 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} x \\ y \\ z[=n] \\ 1 \end{pmatrix} = \begin{pmatrix} xn \\ yn \\ xa_1 + ya_2 + na_3 + a_4 \\ n \end{pmatrix} \mapsto \begin{pmatrix} x \\ y \\ \frac{1}{n}(xa_1 + ya_2 + a_4) + a_3 \\ 1 \end{pmatrix} \overset{!}{=} \begin{pmatrix} x \\ y \\ n \\ 1 \end{pmatrix}$$

This should be independent of x and y, giving $a_1 = a_2 = 0$. Then the second requirement reduces to $a_3 + \frac{a_4}{n} = n$. Combine this with the third requirement:

$$\begin{pmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & a_3 & a_4 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ z[=f] \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ fa_3 + a_4 \\ f \end{pmatrix} \mapsto \begin{pmatrix} 0 \\ 0 \\ \frac{1}{f}a_4 + a_3 \\ 1 \end{pmatrix} \overset{!}{=} \begin{pmatrix} 0 \\ 0 \\ f \\ 1 \end{pmatrix}$$

We have a linear system

$$\begin{cases} a_3 + \frac{1}{n}a_4 = n \\ a_3 + \frac{1}{f}a_4 = f \end{cases}$$

Solving for the system gives the perspective transformation matrix.

2.2.4 Screen Space Transform

After transforming the whole scene into the canonical space, the final step is to map the scene to the actual size of the canvas. This can be achieved by a single diagonal scaling matrix.

> **Note.** Although the z value is not used explicitly in drawing to the canvas, it should not be simply discarded here as it provides information on occlusion, which will be implemented in next section. The scaling factor along z axis should be set to 1.

2.3 Task: Implement Transformation Matrices

The transformation matrices provide important building blocks for the Blinn-Phong shading models. In this part you only need to implement their constructions, and the infrastructure will use the `DrawPrimitiveRaw` function you've implemented in section 1 to draw with uniform white several triangles after applying transformations, via applying linear transformation with the matrix

$$M_{\text{MVP}} = M_{\text{screenspace}} \cdot M_{\text{projection}} \cdot M_{\text{view}} \cdot M_{\text{model}}$$

In the last part of this assignment you will need to implement the whole shading process, using the functions you have already implemented.

The functions that you need to implement are as follows:

```
void Rasterizer::AddModel(MeshTransform transform, glm::mat4 rotation);
void Rasterizer::SetView();
void Rasterizer::SetProjection();
void Rasterizer::SetScreenSpace();
```

Most of the functions do not take any argument, as for each `Rasterizer` it is linked with a `Loader`, whose member `Camera` provides the information for assembling the matrices which are identical throughout the rendering process. For `Rasterizer::AddModel` however, for simplicity (you do not need to do the sanity check and iteration) the `MeshTransform` is directly provided; and the rotation part of the matrix is given to avoid handling the tedious conversion from quaternion to rotation matrix.

A brief overview of the functions are as follows. There are no highlights as you need to implement all of these functions on your own. Refer to the explanations above for details on how these matrices are constructed.

Algorithm 2: Constructing MVP Matrices

```
AddModel(MeshTransform transform, glm::mat4 rotation):
    M_rot is rotation;
    Create M_trans from transform.pos;
    Create M_scale from transform.scale;
    Combine M_rot, M_trans and M_scale to get M_model;

CreateView():
    Get Camera from Loader;
    Create M_rot by orienting the camera towards negative z-axis;
    Create M_trans by moving the camera to the origin;
    Combine M_rot and M_trans to get M_view;

CreateProjection():
    Get camera parameters from Loader;
    Create M_persp to squish the objects to conform to perspective;
    Create M_ortho to rescale and translate the objects to the canonical space;
    Combine M_persp and M_ortho to get M_projection;

CreateScreenSpace():
    Get camera parameters from loader;
    Create M_ss based on the width and height of the canvas, and preserve the z value;
```

The reference output generated by instructor's solution is the following:

