```
    Return $B$
```

Initially, B contains all of T (in fact, is equal to T). By the Grow-Algorithm Corollary, the set B is linearly independent throughout the algorithm. When the algorithm terminates, Span $B = \mathcal{V}$. Hence upon termination B is a basis for $\mathcal{V}$. Furthermore, B still contains all of T since the algorithm did not remove any vectors from B.

There is just one catch with this reasoning. As in our attempt in Section 5.6.3 to show that every vector space has a basis, *we have not yet shown that the algorithm terminates!* This issue will be resolved in the next chapter.

5.7 Unique representation

As discussed in Section 5.1, in a coordinate system for $\mathcal{V}$, specified by generators $\boldsymbol{a}_1, \ldots, \boldsymbol{a}_n$, each vector $\boldsymbol{v}$ in $\mathcal{V}$ has a coordinate representation $[\alpha_1, \ldots, \alpha_n]$, which consists of the coefficients with which $\boldsymbol{v}$ can be represented as a linear combination:

$$\boldsymbol{v} = \alpha_1\, \boldsymbol{a}_1 + \cdots + \alpha_n\, \boldsymbol{a}_n$$

But we need the axes to have the property that each vector $\boldsymbol{v}$ has a *unique* coordinate representation. How can we ensure that?

5.7.1 Uniqueness of representation in terms of a basis

We ensure that by choosing the axis vectors so that they form a basis for $\mathcal{V}$.

Lemma 5.7.1 (Unique-Representation Lemma): Let $\boldsymbol{a}_1, \ldots, \boldsymbol{a}_n$ be a basis for a vector space $\mathcal{V}$. For any vector $\boldsymbol{v} \in \mathcal{V}$, there is exactly one representation of $\boldsymbol{v}$ in terms of the basis vectors.

In a graph G, this corresponds to the fact that, for any spanning forest F of G, for any pair x, y of vertices, if G contains an x-to-y path then F contains exactly one such path.

Proof

Because Span $\{\boldsymbol{a}_1, \ldots, \boldsymbol{a}_n\} = \mathcal{V}$, every vector $\boldsymbol{v} \in \mathcal{V}$ has at least one representation in terms of $\boldsymbol{a}_1, \ldots, \boldsymbol{a}_n$. Suppose that there are two representations;

$$\boldsymbol{v} = \alpha_1\, \boldsymbol{a}_1 + \cdots + \alpha_n\, \boldsymbol{a}_n = \beta_1\, \boldsymbol{a}_1 + \cdots + \beta_n\, \boldsymbol{a}_n$$

Then we can get the zero vector by subtracting one linear combination from the other:

$$\begin{aligned} \mathbf{0} &= \alpha_1\, \boldsymbol{a}_1 + \cdots + \alpha_n\, \boldsymbol{a}_n - (\beta_1\, \boldsymbol{a}_1 + \cdots + \beta_n\, \boldsymbol{a}_n) \\ &= (\alpha_1 - \beta_1)\boldsymbol{a}_1 + \cdots + (\alpha_n - \beta_n)\boldsymbol{a}_n \end{aligned}$$

Since the vectors $\boldsymbol{a}_1, \ldots, \boldsymbol{a}_n$ are linearly independent, the coefficients $\alpha_1 - \beta_1, \ldots, \alpha_n - \beta_n$

must all be zero, so the two representations are really the same. □

5.8 Change of basis, first look

Change of basis consists in changing from a vector's coordinate representation in terms of one basis to the same vector's coordinate representation in terms of another basis.

5.8.1 The function from representation to vector

Let $\boldsymbol{a}_1, \ldots, \boldsymbol{a}_n$ form a basis for a vector space $\mathcal{V}$ over a field $\mathbb{F}$. Define the function $f : \mathbb{F}^n \mapsto \mathcal{V}$ by

$$f([x_1, \ldots, x_n]) = x_1\,\boldsymbol{a}_1 + \cdots + x_n\,\boldsymbol{a}_n$$

That is, f maps the representation in $\boldsymbol{a}_1, \ldots, \boldsymbol{a}_n$ of a vector to the vector itself. The Unique-Representation Lemma tells us that every vector in $\mathcal{V}$ has exactly one representation in terms of $\boldsymbol{a}_1, \ldots, \boldsymbol{a}_n$, so the function f is both onto and one-to-one, so it is invertible.

