Assignment 2: 3D Rasterization Pipeline

CMSC 23700: University of Chicago
Due: Wed January 28th at 11:59PM

Introduction

In this assignment, you will build up each piece of the graphics pipeline, and by the very end, have a full-fledged rasterization pipeline.

Your 3D geometry will now be given in the traditional OpenGL format, with separate arrays for vertex positions, triangles indexing into the vertex array, and either triangle colors or vertex colors (normalized 0-1). You should be able to re-use the coverage function you implemented from the previous assignment. **You should use the same bounding box acceleration from the previous assignment. You should NOT apply antialiasing in this assignment.**

Each question will be associated with its own function, and you should be able to copy (or turn repeated tasks into helper functions and reuse) the code you write into subsequent questions to eventually build up the pipeline. **Three of the questions also have an associated written math question, (see next page)** which you'll answer and submit in your `documentation.pdf`.

Note: We will not be implementing clipping in this assignment, so if you notice that any of your objects are rendered off-screen, then there is a bug in your solution. Also feel free to reuse/adapt the diff script we provided in assignment 1 for debugging purposes.

Textbook Reference

Fundamentals of Computer Graphics Chapter on Viewing (Ch.7 in 4th edition, Ch. 8 in 5th ed)

Code grading

We will be checking for pixelwise correctness of the returned arrays, and we will allow for a tolerance of up to 98% accurate to account for minor errors. In other words, if your solution is 98% correct, you will get full credit. **We also assign partial credit in our manual grading**, even if your solution is not autograder-perfect.

Required packages python >= 3.8, numpy >= 1.21, Pillow >= 10

Written problems

For this assignment, there are two short **written math questions**, corresponding to three of the main parts of this assignment/rasterizing pipeline. **Write your answers in `documentation.pdf`** (you can use TEX math notation if you like. You are also encouraged to draw diagrams to justify or derive your answers, for the diagrams requirement of **`documentation.pdf`**.)

These math problems can be solved by a careful reading of the textbook (chapter 7 in 4th ed., chapter 8 in 5th ed.) and thinking about the transforms involved. They are not meant to be complicated or tricky, and serve more like a 'guided reading' of the chapter, so that you gain an intuition of what the matrices practically *do* in addition to translating them from the textbook into code.

1. **Orthographic projection.** The orthographic projection matrix maps coordinates from the orthographic view volume (a cuboid) to the canonical view volume (a cube). Specifically, it maps the cuboid $[l, r] \times [b, t] \times [f, n]$ to the cube $[-1, 1] \times [-1, 1] \times [-1, 1]$ (where l, r, b, t, n, f are short for `left, right, bottom, top, near, far`).

   (a) The values you'll use in code are $l = 0, r = 12, b = 0, t = 12, f = 0, n = 12$. In general $n > f$ (`near` > `far`). Why is $n > f$ in these conventions?

   (b) Write out the **equations that transform each coordinate**, with a **brief justification or derivation** (1-2 lines or intermediate equations). Give an answer like

   $$x_{\text{canonical}} = (\text{something in terms of } x_{\text{ortho}})$$

   $$y_{\text{canonical}} = (\text{something in terms of } y_{\text{ortho}})$$

   $$z_{\text{canonical}} = (\text{something in terms of } z_{\text{ortho}})$$

   (Once you've written out these equations you should be able to see how the 4×4 matrix $\mathbf{M}_{\text{ortho}}$ is formed.)

2. **Perspective matrix P.** (This is an exercise from the textbook.) Show algebraically that applying the $\mathbf{P}$ matrix followed by the perspective divide preserves z-ordering. i.e. if $z_1 < z_2$ (where z_1 and z_2 have the same sign), then after applying $\mathbf{P}$ and applying the perspective divide to get z-coordinates z_1' and z_2', we still have $z_1' < z_2'$.

   Note that z' does not depend on x, y so you can consider just what happens to the z coordinate when $\mathbf{P}$ is applied followed by a perspective divide.

