TTIC 31040: Introduction to Computer Vision
Winter 2022

Problem Set #2

Out: Jan 21st, 2022

Due: Monday Feb 1st, 11:59pm

Instructions

How and what to submit? Please submit your solutions electronically via Canvas. Put your answer to qualitative discussions in a PDF file (typeset or handwritten/photographed or scanned) named `writeup.pdf`. Then delete the data folder, and submit the entire hw2 directory as a single tarball (not zip please) with the command

```
tar cvf hw2.tar hw2/
```

Late submissions There will be a penalty of **25 points** for any solution submitted within **48 hours** past the deadline. No submissions will be accepted past then. Please email instructors for special circumstances.

What is the required level of detail? When asked to derive something, please clearly state the assumptions, if any, and strive for balance: justify any non-obvious steps, but try to avoid superfluous explanations. When asked to plot something, please include in the `ipynb` file the figure as well as the code used to plot it. If multiple entities appear on a plot, make sure that they are clearly distinguishable (by color or style of lines and markers) and references in a legend or in a caption. When asked to provide a brief explanation or description, try to make your answers concise, but do not omit anything you believe is important. If there is a mathematical answer, provide is precisely (and accompany by only succinct words, if appropriate).

When submitting code (in Jupyter notebook), please make sure it's reasonably documented, runs and produces all the requested results. If discussion is required/warranted, you can include it either directly in the notebook (you may want to use the markdown style for that) or in the PDF writeup. **Please make sure to submit a notebook that has been run with all the outputs generated!**

Collaboration policy Collaboration is allowed and encouraged, as long as you (1) write your own solution entirely on your own, (2) specify names of student(s) you collaborated with in your writeup.

Software Setup

We will use Open3D to visualize 3D data. We recommend installing with

```
pip install open3d
```

and avoid conda install if you can. Open3D plans to deprecate their anaconda support soon.

1 Rendering with Projective Camera

1.1 Background

We provide pointers to background materials that you should be aware of in order to complete this problem.

1.1.1 What does "rendering" mean?

Rendering means to create images, often (but not exclusively) from 3D "assets" (representations of 3D scenes) and properties while simulating the physics of natural image formation. It is one of the core areas of computer graphics.

Camera projection is a critical step in a rendering pipeline. But rendering consists of many other steps, and creating nice-looking images like Pixar's Toy Story involves substantial engineering complexity on the graphics side, which is not the focus of our class.

Computer Vision can be thought of as "inverse graphics". We want to reconstruct and analyze an underlying 3D scene from their 2D projections. In order for us to build up the "inverse" machinery, we need to have an intimate understanding of the "forward" rendering process, especially camera projection.

Therefore in Problem 2 we will write a barebone rendering routine ourselves. This will provide you with a solid grasp of camera intrinsics and extrinsics. Once we verify that our routine is accurate in terms of camera geometry, we will move on to using Open3D as our renderer/visualizer. Open3D allows users to provide intrinsics and extrinsics parameters to manipulate cameras, and takes care of the rest of the rendering pipeline. This is the sweet spot for us at the moment.

We will define our camera model based on the OpenGL standard, discussed below.

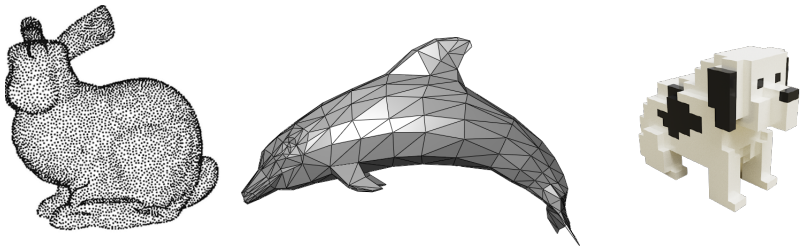

Figure 1: Examples of point cloud (left), mesh (middle), and voxel (right)

1.1.2 3D data representations

We spoke of rendering 3D assets. There are various ways one could represent data in 3D, and we will deal with point cloud and mesh in this assignement.

Point Cloud A common 3D data representation is point cloud. It is just a list of points with their (x, y, z) spatial locations, and optionally color. They are stored in files with extension `.ply`. In this assignment we provide you with a point cloud scan of the facade of our Rockefeller Memorial Chapel to play with[1].