Example 5.8.1: I assert that one basis for the vector space $\mathbb{R}^3$ consists of $\boldsymbol{a}_1 = [2,1,0], \boldsymbol{a}_2 = [4,0,2], \boldsymbol{a}_3 = [0,1,1]$. The matrix with these vectors as columns is

$$A = \left[\begin{array}{c|c|c} 2 & 4 & 0 \\ 1 & 0 & 1 \\ 0 & 2 & 1 \end{array}\right]$$

Then the function $f : \mathbb{R}^3 \longrightarrow \mathbb{R}^3$ defined by $f(\boldsymbol{x}) = A\boldsymbol{x}$ maps the representation of a vector in terms of $\boldsymbol{a}_1, \ldots, \boldsymbol{a}_3$ to the vector itself. Since I have asserted that $\boldsymbol{a}_1, \boldsymbol{a}_2, \boldsymbol{a}_3$ form a basis, every vector has a unique representation in terms of these vectors, so f is an invertible function. Indeed, the inverse function is the function $g : \mathbb{R}^3 \longrightarrow \mathbb{R}^3$ defined by $g(\boldsymbol{y}) = M\boldsymbol{y}$ where

$$M = \begin{bmatrix} \frac{1}{4} & \frac{1}{2} & -\frac{1}{5} \\ \frac{1}{8} & -\frac{1}{4} & \frac{1}{4} \\ -\frac{1}{4} & \frac{1}{2} & \frac{1}{2} \end{bmatrix}$$

Thus M is the inverse of A.

5.8.2 From one representation to another

Now suppose $\boldsymbol{a}_1, \ldots, \boldsymbol{a}_n$ form one basis for $\mathcal{V}$ and $\boldsymbol{b}_1, \ldots, \boldsymbol{b}_m$ form another basis. Define $f : \mathbb{F}^n \longrightarrow \mathcal{V}$ and $g : \mathbb{F}^m \longrightarrow \mathcal{V}$ by

$$f([x_1, \ldots, x_n]) = x_1\,\boldsymbol{a}_1 + \cdots + x_n\,\boldsymbol{a}_n \text{ and } g([y_1, \ldots, y_m]) = y_1\,\boldsymbol{b}_1 + \cdots + \boldsymbol{y}_m\,\boldsymbol{b}_m$$

By the linear-combinations definition of matrix-vector definition, each of these functions can be represented by matrix-vector multiplication:

$$f(\boldsymbol{x}) = \left[\begin{array}{c|c|c} & & \\ \boldsymbol{a}_1 & \cdots & \boldsymbol{a}_n \\ & & \end{array}\right] \left[\begin{array}{c} \boldsymbol{x} \end{array}\right] \text{ and } g(\boldsymbol{y}) = \left[\begin{array}{c|c|c} & & \\ \boldsymbol{b}_1 & \cdots & \boldsymbol{b}_m \\ & & \end{array}\right] \left[\begin{array}{c} \boldsymbol{y} \end{array}\right]$$

Furthermore, by the reasoning in Section 5.8.1, the functions f and g are both invertible. By Lemma 4.13.1, their inverses are linear functions

Now consider the function $g^{-1} \quad f$. It is the composition of linear functions so it is a linear function. Its domain is the domain of f, which is $\mathbb{F}^n$, and its co-domain is the domain of g, which is $\mathbb{F}^m$. Therefore, by Lemma 4.10.19, there is a matrix C such that $C\boldsymbol{x} = (g^{-1} \quad f)(\boldsymbol{x})$.

The matrix C is a *change-of-basis* matrix:

- Multiplying by C converts from a vector's coordinate representation in terms of $\boldsymbol{a}_1, \ldots, \boldsymbol{a}_n$ to the same vector's coordinate representation in terms of $\boldsymbol{b}_1, \ldots, \boldsymbol{b}_n$.

Since $g^{-1} \quad f$ is the composition of invertible functions, it too is an invertible function. By the same reasoning, there is a matrix D such that $D\boldsymbol{y} = (f^{-1} \quad g)(\boldsymbol{y})$.