Submission Instructions

All of your submission code should be contained in **`assignment2_starter.py`**. Do not submit any other python files. Do not zip/tar your submission. Do not put the python file within a folder. Do not import external packages. In addition, submit a PNG file for your custom textured cube. Please remove any print statements before submitting (as this may cause the autograder to time out). Your submission should contain **only** three files: **`assignment2_starter.py`**, **`custom.png`**, **`documentation.pdf`**. Additionally, do not change the image export filenames from their starting names (so lines that save `p1.png`, `p2.png`, and so on must be left exactly with those names.)

If you are planning to use your late bank but are checking your assignment output using the autograder - put the following text in your **`documentation.pdf`** during submission testing: *Please do not grade this submission yet, I am using my late bank.*

Documentation

You should thoroughly and thoughtfully document your work in this project, **both in code** and in a **`documentation.pdf`** file you submit alongside the code and image.

- In **`documentation.pdf`**, document your high-level progress through the assignment: your initial planned approaches, any issues you ran into during implementation, and how you solved them.
- In **`documentation.pdf`**, draw and include an annotated diagram or sketch you use to derive your solution. Graphics algorithms are especially suited for the use of diagrams to aid understanding and debugging. **Please create and include at least one such diagram in `documentation.pdf` as you go through the assignment. Drawing by hand is fine. Diagrams from conversations with a TA/instructor are fine, but must be your own work.** Here are some examples of things you may want to visualize/sketch out for this assignment:
  - how to convert a "windowing transform" (scaling and translation of a rectangle to another rectangle) into a transformation matrix, and how this leads to the derivation of the orthographic projection matrix
  - the z-buffer method
  - the method you use to compute UV coordinates for the cube
  - the method to rasterize and interpolate UV coordinates with perspective correction

  There is no hard standard for how detailed such a diagram must be, but they should be sufficiently illuminating in order to have helped you implement the subproblem in question.
- In **`documentation.pdf`**, cite all resources, online articles, Q&A threads, etc. you use in your code. In line with the Generative AI policy on the course website, you must also cite any use of Generative AI involved:

  - You must include the questions you used and GenAI answers (e.g. a link to the saved conversation if applicable; a text paste of the conversation otherwise)
  - You may **not** use GenAI to describe your code or write any part of `documentation.pdf`
  - **You must only use GenAI for the permissible uses described in the Generative AI policy on the course website.**

- In **code comments**, thoroughly document nontrivial lines and leaps of logic. We encourage documenting the shapes of numpy arrays, even if they may be obvious in the moment. Comment thoughtfully so that you may recall what you did upon re-reading your code in six months; consider the thought process of a hypothetical reader who is in the course but has not done this assignment.

1 Viewport Matrix

Input: `cube` object, coords treated as if they were canonical volume coordinates.
Output: `p1.png`

We will work backwards and start from the viewport matrix $\mathbf{M}_{\text{vp}}$, assuming we already have the canonical volume coordinates. Once you get `render_viewport()` working you should see an output like the one in figure 1 (except figure 1 has the pixels padded out 10 times for visibility; **do not pad the pixels in your submission.**) This should be very similar to what you've already done in the 2D rasterization assignment.

You will go through the array of triangle faces and draw only their vertices using the associated face's **face color**.

Important: Instead of the textbook's matrix, **use the following $\mathbf{M}_{\text{vp}}$**:

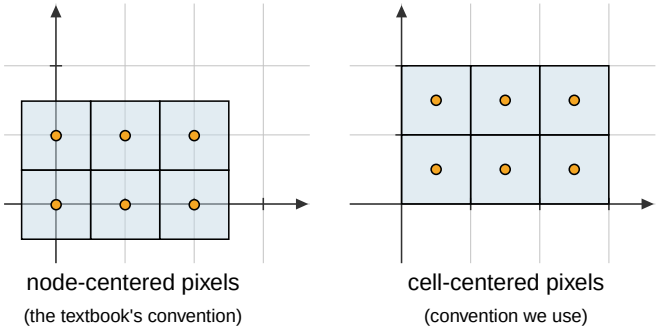
$$\mathbf{M}_{\text{vp}} = \begin{bmatrix} \frac{n_x}{2} & 0 & 0 & \frac{n_x}{2} \\ 0 & \frac{-n_y}{2} & 0 & \frac{n_y}{2} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

This matrix is derived from mapping the canonical square $[-1, 1] \times [-1, 1]$ to the screen-space rectangle $[0, n_x] \times [n_y, 0]$ (where n_x, n_y are the image width and height.)