Mesh Meshes can be thought of as point cloud + connectivity. It consists of a list of 3D vertices, and a list of index triplets with each triplet recording which 3 vertices share the same triangular face. Put it differently they are a bunch of triangles. Triangles can (approximately) represent a surface more accurately and efficiently than a point cloud. They are stored in files with extension `.obj`.

Voxel If you have played with LEGO or Minecraft, you know what voxels are. They are elements of regular grids laid out in 3D, with each cell indicating spatial occupancy and color (and sometimes additional properties like opacity). They are the analogy of pixels in 3D, hence the name "volume pixel – voxel". We are not using voxels in this assignment, but it's important to know what they are.

1.1.3 World axis convention

We use **right-handed** coordinates, with x right, y up and **z backward**. Camera looks towards the **negative z-axis**. Objects in front of a camera have **negative depth**. Shown in Fig 2.

[1]This point cloud was obtained as part of a research project in our group; let us know if you may be interested to help with this research!

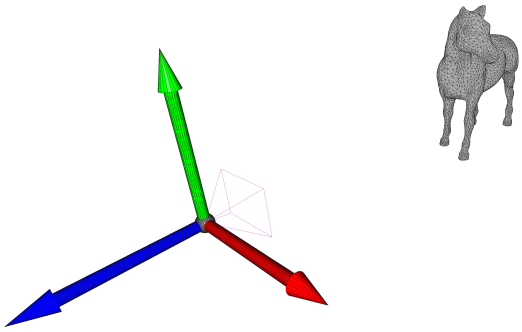

Figure 2: World axis convention. Red: +x. Green: +y. Blue: +z. The camera, visualized as a pink cone at the origin, looks towards the negative z-axis at a horse mesh.

Different fields adopt different coordinate conventions. Roboticists and 3D modelers prefer to put x-y plane on the ground, and z up. Notable examples are **Mujoco** and **Blender**. Vision people prefer z as depth, and x-y for the vertical image plane. Everyone wants what is convenient for their work.

You might find the fact that visible objects have negative depth awkward. But think about it: if we want x right, y up, and a right-handed coordinate system, then z has to point backward. DirectX, which is Microsoft's competitor to OpenGL, uses left-handed coordiantes.

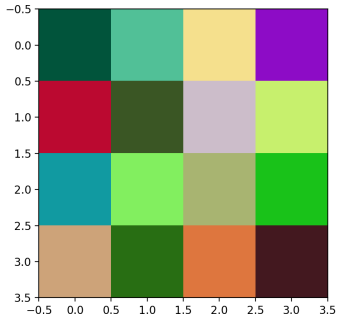

Figure 3: Location of fractional pixel coordinates in a small 4×4 random image. The top left corner is $(-0.5, -0.5)$. The bottom right corner is $(3.5, 3.5)$. The center of the image is $(1.5, 1.5)$. This convention is also used by matplotlib, because this very figure is plotted by matplotlib with its default axis labeling.

1.1.4 Image axis convention

In homework 1 we have seen that the axis convention of an image is that of (origin top-left, x right, y down). Coordinate (0, 0) refers to the pixel at the top-left corner of an image. However in 3D vision and graphics, one frequently deals with fractional pixel coordinate values, and it is important to establish a consistent sub-pixel convention.

As shown in Fig 3, the top left corner of an image is $(-0.5, -0.5)$, not $(0, 0)$. It's the center of a pixel that has intenger coordinates. This has implication on the entires of the intrinsic matrix.

1.1.5 How (not) to get lost during spatial transforms

When you consider the following questions:

- Transform points from camera space to world space;
- How do coordinates of the points change when the camera rotates and translates by a given amount,

you might start wondering which matrix I should invert, and get confused. A good heuristic to maintain a cool head is the following:

For the same point referred to as $\mathbf{p}$ and $\mathbf{q}$ in 2 different coordinate frames (say world and camera, respectively),

$$\mathbf{Ap} = \mathbf{Bq}$$

where $\mathbf{A}$ is the matrix composed of basis vectors for the 1st reference frame, $\mathbf{B}$ is the matrix of basis vectors for the 2nd reference frame. Both of these sets of bases are expressed in some "canonical" reference frame; we can for simplicity assume that that frame is the same as 1st frame, so that $\mathbf{A}$ is the identity. $\mathbf{p}$ are the coordinates of the point in 1st reference frame. $\mathbf{q}$ are the coordinates of the same point in the 2nd reference frame.