- Multiplying by D converts from a vector's coordinate representation in terms of $\boldsymbol{b}_1, \ldots, \boldsymbol{b}_k$ to the same vector's coordinate representation in terms of $\boldsymbol{a}_1, \ldots, \boldsymbol{a}_n$.

Finally, since $f^{-1} \quad g$ and $g^{-1} \quad f$ are inverses of each other, the matrices C and D are inverses of each other.

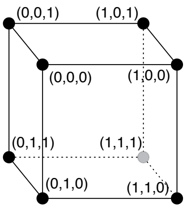
Why would you want functions that map between different representations of a vector? There are many reasons. In the next section, we'll explore one: dealing with perspective in images. Lab 5.12 will similarly deal with perspective using change of basis. Change of basis is crucially important in Chapters 10, 11, and 12.

5.9 Perspective rendering

As an application of coordinate representation, we show how to synthesize a camera view from a set of points in three dimensions, taking into account perspective. The mathematics underlying this task will be useful in a lab, where we will go in the opposite direction, removing perspective from a real image.

5.9.1 Points in the world

We start with the points making up a wire-frame cube with coordinates as shown:



The coordinates might strike you as a bit odd: the point (0,1,0) is vertically *below* the point (0,0,0). We are used to y-coordinates that increase as you move up. We use this coordinate system in order to be consistent with the way pixel coordinates work.

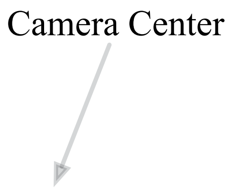
The list of points making up the wire-frame cube can be produced as follows:

```
>>> L = [[0,0,0],[1,0,0],[0,1,0],[1,1,0],[0,0,1],[1,0,1],[0,1,1],[1,1,1]]]
>>> corners = [list2vec(v) for v in L]

>>> def line_segment(pt1, pt2, samples=100):
    return [(i/samples)*pt1 + (1-i/samples)*pt2 for i in range(samples+1)]

>>> line_segments = [line_segment(corners[i], corners[j]) for i,j in
 [(0,1),(2,3), (0,2),(1,3),(4,5),(6,7),(4,6),(5,7),(0,4),(1,5),(2,6), (3,7)]]

>>> pts = sum(line_segments, [])
```

Imagine that a camera takes a picture of this cube: what does the picture look like? Obviously that depends on where the camera is located and which direction it points. We will choose to place the camera at the location (-1,-1, -8) facing straight at the plane containing the front face of the cube.

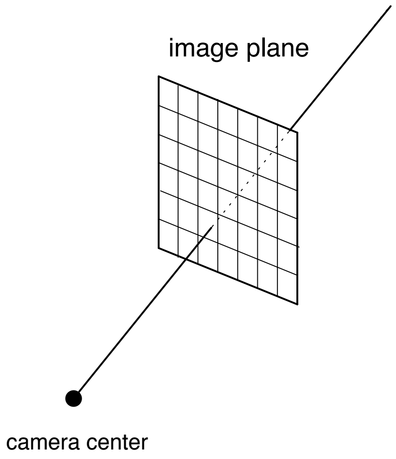
5.9.2 The camera and the image plane

We present a simplified model of a camera, a *pinhole* camera. Assume the location and orientation of the camera are fixed. The pinhole is a point called the *camera center.*



There is an image sensor array in the back of the camera. Photons bounce off objects in the scene and travel through the camera center to the image sensor array. A photon from the scene only reaches the image sensor array if it travels in a straight line through the camera center. The image ends up being reversed.

An even simpler model is usually adopted to make the math easier. In this model, the image sensor array is between the camera center and the scene.



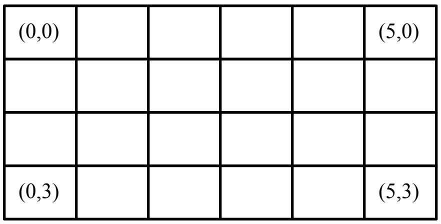
We retain the rule that a photon is only sensed by the sensor array if the photon is traveling in a line through the camera center.

The image sensor array is located in a plane, called the *image plane*. A photon bounces off an object in the scene, in this case the chin of a frog, and heads in a straight line towards the camera center. On its way, it bumps into the image plane where it encounters the sensor array.