The textbook's matrix is different, being derived from mapping $[-1, 1] \times [-1, 1]$ to $[-0.5, n_x - 0.5] \times [-0.5, n_y - 0.5]$ instead. This is because the textbook (as described in section 3.2, figure 3.10 in the textbook 4th ed.) uses a convention where a pixel "owns a unit square centered at integer coordinates". In contrast, notice that in assignment 1, we used a **cell-centered** convention instead. In this convention, as you probably implemented in assignment 1, the point in continuous space that represents a pixel (i.e. the point used for inside-triangle checks) is a $(+0.5, +0.5)$ offset from the pixel corner's integer coordinates.

In addition, we invert the y axis to make it point in the same direction as the row index into the image (i.e. $\mathsf{img}[i, j]$ goes down the image as i increases.)

Using the cell-centered convention means you should be able to reuse your bounding box and coverage method from assignment 1.



Also make sure you process the triangles in the exact order that they are given to get the right pixel colors. As mentioned above, the colors for this part are from `face_colors`, not `vertex_colors`.

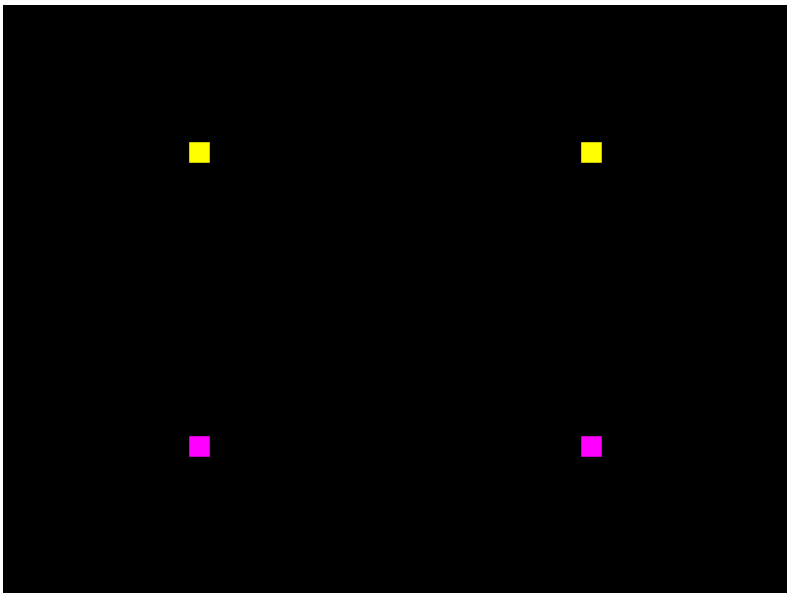


Figure 1: Expected output of `render_viewport()` (with each colored pixel padded out by 10 layers).

(As an aside: the cell-centered convention is the one used by OpenGL and other graphics libraries; also see https://www.realtimerendering.com/blog/the-center-of-the-pixel-is-0-50-5/.)

2 Orthographic Projection

Input: `ortho_cube` object
Output: `p2.png`

Next, build the orthographic projection matrix ($\mathbf{M}_{\text{ortho}}$) and render out the full projected cube. The parameters of the orthographic view volume are `left = 0.0; right = 12.0, bottom = 0.0, top = 12.0; near = 12.0, far = 0.0`. The expected output is shown in Figure 2.

Hint: Just as you likely did for assignment 1, remember to add the pixel center offset to the integer coordinates in order to get the pixel center point used for inside-triangle checks!