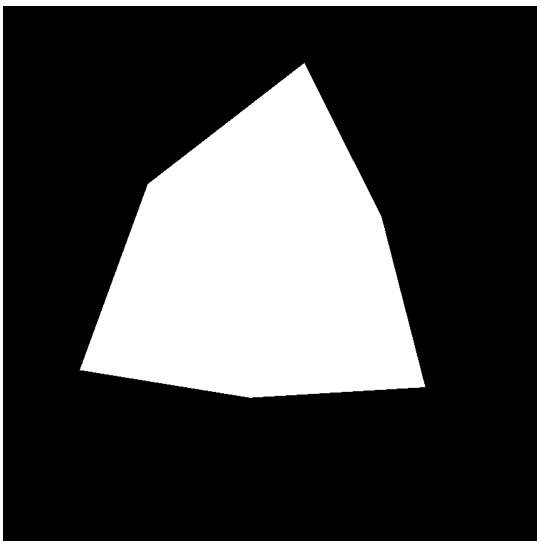


Figure 7: Reference Result for Transformation Test (`test-transform.yaml`)

3 Blinn-Phong Shading Model

3.1 Testfile: Adding Lights

In the final part regarding shading, we need to specify the properties of the lights. Their information are appended at the end of the YAML configuration, of which the following gives an example:

```yaml
task: shading
antialias: SSAA
samples: 16
resolution:
    width: 800
    height: 800
obj: cube
output: output
camera:
    pos: [0.0, 1.0, 2.0]
    lookAt: [0.0, 0.0, 0.0]
    up: [0.0, 2.0, -1.0]
    width: 0.2
    height: 0.2
    nearClip: 0.1
    farClip: 100.0
transforms:
    -
        rotation: [0.886, 0.0897, 0.3455, 0.2958]
        translation: [0.0, 0.0, 0.0]
        scale: [1.0, 1.0, 1.0]
exponent: 4.0
ambient: [10, 10, 10]
lights:
    -
        pos: [0.0, 1.0, 2.0]
        intensity: 2.0
        color: [255, 255, 255]
    -
        pos: [4.0, 0.0, 0.0]
        intensity: 8.0
        color: [179, 87, 181]
```

The `task` will be either `shading` or `shading-depth`, specifying whether only the depth information is needed, or the whole shading process needs to completed. No anti-alias is required, so both the values of the `antialias` and `samples` fields will be discarded. But do specify them (even with a placeholder!) when writing your own configurations as they are required fields in the infrastructure. Failure to do so will result in a runtime error.

The added fields correspond to the lights. Similar to the `MeshTransforms` in section 2, there can be multiple `lights` in the configuration (2 in the sample above). Each `light` has its own position, intensity, and color. Similar to the previous convention, the `pos` will be concatenated with `w` being 1, and the alpha of `color` will be set to 255 (opaque). The additional `exponent` and `ambient` will be detailed later, in the review of the (Blinn-)Phong shading model.

3.2 Review: Barycentric Coordinates

All properties (position, normal, texture coordinates, etc.) are provided to be continuous on a triangle, the most basic element in a mesh. In fact, when we are talking about "triangles", we are in fact referring to the linear interpolation of the vertices given in the mesh[6], treating positions as vectors starting at origin and ending at that specific position.

First consider **linear interpolation**, which is essentially the parametrization of the line connecting the end points, to describe a "smooth transition" between the two points. This can be done multiple times, resulting in n-linear interpolation. A straightforward parametric description is illustrated in the following figure.

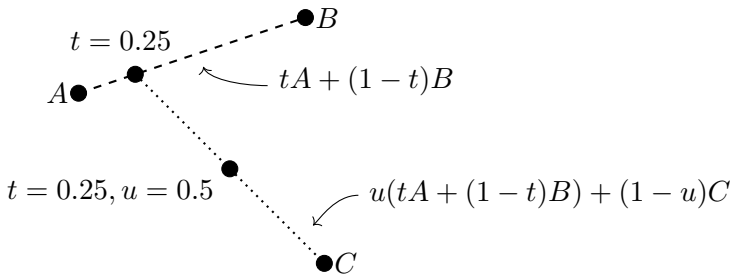


Figure 8: Linear (t) and Bilinear ((u, t)) Interpolation

From the above example, it is clear that in bilinear interpolation t and u are equally important; and applying the t-interpolation first with u-interpolation afterwards, or the other way around does not affect the result. However, the points A, B and C are not attached with equal weight: from the expression we see that parameter t does not take into consideration the point C. The **barycentric coordinate** seeks to define a linear interpolation, where all vertices share the same level of importance. Using similar form of expression a natural definition for a point P in the triangle specified by points A, B and C can be expressed using α and β, when considering positions as vectors:

$$P = \alpha A + \beta B + (1 - \alpha - \beta)C, \quad \text{with barycentric coordinates } (\alpha, \beta, 1 - \alpha - \beta)$$

This can be visualized as the area of the triangle opposite to the corresponding edge. Let the total area of the triangle be s, then for P described using the barycentric coordinates above, the different triangles colored in the following figure should be of area αs, βs and $(1 - \alpha - \beta)s$.