For example, let's say a point has coordinates $\mathbf{p} = (3, 7, 6)$ in world frame, and camera pose/basis is $\mathbf{B}$. Then you have, in homogeneous coordinates,

$$\mathbf{Ap} = \mathbf{Bq}$$

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 3 \\ 7 \\ 6 \\ 1 \end{bmatrix} = \begin{bmatrix} | & | & | & | \\ \mathbf{b}_x & \mathbf{b}_y & \mathbf{b}_z & \mathbf{b}_t \\ | & | & | & | \\ 0 & 0 & 0 & 1 \end{bmatrix} \mathbf{q},$$

where $\mathbf{b}_x$ is the basis vector for the x axis, etc., and $\mathbf{b}_t$ is the 3D translation vector for the camera center (in canonical coordinate frame), with 1 appended to bring it to homogeneous

coordinates. Then

$$\begin{aligned}\mathbf{B}^{-1}\mathbf{A}\mathbf{p} &= \mathbf{q} \\ \mathbf{p} &= A^{-1}B\mathbf{q}\end{aligned}$$

Here $\mathbf{A}$ is the identity matrix. The World2Cam matrix, which transforms $\mathbf{p}$ to $\mathbf{q}$, is thus $\mathbf{B}^{-1}\mathbf{A} = \mathbf{B}^{-1}\mathbf{I} = \mathbf{B}^{-1}$. World2Cam is simply inverse camera pose. How about Cam2World? Well, it takes $\mathbf{q}$ back to $\mathbf{p}$, so it is simply $\mathbf{B}$.

A more complex example (which we will think about more later in the course): let's say we have 2 cameras, with poses $\mathbf{B}_1$ and $\mathbf{B}_2$. How do I transform point coordinates from camera 1 to camera 2?

$$\begin{aligned}\mathbf{B}_1\mathbf{p} &= \mathbf{B}_2\mathbf{q} \\ \mathbf{p} &= \mathbf{B}_1^{-1}\mathbf{B}_2\mathbf{q} \\ \mathbf{B}_2^{-1}\mathbf{B}_1\mathbf{p} &= \mathbf{q}\end{aligned}$$

What if there are 10 cameras chained by some relative pairwise pose relationship? e.g. each camera is a slight rotation and translation of the previous camera. How to transform points in the last camera frame to the first camera frame? You might see this question come up later in the course, as well.

1.2 What's Intrinsic and FoV

In class we discussed camera intrinsics and the role of focal length. Here we provide a more detailed breakdown. In particular we pay attention to how the Field of View (FoV) is related to the focal length.

First let the image axis convention be (origin center, x right, y up). We will move to (origin top-left, x right, y down) later. As an example, let's focus now on the y-axis (vertical direction).

The precise value of an object's y-axis coordinate p_y (physical unit: pixels) is determined by the following:

$$p_y = \underbrace{\frac{y}{z}}_{\text{tan object angle}} \Big/ \underbrace{\underbrace{\frac{0.5h}{f}}_{\text{tan half FoV}} \cdot \underbrace{0.5H}_{\text{half img height}}}_{\text{focal length in intrinsic matrix. unit: pixels}} \tag{1}$$

where

- y and z are height and depth of the object, in meters.

- h is the height of the sensor, or film, in meters.
- f is the focal length of camera lens, or the film distance to camera center, in meters.
- H is the image height, in pixels.
- **Critical:** $\frac{0.5h}{f}$ is the tangent of half vertical field-of-view. In many vision and graphics software, FoV is only parameter provided to you. You don't see the actual focal length/film distance. You see FoV.

You can verify this equality easily by drawing similar triangles.

What's interesting about this equation is that you can group the terms in various ways for different interpretations

- $p_y = \frac{y}{z} \cdot f \cdot \underbrace{\frac{H}{h}}_{\text{pixel/meter}}$ The rightmost term is the pixel density on the film.
- $p_y = 0.5H \cdot \underbrace{(\frac{y}{z} / \frac{0.5h}{f})}_{\text{angle ratio}}$ This can be interpreted as distributing vertical image space according to the angle ratio of the object vs FoV.

Whatever interpretation you adopt, bear in mind that what appears in the intrinsic matrix, loosely referred to as focal length, is actually a parameter that scales y/z. This is referred to as α, β in our lecture slides. The physical unit of this scaling parameter is **pixels**.

Problem 1 [8 points]
Implement the Intrinsic Matrix K.