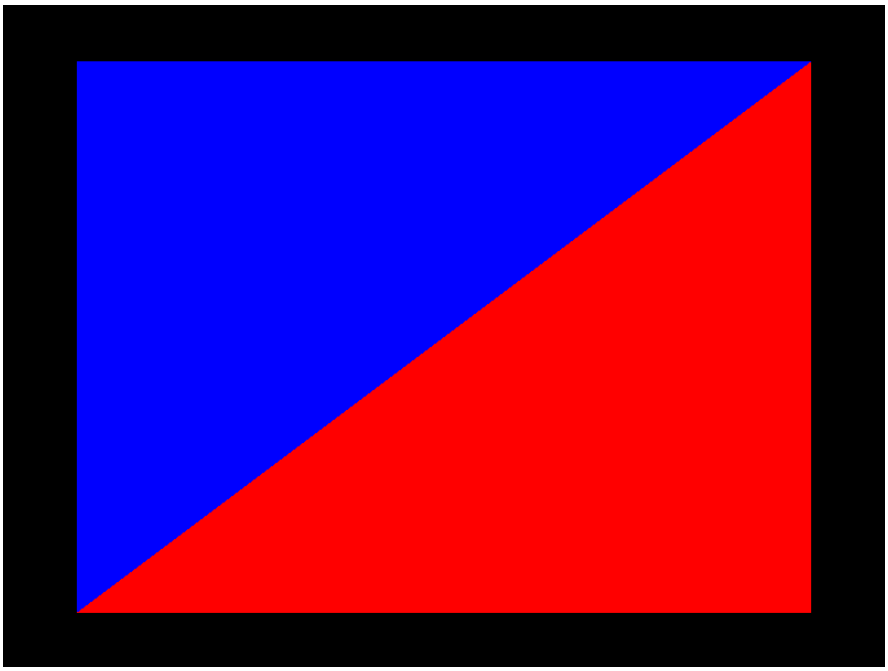


Figure 2: Expected output of `render_ortho()`.

3 Camera Projection

Input: `ortho_cube` object
Output: `p3.png`

Next, build the camera transformation matrix ($\mathbf{M}_{\text{cam}}$). The expected output is shown in Figure 3. The orthographic view volume parameters are the same as the previous question. The camera settings are `eye = (0.2, 0.2, 1), lookAt = (0, 0, 0),` and $\vec{\text{up}} = (0, 1, 0)$. The expected output is shown in Figure 3. Note: `lookAt` is a 3D position, which is different the gaze direction (which is a vector) referenced in the textbook.

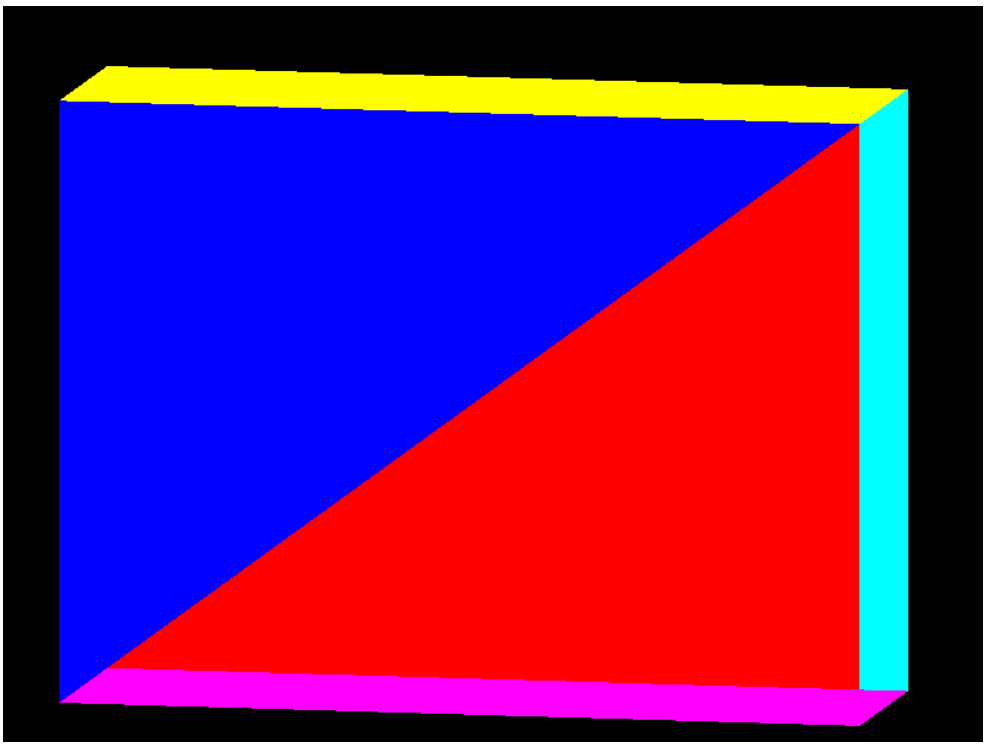


Figure 3: Expected output of `render_camera()`.