This is clearly a linear interpolation as all vertices A, B and C appear linearly on both α and β. It is also required that the three coordinates in the barycentric coordinate must add up to 1 (as is by linear interpolation). Further we can

[6]There can be non-linear elements, for example quadratic triangles, used in Finite Element Analysis (FEA) in simulation, a branch in computer graphics. Interpolation of this kind requires more information (vertices), as two points cannot fix a quadratic function (curve).

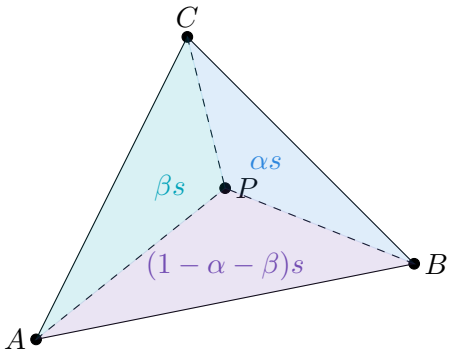


Figure 9: Visualization of Barycentric Coordinates

observe that the coefficients are symmetric: a change of parameter $u := 1 - \alpha - \beta$ and $v := \beta$ gives the change of coordinate

$$(\alpha, \beta, 1 - \alpha - \beta) \quad \longrightarrow \quad (1 - u - v, v, u)$$

which is basically a relabelling of the vertices (and does not change the actual interpolation). These coefficients then allow us to interpolate linearly the properties defined only on the vertices, as listed in the beginning of the section.

3.3 Review: Depth Buffer

A naïve way to resolve occlusion is draw objects in the same order as their distance to the camera. But since order is a *local* property, objects can have intricate "cyclic" orders, for example as in the figure on the right. Although this can be resolved via splitting a triangle into two parts, this also requires rendering pixels which whose value will be eventually discarded. The standard way to handle this problem is to use a **depth buffer (Z-Buffer)**, by traversing through all triangles twice, record the smallest depth value for each pixel in the buffer. When doing the actual rasterization, the depth value of the pixel is compared with the depth value in the depth buffer, and is discarded if it is larger.

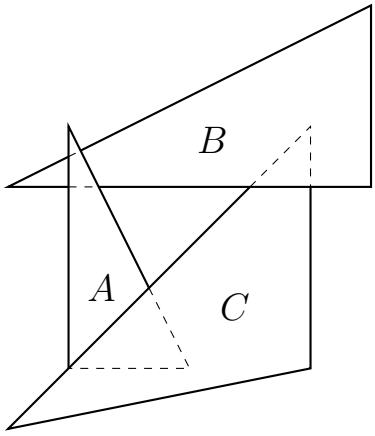


Figure 10: Cyclic Order of Objects

> **Note.** Since the perspective transformation is not linear, the depth value in the canonical space does not change linearly with the actual z-value. Will this cause a problem?

3.4 Review: Phong and Blinn-Phong Shading Model

Phong and Blinn-Phong models are two very similar approximation shading models. Their ideas are summarized in the following figure[7]:

[7] Retrieved from Lecture Slides by Prof. Lingqi Yan at UCSB

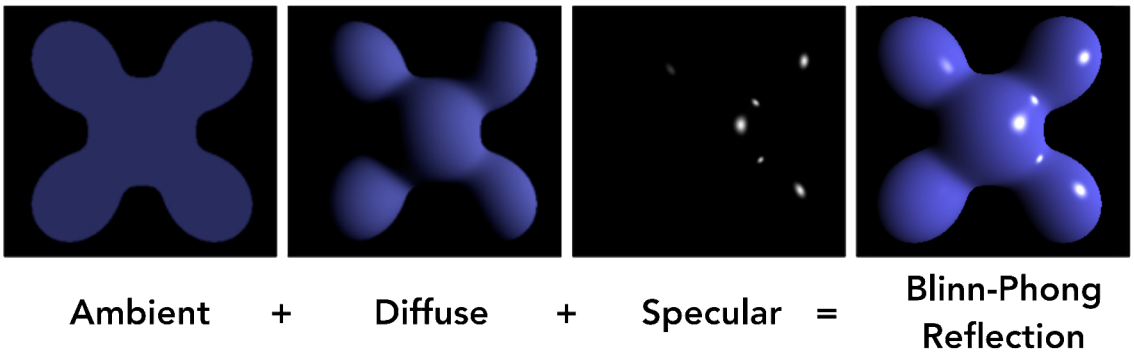


Figure 11: Blinn-Phong Shading Model

We explain the terms separately:

