UCSD CSE 167 Assignment 2:

3D Software Rendering

The previous homework focused on 2D graphics (which is already quite important!). In this homework, we're going 3D. We will define shapes in 3D, project them onto the screen space, and render them. The projection is going to simulate how a camera (or our eyes) works. In particular, we will implement a pinhole camera. The light coming from an object goes through a very small hole and projects on a film (Fig. 1).
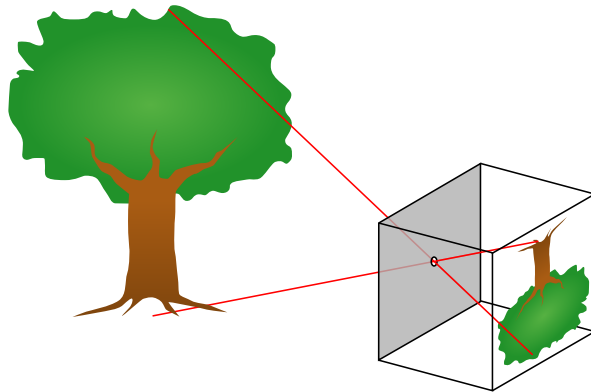

Figure 1: In this homework, we will implement 3D rendering by projecting 3D objects onto images. This is similar to a real-world camera. In particular, we will implement an ideal pinhole camera. Figure taken from Wikipedia.

To facilitate the projection, we will focus on representing 3D surfaces, instead of the full volume. Instead of supporting multiple shapes (circles, squares, etc) like last time, we will focus on a single primitive: triangle. A main reason is that triangle has a very cool property: after perspective projection to 2D, it's still a triangle! (See Fig. 2.) Many other shapes do not have this nice property – a sphere projecting on to 2D can be an ellipse, a square projecting to the screen can become a general quadliteral.
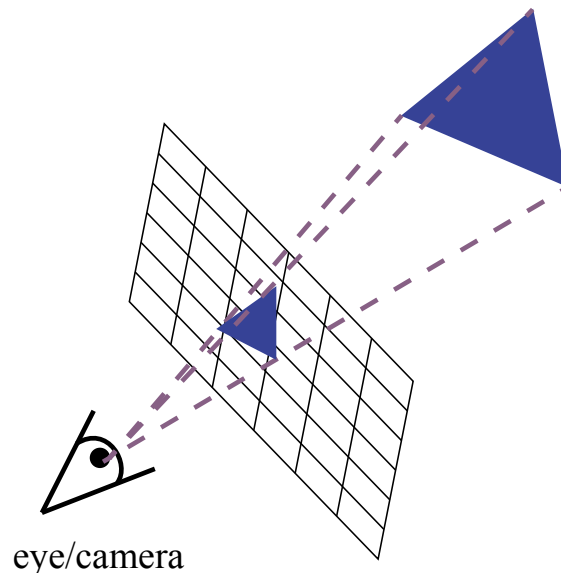

Figure 2: The perspective projection of a triangle.

1 Rendering a single 3D triangle (20 pts)

Let's start simple and render just a single 3D triangle. Our plan is to first project the triangle to a 2D image plane, then we can directly use the code from our previous homework to render the projected triangle.
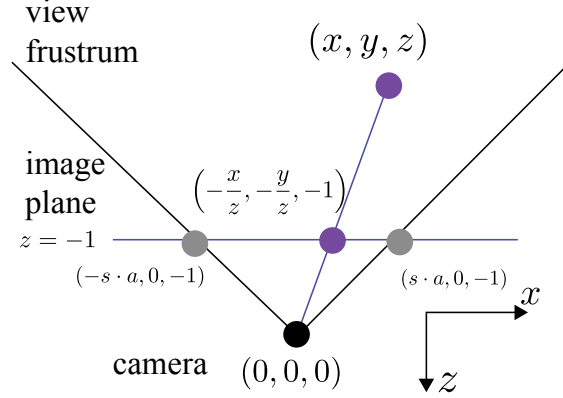

Figure 3: To perform perspective projection, we setup the coordinate system of the camera space so that the origin of the camera is at $(0, 0, 0)$, the camera is facing negative z-axis, x-axis is pointing right, and y-axis is pointing up. We put the image plane at $z = -1$. The perspective projection of a point (x, y, z) is then simply finding the intersection between the line formed by the point and the camera origin with the image plane. Since our image plane has a finite extent, we further clips all point outside of $[-sa, -s, -1] \times [sa, s, -1]$ where a is the aspect ratio ($\frac{\text{image width}}{\text{image height}}$) and s is the scaling factor controlling the size of our image. The clipping defines a *view frustrum*.

How do we project the triangle then? It's significantly easier if we assume a particular coordinate system of our camera. We will lift this assumption in the later part of the homework (by, you guess it, applying 3D transformations). Let's assume in the *camera space*, x-axis is pointing right, y-axis is pointing up, and z-axis is pointing away from the direction the camera is looking at (the z is specified this way so that we are following a right-handed coordinate system). We also assume that the image plane is located at $z = -1$ – this is where we are going to project our triangle onto. Fig. 3 shows that the perspective projection of a point (x, y, z) boils down to computing the intersection between the line formed by the point and the camera origin with the image plane. The projected point (x', y') is

$$(x', y') = \left(-\frac{x}{z}, -\frac{y}{z}\right). \tag{1}$$

Furthermore, we need to define the extent of the image plane (we can't have an image with infinite size!). Note that our image can be a rectangle (instead of a square). We define the extent of the image plane to be $[-sa, -s, -1] \times [sa, s, -1]$: s is a scaling factor that controls the size of the image (it's half of the sensor size), and a is the aspect ratio ($\frac{\text{image width}}{\text{image height}}$). s is related to the vertical field of view α of a camera:

$$s = \tan\left(\frac{\alpha}{2}\right). \tag{2}$$

Any point that lands outside of the extent of the image plane is discarded and not going to show up in our image. We call the set of points that are going to land on the image planes the *view frustrum*. Fig. 3 also visualizes the view frustrum.

Therefore, given a triangle with 3 vertices, we will project all three of them onto the image plane using Eq. (1). This generates a 2D triangle which we can then render using our previous homework's code.