1. Complete the routine `render.py:compute_proj_to_normalized`. Given FoV and img aspect ratio (width / height), this function returns a matrix that projects a 3D point to a normalized image space with axis convention (origin center, x right, y up), and width and height in [-1.0, 1.0].
2. Complete the routine `render.py:compute_normalized_to_img_trans`. Given image width and height in pixels, this function returns a matrix that transforms the projected points in normalized space to image pixel space. Note that image pixel space has convention (origin top-left, x right, y down). Also note that the top-left corner of an image is $(-0.5, -0.5)$.
3. Complete the routine `render.py:compute_intrinsics` that calls the above two functions, and combines their results into a single matrix. This is the camera intrinsic matrix. You will be able to test the correctness of your implementation in problem 3.

End of problem 1

1.3 Camera Pose and Extrinsics

Camera extrinsic matrix maps an object from world space to the camera space. It's also referred to as World2Cam Matrix.

OpenGL parameterizes camera pose using 3 vectors: eye, front and up. You need to derive the camera pose (x,y,z axis direction and camera location) from these 3 vectors.

- Eye is the 3D location of the camera center.
- Front, or gaze, is the direction that the camera looks at. Note that in our world axis convention the camera **looks towards negative z axis**. Hence camera z-axis is the negative of the front direction.
- Up is a little special. It's **not** the camera's y-axis. It's the y-axis vector with allowance for some slant forward or backward along the camera's z-axis (it does not slant left-right along x-axis). Hence the name Up. The benefit of this design will become clear when you do animation over camera trajectory later.
- This assumes that the world axis convention is right-handed. That's how you get the camera x-axis.

Problem 2 **[8 points]**
Complete the routine `render.py:compute_extrinsics`. Given Eye, Front, Up direction, this function returns the World2Cam matrix E.

End of problem 2

1.4 Rendering Point Cloud

Now that we have the intrinsic and extrinsic matrices ready. let's render some point clouds. We will write a simple rendering routine.

For each point, after computing its pixel locations (p_x, p_y) using the projection matrix, we will round them to the nearest integer coordinate and color the pixel red. Note that actual point cloud rendering algorithm used in computer graphics is substantially more complex. (How to blend points that fall into the same pixel?)

Problem 3 **[4 points]**
Render 4 points. $\begin{bmatrix} 0 \\ 0 \\ -1 \end{bmatrix}$, $\begin{bmatrix} 1 \\ -1 \\ -1 \end{bmatrix}$, $\begin{bmatrix} -1 \\ -0.5 \\ -1 \end{bmatrix}$, and $\begin{bmatrix} 0 \\ 1 \\ -1 \end{bmatrix}$.

Intrinsics: FoV 90°; image width 80 pixels, height 60 pixels.

Extrinsics: eye location $\begin{bmatrix}0\\0\\2\end{bmatrix}$, front direction $\begin{bmatrix}0\\0\\-1\end{bmatrix}$, up direction $\begin{bmatrix}0\\1\\-1\end{bmatrix}$.

We have chosen the numbers in such a way that you should be able to calculate the projected point locations by hand with pencil and paper (to help verify your implementation).

Run the function `q1.py:debug_four_points`. Save the figure to `four_points.png`.

End of problem 3

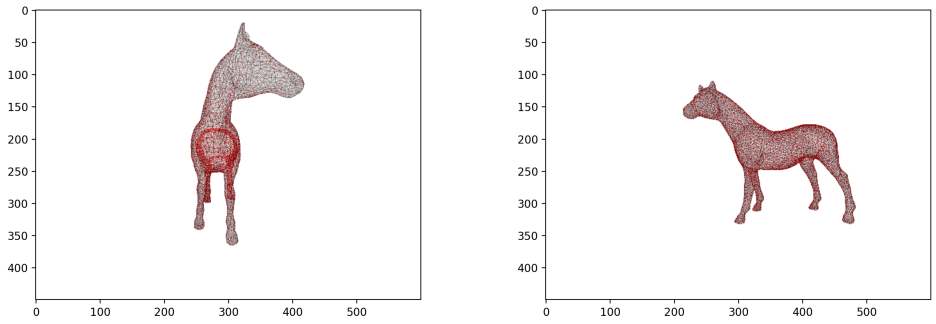

Figure 4: The expected outputs of our horse point cloud rendering superimposed on top of open3d renderings.