- The ambient term is a uniform color added to the object, independent from the light source. The color is specified in the `ambient` field in the YAML configuration.
- The diffuse (lambertian) term is proportional to the cosine of the angle between the normal vector and the light source, clamped above at 0 (as occlusion should not cancel out the effects of other lights). It is also inverse proportional to the square of distance between the light source and the object. The "square" here results from the reasoning that the energy from the light is distributed evenly on the surface of sphere for any fixed distance (radius of the sphere), and the surface area of the sphere is proportional to the radius of the sphere. Denoting the distance from the light to be r, and the light direction and the normal vector as $\boldsymbol{\ell}$ and $\boldsymbol{n}$, the diffuse term is given by
$$L_d = \text{LightColor} \cdot \text{LightIntensity} \cdot \frac{1}{r^2} \max\{0, \boldsymbol{\ell} \cdot \boldsymbol{n}\}$$
- The specular term, modelling the "shiny" reflection of light, is defined a bit differently between Phong and Blinn-Phong models. Denoting $\boldsymbol{v}$ as the direction from the object to the camera, and $\boldsymbol{h} := \frac{1}{2}(\boldsymbol{\ell} + \boldsymbol{v})$ as the half vector of $\boldsymbol{\ell}$ and $\boldsymbol{v}$, in Blinn-Phong model the specular term is given by
$$L_s = \text{LightColor} \cdot \text{LightIntensity} \cdot \frac{1}{r^2} \underline{\max\{0, \boldsymbol{n} \cdot \boldsymbol{h}\}^e}$$
where e is specified in the `exponent` field in the YAML configuration. The Phong model is similar, but uses $\max\{0, \boldsymbol{v} \cdot \boldsymbol{r}\}^e$ instead of the underlined part in the equation above, where $\boldsymbol{r} := 2(\boldsymbol{\ell} \cdot \boldsymbol{n})\boldsymbol{n} - \boldsymbol{\ell}$ which is the reflection vector of $\boldsymbol{\ell}$. Since the angle formed by $\boldsymbol{v}$ and $\boldsymbol{r}$ is twice as the angle formed by $\boldsymbol{v}$ and $\boldsymbol{n}$, the exponent e needs to be changed accordingly to achieve the same effect, which in Phong model is often set to be 4 times as the exponent in Blinn-Phong model.

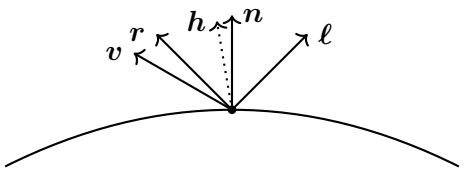
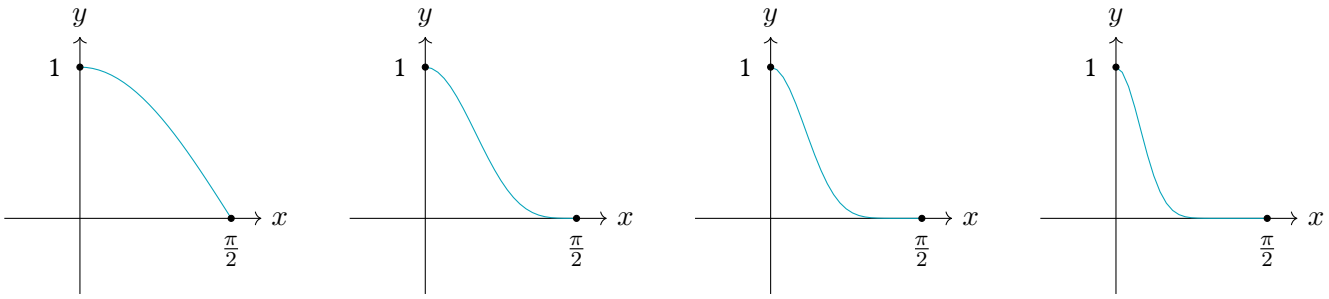