There are a few other complications though. Firstly, the point (x', y') is in the (projected) camera space after the projection. We will need to transform it into the screen space. Recall that our screen space has x-axis pointing right with y-axis pointing down, and the extent is $[0, 0] \times [\text{width}, \text{height}]$. See Fig. 4. So we need to further convert the point (x', y') from the projected camera space to image space. Looking at Fig. 4,
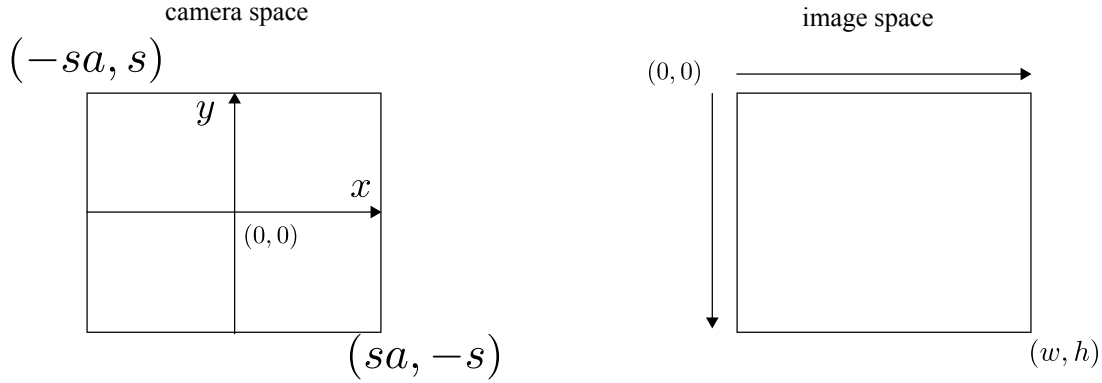
Figure 4: The different coordinate systems between our camera space and image space.

we can see that the we need to translate the origin, scale the axes, and flip the y axis. We will give you the formula for the x-axis, and you should figure out the y-axis formula yourself:

$$x'' = w\frac{x' + sa}{2sa}, \tag{3}$$

where w is the width of the image.

The second complication is that a triangle vertex can be behind the camera i.e., $-z < 0$. In fact, even if z is very close to zero, it can be still problematic: recall that in Eq. (1), we need to divide by $-z$ for the projection. When z is very close to zero, the division can be unstable due to limited floating point precision. Hence, we consider all points such that $-z < z_{\text{near}}$ to be behind the camera. z_{near} is usually called the *near clipping plane*.

Dealing with triangles with only one or two vertices behind the camera, and the other in front of the camera is tricky: we will need to implement *clipping* (described below as a bonus). Instead, in this homework, we only require you only render the triangles where all three vertices are in front of the near plane. If even one of the vertices of a triangle is behind the near clipping plane, you should reject the triangle and not render it.

We'll use the same antialiasing scheme we used in Homework 1, i.e., a 4×4 supersampling.

Implement your triangle rendering in `hw_2_1()` in `hw2.cpp`. To test your rendering, use the following commands:

```
./balboa -hw 2_1 -s 1 -p0 -1 -1 -3 -p1 0 1 -3 -p2 1 -1 -3 -color 0.7 0.3 0.4 -znear 1e-6
./balboa -hw 2_1 -s 1 -p0 -1 0 -2 -p1 0 2 -4 -p2 1 -2 -5 -color 0.7 0.3 0.8 -znear 1e-6
./balboa -hw 2_1 -s 1 -p0 2 2 -1 -p1 -1 -1 -2 -p2 2 0 -3 -color 0.3 0.8 0.2 -znear 1e-6
./balboa -hw 2_1 -s 1 -p0 200 200 -100 -p1 1 3 -4 -p2 -2 -2 -2 -color 0.8 0.1 0.3 -znear 1e-6
```

We provide our rendering of the first command in Fig. 6.

In your submission, save the images generated by the second to fourth commands as

```
outputs/hw_2_1_1.png
outputs/hw_2_1_2.png
outputs/hw_2_1_3.png
```

We'll compare your output with our rendering for grading.

Bonus: triangle clipping (15 pts). In practice, instead of rejecting a triangle if one or two verices are behind the near clipping plane, graphics pipelines would implement triangle clipping (Fig. 5). As a bonus, you will implement the clipping of the triangles and render them correctly even when some vertices are behind the near clipping plane. To speed up the computation, clipping is also often done with the entire 3D
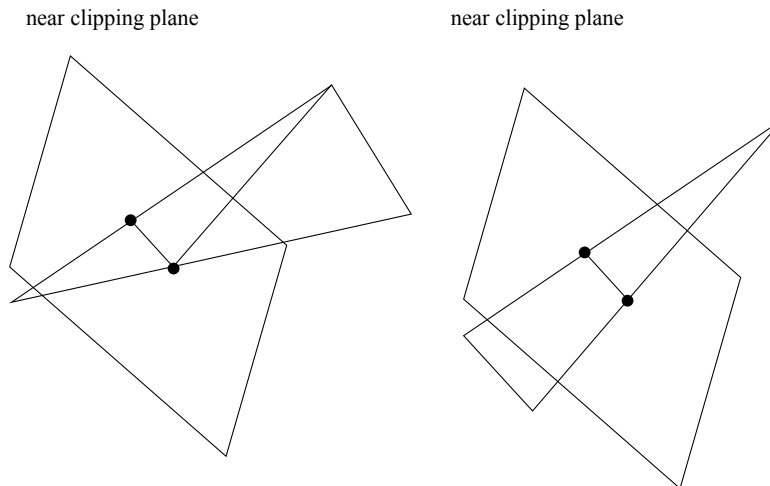
Figure 5: If one or two vertices of a triangle is behind the near clipping plane, then (ideally) we would need to clip the triangle. If only one vertex is behind the clipping plane (left), then we need to clip and further split the resulting quadliteral into two triangles. If two vertices are behind the clipping plane (right), the clipped triangle is still a triangle.

view frustum, discarding any triangles that are outside of the screen or too far away from the camera. We don't do it in this homework.

2 Rendering a triangle mesh (20 pts)

Next, we'll extend our previous code to handle more than one triangle. In computer graphics, we often store these triangles in a data structure called "triangle mesh" (Fig. 7). Triangle mesh is a more efficient data structure than simply storing a list of triangles since many of the triangles would share vertices in graphics (really? when will it be better and when will it not be better?). To render triangle meshes, you need to go through the face array and loop through all the triangles.

An important difference, compared to the list of triangles we have in Homework 1, is to determine the depth order between all these triangles. That is, for a point inside a pixel, we need to decide which triangle is the one we actually see. The tricky part is, unlike Homework 1, it is impossible to globally sort the triangles to determine a depth order (Fig. 8).

Therefore, we need to know the depth value of all the triangles overlapping with the point in the pixel. That is, given a image space point inside a projected triangle, we need to *interpolate* from the depth values of the three vertices. This turns out to be slightly involved mathematically (code-wise it's actually not that much though, my implementation of the depth interpolation takes around 70 lines including lots of comments).