The sensor array is a grid of rectangular sensor elements. Each element of the image sensor array measures the amount of red, green, and blue light that hits it, producing three numbers. Which sensor element is struck by the photon? The one located at the intersection between the image plane and the line along which the photon is traveling.

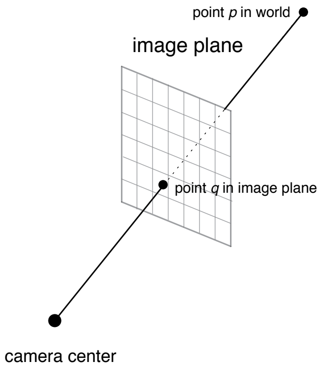
The result of all this sensing is an image, which is a grid of rectangular picture elements (pixels), each assigned a color. The pixels correspond to the sensor elements.

The pixels are assigned coordinates, as we have seen before:



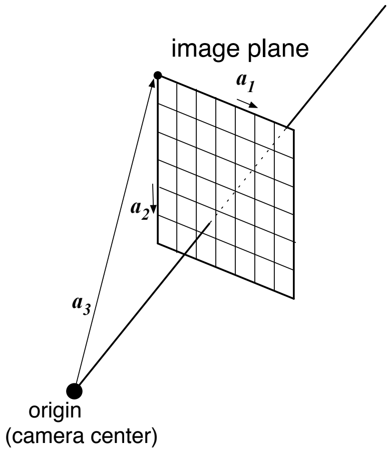
5.9.3 The camera coordinate system

For each point $\boldsymbol{q}$ in the sensor array, the light that hits $\boldsymbol{q}$ is light that travels in a straight line towards the camera center. Thus the color detected by the sensor at $\boldsymbol{q}$ is located at some point $\boldsymbol{p}$ in the world such that the line through $\boldsymbol{p}$ and the origin intersects the image plane at $\boldsymbol{q}$.



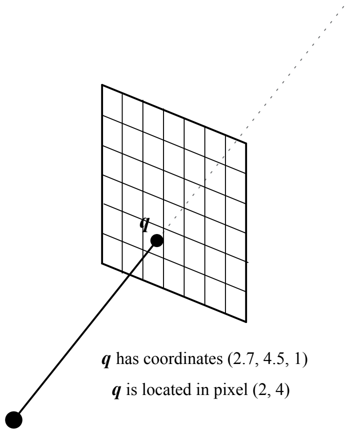
To synthesize an image of the wire-frame cube, we need to define a function that maps points $\boldsymbol{p}$ in the world to the pixel coordinates of the corresponding point $\boldsymbol{q}$ in the image plane.

There is a particularly convenient basis that enables us to simply express this function. We call it the *camera coordinate system.*



The origin is defined to be the camera center. The first basis vector $\boldsymbol{a}_1$ goes horizontally from the top-left corner of a sensor element to the top-right corner. The second vector $\boldsymbol{a}_2$ goes vertically from the top-left corner of a sensor element down to the bottom-left corner. The third vector $\boldsymbol{a}_3$ goes from the origin (the camera center) to the top-left corner of sensor element (0,0).

There's something nice about this basis. Let $\boldsymbol{q}$ be a point in the image plane


q
q has coordinates (2.7, 4.5, 1)
q is located in pixel (2, 4)

and let the coordinates of $\boldsymbol{q}$ in this coordinate system be $\boldsymbol{x} = (x_1, x_2, x_3)$, so $\boldsymbol{q} = x_1\,\boldsymbol{a}_1 + x_2\,\boldsymbol{a}_2 + x_3\,\boldsymbol{a}_3$.

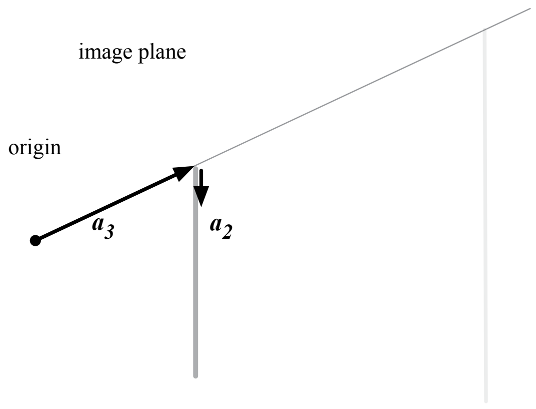
Then the third coordinate x_3 is 1, and the first and second x_1, x_2 tell us which pixel contains the point $\boldsymbol{q}$.

```
def pixel(x): return (x[0], x[1])
```

We can round x_1 and x_2 down to integers i, j to get the coordinates of that pixel (if there exists an i, j pixel).