Figure 12: Vectors used in Phong/Blinn-Phong Model

But both achieve a sharp attenuation of the specular term as the angle between $\boldsymbol{n}$ and $\boldsymbol{h}$ increases, i.e. the viewing angle deviates from the reflection angle, from the sharp decreasing of the cosine function raised to an exponent:



$\cos x, x \in [0, \pi/2]$ $\cos^4 x, x \in [0, \pi/2]$ $\cos^8 x, x \in [0, \pi/2]$ $\cos^{16} x, x \in [0, \pi/2]$

Note. When calculating the depth/color of a pixel, we are using linear interpolation (barycentric coordinates) between the vertices. As in the shading process we need to compute the color of a pixel based on its world coordinates, applying the inverse MVP transformation is necessary.

This certainly works (you are allowed to use this strategy in your implementation), but computing the inverse of a matrix is expensive. Extending the thoughts of barycentric coordinates and looking deeper into the MVP transformation, is there a simpler (and less costly) way to retrieve the world coordinates correspoding to a pixel?

3.5 Task: Implement Blinn-Phong Shading Model

There are three functions and one static variable that needs to be implemented in this part:

```
1  glm::vec3 Rasterizer::BarycentricCoordinate(glm::vec2 pos, Triangle trig)
2  float Rasterizer::zBufferDefault;
3
4  void Rasterizer::UpdateDepthAtPixel(uint32_t x, uint32_t y, Triangle original, Triangle
   transformed, ImageGrey& ZBuffer)
5  void Rasterizer::ShadeAtPixel(uint32_t x, uint32_t y, Triangle original, Triangle transformed
   , Image& image)
```

The first function `BarycentricCoordinate` will not be used in the infrastructure, but since it is essential for your implementation, it is included in the list. For `trig` with vertices `v1`, `v2` and `v3` whose `z` value being set to 0, the function

should return `bc` s.t.

$$\begin{pmatrix} \texttt{pos.x} \\ \texttt{pos.y} \\ 0 \end{pmatrix} = \texttt{bc[0]} * \texttt{v1.pos} + \texttt{bc[1]} * \texttt{v2.pos} + \texttt{bc[2]} * \texttt{v3.pos}$$

The first subtask is to implement functions related to rendering the depth buffer. You need to first initialize the static variable `Rasterizer::zBufferDefault` whose value representing to the infinite depth. The infrastructure will initialize the `ZBuffer` with `zBufferDefault`.

The rest two functions correspond to the depth testing and the actual shading process. `UpdateDepthAtPixel` computes for each pixel the depth value. The parameters are:

- `x` and `y` are the coordinates of the pixel in the image.
- `original` is the triangle in the world space.
- `transformed` is the triangle in the canonical space.
- `ZBuffer` is the depth buffer that you need to write to.

The values of the depth buffer (Z-Buffer) must be between -1 and 1, to allow the infrastructure to correctly output to a greyscale image. The initialization of depth buffer should be consistent with your depth testing.

`ShadeAtPixel` computes the color of the pixel, using Blinn-Phong shading model. The parameters are interpreted similarly, with the only difference being that the image handle is an `Image`, where each pixel is a `Color` instead of a `float`.

Notice that all the `Light`s are available in `rasterizer.loader`. Also see what other members `rasterizer` class have, as they may be useful in your implementation. Do not forget to write to the `ImageGrey` or `Image` after computing the depth or color.

It is recommended to first implement `UpdateDepthAtPixel`, and use `shading-depth` type in YAML configuration to see if a reasonable depth buffer can be output. The proceed to complete the shading part. As a reference a pseudocode for the overall process is presented below. The parts that you need to implement are marked with green.

Algorithm 3: Depth Test and Phong Shading

```
// Depth Test
Set the ZBuffer to be infinite;
foreach triangle in the scene do
    Transform the triangle to the canonical space (Your implementation in the previous part will be used);
    foreach pixel in triangle do
        Compute the depth of the pixel using barycentric coordinates of triangle;
        if depth is smaller than the value in ZBuffer then  Update the ZBuffer with the new depth ;
    end
end
// Shading
foreach triangle in the scene do
    Transform the triangle to the canonical space (Your implementation in the previous part will be used);
    foreach pixel in triangle do
        Compute the depth of the pixel using barycentric coordinates of triangle;
        if depth is exactly the value in ZBuffer then
            Compute the position of depth in the world space;
            Retrieve its normal using barycentric coordinates;
            Compute its color based on normal and light direction, using Blinn-Phong model;
            Write the color to the image;
    end
end
```