4 Perspective Projection

Input: `cube` object
Output: `p4.png`

We are now going to introduce perspective. Refer to 8.3 and 8.5 in the textbook (5th ed.), or 7.3 and 7.5 (4th ed.); use M_{per} and not M_{OpenGL}. Recall that you can build the viewing volume using just the values `fovy, aspect, near,` and `far` (note fov = field of view). The camera settings are `fovy = 65.0, aspect = 4/3, near = -1, far = -100, eye = (1, 1, 1), lookAt = (0, 0, 0), and` $\vec{\text{up}} = (0, 1, 0)$. The expected output is shown in Figure 4.

Hint: Remember to do the perspective divide after applying the matrices.

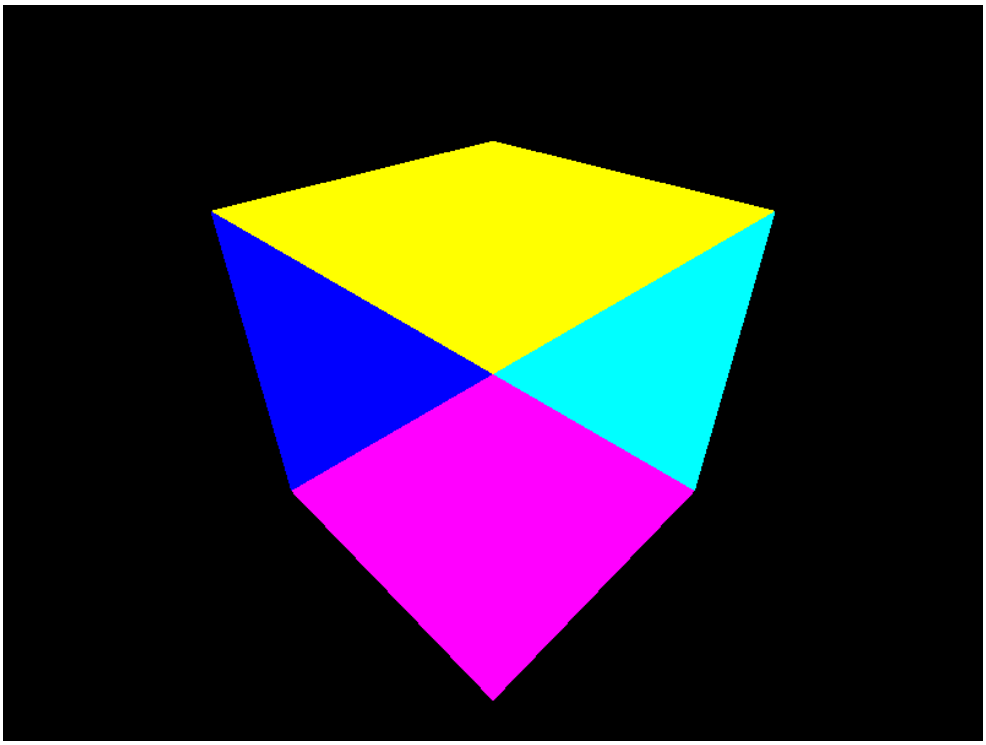


Figure 4: Expected output of `render_perspective()`.

5 Z-Buffering & Color Interpolation

Input: `cube` object
Output: `p5.png`

Now we apply our finishing touches by implementing a z-buffer and apply color interpolation over triangle vertices using **barycentric coordinates** (refer to 2.7.1 in the 4th ed. textbook, 2.9.1 in the 5th ed. textbook). The expected output is shown in Figure 6.