To explain how to find the desired depth value, we need to explain what are barycentric coordinates. Any point $\mathbf{p}$ inside a triangle with three vertices $\mathbf{p}_0$, $\mathbf{p}_1$, $\mathbf{p}_2$ can be expressed using linear combination of the three vertices:

$$\mathbf{p} = b_0\mathbf{p}_0 + b_1\mathbf{p}_1 + b_2\mathbf{p}_2, \tag{4}$$

where $b_0 + b_1 + b_2 = 1$, $b_0 \geq 0$, $b_1 \geq 0$, and $b_2 \geq 0$. Given a point $\mathbf{p}$, we can compute the **unique** coefficients
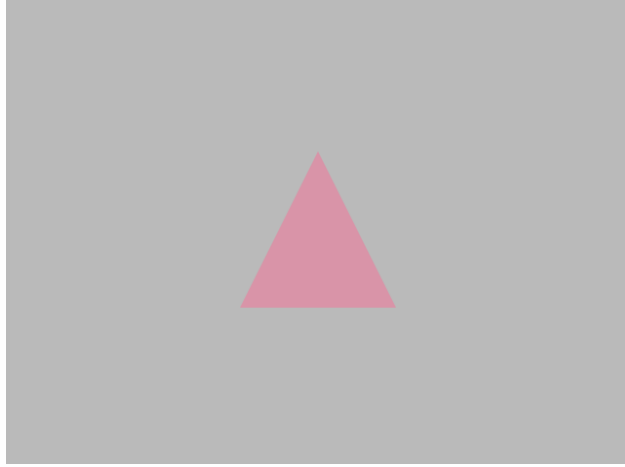
Figure 6: Reference rendering for homework 2.1.

using Cramer's rule:

$$b_0 = \frac{\text{area}\left(\mathbf{p}, \mathbf{p}_1, \mathbf{p}_2\right)}{\text{area}\left(\mathbf{p}_0, \mathbf{p}_1, \mathbf{p}_2\right)}$$
$$b_1 = \frac{\text{area}\left(\mathbf{p}_0, \mathbf{p}, \mathbf{p}_2\right)}{\text{area}\left(\mathbf{p}_0, \mathbf{p}_1, \mathbf{p}_2\right)} \tag{5}$$
$$b_2 = \frac{\text{area}\left(\mathbf{p}_0, \mathbf{p}_1, \mathbf{p}\right)}{\text{area}\left(\mathbf{p}_0, \mathbf{p}_1, \mathbf{p}_2\right)}$$

To get the area of a triangle, you can use the length of the cross product between two of the edge vectors, which will give you the area of the parallelogram, you can then divide the parallelogram area by two. This works also for 2D triangles, you only need to pretend the 2D triangle is on a 3D plane (e.g., $z = 0$).

As illustrated in Fig. 9, our goal is, given an image plane point $\mathbf{p}'$ and a triangle with three vertices $\mathbf{p}_0$, $\mathbf{p}_1$, and $\mathbf{p}_2$, figure out the z coordinate of the corresponding point $\mathbf{p}$. We want to use barycentric coordinates to help us. So we need to figure out what b_0, b_1, and b_2 are, so that we can interpolate the z coordinates from $\mathbf{p}_0$, $\mathbf{p}_1$, and $\mathbf{p}_2$. Unfortunately, we cannot directly use Eq. (5), since we don't know $\mathbf{p}$ (otherwise we would have known the answer already!).

What we do know are the original vertices $\mathbf{p}_0$, $\mathbf{p}_1$, and $\mathbf{p}_2$, projected vertices $\mathbf{p}'_0$, $\mathbf{p}'_1$, and $\mathbf{p}'_2$, and our image plane point $\mathbf{p}'$. Furthermore, we know that $\mathbf{p}$ projects to $\mathbf{p}'$ and $\mathbf{p}_i$ projects to $\mathbf{p}'_i$. Given the projected vertices and the image plane point, we can compute the projected barycentric coordinates b'_0, b'_1, b'_2 using Eq. (5). We want to relate the projected barycentric coordinates with the original ones.

Since $\mathbf{p}$ projects to $\mathbf{p}'$, we know that

$$\mathbf{p}' = \left(-\frac{\mathbf{p}.x}{\mathbf{p}.z}, -\frac{\mathbf{p}.y}{\mathbf{p}.z}, -1\right) = -\frac{\mathbf{p}}{\mathbf{p}.z}. \tag{6}$$

Plugging in $\mathbf{p} = b_0\mathbf{p}_0 + b_1\mathbf{p}_1 + b_2\mathbf{p}_2$ and $\mathbf{p}.z = b_0\mathbf{p}_0.z + b_1\mathbf{p}_1.z + b_2\mathbf{p}_2.z$:

$$\mathbf{p}' = -\frac{b_0\mathbf{p}_0 + b_1\mathbf{p}_1 + b_2\mathbf{p}_2}{b_0\mathbf{p}_0.z + b_1\mathbf{p}_1.z + b_2\mathbf{p}_2.z}. \tag{7}$$

Next, we know that $\mathbf{p}_0$ projects to $\mathbf{p}'_0$, $\mathbf{p}_1$ projects to $\mathbf{p}'_1$, and $\mathbf{p}_2$ projects to $\mathbf{p}'_2$:

$$-\left(\mathbf{p}_0.z\right)\left(\mathbf{p}'_0\right) = \mathbf{p}_0$$
$$-\left(\mathbf{p}_1.z\right)\left(\mathbf{p}'_1\right) = \mathbf{p}_1 \tag{8}$$
$$-\left(\mathbf{p}_2.z\right)\left(\mathbf{p}'_2\right) = \mathbf{p}_2.$$

Plugging in, we get

$$\mathbf{p}' = \frac{b_0\left(\mathbf{p}_0.z\right)\mathbf{p}'_0 + b_1\left(\mathbf{p}_1.z\right)\mathbf{p}'_1 + b_2\left(\mathbf{p}_2.z\right)\mathbf{p}'_2}{b_0\mathbf{p}_0.z + b_1\mathbf{p}_1.z + b_2\mathbf{p}_2.z}. \tag{9}$$