The reference output generated by instructor's solution is the following:

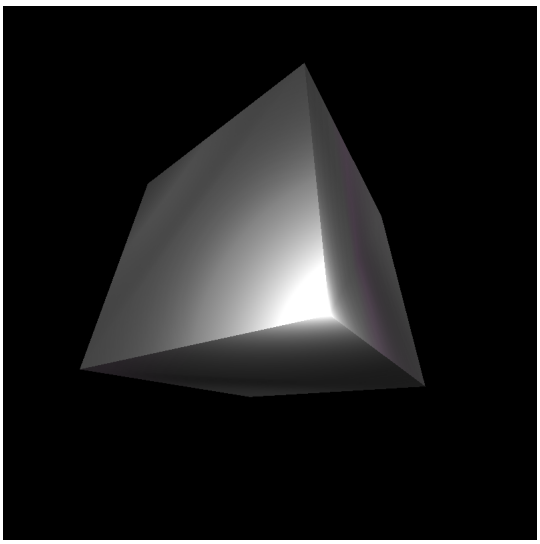


Figure 13: Reference Result for Blinn-Phong Shading (`test-shading.yaml`)

4 Grading

This assignment has 100 points in total, distributed as follows:

- **Part 1: Triangle Rasterization** (30 points). A robust algorithm that can handle all types of triangles will get full points. Triangles are guaranteed to be non-degenerate (three vertices will not be on the same line).
- **Part 2: MVP Transformation** (30 points). A correct implementation of the transformations will get full points.
- **Part 3: Depth Test and Shading** (40 points). Depth test will account for 20 points, and will be tested via rendering depth buffer. Shading will account for the rest 20 points, and a correct implementation (either Phong or Blinn-Phong) will get full points.

For points of extra features below and the portal for submission, please refer to the course logistics.

5 (Optional) Extra Features*

The rasterization pipeline that we have implemented above is a very basic one. There are many features that wer have not implemented, for example texture and shadow, which are essential in modern rendering. Furthermore, we are doing the rendering process on the CPU, which is not efficient. The following lists some of the tasks that you can further implement, and can be regarded as extra features, in the order of ascending difficulty:

- Implement MSAA (Multi-Sample Anti-Aliasing) in shading model[8]. In a nutshell, this outperforms SSAA by computing shading per pixel, and compute the depth for each sample point.
- Implement deferred shading[9]. The main idea is to decouple each stage in rasterization, creating a multi-stage yet simpler rendering pipeline. This is especially useful when the scene is rather complex.
- Implement texture mapping[10]. This requires you to load the texture image, and compute the mipmaps, to order aliasing artifacts for sharp triangles.
- Implement shadows[11]. The high level idea is that shadows are essentially parts of the object that is not occluded from camera, but is occluded by light. Therefore, rendering depth buffers from the light gives the occlusion information; and the Blinn-Phong shading model can be turned off for the occluded parts. This is referred to as "shadow mapping".

[8] This article gives a good explanation of the algorithm.
[9] Wikipedia provides a good introduction to the topic.
[10] Wikipedia gives a basic introduction. You may also want to look into mipmaps for quality texture application.
[11] LearnOpenGL gives an explanation and implementation in OpenGL. You do not need to understand the code, but the explanation of concept there is clear enough. However, due to the one-bounce nature of rasterization, shadow thus generated is "hard", i.e. has sharp edges. In homework 2, you will see that raytracing can also generate shadows, which are far better in quality, but also far slower.

- Transplant the rendering process to any rendering API[12]. Notice that in this case you do not need to compute the depth buffer as the API will handle this for you. However, you need to understand how the pipeline is constructed, and write a bit of shader code to implement the actual shading. For this specific task, your are welcome to use any external library that you see helpful.

You are welcome to implement features not listed here, but make sure to discuss with us to ensure that it is both feasible and accountable for extra credit.

[12]There are various APIs to choose from, for example OpenGL (multi-platform), DirectX (Windows), Metal (Mac OS) and Vulkan (multi-platform). We strongly recommend OpenGL, which is complex enough for customization, but not too complex to extend much beyond the algorithms. For OpenGL, a good reference is LearnOpenGL. Reading the first and a bit of the second chapter will be sufficient for the task.