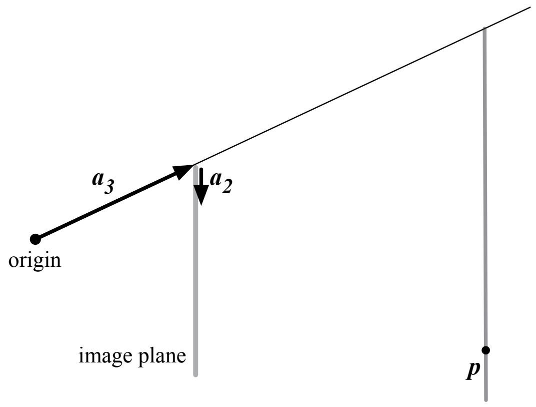
5.9.4 From the camera coordinates of a point in the scene to the camera coordinates of the corresponding point in the image plane

Let's take a side view from a point in the image plane, so we only see the edge of the sensor array:



In this view, we can see the basis vectors $\boldsymbol{a}_2$ and $\boldsymbol{a}_3$ but not $\boldsymbol{a}_1$ since it is pointed directly at us.

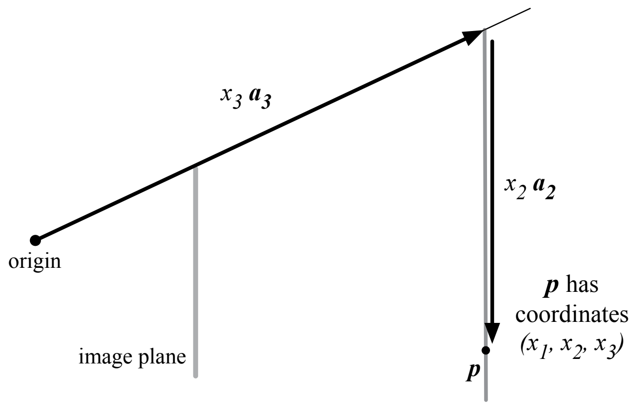
Now suppose $\boldsymbol{p}$ is a point in the scene, way beyond the image plane.



You can see the edge of the plane through $\boldsymbol{p}$ that is parallel to the image plane. We write $\boldsymbol{p}$ as a linear combination of the vectors of the camera basis:

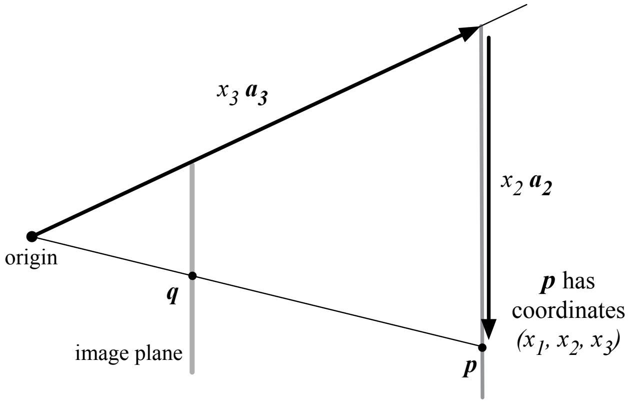
$$\boldsymbol{p} = x_1\,\boldsymbol{a}_1 + x_2\,\boldsymbol{a}_2 + x_3\,\boldsymbol{a}_3$$

Think of the vector $x_3\,\boldsymbol{a}_3$ extending through the bottom-left corner of the sensor array all the way to the plane through $\boldsymbol{p}$ that is parallel to the image plane.