Hint: If the cube looks "open"/the triangles look like they have the wrong overlapping order, you may have had your z-order comparison flipped. Remember which way the z axis of the view volume points.

Hint: If you're reading this again after finishing P7 and wondering whether you forgot perspective-correct interpolation for P5 and P6: you did not, these parts use barycentric color interpolation *without* the correction needed in P7.

`documentation.pdf` diagram: It may be helpful to draw a diagram of the view volume with the z axis, as well as some points or shapes along this axis, to visualize how you should fill the z buffer.

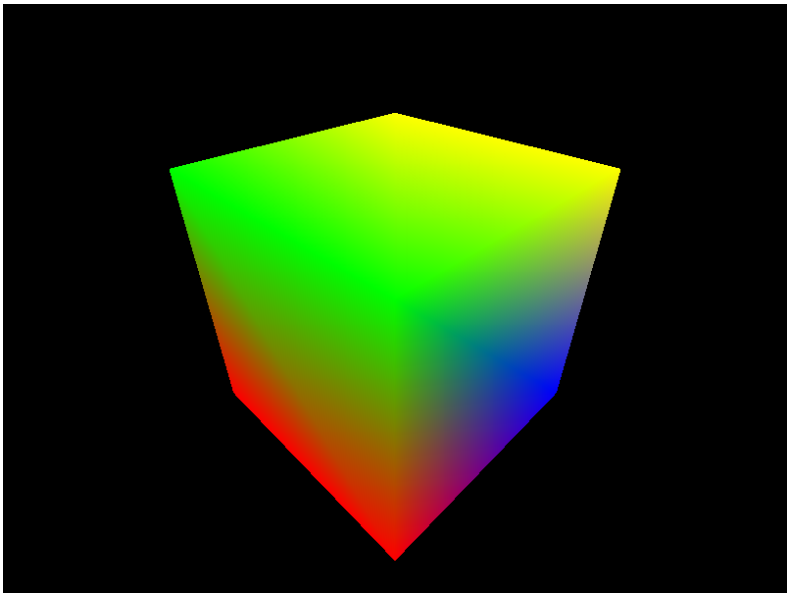


Figure 5: Expected output of `render_zbuffer_with_color()`

We included an additional viewing angle for your reference. All the viewing parameters are the same as before, except for `eye = (0.75, 2, 1)`.

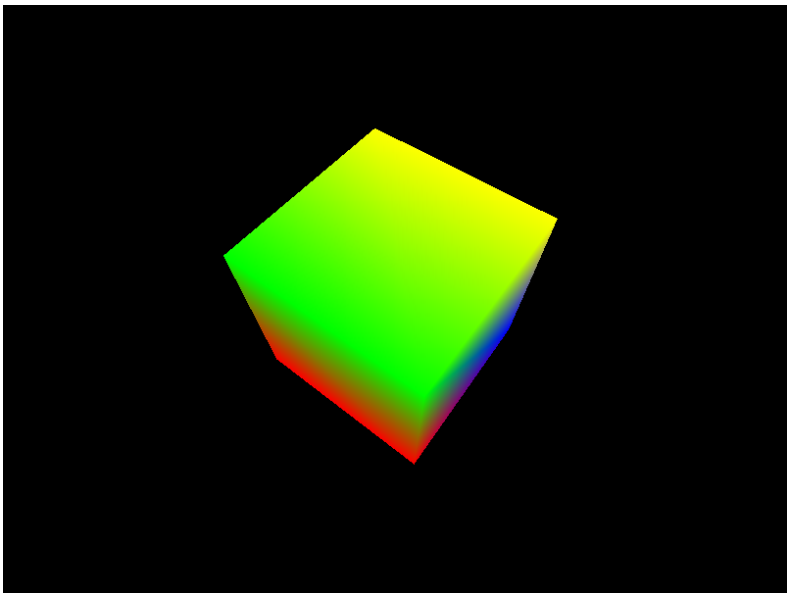


Figure 6: Expected output of `render_zbuffer_with_color()` from a **different** angle.