Problem 4 **[4 points]**

We provide you with a horse point cloud. Run the function `q1.py:mine_vs_open3d_pc`. It superimposes our naive rendering of the horse point cloud on top of open3d mesh rendering, and you should see that the two images align perfectly.

Render the superimposed point cloud image from 2 viewpoints:

Eye: $\begin{bmatrix}0\\0\\2\end{bmatrix}$, front $\begin{bmatrix}0\\0\\-1\end{bmatrix}$, up $\begin{bmatrix}0\\1\\-1\end{bmatrix}$

Eye: $\begin{bmatrix}2\\0\\0.5\end{bmatrix}$, front $\begin{bmatrix}-1\\0\\0\end{bmatrix}$, up $\begin{bmatrix}0\\1\\0\end{bmatrix}$

Save the figures from the two viewpoints as `mine_vs_ref_view1.png` and `mine_vs_ref_view2.png`. We have attached the expected output for your reference in Fig 4.

We know at this point our camera projection is correct in terms of geometry. We can switch to using only Open3D. By now you know how the API works "under the hood".

End of problem 4

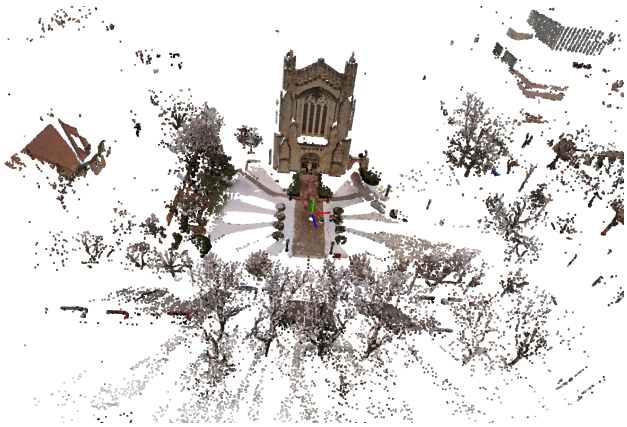

Figure 5: A point cloud scan of Rockfeller Memorial Chapel

Problem 5 **[4 points]**
We provide you with a point cloud scan of the Rockfeller Memorial Chapel in `rocefeller_chapel.ply`. It has been subsampled to 3% of the original point density and currently has 7 million points.

Render this point cloud using Open3D. Drag and zoom manually. Pick your favorite viewpoint and take a screenshot by pressing the "P" key. Press "h" to print the Open3D help manual. It has a whole suite of convenience features you can play with.

Run the function `q1.py:vis_rockfeller`. Save the image to `chapel.png`. We have attached a reference output in Fig 5.

End of problem 5

1.5 Rendering Mesh under Perspective Distortion

Problem 6 **[8 points]**
In class we talked about perspective distortion. This happens when the FoV is very large and objects are on near the periphery of the image. To see this effect, render the horse mesh, and manually rotate the camera so that the horse is near the edge of the image.

First let the FoV be 120°, move around, what do you see?

First let the FoV be 160°, what do you see now?

Run the function `q1.py:vis_horse`. Save the figure from the two FoV as `horse_fov_120.png` and `horse_fov_160.png`, and write in your written solution the qualitative description of the visual phenomena you observe when moving each of the cameras, and the difference between the two cameras.

End of problem 6

1.6 Draw Cameras by Shooting Rays through Pixels

We have projected points to pixels. Now we reverse the process and shoot rays through pixels to the world. We use this technique to visualize our camera as a pyramid (whose top vertex is the camera center, and the base is the image "canvas").

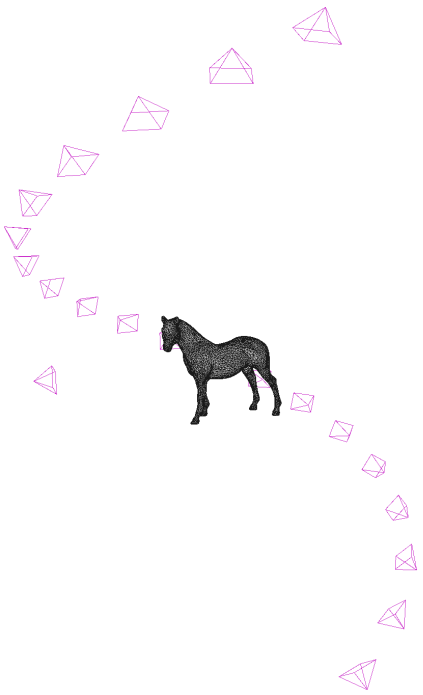

Figure 6: Visualization of Camera Trajectory

Problem 7 **[8 points]**
Complete the routine `render.py:rays_through_pixels`. Given a list of 2D pixel coordinates (shape $[n, 2]$), the function returns a list of ideal points / 3D directions (shape $[n, 4]$).