5

vertices:
0: (0, 1, 0)
1: (1, 0, 0)
2: (0, 0, 1)
3: (-1, 0, 0)

faces:
0: (0, 1, 2)
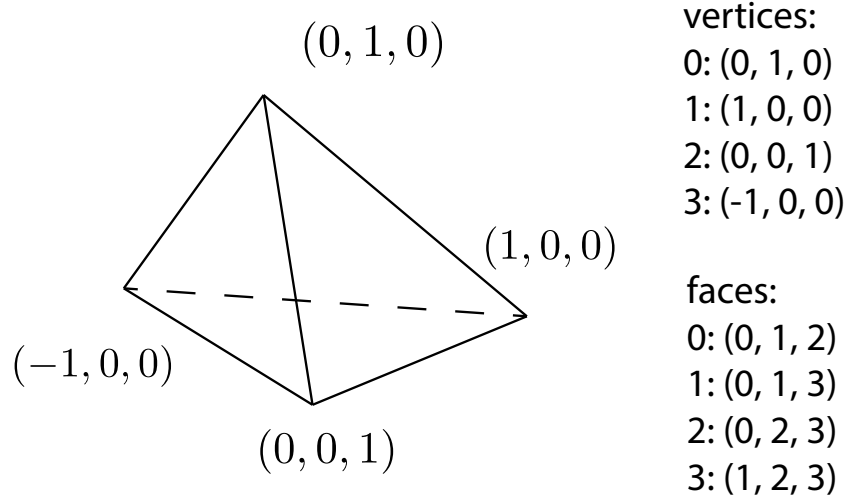1: (0, 1, 3)
2: (0, 2, 3)
3: (1, 2, 3)

Figure 7: A triangle mesh contains a list of vertices (3D positions) and a list of faces (3 integers pointing towards the index of the vertices). The figure shows a case of a triangle mesh with 4 vertices and 4 faces/-triangles. The first face represents the triangle on the right side, the second face represents the triangle at the back, the third face represents the triangle on the left, and the last face represents the triangle at the bottom.

Compare the above with:

$$\mathbf{p}' = b_0'\mathbf{p}_0' + b_1'\mathbf{p}_1' + b_2'\mathbf{p}_2', \tag{10}$$

using the uniqueness of barycentric coordinates, we have:

$$
\begin{aligned}
b_0' &= \frac{b_0\mathbf{p}_0.z}{b_0\mathbf{p}_0.z + b_1\mathbf{p}_1.z + b_2\mathbf{p}_2.z} \\
b_1' &= \frac{b_1\mathbf{p}_1.z}{b_0\mathbf{p}_0.z + b_1\mathbf{p}_1.z + b_2\mathbf{p}_2.z} \\
b_2' &= \frac{b_2\mathbf{p}_2.z}{b_0\mathbf{p}_0.z + b_1\mathbf{p}_1.z + b_2\mathbf{p}_2.z}.
\end{aligned}
\tag{11}
$$

We are almost there: we now want to express b_0, b_1, and b_2 using b_0', b_1', and b_2'. Let $B = b_0\mathbf{p}_0.z + b_1\mathbf{p}_1.z + b_2\mathbf{p}_2.z$, we have:

$$
\begin{aligned}
b_0 &= \frac{b_0'B}{\mathbf{p}_0.z} \\
b_1 &= \frac{b_1'B}{\mathbf{p}_1.z} \\
b_2 &= \frac{b_2'B}{\mathbf{p}_2.z}
\end{aligned}
\tag{12}
$$

Furthermore, we know that $b_0 + b_1 + b_2 = 1$, so:

$$B = \frac{1}{\frac{b_0'}{\mathbf{p}_0.z} + \frac{b_1'}{\mathbf{p}_1.z} + \frac{b_2'}{\mathbf{p}_2.z}}. \tag{13}$$
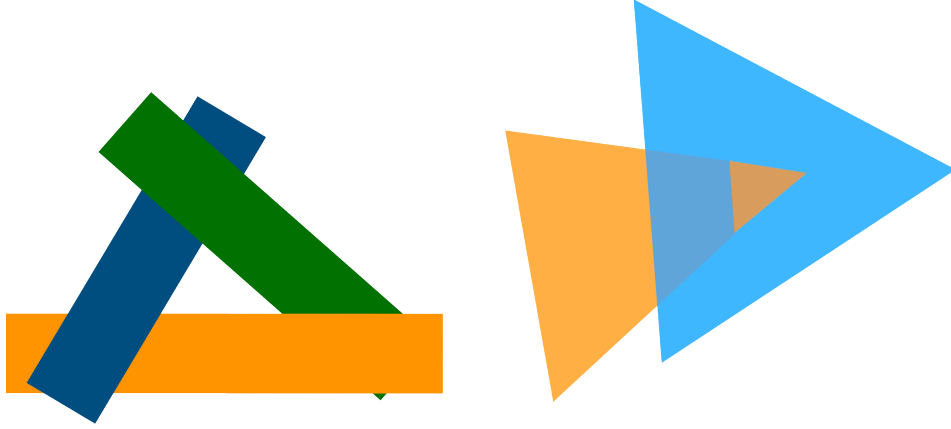
6

Figure 8: We cannot globally sort all objects into a single order by depth. The objects can form a cyclic order (left) or they can have interpenetration (right).

Rewriting Eq. (11) using the equation above, we have:

$$\frac{\frac{b'_0}{\mathbf{p}_0.z}}{\frac{b'_0}{\mathbf{p}_0.z} + \frac{b'_1}{\mathbf{p}_1.z} + \frac{b'_2}{\mathbf{p}_2.z}} = b_0$$

$$\frac{\frac{b'_1}{\mathbf{p}_1.z}}{\frac{b'_0}{\mathbf{p}_0.z} + \frac{b'_1}{\mathbf{p}_1.z} + \frac{b'_2}{\mathbf{p}_2.z}} = b_1. \tag{14}$$

$$\frac{\frac{b'_2}{\mathbf{p}_1.z}}{\frac{b'_0}{\mathbf{p}_0.z} + \frac{b'_1}{\mathbf{p}_1.z} + \frac{b'_2}{\mathbf{p}_2.z}} = b_2$$

The equation above has an intuitive meaning: **the original barycentric coordinates can be obtained by the projected barycentric coordinate weighted by inverse depth**. We will use these equations again later in the homework.

Having the barycentric coordinates, we can finally get the desired depth:

$$\mathbf{p}.z = b_0\mathbf{p}_0.z + b_1\mathbf{p}_1.z + b_2\mathbf{p}_2.z. \tag{15}$$

To recap: given an image plane point and a triangle, we first convert it from screen space to camera space (using the inverse of Eq. (3), recall Fig. 4). Then, we compute the projected barycentric coordinates b'_0, b'_1, and b'_2 using Eq. (5). Using the projected barycentric coordinates and the depth at the three vertices, we convert them to the original barycentric coordinates using Eq. (14). Finally, we obtain the depth using Eq. (15) and use the depth to pick the triangle that is the closest to the pixel sample, but in front of the clipping plane.