6 Render Big Scene

Input: `objlist` list of objects
Output: `p6.png`

Let's kick the tires a bit on your renderer by rendering a more complex scene. All the camera parameters are the same as the previous problem **except for** `eye = (-0.5, 1, 1)`. The expected output is shown in Figure 7.

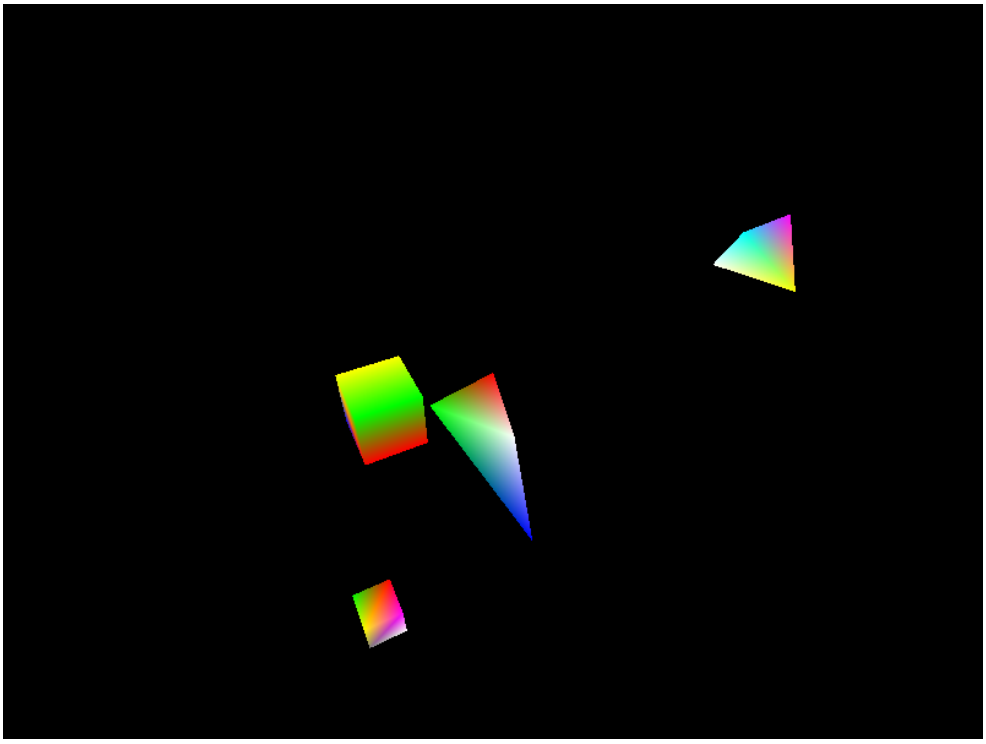


Figure 7: Expected output of `render_big_scene()`.

7 Texture Mapping

Input: `cube` object, `flag.png`
Output: `p7.png`

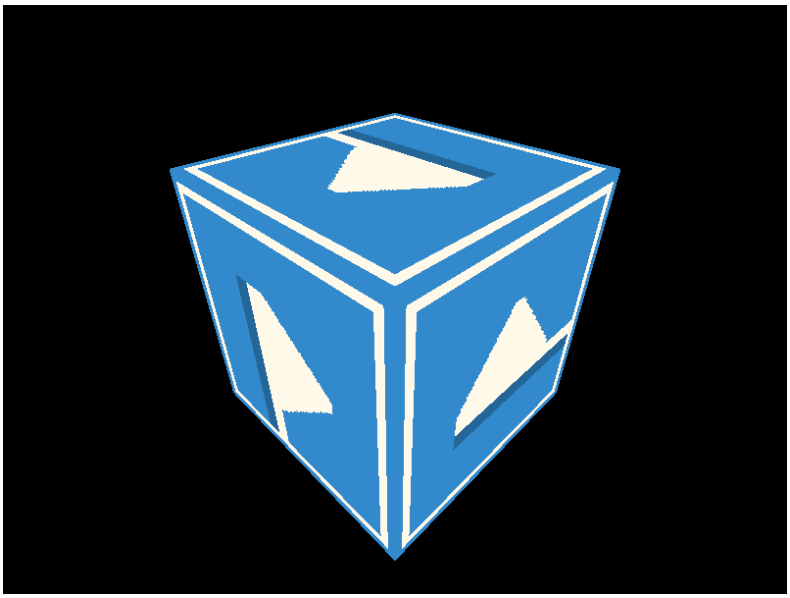


Figure 8: One possible output of `texture_map()` using `flag.png`.