Complete the routine `vis.py:draw_camera`, and shoot rays through the 4 corner pixel locations. Recall that top left corner is $(-0.5, -0.5)$. Bottom right corner is $(w - 0.5, h - 0.5)$. Note that you have to move the cone according to the camera pose.

Run the function `p1.py:vis_camera_trajectory`. This will load a camera trajectory with 20 poses arranged in a spiral.

Drag around, pick your favorite viewpoint, and save the figure to `cam_traj.png`. We have attached a reference image in Fig 6.

This is useful in future assignments when you want to generate multiples views of an object and need to visualize your camera trajectory.

End of problem 7

2 Vanishing Points

Problem 8 **[8 points]**
Prove that all lines corresponding to a given direction in 3D space, i.e., $\{\mathbf{a} + t\mathbf{n} \,|\, \forall \mathbf{a} \in \mathbb{R}^3\}$, with $\mathbf{n}$ non-orthogonal to the optical axis of the camera, have the same vanishing point in the image plane.

Advice: consider the (mathematical) limit of the projection of a point on a line as the depth of the point increases.

End of problem 8

Problem 9 **[8 points]**
Consider a set of parallel lines within a (non fronto-parallel) 3D plane; these lines are associated with a vanishing point (as proven in the previous problem). Prove that the vanishing points for all such sets for a given 3D plane form a 2D line in the image plane.

End of problem 9

3 Homography

Problem 10 **[10 points]**
Prove that two images of a static scene are related by 2D homography under pure camera rotation (zero translation). You can assume that there is no change in intrinsics over different images.

End of problem 10

As discussed in class, planar homography also applies to two views of the same 3D plane. Let's use this technique to rectify the image of a football field.

3.1 Image Rectification

Problem 11 **[10 points]**
Complete the routine `q3.py:compute_homography`. It uses the DLT algorithm. Now run the function `q3.py:rectify_image`. It applies the computed homography from the 4 corners to warp the image. The expected output is shown for you in Fig 8. Why does the warped image appear dimmer on the top right? Note that we are using a warping routine written entirely in numpy.

Figure 7: A sports stadium. The boundaries of the football field are highlighted in cyan.

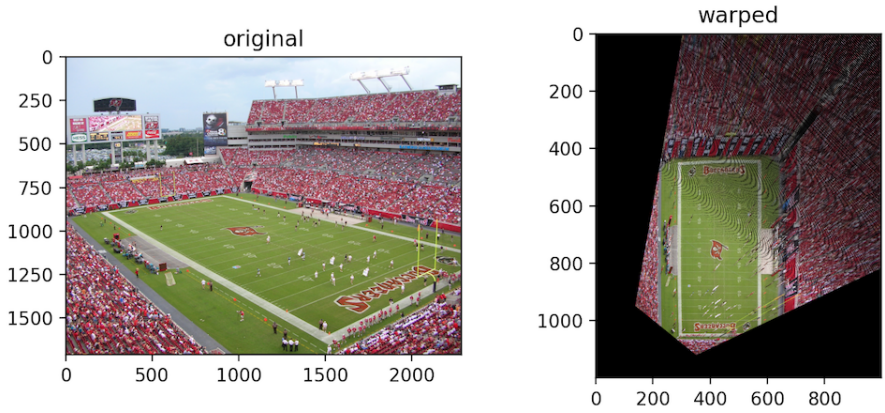

Figure 8: A sports stadium. The boundaries of the football field are highlighted in cyan.

Save the figure in file `rectified.png`.

End of problem 11

4 Camera Calibration

The PnP calibration algorithm requires ground truth 3D location to calibrate a camera, and it is often an unrealistic assumption. But it is a good exercise nevertheless.

Problem 12 **[10 points]**
Complete `q4.py:pnp_calibration` and run the function `q4.py:main`. Make sure you can pass this assertion test.

End of problem 12

The best way to calibrate camera intrinsics is Zhang's algorithm. It only requires a 2D planar pattern. We will provide you with a script in case you need to calibrate camera intrinsics for your project. Also look it up. It's not hard to understand.