One final detail, remember we discussed two variants for rendering multiple objects in a scene? The depth testing makes the two variants differ more. For the *ray tracing* style (i.e., for each pixel, check all triangles), you need to maintain a minimal z value when traversing all the triangles. For the *rasterization* style (i.e., for each triangle, check all pixels), you need to maintain a whole image of z values, and this is usually called the *Z-buffer*.

```
# For each pixel, check all triangles
for each pixel:
    z_min = infinity
    for each triangle:
        # check if the pixel center hits the triangle
        # overwrite color and z_min if the triangle is closer
```
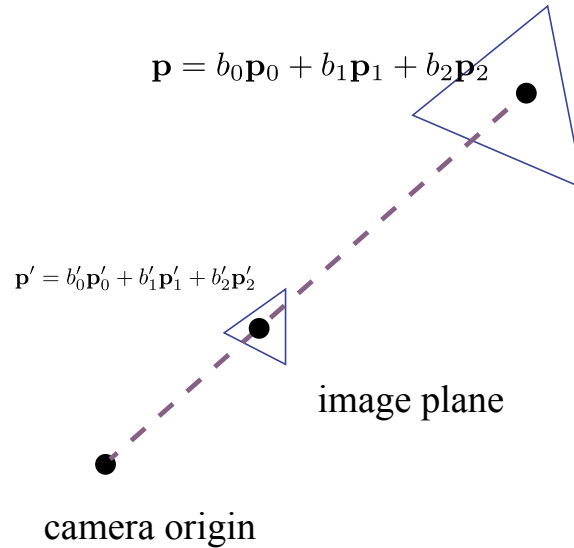
$$\mathbf{p} = b_0\mathbf{p}_0 + b_1\mathbf{p}_1 + b_2\mathbf{p}_2$$

$$\mathbf{p}' = b'_0\mathbf{p}'_0 + b'_1\mathbf{p}'_1 + b'_2\mathbf{p}'_2$$

image plane

camera origin

Figure 9: Given a point on image plane p', and a triangle with three vertices $\mathbf{p}_0$, $\mathbf{p}_1$, $\mathbf{p}_2$, we want to know the depth, i.e., the z coordinate at the corresponding point $\mathbf{p}$.

```
# For each triangle, check all pixels
Z_buffer = Image(w, h)
for each triangle:
    # project the triangle
    for each pixel:
        # check if the pixel center hits the triangle
        # overwrite color and Z_buffer[pixel] if the triangle is closer
```

We will discuss the pros and cons of the two approaches in depth in the lectures.

Now you should be ready to implement the rendering of a triangle mesh! In this part, to specify the color of the triangles, in our triangle mesh, we further store a color per triangle.

Our `TriangleMesh` data structure looks like this:

```
struct TriangleMesh {
    std::vector<Vector3> vertices; // 3D positions of the vertices
    std::vector<Vector3i> faces; // indices of the triangles
    std::vector<Vector3> face_colors; // per-face color of the mesh, only used in HW 2.2
    std::vector<Vector3> vertex_colors; // per-vertex color of the mesh, used in HW 2.3 and later
    Matrix4x4 model_matrix; // used in HW 2.4
};
```

`faces` and `face_colors` will always be the same length.

Go implement your triangle mesh rendering code in `hw_2_2`. To test your implementation, use the command

```
./balboa -hw 2_2 -s 1 -znear 1e-6 -scene_id 0
./balboa -hw 2_2 -s 1.5 -znear 1e-6 -scene_id 1
./balboa -hw 2_2 -s 0.4 -znear 1e-6 -scene_id 2
./balboa -hw 2_2 -s 0.5 -znear 1e-6 -scene_id 3
```

where `[scene_id]` is the scene you want to render (0-3).

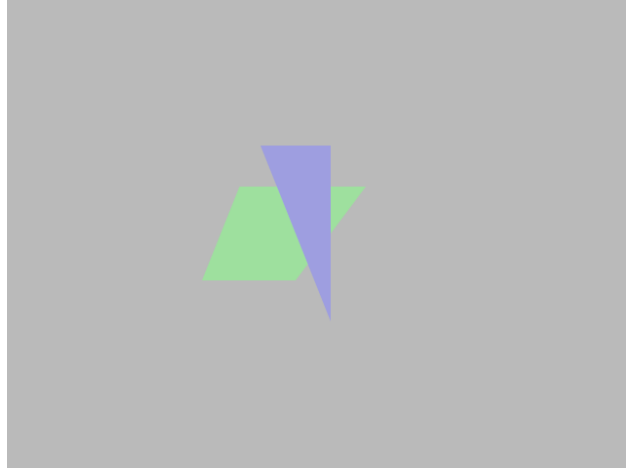Our rendering for `scene_id` 0 is in Fig. 10.

Figure 10: Reference rendering for homework 2.2.

In your submission, save your rendering of `scene_id=1-3` as

```
outputs/hw_2_2_1.png
outputs/hw_2_2_2.png
outputs/hw_2_2_3.png
```

We'll compare your output with our rendering for grading.

Bonus: occlusion culling (20 pts). During rasterization-style rendering, if we can prove that a triangle is completely blocked by some other triangles (using the current Z buffer), we can completely skip the rendering of the triangle to speed up our rendering. This is called occlusion culling or Z-culling. Read the paper Hierarchical Z-Buffer Visibility from Greene et al. and implement occlusion culling using a hierarchical Z buffer. Our test scenes do not have heavy occlusion, so you will need to design your own scenes to show the speedup.

3 Perspective-corrected interpolation (20 pts)

So far, we have been rendering constant color triangles. It's time to make things more colorful. Instead of specifying *face colors*, we'll specify *vertex colors*, and interpolate them using barycentric coordinates.

It might be tempting to use the projected barycentric coordinates $(b'_0, b'_1$ and $b'_2)$ to interpolate the vertex colors, but this is incorrect: our triangle would be changing color based on the vertex depth if we do this (try it!). Instead, we want to interpolate using the original barycentric coordinates b_0, b_1, and b_2. Fortunately, we already know how to get them using Eq. (14)! After computing the original barycentric coordinates, we'll just interpolate the vertex colors C_0, C_1, C_2 defined at the triangle vertices:

$$C = b_0 C_0 + b_1 C_1 + b_2 C_2. \tag{16}$$

This is what people mean if they say they are doing *perspective-corrected interpolation* in a renderer.