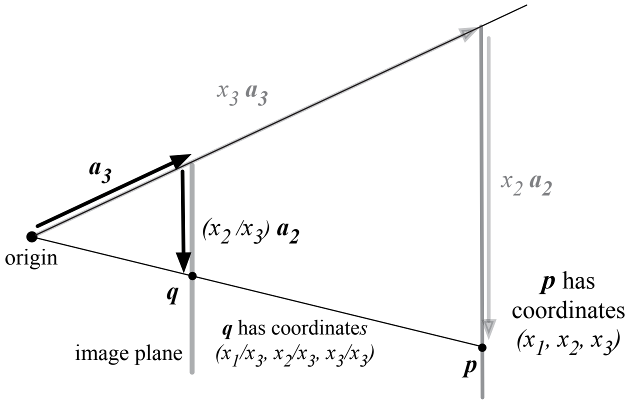
The vector $x_2\,\boldsymbol{a}_2$ extends vertically downward, and the vector $x_1\,\boldsymbol{a}_1$, which is not visible, extends horizontally towards us.

Let $\boldsymbol{q}$ be the point where the line through $\boldsymbol{p}$ and the origin intersects the image plane.



What are the coordinates of $\boldsymbol{q}$?

We see that the triangle formed by the origin, the head of $\boldsymbol{a}_3$ (when the tail is located at the origin), and $\boldsymbol{q}$ is a scaled-down version of the triangle formed by the origin, the head of $x_3\,\boldsymbol{a}_3$, and the point $\boldsymbol{p}$. Since the side formed by $\boldsymbol{a}_3$ has length $1/x_3$ times that of the side formed by $x_3\,\boldsymbol{a}_3$, a little geometric intution tells us that the coordinates of $\boldsymbol{q}$ are $1/x_3$ times each of the coordinates of $\boldsymbol{p}$, i.e. that $\boldsymbol{q}$'s coordinates are $(x_1/x_3, x_2/x_3, x_3/x_3)$.


$x_3\,\boldsymbol{a}_3$
$\boldsymbol{a}_3$
$x_2\,\boldsymbol{a}_2$
$(x_2/x_3)\,\boldsymbol{a}_2$
origin
$\boldsymbol{q}$
$\boldsymbol{q}$ has coordinates $(x_1/x_3, x_2/x_3, x_3/x_3)$
image plane
$\boldsymbol{p}$ has coordinates (x_1, x_2, x_3)
$\boldsymbol{p}$

Thus, when the camera basis is used, it is easy to go from the representation of $\boldsymbol{p}$ to the representation of $\boldsymbol{q}$: just divide each of the entries by the third entry:

```
def scale_down(x): return list2vec([x[0]/x[2], x[1]/x[2], 1])
```

5.9.5 From world coordinates to camera coordinates

We now know how to map

- from the representation in camera coordinates of a point in the world
- to the coordinates of the pixel that "sees" that point.

However, to map the points of our wire-frame cube, we need to map from the coordinates of a point of the cube to the representation of that same point in camera coordinates.

First we write down the camera basis vectors.

We do this in two steps. The first step accounts for the fact that we are locating the camera center at (-1,-1,-8) in world coordinates. In order to use the camera coordinate system, we need to locate the camera at (0,0,0), so we translate the points of the wire-frame cube by adding (1,1,8) to each of them:

```
>>> shifted_pts = [v+list2vec([1,1,8]) for v in pts]
```

In the second step, we must do a change of basis. For each point in `shifted_pts`, we obtain its coordinate representation in terms of the camera basis.

To do this, we first write down the camera basis vectors. Imagine we have 100 horizontal pixels and 100 vertical pixels making up an image sensor array of dimensions 1×1. Then $a_1 = [1/100, 0, 0]$ and $\boldsymbol{a}_2 = [0, 1/100, 0]$. For the third basis vector $\boldsymbol{a}_3$, we decide that the sensor array is positioned so that the camera center lines up with the center of the sensor array. Remember that $\boldsymbol{a}_2$ points from the camera center to the top-left corner of the sensor array, so $\boldsymbol{a}_2 = [0, 0, 1]$.

```
>>> cb = [list2vec([1/xpixels,0,0]),
          list2vec([0,1/ypixels,0]),
          list2vec([0,0,1])]
```

We find the coordinates in the camera basis of the points in `shifted_pts`:

```
>>> reps = [vec2rep(cb, v) for v in shifted_pts]
```