Now see if you can map an image texture to each face of the cube. This involves

- a way to map triangle corners to UV coordinates (write this in `my_cube_uvs`).
  - A "corner" is a vertex in the context of a specific face it is adjacent to; you can think of a corner as the slice/wedge of a vertex that "looks at" a specific adjacent face. There are 3 corners per face, and there should be (u, v) coordinates for each corner. As such, the full array of UVs has shape $(\#F, 3, 2)$, that is, (number of faces $\times$ 3 corners per face $\times$ 2 coordinates u and v).
- rasterizing with colors sampled from the texture image using interpolated UV coordinates. (write this in `texture_map`).
  - Don't forget perspective-correct UV coordinate interpolation! (Refer to section 11.2.4 from 4th edition of the textbook. **Warning!** The *fifth* edition of the textbook does not contain the perspective-correct interpolation section! Additionally, the 4th edition contains a typo in the calculation for 1_s. Rather than $\gamma(2/w_2)$, use $\gamma(1/w_2)$.)

– Once you have interpolated UV coordinates, sample the texture image using the **clamping** nearest-neighbor rounding function. This is given in section 11.2.3 of the textbook (both 4th and 5th editions). Assume the convention that the UV domain origin $(0, 0)$ corresponds to the bottom left corner of the texture image, and $(1, 1)$ corresponds to the top right. Here is some pseudocode for this texture sampling method:

```
i = clamp(round((1 — v) * texture_height — 0.5), 0, texture_height—1)
j = clamp(round(u * texture_width — 0.5), 0, texture_width—1)
color = texture[i, j]
```

Things to note:

- The camera parameters for this part are the same as P4/P5.
- `my_cube_uvs()` is only for you to create test input UVs for the purpose of implementing `texture_map`, reproducing `p7.png` and creating your `custom.png`. The autograder will **not** run this function.

  In producing `p7.png` and `custom.png`, there can be different UV mappings that result in differing orientations of the images on the cube faces. **You can write UVs for whichever orientation you want**; the given `p7.png` is just one possible orientation (see Figure 8.)
- The function `texture_map()` which you implement should work with any input UV mapping for the cube, given as the array `uvs` of shape $(\texttt{n\_faces}, 3, 2)$. The autograder will test your implementation on different UV mappings of the cube.

`documentation.pdf` diagram: It may be helpful to draw a diagram of the method you use to compute UV coordinates for the corners of the triangles of the cube.

You may also wish to sketch out how the perspective-correct UV coordinate interpolation works (in terms of shuttling the extra values around in the vectors put through barycentric interpolation).

8 Create your own texture

Replace the image in this assignment with a custom texture image that you created. Again, the orientation for each face can be completely up to you. Rasterize the result with your pipeline and include the resulting png with your submission.

Notes on debugging with diff script

If you do use the diff functionality (or derivatives thereof) from the `utils.py` file that we provided in assignment 1, the most reliable diff pair to run should be between

your output `.npy.gz` file and the reference `.npy.gz` file. Diffing the `.png` files is also fine, but we've seen PNGs end up *subtly different* when created from the same float array but on different machines. (This is not an issue during autograding.)

Also remember that for this assignment, we tolerate 98% accuracy in the autograder; if your solution is 98% correct, you get full credit.

Debugging tip

If your solution is only different from the expected images at **edge pixels** of the rendered primitives, try adding a `1e-15` tolerance to both sides of your $\leq$ or $\geq$ comparisons involved in bounding box or inside-outside calculations. For instance, for a test like `a <= x <= b`, try `(a - 1e-15) <= x <= (b + 1e-15)`. This can compensate for any floating point differences that have accumulated in your solution that aren't actually mathematically significant, only appearing because of floating point precision.