That's all you need to know to implement vertex color rendering! (I hope it's easier than the previous two.) Go and implement it in `hw_2_3`. Test it using

```
./balboa -hw 2_3 -s 1 -znear 1e-6 -scene_id 0
./balboa -hw 2_3 -s 1.5 -znear 1e-6 -scene_id 1
./balboa -hw 2_3 -s 0.4 -znear 1e-6 -scene_id 2
./balboa -hw 2_3 -s 0.5 -znear 1e-6 -scene_id 3
```

As usual, our rendering for `scene_id 0` is in Fig. 11.

Figure 11: Reference rendering for homework 2.3.

As usual, save your outputs for `scene_id=1-3` as following.

```
outputs/hw_2_3_1.png
outputs/hw_2_3_2.png
outputs/hw_2_3_3.png
```

4 3D transformation (20 pts)

So far we have assumed that everything is in the camera space and camera is always located at $(0, 0, 0)$ facing negative z with up vector y. Let's add some transformations so that things are less restricted. Remember that in the 2D case in Homework 1, we use a 3×3 matrix to represent affine transformations. Here, we will use a 4×4 matrix. Like in 2D, we will support scaling, translation, and rotation (shearing is also possible, but we want to safe you some typing). Furthermore, we will support two kinds of new transformations: lookAt transform and perspective transform that we will talk about soon. The perspective transform implementation is optional, but implementing it might helpful understand parts of the next homework.
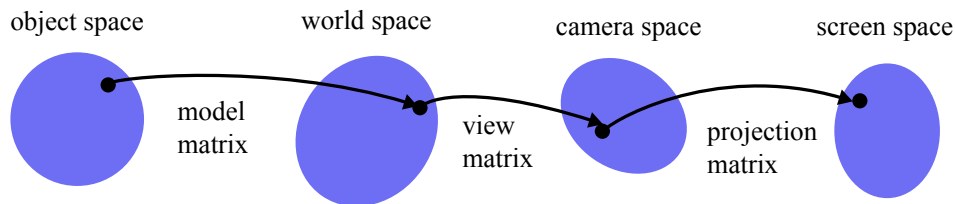

Figure 12: Spaces and the transformations between them in 3D.

Before we explain the 3D transformations, we need to explain our spaces in 3D. Previously in 2D, we only had *object space* and *screen space*. The existence of the camera makes things slightly more complicated: the camera lives in the *camera space* (the coordinate system in the previous parts), and the matrices now transform the objects into *world space*. As a convention, we usually call the object-to-world transformation the *model matrix* M, the world-to-camera transformation the *view matrix* V, the camera-to-screen transformation the *projection matrix* P (yes, we can express the projection as a matrix! more on this later). Given a vertex v on a 3D triangle mesh, we can chain the transformations to project it onto the screen space: $v' = PVMv$ (unfortunately, the combined matrix PVM is usually called the "MVP" matrix). Fig. 12 illustrates this.

Scaling and translation. Scaling and translations are basically the same as in 2D:

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \tag{17}$$

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \tag{18}$$

Rotation. Rotation in 3D is a lot more complicated. Unlike 2D, which we pretty much only have two ways to rotate (clockwise or counterclockwise), in 3D there are infinitely many ways to rotate. We will talk about a lot of them in the lectures, but for this homework we will focus on rotating over a given axis $\mathbf{a}$ by θ degree. This is usually called an Axis/Angle representation of rotation. The following derivation is taken from the book Physically-based Rendering: From Theory to Implementation, which is based on the famous Rodrigues' rotation formula. Also see this excellent article from Max Slater if you want to read more.
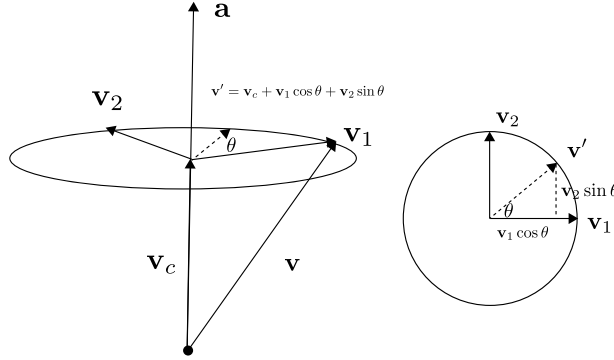

Figure 13: To rotate a vector $\mathbf{v}$ about an axis $\mathbf{a}$, we construct a plane with normal $\mathbf{a}$ with coordinate basis $\mathbf{v}_1$ and $\mathbf{v}_2$. The rotation can then be expressed as $\mathbf{v}' = \mathbf{v}_c + \cos\theta\mathbf{v}_1 + \sin\theta\mathbf{v}_2$.

Our idea is to construct a coordinate system around the axis $\mathbf{a}$, then apply trigonometric identities in the 2D rotation plane. Instead of directly constructing the rotation matrix, it's easier to first derive what happens after we rotate a vector $\mathbf{v}$. Fig. 13 illustrates the geometry: we first project the vector $\mathbf{v}$ to axis $\mathbf{a}$:

$$\mathbf{v}_c = (\mathbf{v} \cdot \mathbf{a})\,\mathbf{a}. \tag{19}$$

Next, we want to construct a coordinate system at $\mathbf{v}_c$. We choose the first axis to be:

$$\mathbf{v}_1 = \mathbf{v} - \mathbf{v}_c. \tag{20}$$

The next axis $\mathbf{v}_2$ needs to be perpendicular to $\mathbf{v}_1$, so:

$$\mathbf{v}_2 = \mathbf{a} \times \mathbf{v}_1. \tag{21}$$

With some trigonometry, we can then derive the rotated vector $\mathbf{v}'$:

$$\mathbf{v}' = \mathbf{v}_c + \cos\theta\mathbf{v}_1 + \sin\theta\mathbf{v}_2. \tag{22}$$

(Note that our formulas are slightly different from the textbook above, since we use a right-handed coordinate system instead of a left-handed one.)

We can do a quick sanity test: let $\mathbf{a}$ be the z axis. We have $\mathbf{v}_c = (0, 0, \mathbf{v}.z)$, $\mathbf{v}_1 = (\mathbf{v}.x, \mathbf{v}.y, 0)$, and $\mathbf{v}_2 = (-\mathbf{v}.y, \mathbf{v}.x, 0)$. Thus $\mathbf{v}' = (\cos\theta\mathbf{v}.x - \sin\theta\mathbf{v}.y, \cos\theta\mathbf{v}.x + \sin\theta\mathbf{v}.y, \mathbf{v}.z)$. We've recovered the standard 2D rotation.

To obtain the rotation matrix, we can plug in $\mathbf{v} = (1, 0, 0)$ to get the first column of the matrix, $\mathbf{v} = (0, 1, 0)$ to get the second column, and $\mathbf{v} = (0, 0, 1)$ to get the third column. Specifically, the first column of the 4×4 rotation matrix R is given by:

$$\begin{bmatrix} \mathbf{a}.x\mathbf{a}.x + (1 - \mathbf{a}.x\mathbf{a}.x)\cos\theta \\ \mathbf{a}.x\mathbf{a}.y(1 - \cos\theta) + \mathbf{a}.z\sin\theta \\ \mathbf{a}.x\mathbf{a}.z(1 - \cos\theta) - \mathbf{a}.y\sin\theta \\ 0 \end{bmatrix}. \tag{23}$$

The second column is

$$\begin{bmatrix} \mathbf{a}.y\mathbf{a}.x(1 - \cos\theta) - \mathbf{a}.z\sin\theta \\ \mathbf{a}.y\mathbf{a}.y + (1 - \mathbf{a}.y\mathbf{a}.y)\cos\theta \\ \mathbf{a}.y\mathbf{a}.z(1 - \cos\theta) + \mathbf{a}.x\sin\theta \\ 0 \end{bmatrix}. \tag{24}$$

The third column is

$$\begin{bmatrix} \mathbf{a}.z\mathbf{a}.x(1 - \cos\theta) + \mathbf{a}.y\sin\theta \\ \mathbf{a}.z\mathbf{a}.y(1 - \cos\theta) - \mathbf{a}.x\sin\theta \\ \mathbf{a}.z\mathbf{a}.z + (1 - \mathbf{a}.z\mathbf{a}.z)\cos\theta \\ 0 \end{bmatrix}. \tag{25}$$
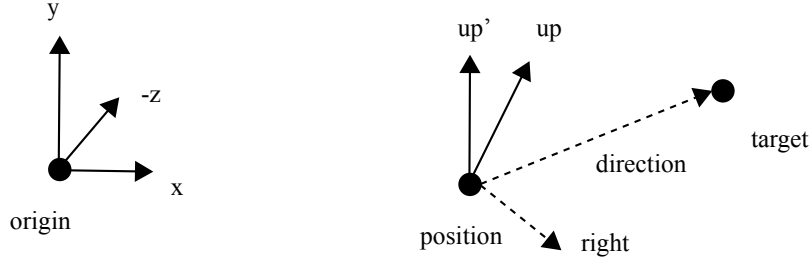

Figure 14: The LookAt transform maps a coordinate frame based on the parameters `position`, `target`, and `up`. $-z$ axis is mapped to the `direction` of `target - position`, x axis is mapped to the cross product of `direction` and `up`, and y axis is mapped to the cross product of `right` and `direction`.

LookAt. The LookAt transform is useful for positioning cameras. The LookAt transform takes 2 3D points and a 3D vector as parameters: the `position` of the object $\mathbf{p}$, the `target` the object is looking at $\mathbf{t}$, and the `up` vector $\mathbf{u}$ that describe the orientation of the object. Fig. 14 illustrates the geometry.

Given the input, we first compute where the camera should be facing using $\mathbf{p}$ and $\mathbf{t}$:

$$\mathbf{d} = \text{normalize}(t - p). \tag{26}$$

Given the `direction` $\mathbf{d}$, we can then compute the `right` vector $\mathbf{r}$ using cross product with the given `up` vector $\mathbf{u}$:

$$\mathbf{r} = \text{normalize}\left(\mathbf{d} \times \mathbf{u}\right). \tag{27}$$

We are not done yet though. Since the up vector $\mathbf{u}$ is not necessarily perpendicular to the camera direction $\mathbf{d}$, we do not have an orthonormal basis yet. Thus we recompute the up vector using the cross product between the right vector and the camera direction:

$$\mathbf{u}' = \mathbf{r} \times \mathbf{d}. \tag{28}$$

Given these information, we can build our transformation matrix:

$$L = \begin{bmatrix} \mathbf{r}.x & \mathbf{u}'.x & -\mathbf{d}.x & \mathbf{p}.x \\ \mathbf{r}.y & \mathbf{u}'.y & -\mathbf{d}.y & \mathbf{p}.y \\ \mathbf{r}.z & \mathbf{u}'.z & -\mathbf{d}.z & \mathbf{p}.z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{29}$$

The first column sends x axis $((1,0,0))$ to $\mathbf{r}$, the second column sends y axis $((0,1,0))$ to $\mathbf{u}'$, the third column sends z axis to $\mathbf{d}$, and the last column translates the coordinate systems by $\mathbf{p}$.

Note that when applied to cameras, the matrix L is designed to transform from camera space to world space. When constructing the view matrix (Fig. 12), we need to go from world space to the camera space.

Perspective projection. Now we are at the coolest part. It turns out that with some math tricks, we can turn the perspective projection we did in the first part (Fig. 3) into a matrix multiplication as well! The trick is to introduce something called the homogeneous coordinates. The idea is to make use of the 4th component of the vector. So far, we always assumed that the 4th component to be either 1 or 0 (when it's 0, it's a vector instead of a point). From now on, our vectors can have arbitrary 4th component:

$$\begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}. \tag{30}$$

The clever part is how we obtain the 3D vector from the 4D vector above: we define everything to be equivalent up to an arbitrary non-zero scaling:

$$\begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} = \begin{bmatrix} \frac{x}{w} \\ \frac{y}{w} \\ \frac{z}{w} \\ 1 \end{bmatrix}. \tag{31}$$

Following this definition, we can implement Eq. (1) using the following matrix multiplication:

$$\begin{bmatrix} x' \\ y' \\ z' \\ w' \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}. \tag{32}$$

Notice that

$$\begin{bmatrix} x' \\ y' \\ z' \\ w' \end{bmatrix} = \begin{bmatrix} x \\ y \\ 1 \\ -z \end{bmatrix} = \begin{bmatrix} -\frac{x}{z} \\ -\frac{y}{z} \\ -\frac{1}{z} \\ 1 \end{bmatrix}. \tag{33}$$

A bonus we get is that the new z' component becomes the reciprocal of $-z$: we can directly use it for getting the barycentric coordinates (Eq. (14))!

Remember that the projection above only projects within the camera space. We need to further convert to screen space (Eq. (3)). I'll let you figure out how the matrix should look like!

Now you should have everything you need to complete this homework.

Like Homework 1, we will describe the camera transformation and the meshes in a JSON scene file.

A scene file would look like this:

```
{
    "camera":
    {
        "resolution": [640,480],
        "transform": [
            {
                "lookat":
                {
                    "position": [0,1,0],
                    "target": [0,0,-5],
                    "up": [0,1, 0]
```

```
                    }
                }
            ],
            "s": 1,
            "z_near": 1e-6
        },
        "background": [
            0.5, 0.5, 0.5
        ],
        "objects": [
            {
                "vertices": [
                    -1.7,  1.0, -5.0,
                     1.0,  1.0, -5.0,
                    -0.5, -1.0, -5.0,
                    -2.5, -1.0, -5.0,
                    -1.0,  1.5, -4.0,
                     0.2,  1.5, -4.0,
                     0.2, -1.5, -4.0
                ],
                "faces": [
                    0, 1, 2,
                    0, 2, 3,
                    4, 5, 6
                ],
                "vertex_colors": [
                    0.75, 0.35, 0.35,
                    0.35, 0.75, 0.35,
                    0.35, 0.35, 0.75,
                    0.75, 0.35, 0.75,
                    0.35, 0.75, 0.75,
                    0.75, 0.35, 0.75,
                    0.75, 0.75, 0.35
                ],
                "transform": [
                    {
                        "rotate": [45,1,1, 1]
                    }
                ]
            }
        ]
}
```

Instead of directly specifying the vertices/faces in the JSON file, we can also use the Stanford PLY format:

```
{
    "camera":
    {
        "resolution": [640,480],
        "transform": [
            {
                "lookat":
                {
```

```
                    "position": [1,1,0],
                    "target": [0,0,-5],
                    "up": [0,1, 0]
                }
            }
        ],
        "s": 1,
        "z_near": 1e-6
    },
    "background": [
        0.5, 0.5, 0.5
    ],
    "objects": [
        {
            "filename": "tetrahedron.ply",
            "transform": [
                {"scale": [1.5, 0.8, 0.7]},
                {"translate": [0.5, -0.3, 0.2]}
            ]
        }
    ]
}
```

The PLY file is human readable (when in the "ascii" mode) and looks like this:

```
ply
format ascii 1.0
comment Created by Blender 3.4.1 - www.blender.org
element vertex 4
property float x
property float y
property float z
property float red
property float green
property float blue
element face 4
property list uchar uint vertex_indices
end_header
-0.483444 0.826069 -2.307457 0.75 0.35 0.35
1.152029 -1.111880 -1.928058 0.35 0.75 0.35
1.485437 0.880605 -2.239198 0.35 0.35 0.75
0.594299 0.143237 -1.466344 0.75 0.35 0.75
3 0 1 2
3 1 3 2
3 3 1 0
3 2 3 0
```

The scene will be parsed into the following scene data structure:

```
struct Camera {
    Matrix4x4 cam_to_world;
    Vector2i resolution;
    Real s;
    Real z_near;
};

struct Scene {
```

```
    Camera camera;
    Vector3 background;
    std::vector<TriangleMesh> meshes;
};
```

Like in Homework 1, your first task is to implement the transformations in `parse_transformation` in `hw2_scenes.cpp`. Next, you'll render the parsed scene in `hw_2_4` in `hw2.cpp`.

Important: the `transform` in the camera describes the transformation from camera to world, and that's what we store in `Camera::cam_to_world`. The "view matrix" in Fig. 12 is the inverse of `cam_to_world`.

To test your rendering, use the following commands:

```
./balboa -hw 2_4 ../scenes/two_shapes.json
./balboa -hw 2_4 ../scenes/tetrahedron.json
./balboa -hw 2_4 ../scenes/cube.json
./balboa -hw 2_4 ../scenes/prism.json
./balboa -hw 2_4 ../scenes/teapot.json
```

Our renderings of the `two_shapes` and `tetrahedron` scenes are shown in Fig. 15.
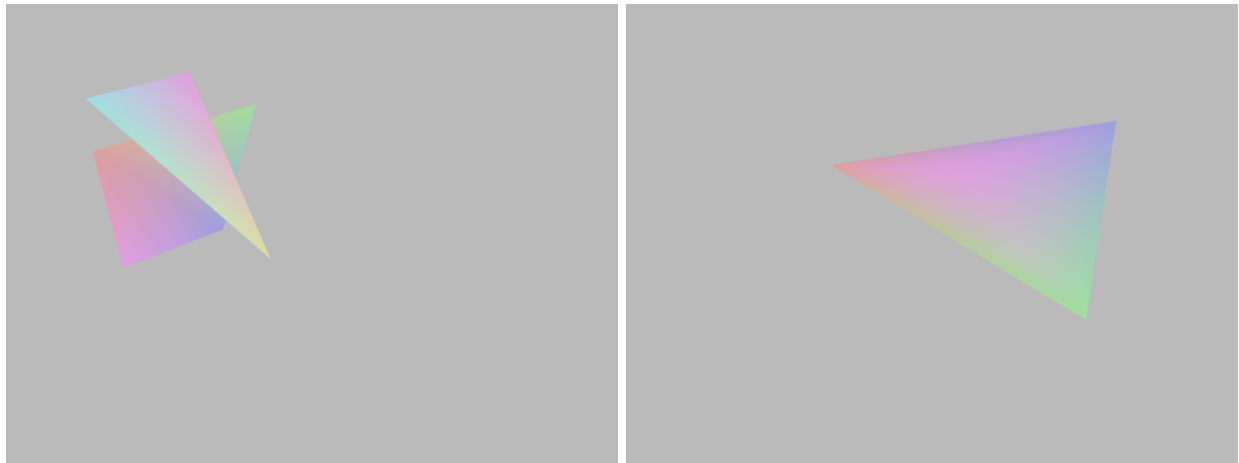

Figure 15: Reference renderings for homework 2.4.

Save your outputs for the scenes above as following.

```
outputs/hw_2_4_cube.png
outputs/hw_2_4_prism.png
outputs/hw_2_4_teapot.png
```

Note that the last teapot scene contains 1570 triangles, so it might take a bit of time to render it (it took 30 seconds in my implementation). By the way, the teapot is the famous Utah teapot made by Martin Newell when doing his Ph.D. at Utah in the early days of computer graphics.

Bonus (15 pts). Like Homework 1, generate an animation by interpolating between transformations. To interpolate between rotations, read the article from Max Slater. Feel free to modify any part of the code in balboa. You can use ffmpeg to convert a sequence of images into a video file.

5 Design your own scenes (20 pts)

As usual, design a scene yourself and submit the scene and your rendering to us. We will give extra credits to people who impress us. Feel free to download 3D model files on the internet (please credit the authors and let us know where you downloaded it). Note that our