5.9.6 ... to pixel coordinates

Next we obtain the projections of these points onto the image plane:

```
>>> in_camera_plane = [scale_down(u) for u in reps]
```

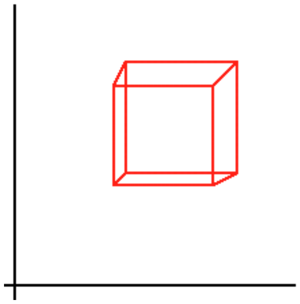
Now that these points lie in the camera plane, their third coordinates are all 1, and their first and second coordinates can be interpreted as pixel coordinates:

```
>>> pixels = [pixel(u) for u in in_camera_plane]
```

To see the result, we can use the `plot` procedure from the `plotting` module.

```
>>> plot(pixels, 30, 1)
```

However, keep in mind that increasing second pixel coordinate corresponds to moving downwards, whereas our `plot` procedure interprets the second coordinate in the usual mathematical way, so the plot will be a vertical inversion of what you would see in an image:



5.10 Computational problems involving finding a basis

Bases are quite useful. It is important for us to have implementable algorithms to find a basis for a given vector space. But a vector space can be huge—even infinite—how can it be the input to a procedure? There are two natural ways to specify a vector space $\mathcal{V}$:

1. Specifying generators for $\mathcal{V}$. This is equivalent to specifying a matrix A such that $\mathcal{V} = \text{Col } A$.

2. Specifying a homogeneous linear system whose solution set is $\mathcal{V}$. This is equivalent to specifying a matrix A such that $\mathcal{V} = \text{Null } A$.

For each of these ways to specify $\mathcal{V}$, we consider the Computational Problem of finding a basis.

Computational Problem 5.10.1: *Finding a basis of the vector space spanned by given vectors*

- *input:* a list $[\boldsymbol{v}_1, \ldots, \boldsymbol{v}_n]$ of vectors
- *output:* a list of vectors that form a basis for Span $\{\boldsymbol{v}_1, \ldots, \boldsymbol{v}_n\}$.

You might think we could use the approach of the Subset-Basis Lemma (Lemma 5.6.11) and the procedure `subset_basis(T)` of Problem 5.14.17, but this approach depends on having a way to tell if a vector is in the span of other vectors, which is itself a nontrivial problem.

Computational Problem 5.10.2: *Finding a basis of the solution set of a homogeneous linear system*

- *input:* a list $[\boldsymbol{a}_1, \ldots, \boldsymbol{a}_m]$ of vectors
- *output:* a list of vectors that form a basis for the set of solutions to the system $\boldsymbol{a}_1 \cdot \boldsymbol{x} = 0, \ldots, \boldsymbol{a}_m \cdot \boldsymbol{x} = 0$

This problem can be restated as

Given a matrix

$$A = \begin{bmatrix} \boldsymbol{a}_1 \\ \hline \vdots \\ \hline \boldsymbol{a}_m \end{bmatrix},$$

find a basis for the null space of A.

An algorithm for this problem would help us with several Questions. For example, having a basis would tell us whether the solution set was trivial: if the basis is nonempty, the solution set is nontrivial.

In Chapters 7 and 9, we will discuss efficient algorithms for solving these problems.

5.11 The Exchange Lemma

Do the Minimum Spanning Forest algorithms find truly minimum-weight spanning forests? We have seen one example of a computational problem—Minimum Dominating Set–for which greedy algorithms sometimes fail to find the best answer. What makes Minimum Spanning Forest different?

5.11.1 The lemma

We present a lemma, the Exchange Lemma, that applies to vectors. In Section 5.4.3 we saw the close connection between MSF and vectors. In Section 5.11.2, we use the Exchange Lemma to prove the correctness of the Grow algorithm for MSF.

Lemma 5.11.1 (Exchange Lemma): Suppose S is a set of vectors and A is a subset of S. Suppose $\boldsymbol{z}$ is a vector in Span S and not in A such that $A \cup \{\boldsymbol{z}\}$ is linearly independent. Then there is a vector $\boldsymbol{w} \in S - A$ such that Span $S =$ Span $(\{\boldsymbol{z}\} \cup S - \{\boldsymbol{w}\})$.

It's called the *Exchange* Lemma because it says you can inject a vector $\boldsymbol{z}$ and eject another vector without changing the span. The set A is used to keep certain vectors from being ejected.

Proof

Write $S = \{\boldsymbol{v}_1, \ldots, \boldsymbol{v}_k, \boldsymbol{w}_1, \ldots, \boldsymbol{w}\ \}$ and $A = \{\boldsymbol{v}_1, \ldots, \boldsymbol{v}_k\}$. Since $\boldsymbol{z}$ is in Span S, it can be expressed as a linear combination of vectors in S:

$$\boldsymbol{z} = \alpha_1\, \boldsymbol{v}_1 + \cdots + \alpha_k\, \boldsymbol{v}_k + \beta_1\, \boldsymbol{w}_1 + \cdots + \beta\ \ \boldsymbol{w} \tag{5.3}$$

If the coefficients $\beta_1, \ldots, \beta\ $ were all zero then we would have $\boldsymbol{z} = \alpha_1\, \boldsymbol{v}_1 + \cdots + \alpha_k\, \boldsymbol{v}_k$, contradicting the linear independence of $A \cup \{\boldsymbol{z}\}$. Thus the coefficients $\beta_1, \ldots, \beta\ $ cannot all be zero. Let β_j be a nonzero coefficient. Then Equation (5.3) can be rewritten as

$$\begin{aligned}\boldsymbol{w}_j = (1/\beta_j)\ \boldsymbol{z} + (-\alpha_1/\beta_j)\ \boldsymbol{v}_1 + \cdots + (-\alpha_k/\beta_j)\ \boldsymbol{v}_k + (-\beta_1/\beta_j)\ \boldsymbol{w}_1 + \ldots + (-\beta_{j-1}/\beta_j)\ \boldsymbol{w}_{j-1} \\ +(-\beta_{j+1}/\beta_j)\ \boldsymbol{w}_{j+1} + \cdots + (-\beta\ /\beta_j)\ \boldsymbol{w}\end{aligned} \tag{5.4}$$

By the Superfluous-Vector Lemma (Lemma 5.5.1),

$$\text{Span }(\{z\} \cup S - \{\boldsymbol{w}_j\}) = \text{Span }(\{z\} \cup S) = \text{Span } S$$

□

We use the Exchange Lemma in the next section to prove the correctness of the Grow algorithm for MSF. In the next chapter, we use the Exchange Lemma in a more significant and relevant way: to show that all bases for a vector space $\mathcal{V}$ have the same size. This is the central result in linear algebra.

5.11.2 Proof of correctness of the Grow algorithm for MSF

We show that the algorithm GROW(G) returns a minimum-weight spanning forest for G. We assume for simplicity that all edge-weights are distinct. Let F^* be the true minimum-weight spanning forest for G, and let F be the set of edges chosen by the algorithm. Let $e_1, e_2, \ldots, e_m$ be the edges of G in increasing order. Assume for a contradiction that $F = F^*$, and let e_k be the minimum-weight edge on which F and F^* disagree. Let A be the set of edges before e_k that are in both F and F^*. Since at least one of the forests includes all of A and also e_k, we know $A \cup \{e_k\}$ has no cycles (is linearly independent).

Consider the moment when the Grow algorithm considers e_k. So far, the algorithm has chosen the edges in A, and e_k does not form a cycle with edges in A, so the algorithm must also choose e_k. Since F and F^* differ on e_k, we infer that e_k is *not* in F^*.

Now we use the Exchange Lemma:

- A is a subset of F^*.
- $A \cup \{e_k\}$ is linearly independent.
- Therefore there is an edge e_n in $F^* - A$ such that Span $(F^* \cup \{e_k\} - \{e_n\}) =$ Span F^*.

That is, $F^* \cup \{e_k\} - \{e_n\}$ is also spanning.

But e_k is cheaper than e_n so F^* is not a minimum-weight solution. **Contradiction.** This completes the proof of the correctness of GROW(G).

5.12 *Lab: Perspective rectification*

The goal for this lab is to remove perspective from an image of a flat surface. Consider the following image (stored in the file `board.png`) of the whiteboard in my office: