Closed-Environment Navigation System Specification

By Deegan Osmundson

Justification & Overview

Navigation in a closed environment benefits from having a memory being continuously updated by perception of that environment. New entities in the environment should be treated with caution until they can be established as a constant and unmoving presence, and if they cannot (because they are moving or changing), they should be treated with extra caution. This is because animals must be avoided and new structures can be ignored. This differentiation between the familiar and the unfamiliar can only be performed when one possesses an ongoing memory (allowing the *familiar* to exist). Additionally, a working memory of the operating environment is necessary to plan routes that are bound by a predefined border or that minimize the distance traveled.

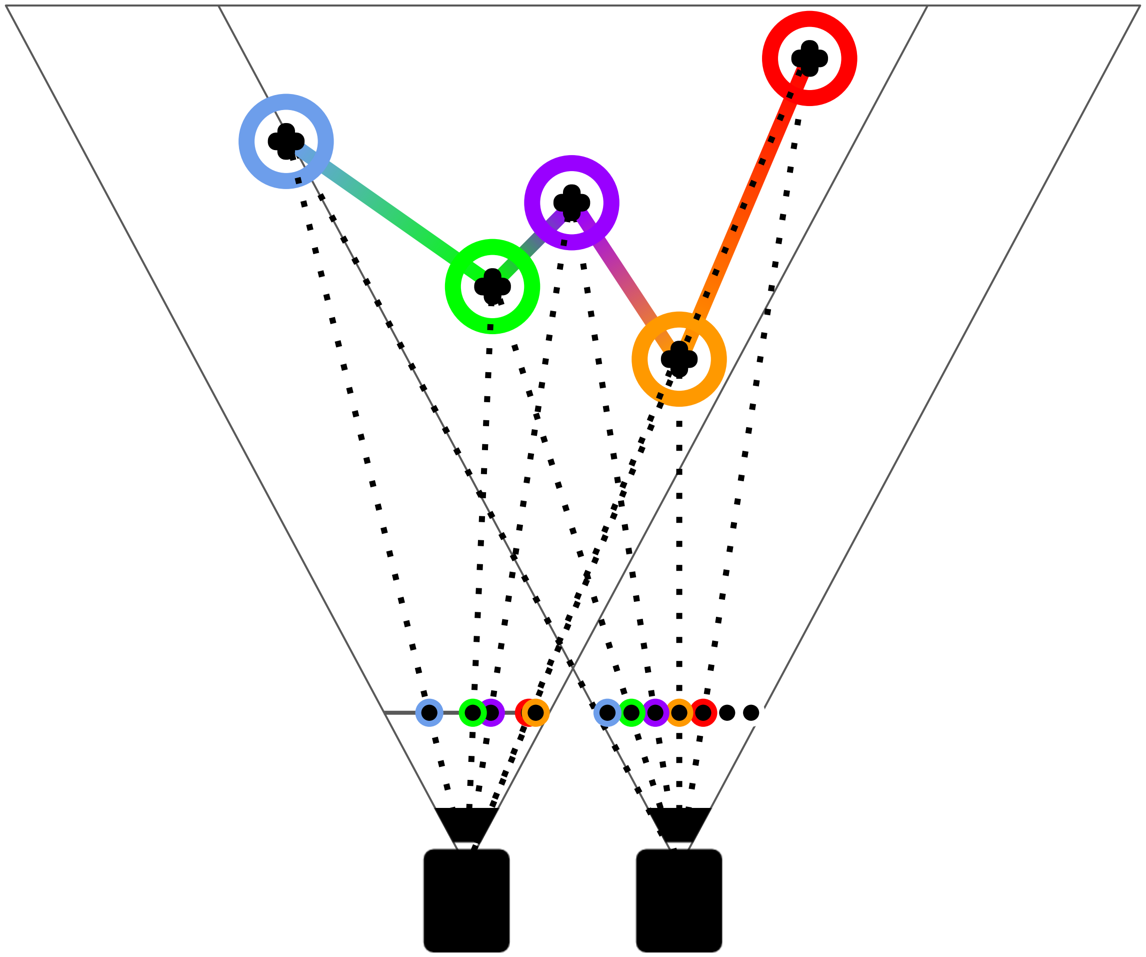
Many 3D video games use voxels to store terrain. The advantage to voxels is that they can be held in an easily readable/writable three dimensional array, in which every one voxel represents one cubic volume (of a particular size) of the environment. Unfortunately, voxels are better suited for representing volumes than surfaces, which are what we'll be working with, given that a camera can only truly observe surfaces. Suppose that each voxel is replaced by a surface defined by a plane that extends to the walls of the cubic box that bounded the voxel's volume. These surfaces would far more adequately model camera-produced visual information.

However, a three dimensional array of uniformly-sized surfaces on its own is not very searchable. Say the camera is not where it thinks it is; how can it find out where it is located based on the combination of what is memorized and what is perceived? Identifying its surroundings will be difficult without having first extrapolated recognizable patterns from its memory. Since we're focusing on surfaces on the smaller scale, we could generalize adjacent and aligned surfaces into larger faces of nonuniform size/shape. From these structures, visual/physical patterns could be extracted, memorized, and searched. Because they are nonuniform, faces wouldn't be nearly as efficient to write to, so they shouldn't replace surfaces altogether. Instead, a 3D array of surfaces should be targeted for frequent updates, whereas a list of faces should be recompiled (off of the surfaces) more periodically.

This specification describes how such a system might be made to work. As indicated by the final section, the navigation system is intended for use in automatic lawn mowers. Throughout the specification, procedures are written in a sort of pseudo-code whose syntax is built upon by every consecutive procedure, so if syntax isn't defined on a page, it was probably defined previously. The wording of a procedure's heading is particularly important to remember, since references to the procedure aren't as explicit as a function call in actual code. Also, bits and pieces of vocabulary are informally dropped throughout, so if a phrase isn't defined somewhere, it may have been defined earlier.

Contents

Environmental Perception



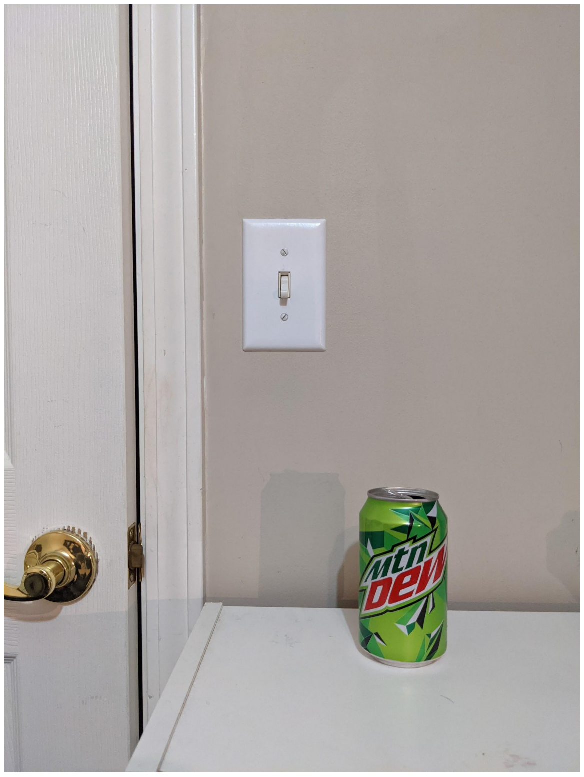


Photo 1 (Left Camera)

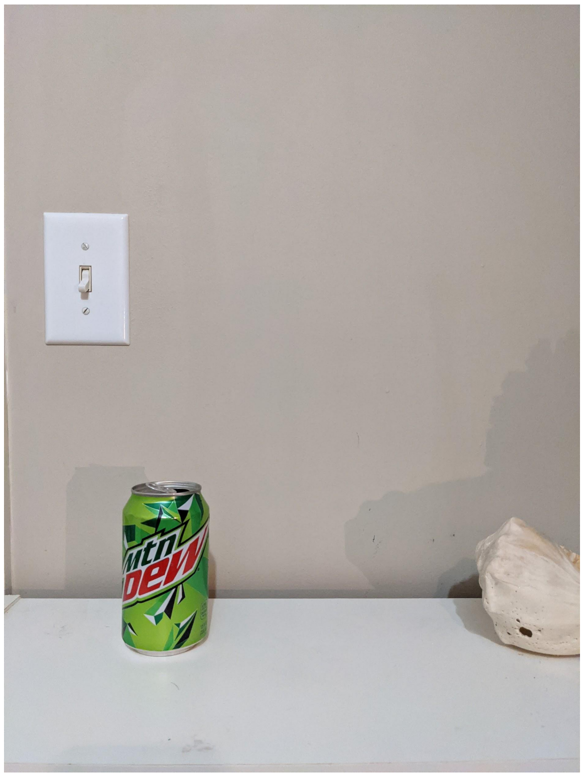
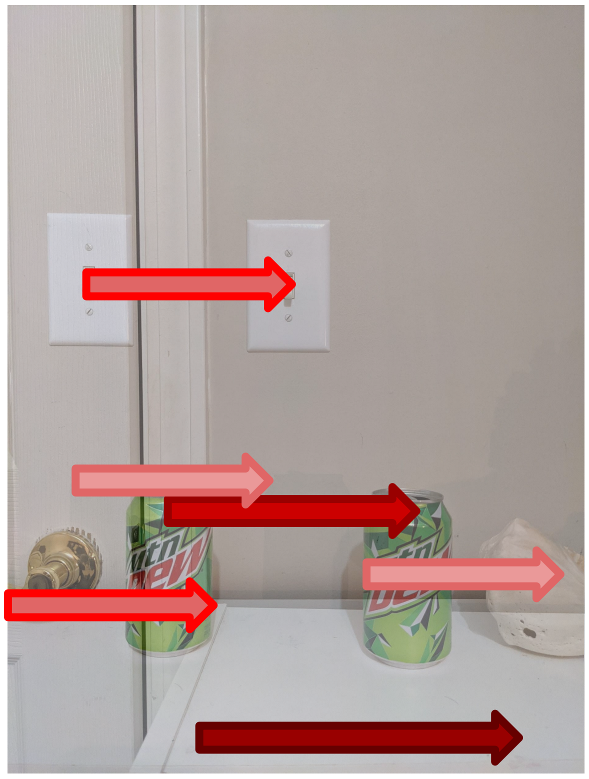


Photo 2 (Right Camera)

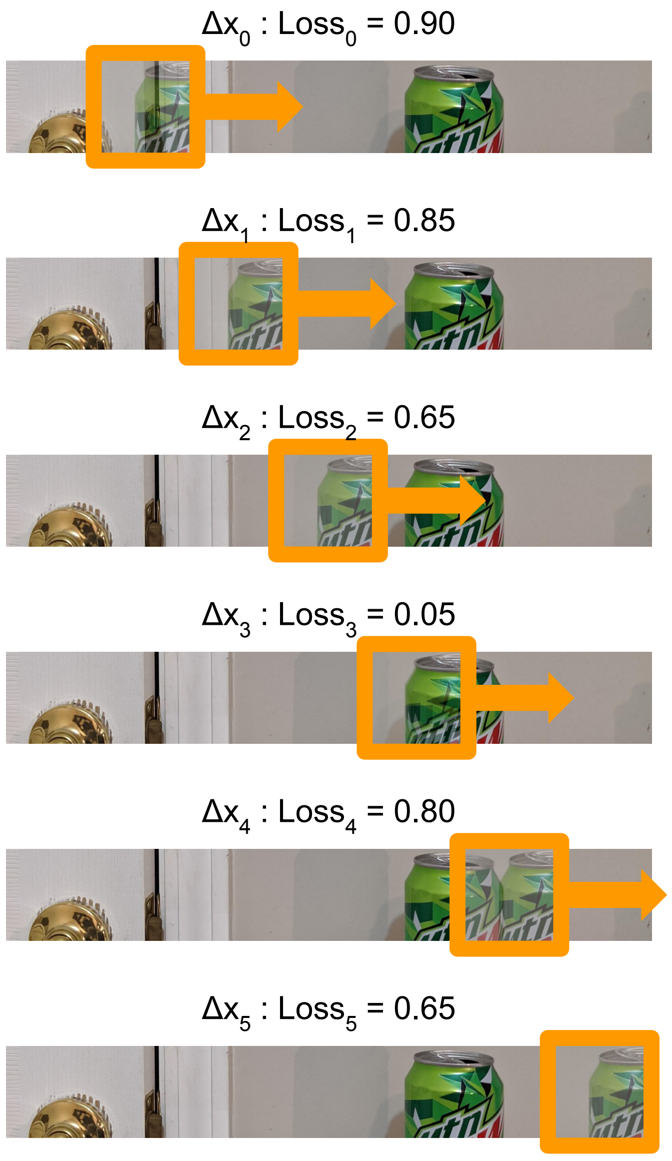
Depth perception requires two separate perspectives of the same scene. Here, two photos were taken by side-by-side cameras at equal depths with identical orientations.

In order to determine the depth of a real-world point visible in both photos, the difference between its projections to each of the cameras must first be measured. This measurement is the horizontal shift in pixels from that point's location in one image to its location in the other. Each arrow represents a shift from Photo 2 to Photo 1. Notice that the shorter shifts, represented by lighter shaded arrows, occur at slightly greater depths. Also notice that all shifts are in the same direction, from left to right. The opposite would be true for shifts from Photo 1 to Photo 2.

Pixel Shift Introduction


Photos Overlapped

Determining Pixel Shift

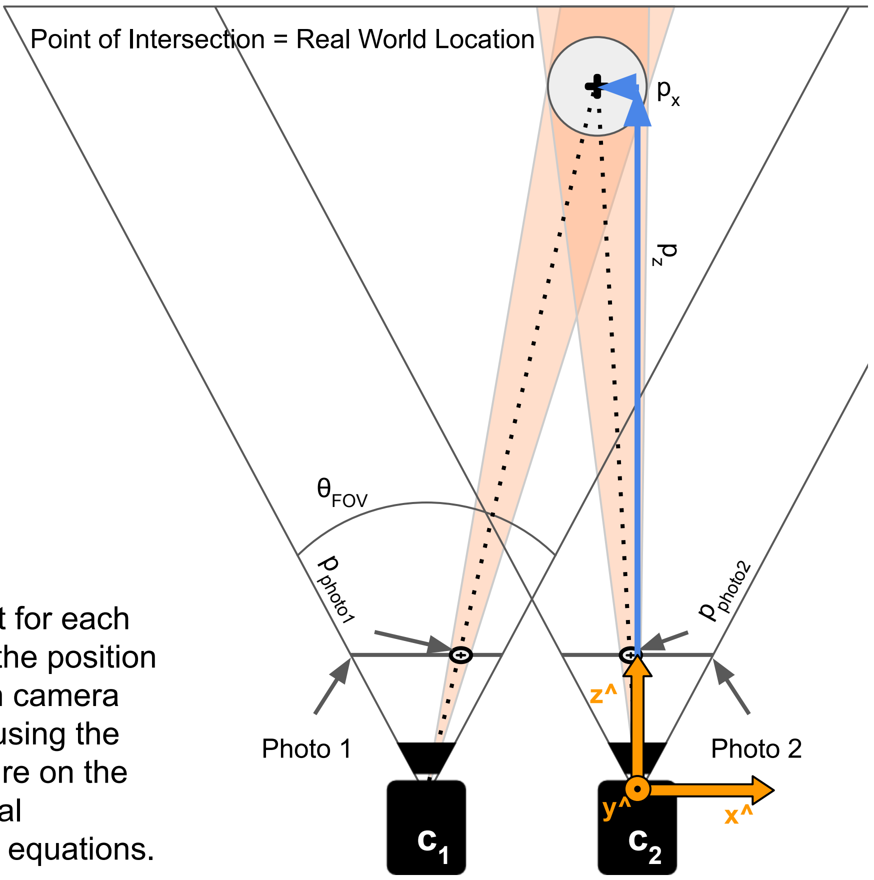

Δx_0 : $\text{Loss}_0 = 0.90$
Δx_1 : $\text{Loss}_1 = 0.85$
Δx_2 : $\text{Loss}_2 = 0.65$
Δx_3 : $\text{Loss}_3 = 0.05$
Δx_4 : $\text{Loss}_4 = 0.80$
Δx_5 : $\text{Loss}_5 = 0.65$

In order to find the pixel shift for a given point on Photo 2, the point must be compared with several points along the same row of Photo 1. The difference between the point's position on Photo 2 and on Photo 1 defines its pixel shift.

The figure on the left shows a small square region of Photo 2, also called a pixel batch, being compared with regions of equal size along a row of Photo 1. Because the projection of a point to Photo 1 is always farther to the right than its projection to Photo 2, pixel batches, which are these projections to Photo 2, can only be shifted to the right. In the figure, the step size (distance between tested shifts) appears a little less than the width of the batch. However, this does not necessarily have to be the case; the step size could be set to as little as 1 pixel for maximum precision.

A loss value is calculated for each tested shift, measuring the difference between the pixel batch and the region of Photo 1. The figure shows the loss Loss_n calculated for each shift Δx_n. It can be observed that $\text{Loss}_3 < \text{Loss}_0, \text{Loss}_1, \dots, \text{Loss}_5$, so Δx_3 is the most accurate shift for this pixel batch.

This process of determining a pixel shift is performed for every pixel batch. Accompanying each batch's loss-minimizing shift is a measure of certainty, which measures how much more accurate the selected shift is than the other options (certainty = avg loss / min loss).



Once the pixel shift for each pixel batch is calculated, the position of the point at its center in camera space can be calculated using the below equations. The figure on the right illustrates the physical parameters used in these equations.

Known Variables:

θ_{FOV} = angle of field of view

$p_{photo1[x]}$ = horizontal component of the point's position (in units of pixels) on Photo 1

$p_{photo2[x]}$ = horizontal component of the point's position (in units of pixels) on Photo 2

$p_{photo1[x]} - p_{photo2[x]}$ = batch's pixel shift from Photo 2 to 1

w = width of photo in units of pixels

h = height of photo in units of pixels

$c_{1[x]}$ = horizontal component of position of camera 1

$c_{2[x]}$ = horizontal component of position of camera 2

Unknown Variables

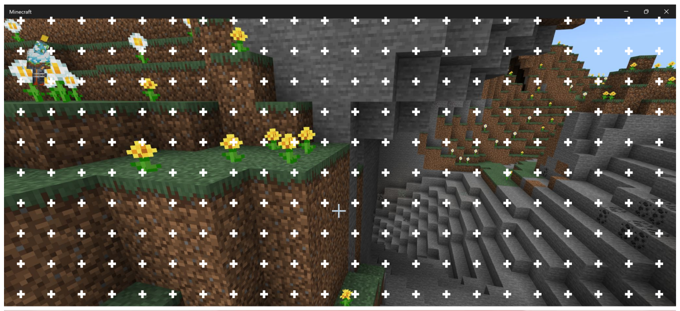
p_z, p_x, p_y = depth, horizontal, and vertical components of the point's position in camera space (defined by orange basis vectors)

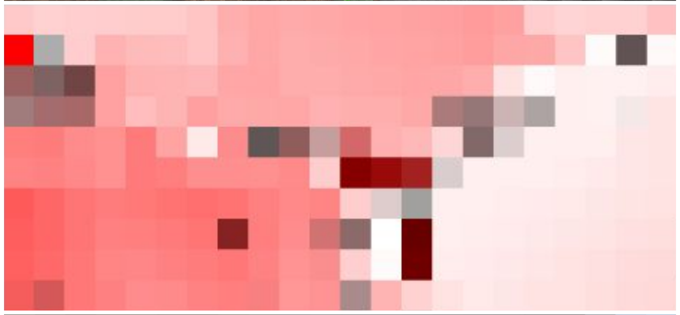
$p_z = (c_{2[x]} - c_{1[x]})\max(w, h) / 2(p_{photo1[x]} - p_{photo2[x]})\tan(\theta_{FOV})$

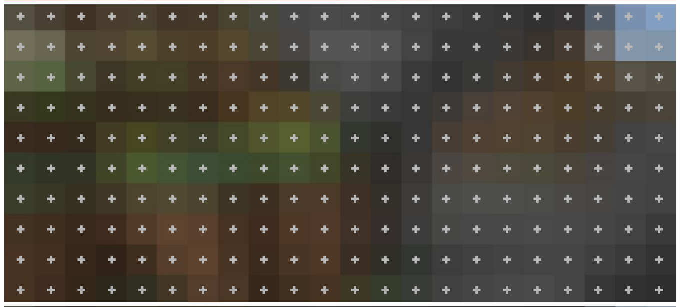
$p_x = 2p_z*(p_{photo2[x]} - w/2)*\tan(\theta_{FOV}) / \max(w, h)$

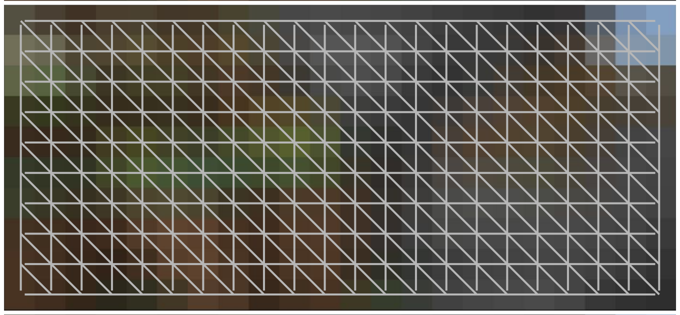
$p_y = 2p_z*(p_{photo2[y]} - h/2)*\tan(\theta_{FOV}) / \max(w, h)$

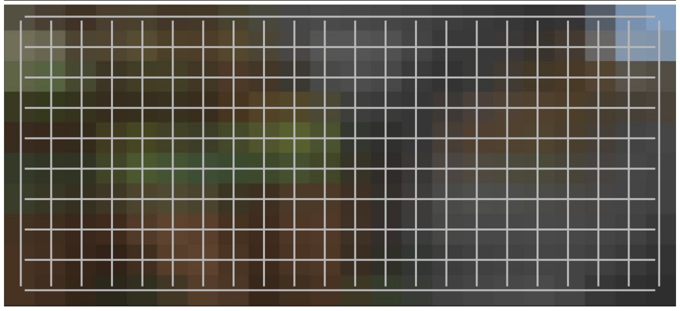
Point Location











Quad Extraction

Shifts Phase:

For each pixel batch (each center is marked by a '+' in figure 1), a shift and certainty are found as previously described. Figure 2 illustrates both the shift and the certainty determined for each batch, with redness indicating greater shift, white indicating lesser shift, and darkness indicating lesser certainty.

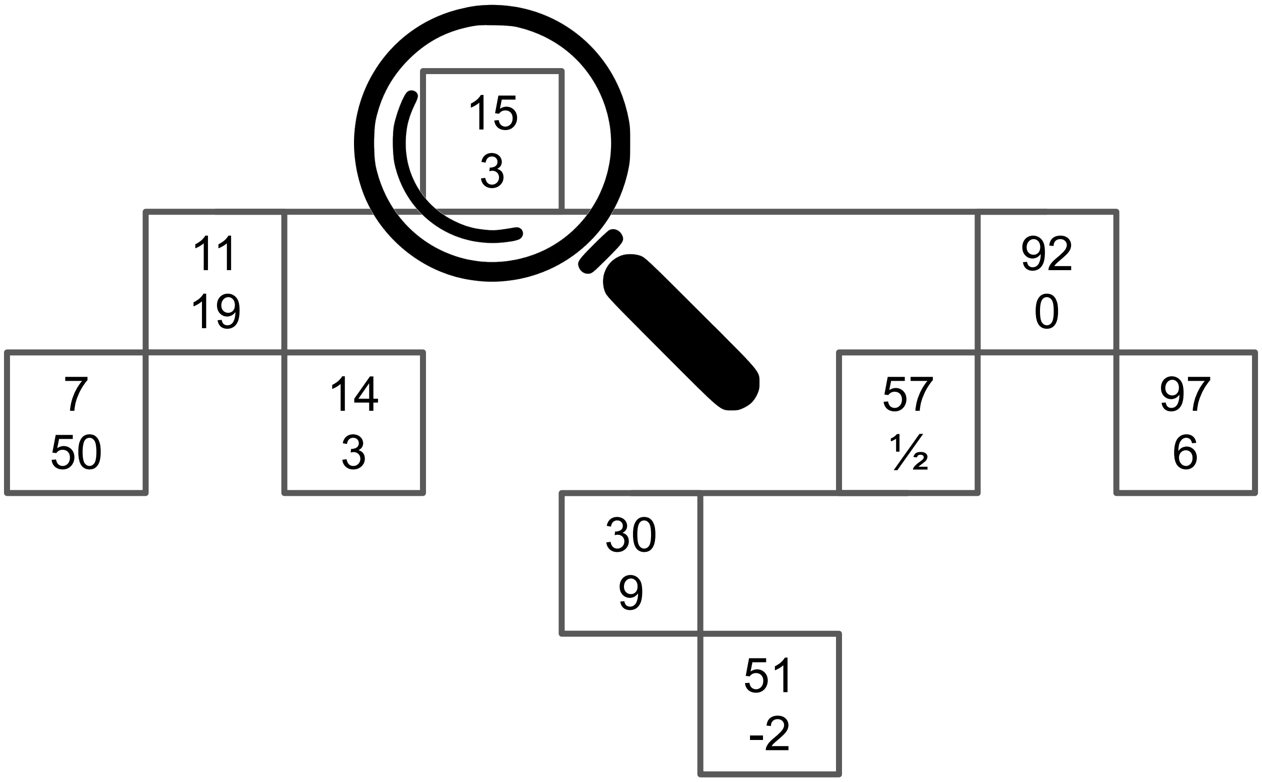
Vertices Phase:

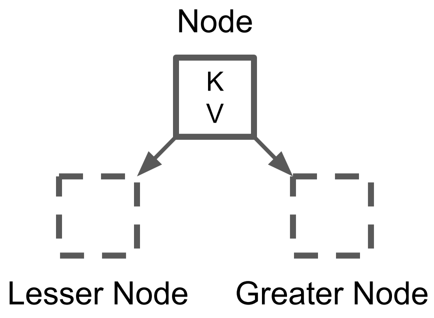
The position in camera space of each batch's center point is calculated as previously described, and each batch's average color (shown in figure 3) is also computed. Together, the point, color, and certainty corresponding to each batch describe a vertex.

Quads Phase:

Vertices are joined with adjacent vertices to form triangles (shown in figure 4), each having its own normal vector. Every pair of diagonally adjacent triangles is combined into a quad (shown in figure 5), whose normal is an average of the two triangles' normals. Each quad is also assigned a center point, color, and certainty, all of which are weighted averages (weighed by certainty) of its vertices. The resulting array of quads is the final product of environmental perception and is named the Perceived Quads.

Search Structures





Anatomy of the Node

Each node contains the following:

- key (K) - a value by which the node is sorted
- value (V)
- size (1 + number of nodes below)
- pointer to a "lesser" node with a key < K
- pointer to a "greater" node with a key > K

It is possible for a node's lesser node or greater node pointers to be null (point to nothing).

Syntax Notes:

"[K:V]" represents a key-value pair of key K and value V, and "//" prefaces a note or comment.

Inserting/adding [K:V] into a node N:

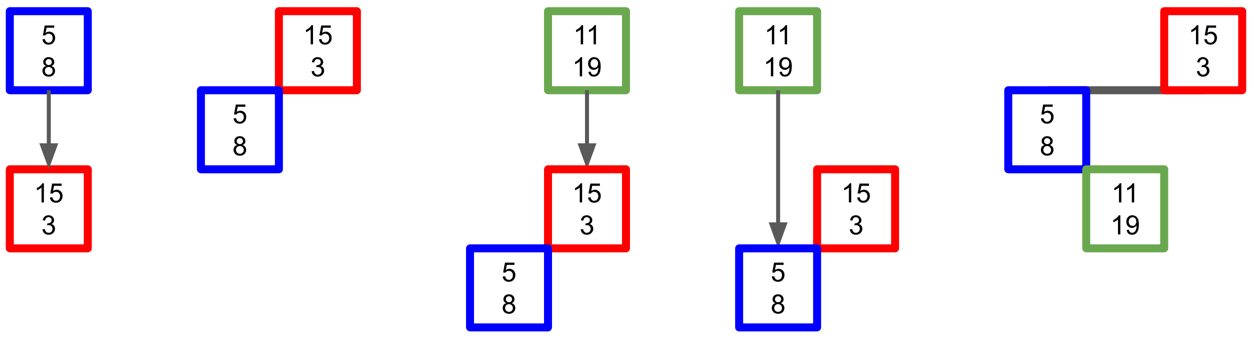
```
// For N to "pass" the pair [K:V] to another node M
    would mean for it to insert the pair into M if it had
    been inserted into N or add the pair to M if it had
    been added to N.
If K = N's key:
    If inserting, N's value is overwritten with V;
    If adding, V is added to N's value;
        //If the value of N is a list, adding will append
            N's list with the value or list of V;
If K < N's key:
    If N already points to a lesser node, [K:V] is
        passed to N's lesser node;
    Otherwise, N's lesser node pointer is pointed to a
        new node with key K and value V;
If K > N's key:
    If N already points to a greater node, [K:V] is
        passed to N's greater node;
    Otherwise, N's greater node pointer is pointed to a
        new node with key K and value V;
```

Note:

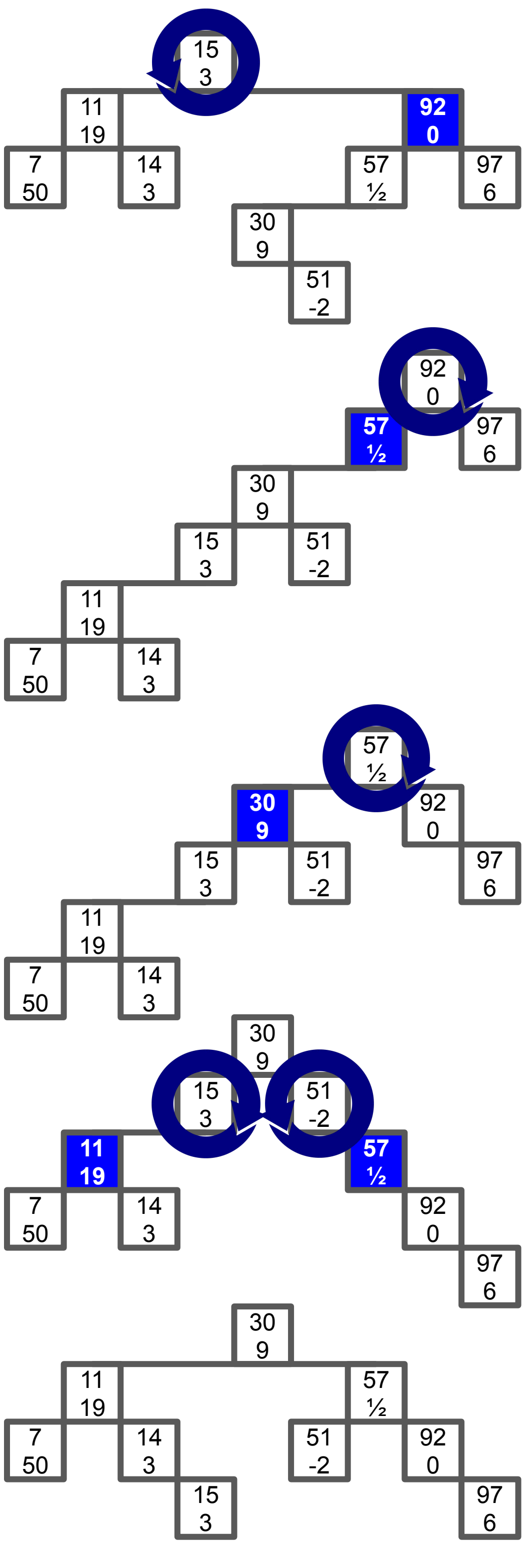
If node N is inserted into another node, N's pointers overwrite the pointers of the node that its key-value pair is copied to.

Example (referring to stages of the figure)**:**

1. The blue node is inserted (represented by arrow) into the red node.
2. Blue's key is greater than that of red, and because red does not yet point to a lesser node, blue is made red's new lesser node.
3. Green is inserted into red.
4. Green's key is less than that of red, but because red already points to blue as its lesser node, green is inserted into blue.
5. Green's key is greater than that of blue, and because blue does not yet point to a greater node, green is made blue's new greater node.



Constructing a Binary Search Tree



When is a binary search tree unbalanced?

A binary search tree is considered unbalanced if the difference between the sizes of each side of any node is > 1. After multiple insertions, a binary tree is likely to have become unbalanced.

Why balance binary search trees?

Inserting new nodes into or searching a balanced tree take fewer steps on average because all nodes are as close as possible to the top node.

How are binary trees balanced?

A binary search tree is balanced by balancing each individual node. To balance node N, N must be rotated until the side-side size difference is less than or equal to one. The procedure for rotating N once is as follows:

1. Identify which of N's lesser or greater nodes is larger in size.
2. Copy N's key, N's value, and N's smaller node pointer into a new node M (but don't insert M anywhere yet).
3. Replace N's contents (key, value, and pointers) with those of its larger node.
4. Insert M into the larger node. Because M and everything below it (the former smaller side of N) is all lesser than or all greater than N and everything below it, M will settle at an extreme side of the tree.

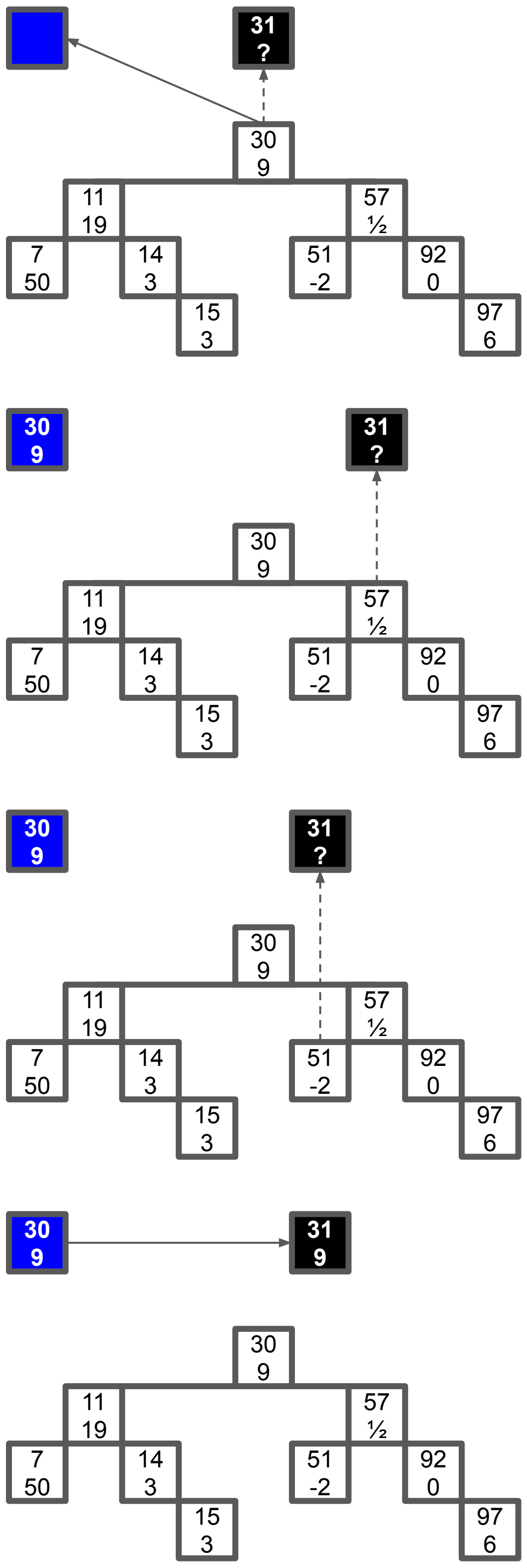
The top node is first balanced, then the nodes below it (its lesser and greater nodes), then the nodes below them, and so on and so forth.

Example (referring to stages of the figure)**:**

1. The top node's greater (right) side is two nodes **larger**, so the top node is rotated down to the left, moving it and its lesser (left) side to the far left of its right side.
2. The new top node's lesser (left) side is six nodes **larger**, so the top node is rotated down to the right.
3. The new top node is again rotated down to the right because its left side is still four nodes **larger**.
4. The top node is finally balanced, so the nodes below it are balanced next. Each of these lower nodes has a side-side size difference of three, so they are rotated accordingly.
5. All nodes are at equilibrium because no node shows a side-side size difference greater than one.

Notice that in every stage, the left-right order of nodes remains the same, despite their heights in the tree frequently changing.

Balancing a Binary Search Tree



Searching a BST

What is the purpose of searching?

Binary search trees store values and sort them by key, similar to how dictionaries store definitions and sort them alphabetically by the words they define. Therefore, the objective of searching is to find the value corresponding to a provided key. In the context of this project, in the case that a search doesn't find a node with a matching key, the search will instead return the value of the closest node it can find.

How does searching work?

Searching works similar to insertion, and the procedure below is written using the same syntax (as will be many future procedures).

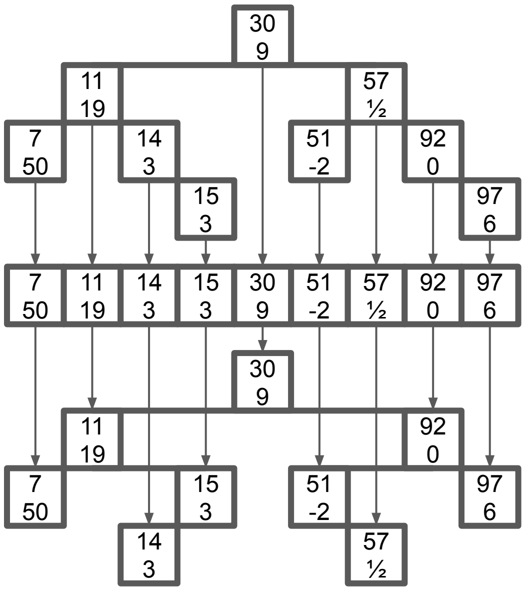
Searching node N for key K:

```
// [$K_C$:$V_C$] is the key-value pair whose key is the closest to K
  out of all nodes that have already been searched. $V_C$ is only
  ever returned if no node has a key exactly equal to K.
  [$K_C$:$V_C$] starts out by default as the key-value pair of the
  tree's top node, and if N searches its lesser or greater node
  for K, that node refers to the same [$K_C$:$V_C$].
If K = N's key, return N's value;
If |N's key - K| < |$K_C$ - K|:
  Set [$K_C$:$V_C$] equal to a copy of N's key-value pair;
If K < N's key:
  If N points to a lesser node, search N's lesser node for K and
    return the value returned by the search;
  Otherwise, return $V_C$;
If K > N's key:
  If N points to a greater node, search N's greater node for K
    and return the value returned by the search;
  Otherwise, return $V_C$;
```

Example (refers to stages of the figure):

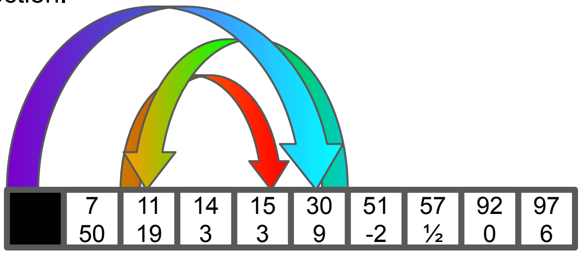
The blue box represents a copy of the current closest key-value pair (called [K_C:V_C] in the procedure), which is relied on for a return value if no node is found to have the exact desired key. The black box just represents the provided key and the empty spot in which the search's corresponding return value will be placed. The box from which the dashed arrow is drawn is the node actively being searched.

1. The tree's top node is searched for a key of 31, and its contents are also copied to [K_C:V_C].
2. 31 is greater than 30, the key of the top node, so the node on the right (the greater node) is searched next. Its key of 57 is no closer to 31 than 30, the current K_C, so [K_C:V_C] stays the same.
3. 31 is lesser than 57, so the node down to the left (the lesser node) is searched next. Its key of 51 too is no closer to 31 than K_C.
4. 31 < 51, but there are no lower nodes left to search through, so V_C becomes the return value.



The figure on the left first illustrates the conversion of a BST to a BSA. Below that is a BST-style representation of the order in which the new search algorithm will search through the array's key-value pair elements.

The figure below illustrates this search order, as described in the procedure for searching an array (see text below), for the search of a key of 15. First the 30:9 pair is searched, then 11:19, and lastly 15:3. The bottom BST from the figure to the left corroborates this pattern upon inspection.



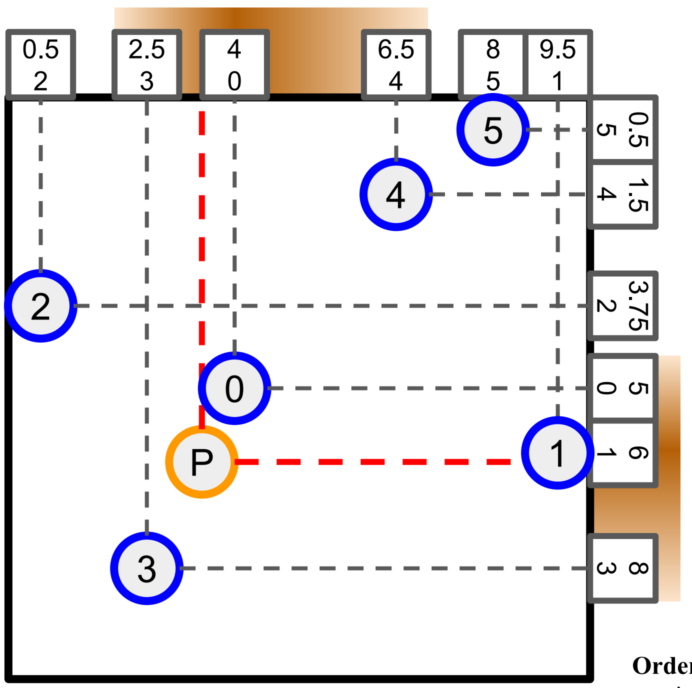
A binary search array is a one-dimensional array of key-value pairs, ordered by key, that cannot (as far as we're concerned) be modified. A BST is first used to collect and sort all of the elements (as nodes), which can then be exported to this array. Searching a BSA is similar to searching a BST, but a BSA returns the index of the key-value pair with the matched/closest key instead of returning its value. This allows for easier access to not only that one key-value pair, but also the ones surrounding it.

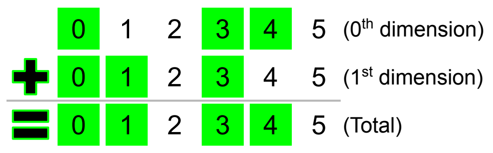
Searching array A for key K:

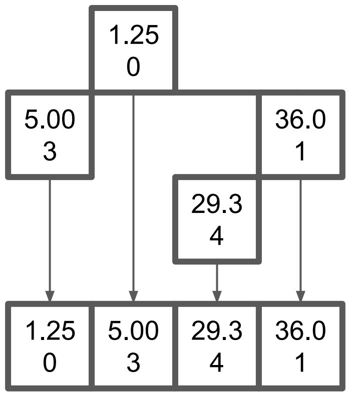
```
// A is an array of key-value pairs
// A[i] represents the $i^{th}$ key-value pair of A
The Marker is the index of the array element being
    searched; Marker is initialized to -1;
Direction is either + or -, representing whether the
    next element to be searched is to the right (+) or
    left (-) of the Marker; Direction is initialized to +;
Continuous is a binary digit (either 0 or 1) that tells
    whether or not the former Direction is equal to the
    new Direction;
Size measures the number of array elements left to
    search through; Size is initialized equal to the full
    length of the array;
Range is a measurement used to determine how far
    the Marker will move to arrive at its next location
    (the actual distance moved will be Range + 1);
$K_C$ is the closest key out of the keys of all of the
    items of A that have been searched by the present
    iteration of the loop below; $K_C$ is initialized equal
    to the key of A[0];
$I_C$ is the index of the key-value pair from which $K_C$
    was taken; $I_C$ is initialized equal to 0;

1.If Size = 0, return $I_C$;
2.Range is set equal to the floored division of Size
    by 2;
3.Direction * (Range + 1) is added to the Marker;
4.K is compared with the key of A[Marker];
    a.If equal, Marker is returned;
    b.If the key of A[Marker] is closer to K than $K_C$,
        $K_C$ is set equal to A[Marker] and $I_C$ is set equal
        to Marker;
    c.If K is lesser, Direction is set equal to -1;
    d.If K is greater, Direction is set equal to +1;
5.The former Direction is compared with the new
    one;
    a.If equal, Continuous is set equal to 1;
    b.Otherwise, Continuous is set equal to 0;
6.Size is set equal to Range - Continuous * (1 + 2 *
    Range - Size);
    // (1 + 2 * Range - Size) returns 0 if Size is odd
        and 1 if Size is even.
7.Jump back to Step 1;
```

Binary Search Arrays







Spacial Sets

A BSA can be used to order a collection of points by their positions along one dimension, where the keys are their positions (along that one dimension) and the values are their serial numbers. A set of n BSAs of equal length, the ith of which orders the same collection of points by their positions along the ith dimension, is called an n-dimensional spacial set.

When trying to order a long list of points by their distance to point p, it can be a huge time-saver to only look through the ones that are somewhat close to p in at least one dimension. Once these points are identified, their serial numbers can be ordered by the points' squared distance to p using a single BST, which can finally be converted into a BSA. This final ordered list (the BSA) won't include the points that are far away, but that's fine, assuming only the closest ones matter.

Example (referring to figures above and procedure on right):

The figure at the top left shows six points (labeled by serial number) in a 2-dimensional space which, for each dimension, are ordered by their positions in the BSA displayed on that dimension's axis. The gold point is the searched 2D position. The inclusion ranges are shown as golden rectangles behind their respective dimension's BSA, centered on the item corresponding to the closest point in that dimension. The top right figure shows two binary arrays, each defining what points fall under the inclusion range in its respective dimension, being added to make a new array (called B in the procedure). The bottom right figure first shows the BST used to order the serial numbers of the nearby points by the points' distances, then shows it being converted into the returned ordered list.

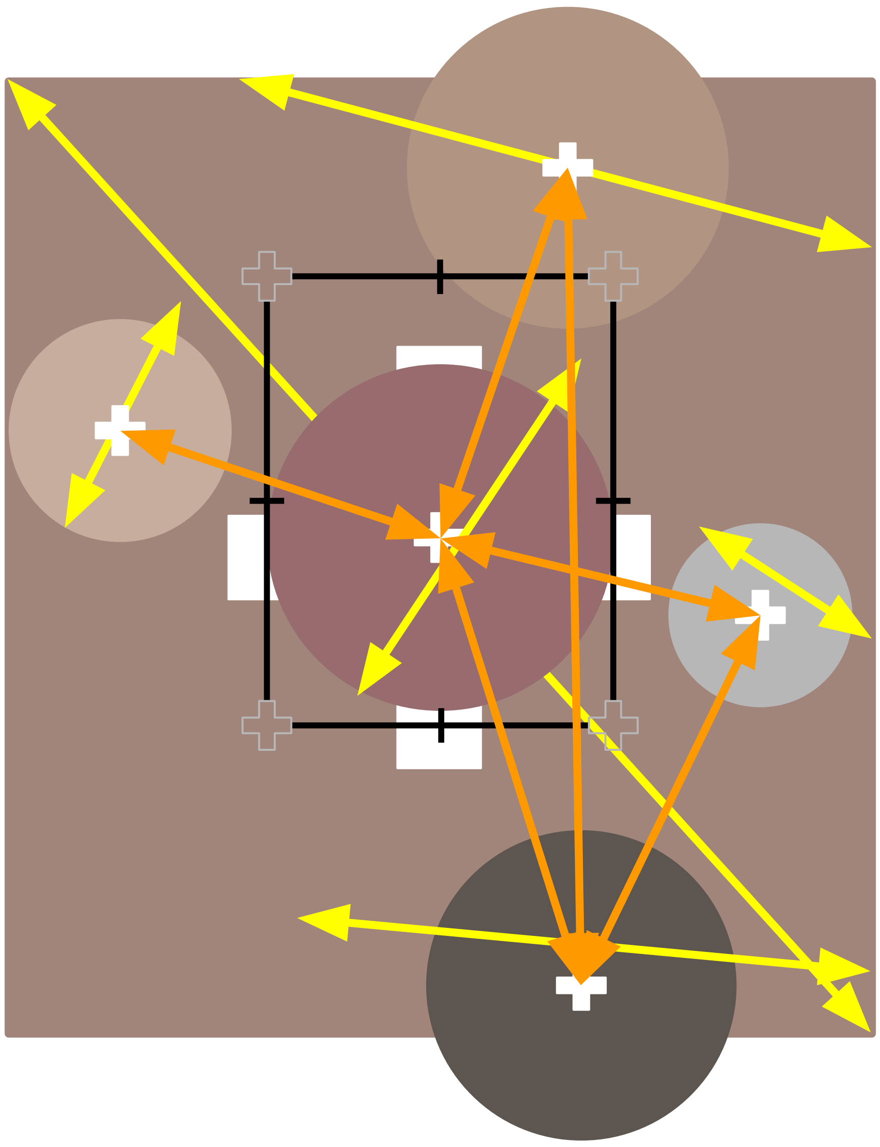
Ordering list of points L by distance to point P:

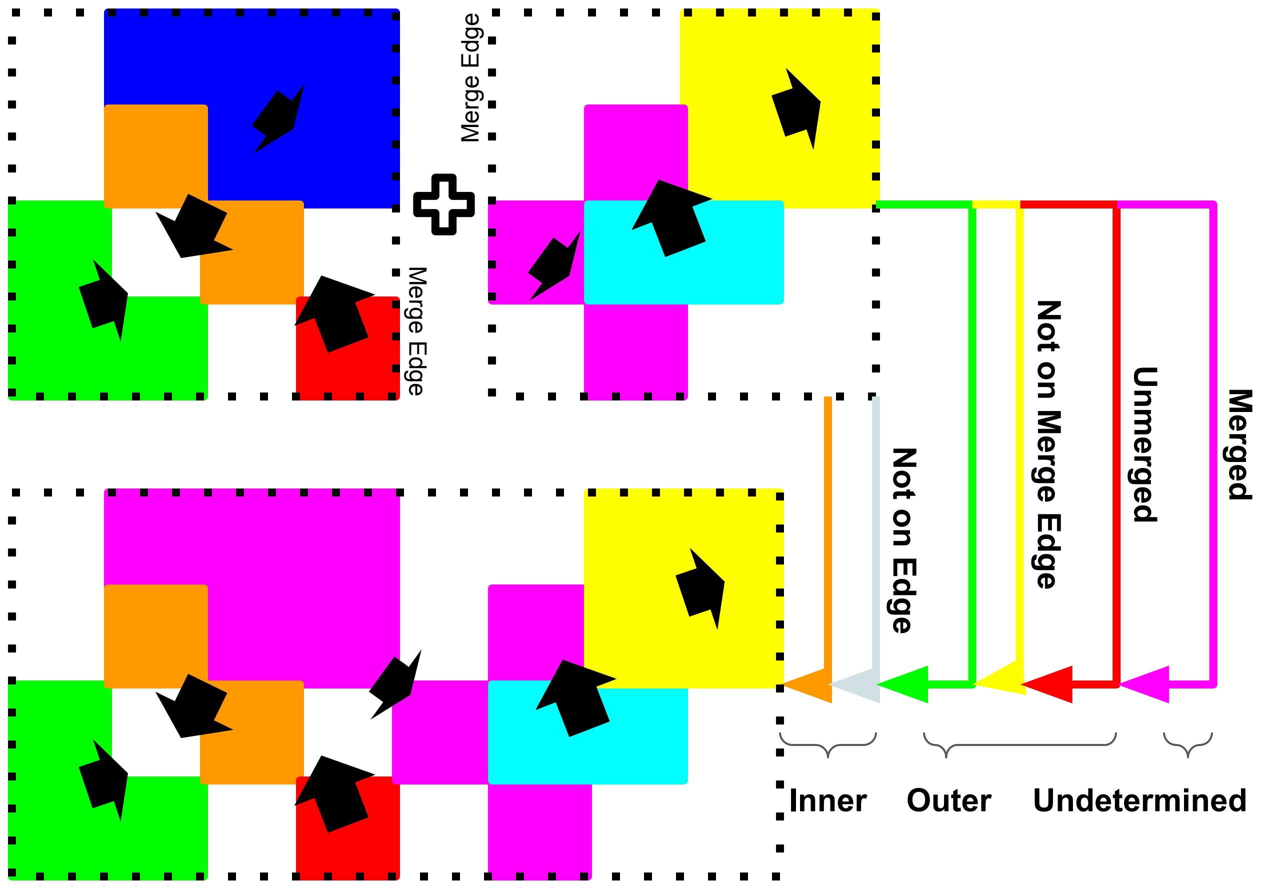
```
n is the number of dimensions of the space in which P and
    the points of L exist;
$l$ is the number of points in L;
S is the spacial set that orders the points of L in each one
    of the n dimensions;
R is the list of each dimension's inclusion range;
// $P_i$ represents the position of P in the ith dimension.
// $S_i$ represents the ith BSA of S, which orders the points of
    L by their position in the ith dimension.
// $R_i$ represents the range of inclusion for the ith dimension.
// For any list U, U[i] represents the ith item of U.
// The serial number of L[i] is given by i;
B is a new boolean array of length $l$, and every item of B is
    initialized to be false;
// B[i] says whether L[i] is somewhat close to P in at least
    one dimension. The range of inclusion for "somewhat
    close" is defined differently for each ith dimension by $R_i$.
T is a new BST whose keys are squared distances and
    whose values are serial numbers;

For each integer d $\in$ [0, n):
    c is returned by the search of the BSA $S_d$ for $P_d$;
    // The value of $S_d$[c] is the serial number of the point of
        L closest to P in the dth dimension.
    For each integer i $\in$ [max(c - $R_d$, 0), min(c + $R_d$, $l$ - 1)]:
        N is the value (serial number) of $S_d$[i];
        B[N] is set equal to true;
For each integer i $\in$ [0, $l$), only if B[i] is true:
    The key-value pair [$||L[i] - P||^2$ : i] is inserted into T;
T is converted into an array/BSA and returned;
```

Binary Search for Points

Feature Extraction



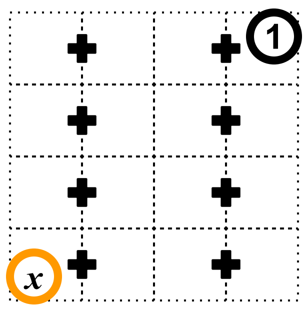


A face is a data structure used to summarize the attributes of one large nonuniform surface that is composed of many smaller uniform surfaces that are adjacent and similarly oriented. Adjacent containers of faces can be combined by one fundamental operation: the merge. Any given merge occurs along one dimension (left to right in the figure). In a merge, any face may only be combined with another face from the opposite container, which only occurs in the case that the two are adjacent and lie on similar planes. In the figure, the order of arrows (left to right) indicates the order in which the different types of faces can be copied from the original two containers to the new one. When merging containers, the first faces to carry over are those that do not touch an edge, because without touching an edge, it would be impossible to touch faces from the other container. Next, for that same reason, faces that don't touch the merge edge (the edge on which the containers come in contact) can be carried over. Now, any remaining faces whose planes match are combined into one face, and the rest that weren't matched are copied over last.

Face Compilation

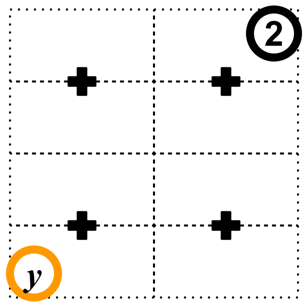
Each face, in addition to storing a plane, stores a list of indices, which may refer to its constituent perceived quads or plane cubes (described later), both of which are stored in their own array or array-like structure. The squares in the figure represent these constituent surfaces, and each collection of squares of all the same color represent a face whose normal vector is the black arrow positioned at its center. When two faces are combined, the new face adopts all constituent surfaces of both original faces. The purpose of keeping track of a face's constituents is to be able to create a texture for the face. This is done by rendering its constituent surfaces from a direction orthogonal to the face (and therefore roughly orthogonal to its constituent surfaces).

The figures on the right illustrate the face containers (container contents not shown) within a 2D array being progressively combined until they are represented by just one face container. The dimension in which these combinations occur cycles between the 2 dimensions, which are called x and y. The number of cycles necessary to fully combine an array of face containers is $\lceil \log_2(n) \rceil$, where n is the width of the square/cube array.



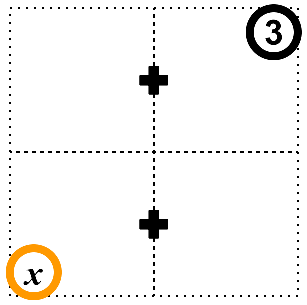
Combination Steps:

1. Face containers within a 4×4 2D array are combined with their neighbors in the x dimension.
2. The face containers (which are each twice as wide as the ones from Step 1) in the new 2×4 2D array are combined with their neighbors in the y dimension.
3. The face containers in the new 2×2 2D array are combined with their neighbors in the x dimension.
4. The last two face containers are combined with each other in the y dimension.

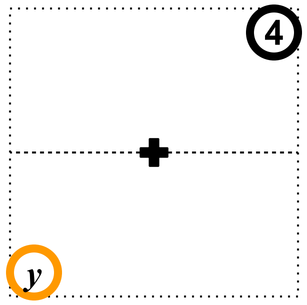


Cycle (period = 2 combination steps (for 2D array)):

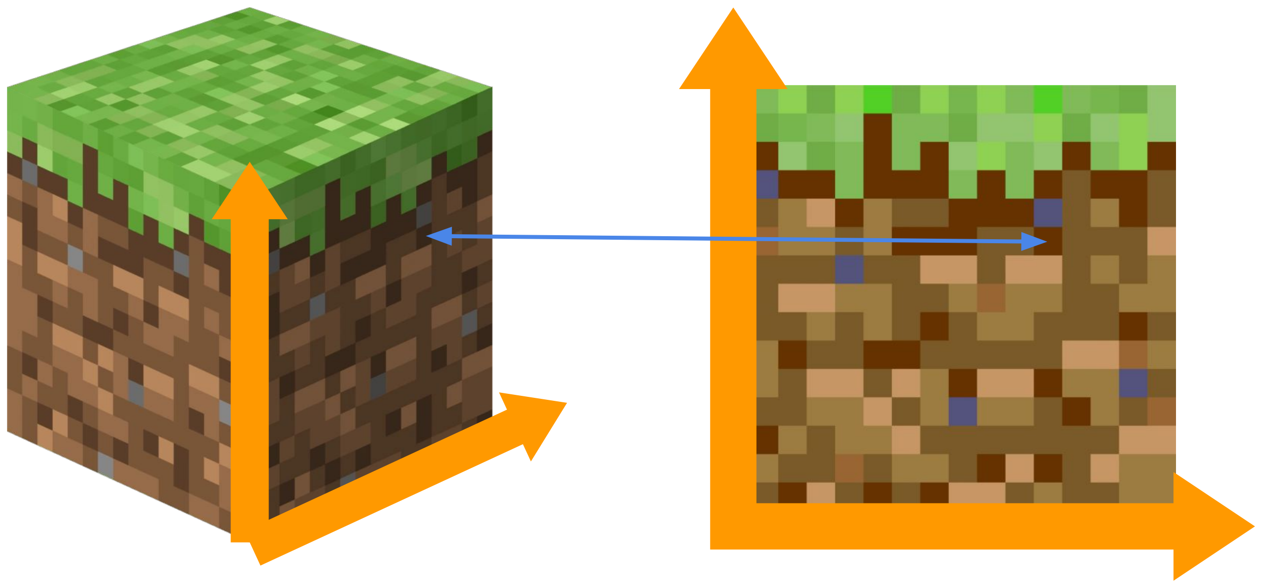
1. Face containers are combined with their neighbors in the x dimension.
2. Face containers are combined with their neighbors in the y dimension.



If a 4×4×4 3D array were used instead, the same number of cycles would occur (2 cycles). However, the period of each cycle would be 3 combinations instead of 2, since the 3rd step of the cycle would be to combine in the z dimension (so there would be 6 total steps/combinations).



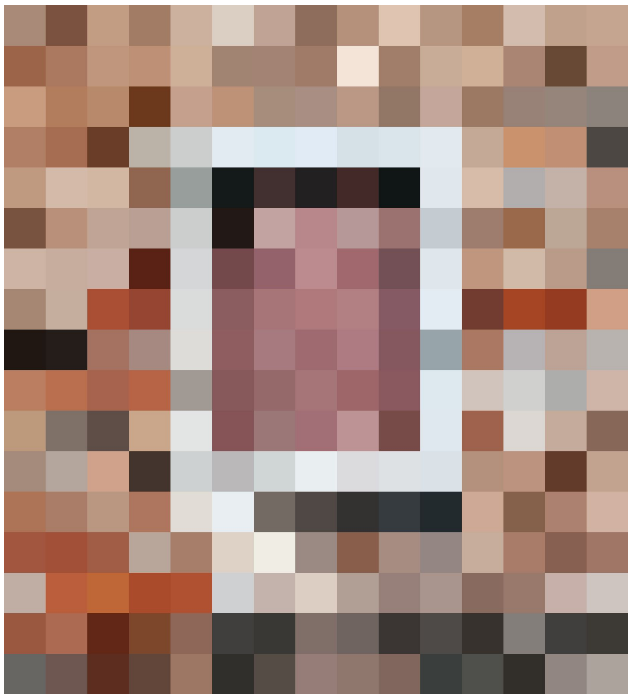
Face Container Combinatory Cycles

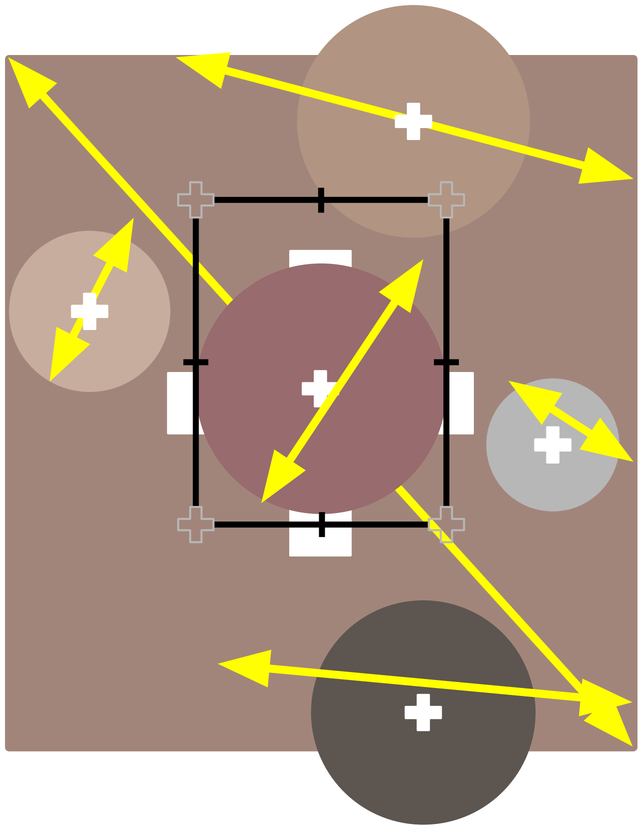


The figure on the left shows a rendering of a face in camera space. The orange arrows are the basis vectors of the face’s texture space, and the point from which the arrows originate is the texture space’s origin. The figure on the right shows the face’s texture rendered in its own texture space. A point that lies on the face can be projected into the texture’s space, and a point on the texture can be transformed into camera space. This two way transformability of a point is illustrated with the blue arrow.

A texture is simply a 2D array of pixels (an image) that describes how a face looks. As mentioned earlier, the face’s texture can be generated by rendering the face from a direction perpendicular to its surface. Once all combinations are complete within a collection of faces, the textures of the remaining faces (which may first be filtered by a minimum face size) are rendered and saved temporarily.

Face Textures





Facial Vertices Extraction

Types of Facial Vertices:

Phrases in parenthesis indicate how each attribute is illustrated.
Underlined attributes are 3D vectors that really represent three separate attributes, one for each component.

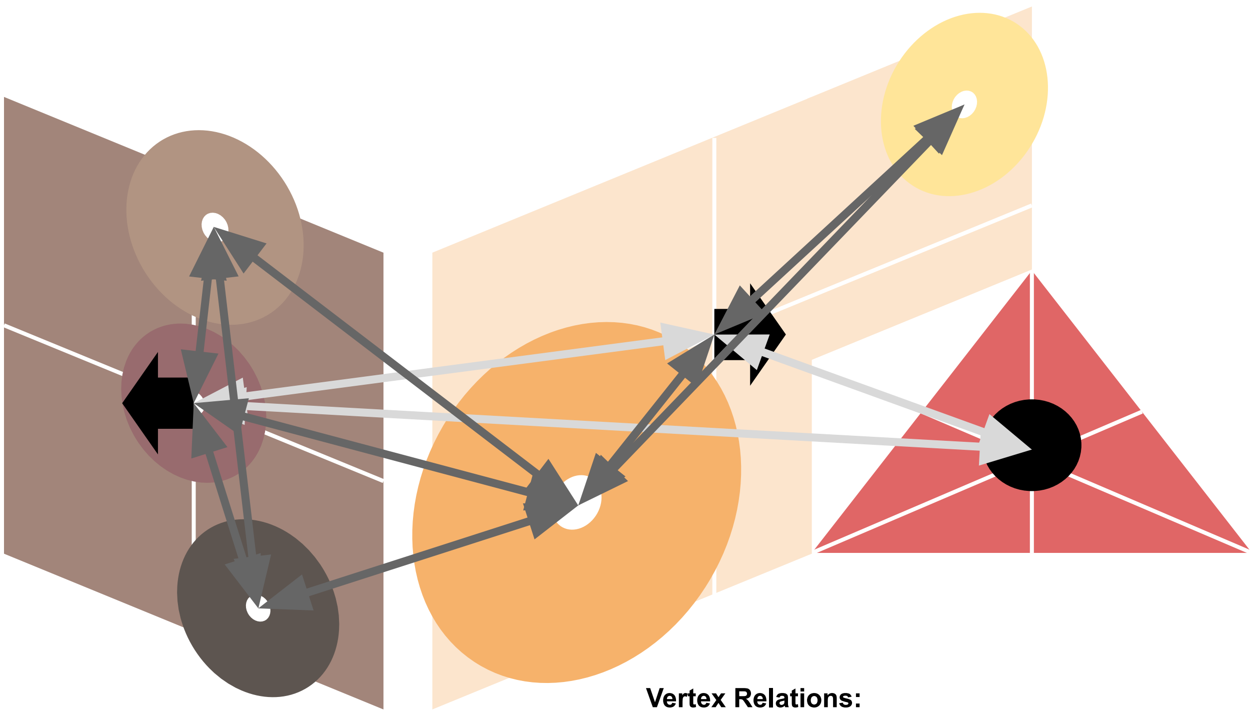
- Overall face
    - Attributes: Center point of area (large white '+'), average color, total area (rectangle), maximum width (yellow arrow), normal (not shown)
- Regions of similar color
    - Attributes: Center point of area (small white '+'), average color, total area (circle), maximum width (shown as yellow arrow)
- Lines
    - Attributes: Midpoint (tick on line), average color (not shown), length (black line), direction

The image on the top left shows the texture (without pixel-pixel interpolation) of some rectangular face, where each square represents one pixel. The figure on the bottom left illustrates the vertices and their attributes extracted from the texture.

Above, the different types and attributes of vertices are listed, along with the symbols by which attributes are represented in the bottom left figure. These patterns would be extracted by a process somewhat similar to that of face compilation, in the sense that containers of patterns would be combined with their neighbors on one side/dimension at a time.

Extraction of facial vertices is performed on a list of faces, each whose serial number is given by its index in the list. Each of the vertices produced from this list of faces is stored in one of three lists, each for a different vertex type. To provide identification for each vertex, the serial numbers of the vertices' faces are interleaved with the vertices. For each type of vertex is a spacial set that sorts its respective list of vertices by their attributes (where the n^{th} attribute represents a position in the n^{th} dimension). When searching these each spacial sets, the attributes/dimensions can be scaled differently to weigh them by importance.

These three spacial sets are half of the six feature spacial sets. Each type of vertex is one of the six types of features. The other three feature types are described on the next page.



Vertex Relations:

The relations between nearby vertices can be made features of their own. A relation feature is a feature that describes how some vertex is related to another vertex of the same type, from the perspective of one of vertices. Each of the vertices involved in a relation has its own relation feature to describe that relation from its own perspective. These relations are created only if the involved vertices are within a certain physical range, which may be different for each type as seen fit, of each other. A size (area or length) minimum for the vertices may also be implemented to prevent relations from being drawn between too small of vertex features.

The figure above illustrates three faces, their facial vertices (located at the white circles or the intersection of white lines), and their relations. The list on the left describes what attributes are collected for each relation feature, as well as how each type of relation is distinguished in the figure. The figure does not include any lines (for no particular reason), but it does include the other facial vertex types.

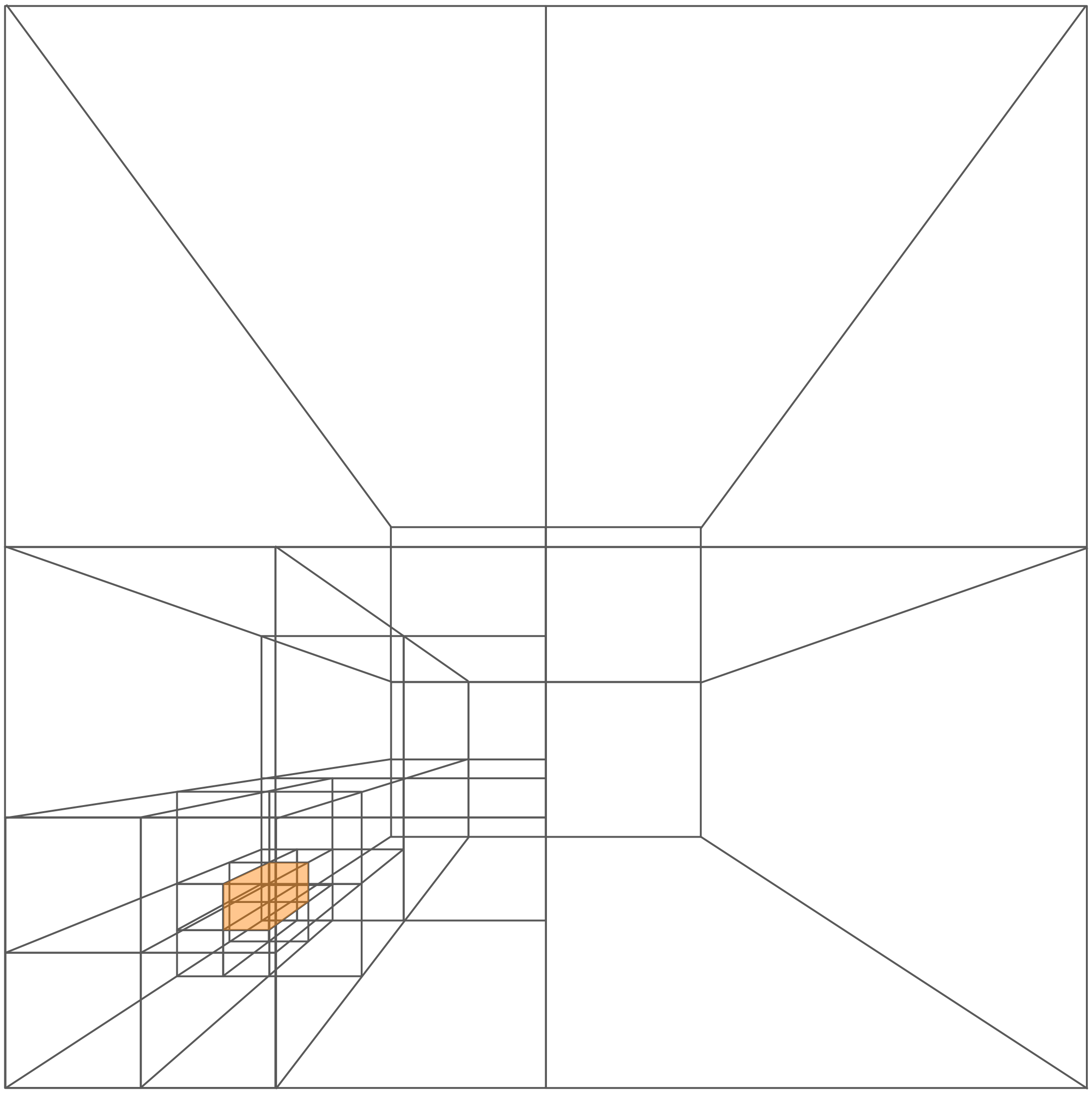
Just like vertex features, relation features are interleaved with their faces' serial numbers in a list shared with all of the other relations for vertices of the same type. Again, a feature spacial set is built for each of these relation feature arrays, completing the six feature spacial sets extracted from the given list of faces.

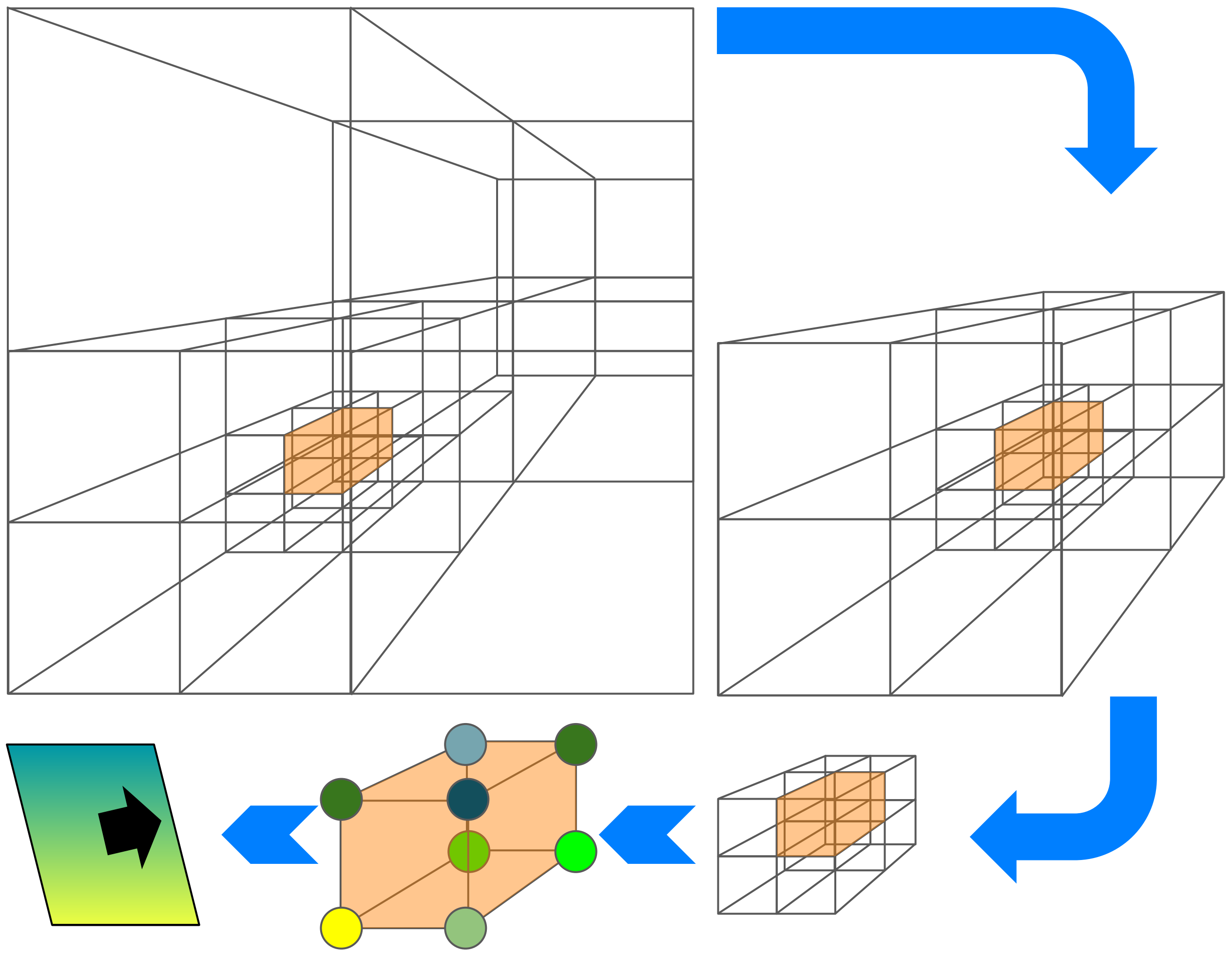
Relation Feature Attributes for each Vertex Type:

- Face
  (relations shown as light grey arrows)
  - Attributes: distance to other face, color of other face, area of other face, maximum width of average face, magnitude of dot product between normals
- Color region
  (relations shown as dark grey arrows)
  - Attributes: distance to other color region, color of other color region, area of other color region, maximum width of average color region
- Line
  (no lines or line relations shown in the figure)
  - Attributes: distance to other line, color of other line, length of other line, magnitude of dot product between directions

Vertex Relations Extraction

Memory





Plane Map

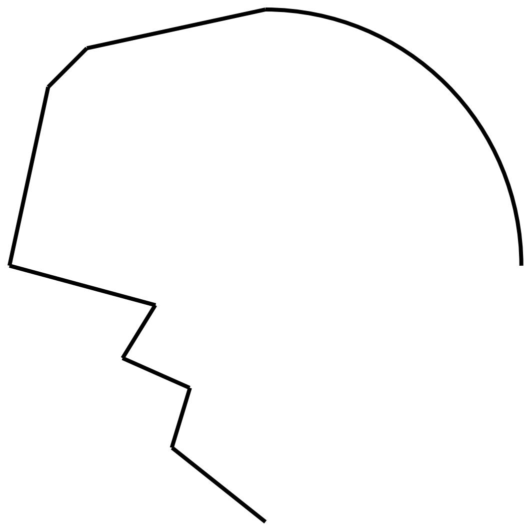
The Plane Map keeps track of the physical operating environment and is relied on heavily in navigation for relating the camera’s current position to the positions of remembered obstacles and the user-defined borders of the operating environment.

The figure above illustrates a plane being unwrapped from the plane container hierarchy (the Plane Map), a hierarchy of nested cubic containers that, at the lowest level, hold plane cubes (cubes that each contain one plain). Each plane, defined in this context by a normal and a point, describes the general structure of the volume occupied by its plane cube. The color of any point within the volume occupied by the plane cube is actually completely independent of the plane itself. Instead of the plane having some sort of texture, each corner of its plane cube has a color, and the color at any point between them is simply linearly interpolated between the eight corners. However, the only points where color is said to exist are those that lie on the plane.

Because the Plane Map is updated off of the Perceived Quads (as described in the Plane Map Update section), which themselves are each given a certainty value, every plane cube is assigned a certainty value of its own.

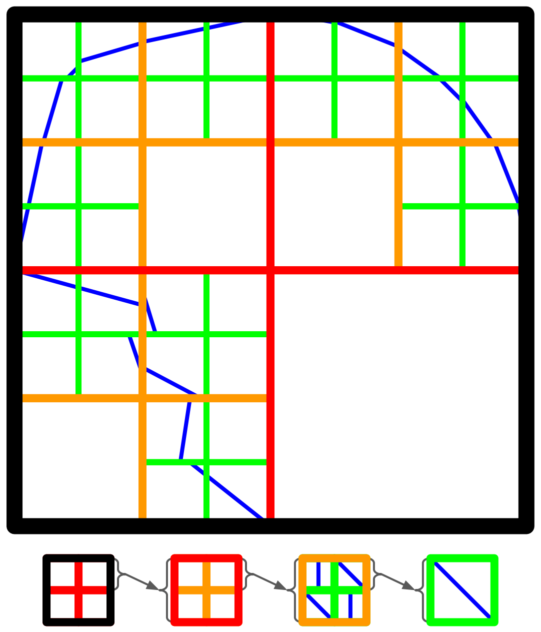
Unwrapping the Plane Container Hierarchy

Illustrating the Plane Map and faces is easier to do in two dimensions. For the sake of simplicity, the figures on the right all use lines to represent planes. These lines could be interpreted as cross sections of the planes. The top figure shows the surfaces of some real-world object or scene, with all color being omitted.



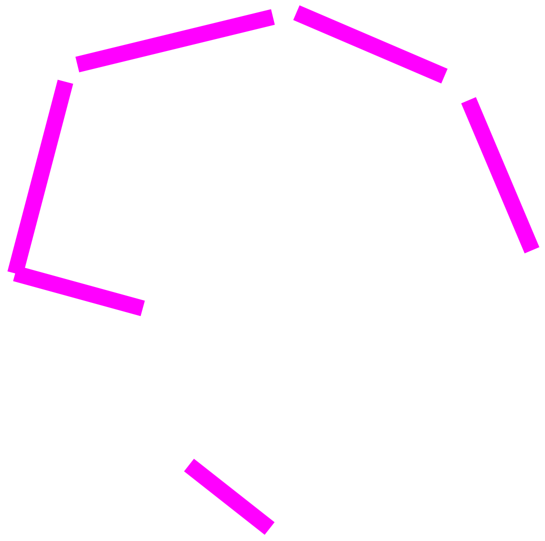
Plane Map

The figure below shows the scene being subdivided into nested square (rather than cubic, because it's in 2D) areas that represent the plane container hierarchy. Each square contains smaller subdivisions only if surfaces exist within. At the lowest level of subdivision (green, the third level), if a surface is found within, its average orientation and position is recorded as a plane (illustrated as a blue line for simplicity). Because the surfaces from the first figure are approximated to be flat over large intervals in the second figure, the blue planes do not always closely resemble the original surfaces that they generalize.



Face Map

The Face Map is the list of faces compiled from the Plane Map. The bottom figure illustrates the positions, orientations, and sizes of what the faces of the Face Map might look like. From the Face Map, the six feature spacial sets (as described in the Feature Extraction section) are extracted and kept around for as long as this version of the Face Map is.



Face Map 2D Representation

Integrated Movement Positioning

B is a 4x4 matrix composed of four homogeneous column vectors. B[i] represents the i^{th} basis vector defining camera space from the perspective of Plane Map space. B[3] represents the origin of camera space from the perspective of Plane Map space. w is the width, or the distance between the wheels.

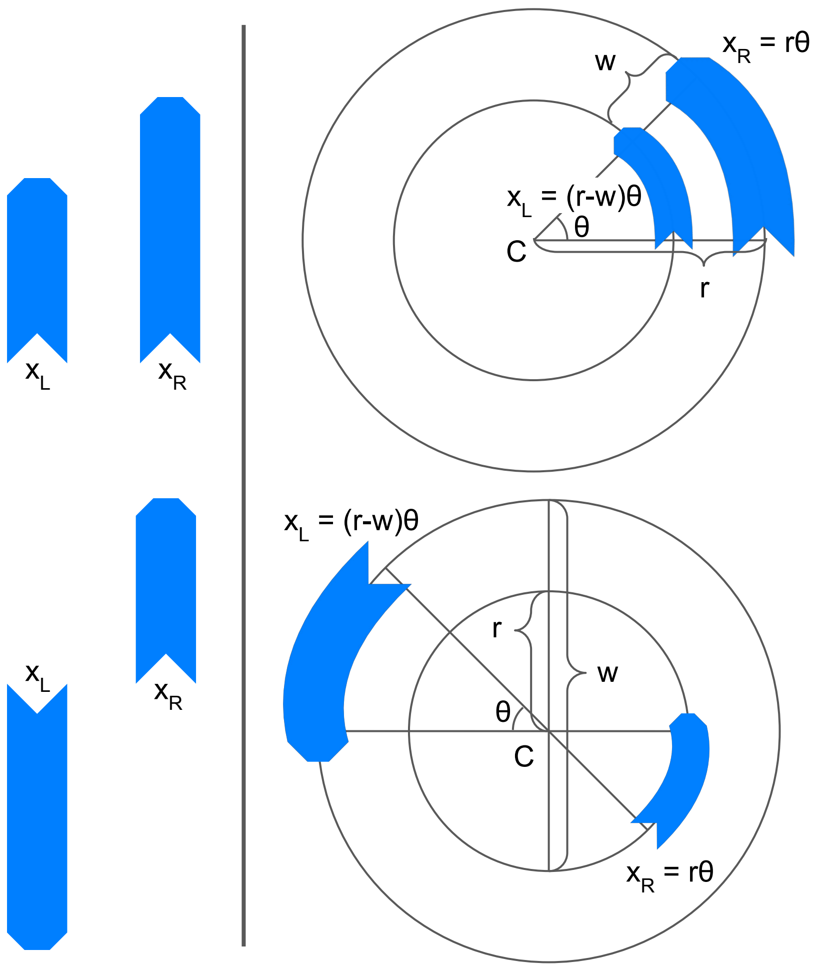
The short list of instructions below describes all necessary computations. Anything after a "//" is a comment.

```
$x_L = (dx_L/dt)\Delta t$
$x_R = (dx_R/dt)\Delta t$
If $x_L \neq x_R$:
    $\theta = x_R/r = x_L/(r-w)$
    $r = wx_R/(x_R-x_L)$
    C = B[3] - B[0] * (r - w/2)
    // C is the center of rotation
    B[3] = B[3] - C
    B = B * RotationMatrix(θ)
    // This rotation winds from x to y within the xy plane
    B[3] = B[3] + C
Else:
    B[3] = B[3] + B[2] * $(x_L+x_R)/2$
    // $(x_L+x_R)/2$ is the average x
```

Over a small interval of time Δt, the right wheel rolls a distance of x_R and the left wheel rolls a distance of x_L. Integrated Movement Positioning calculates and accumulates the resulting small changes in position and orientation over time.

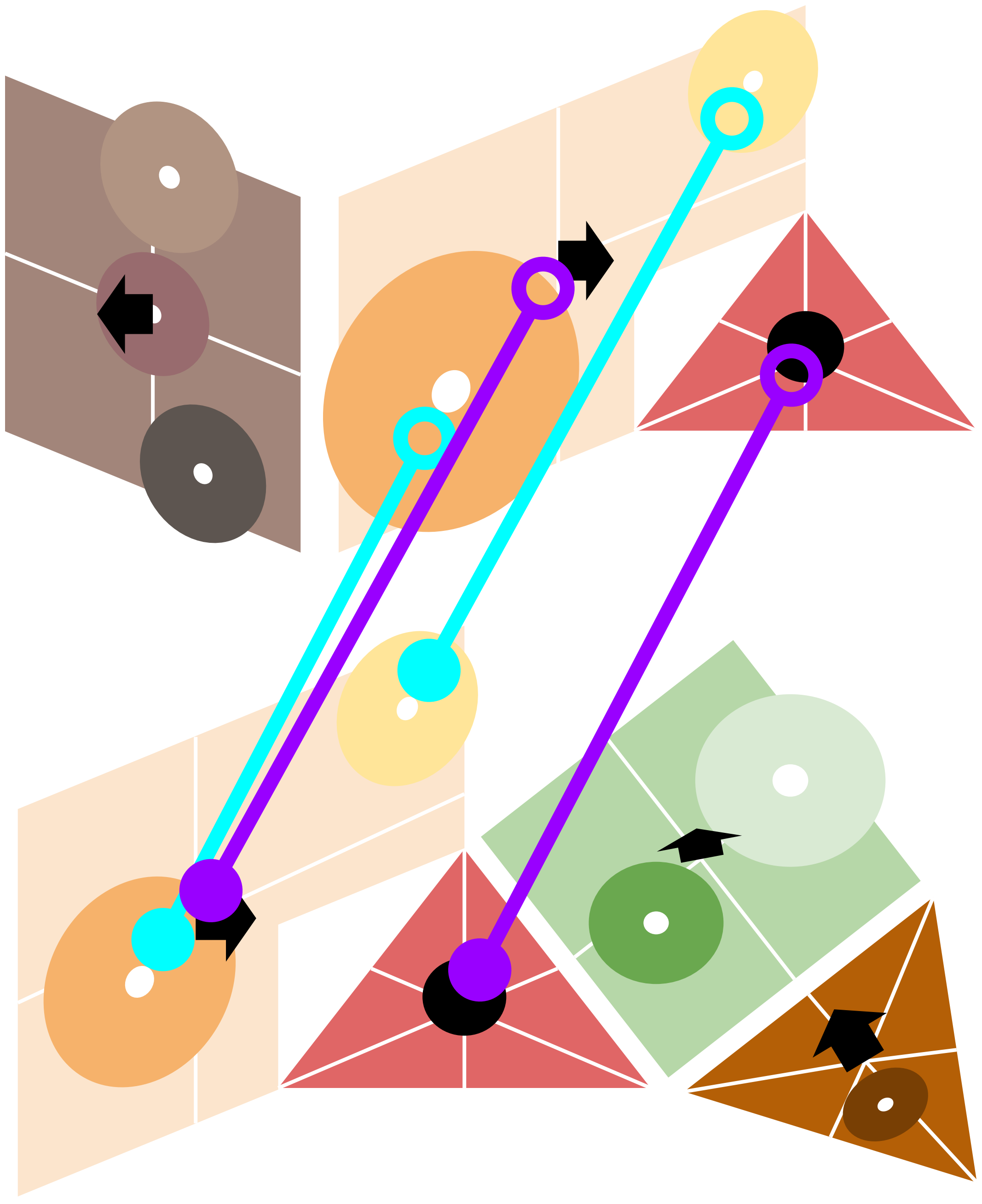
If $x_L \neq x_R$: The device must be rotating around some point, changing both its position and its orientation. It is assumed that this will almost always be the case to some extent.

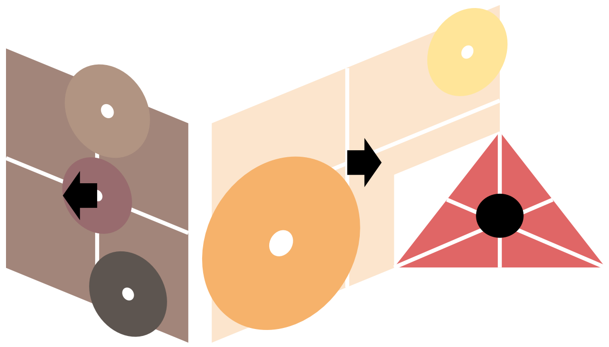
If $x_L = x_R$: The mower must have traveled in a perfectly straight line, so no rotation or curved motion is necessary.

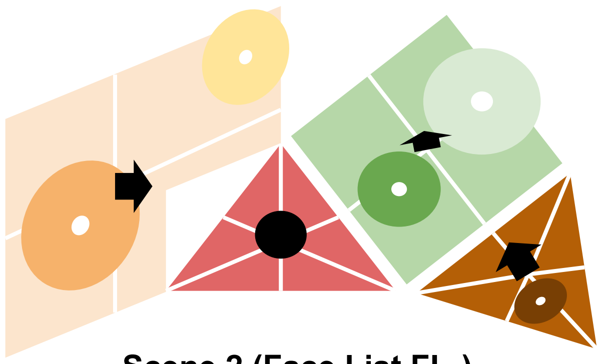


Exception:
$x_L = x_R$

Face Match Search




Scene 1 (Face List FL_1)


Scene 2 (Face List FL_2)

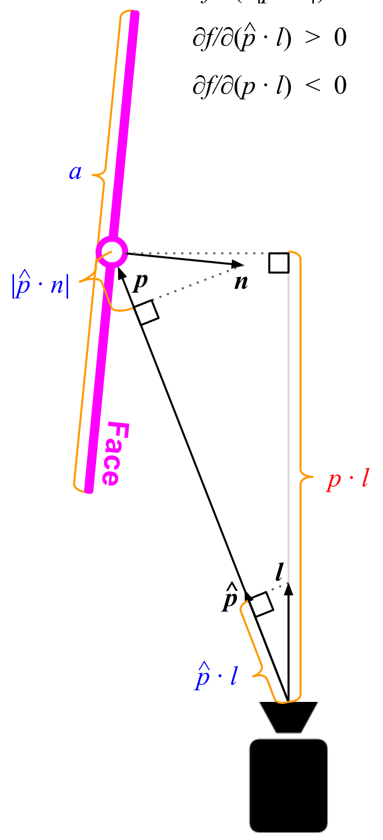
Expected Camera Placement is the expected position and orientation of Scene 2's observer relative to the space of Scene 1. This provides an early placement estimate to transform any FL_2 vertex feature centerpoints from Scene 2 space into Scene 1 space before comparing them with FL_1 vertex features, which are already in Scene 1 space.

$$\text{Visibility} = f\left(a|\hat{p} \cdot n|, \hat{p} \cdot l, p \cdot l \right)$$

$$\partial f / \partial (a|\hat{p} \cdot n|) > 0$$

$$\partial f / \partial (\hat{p} \cdot l) > 0$$

$$\partial f / \partial (p \cdot l) < 0$$



Face Match Search

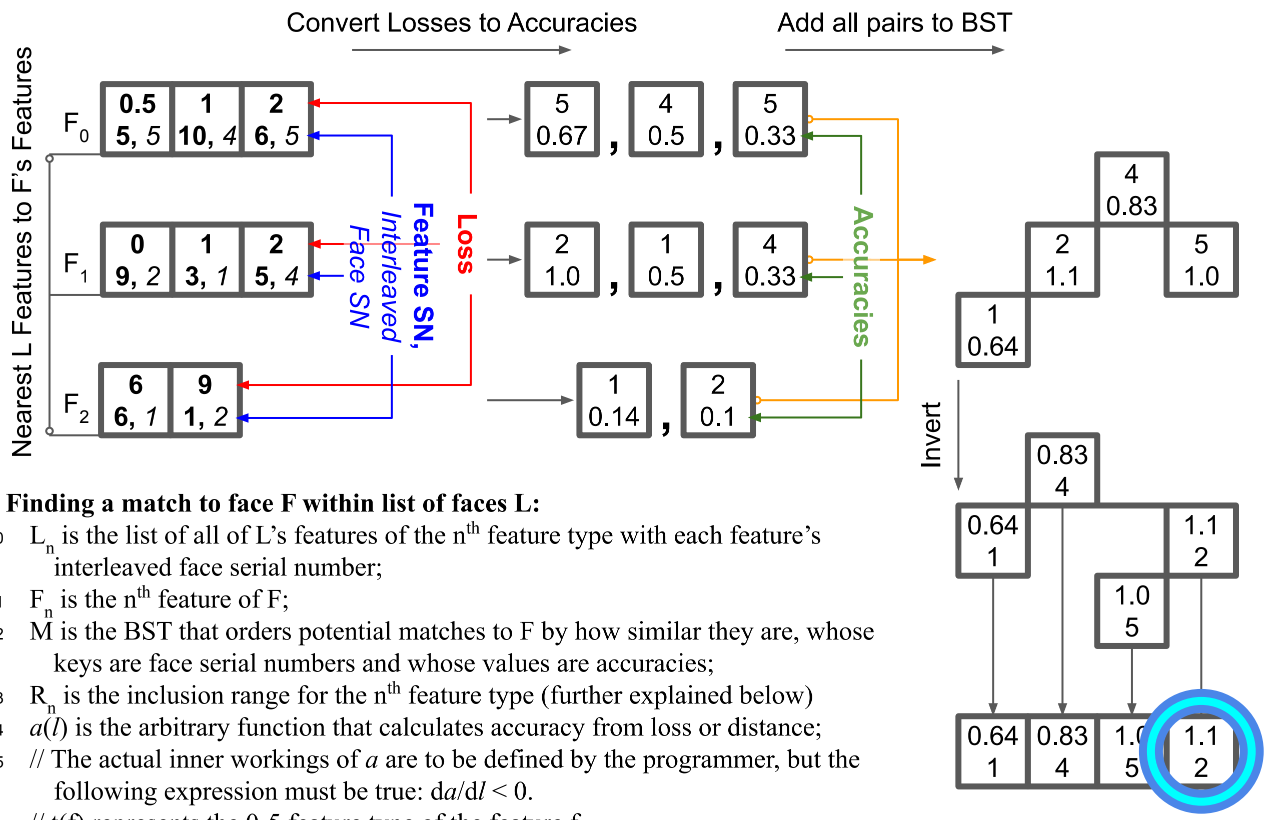
Implicitly Passed Resources:

Above, the explicitly passed resources for the Face Match Search are listed. However, some resources are left out of this list. Scene 1 is expected to have come from memory, so it is feasible to expect FL_1 to come with its six feature spacial sets. Scene 2 is expected to be more fresh, so FL_2 only needs to come with its six feature lists (no ordering or spacial sets). Additionally, a minimum accuracy threshold and a requested return quantity must be passed to the Face Match Search.

Assigning Priority

In order to find matches between the two scenes, FL_1 is searched for matches to specific faces from FL_2. An FL_2 face of greater visibility (closer to camera, perpendicular to line of sight, visible to both cameras) is more likely accurate, so it is more likely to match something in FL_1. Therefore, faces of greater visibility should be prioritized and searched for first. Let's call **p** the displacement from the camera to the center of an FL_2 face, **n** the normal of that face, **a** the total area of that face, and **l** the camera's line of sight, as illustrated in the left figure. The visibility of the face is an arbitrary function of three factors: **a*|(p/||p|| · n)|**, **p · l**, and **p/||p|| · l**. Each factor listed is red or blue if it decreases or increases visibility, respectively. The specific innards of this visibility function are up to the discretion of the programmer.

Apart from prioritizing faces of greater visibility, it is also practical to prioritize faces with a greater number of features, which may be more easily identifiable than those with fewer features.



Finding a match to face F within list of faces L:

```
0  $L_n$ is the list of all of L's features of the nth feature type with each feature's
     interleaved face serial number;
1  $F_n$ is the nth feature of F;
2  M is the BST that orders potential matches to F by how similar they are, whose
     keys are face serial numbers and whose values are accuracies;
3  $R_n$ is the inclusion range for the nth feature type (further explained below)
4  $a(l)$ is the arbitrary function that calculates accuracy from loss or distance;
5  // The actual inner workings of $a$ are to be defined by the programmer, but the
     following expression must be true: $da/dl < 0$.
6  // t(f) represents the 0-5 feature type of the feature f.
7  For each integer $i \in [0, \text{\# of features belonging to F})$:
8      The list A of key-value pairs is produced by ordering the features of $L_{t(F_i)}$ by
         their distance to $F_i$ (with the Binary Search for Points procedure);
9      $R_{t(F_i)}$ is the arbitrary maximum number of elements in A that will be looked at;
10     For each integer $j \in [0, \min(R_{t(F_i)}, \text{length of A} - 1)]$:
11         The key-value pair [face serial number of L[value of A[j]] : $a$(key of A[j])] is
             added (not inserted) to M;
12 M is inverted (every [key : value] of the original M is reinserted to the new M as
     [value : key]);
13 M is converted into an array/BSA, and the last key-value pair is returned as the
     [accuracy : face serial number] of F's closest match;
```

Face Match Search Example (referring to figures and procedure above):

The figure illustrates the later processes of the procedure, starting around the 9th line. The face for which we are finding a match, F, has three features, and the first two are of the same type. The inclusion range for the feature type of F_0 and F_1 is 3, and the inclusion range for the feature type of F_2 is 2. The three features from L most similar to F_0 are the features 5, 10, and 6 of type $t(F_0)$. The figure in the top left shows these features' [distance to F_0 : feature serial number] pairs with interleaved face serial numbers. To clarify, multiple completely separate features can have the same feature serial number if they are of different feature types. Another clarification: 'distance' in this context is used in the sense that each feature is a point whose position is defined by the feature's attributes. In a similar fashion, the features from L most similar to F_1 and F_2 are shown in the rows below the first.

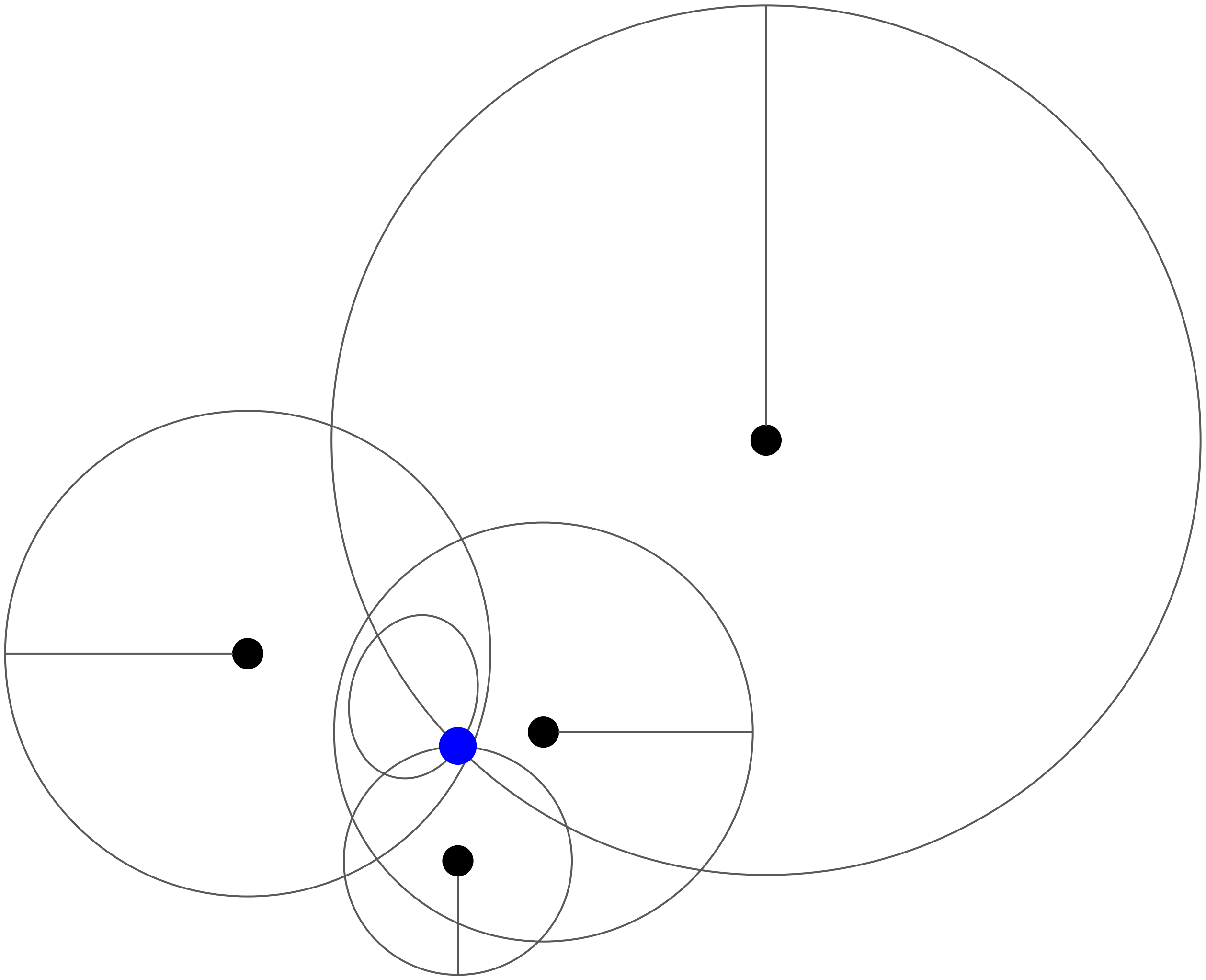
To the right of those three rows are another three rows of key-value pairs, each pair corresponding to the feature in the first three-row set at the same row and column. Each of these new pairs has a key equal to its feature's face serial number and value equal to its feature's distance/loss converted into an accuracy. In this example, the accuracy function is given by the expression $a(l) = 1/(1 + l)$.

Next, all [face serial number : accuracy] pairs are added to the BST M (illustrated further to the right), which is then inverted (illustrated below that) and converted into an array (lowest illustration). The last pair of the array, which has the greatest accuracy (1.1 in this example), contains the serial number (2 in this example) of L's closest match to F. This last pair is the final result of the search.

Search Order

In the order of priority, a match from FL_1 is found for each FL_2 face until the total number of matches (each match is only counted if its accuracy exceeds the minimum accuracy threshold) is equal to the requested return quantity. These matches of satisfactory accuracy, along with the serial numbers of their involved FL_2 faces, are returned.

Camera Placement Update



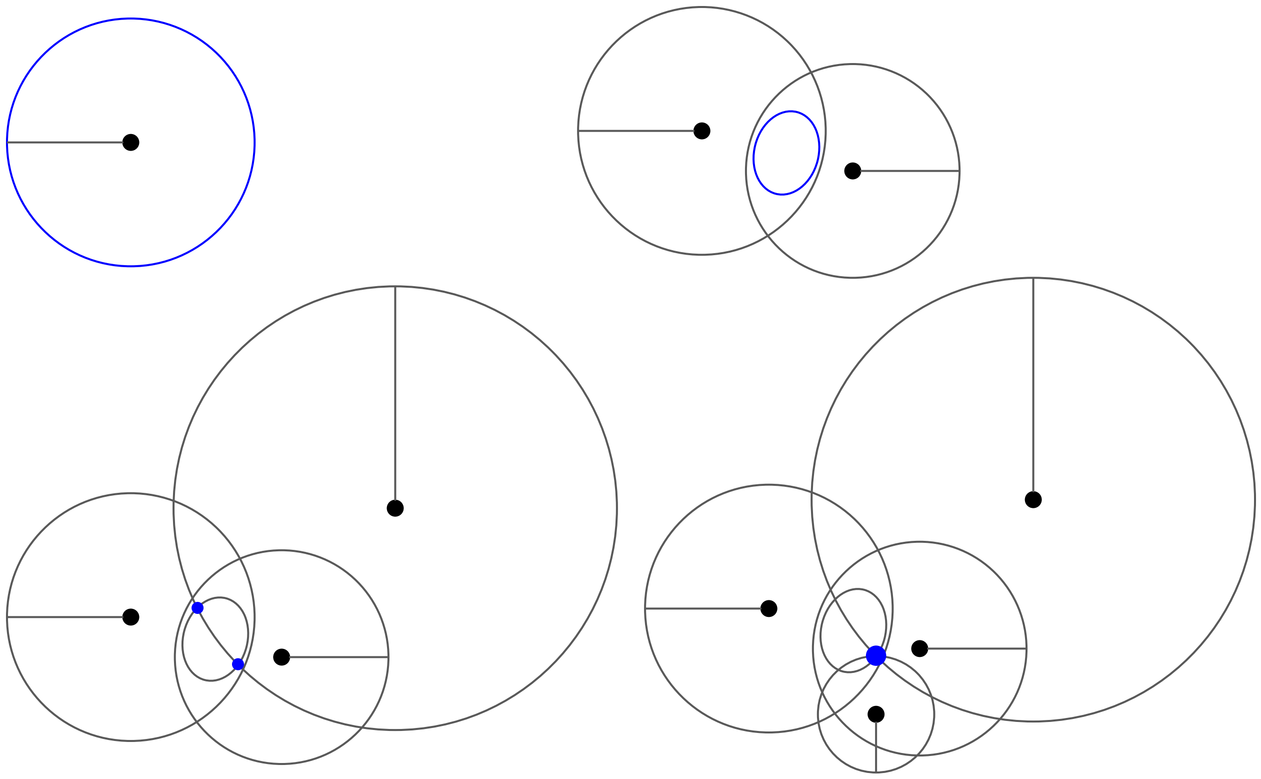
Quadrilateration

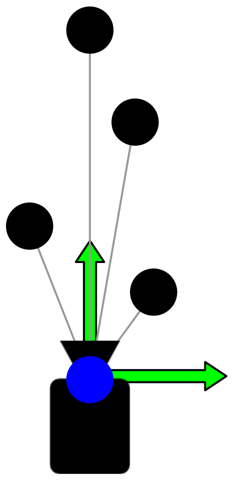
(because triangulation is for 2D)

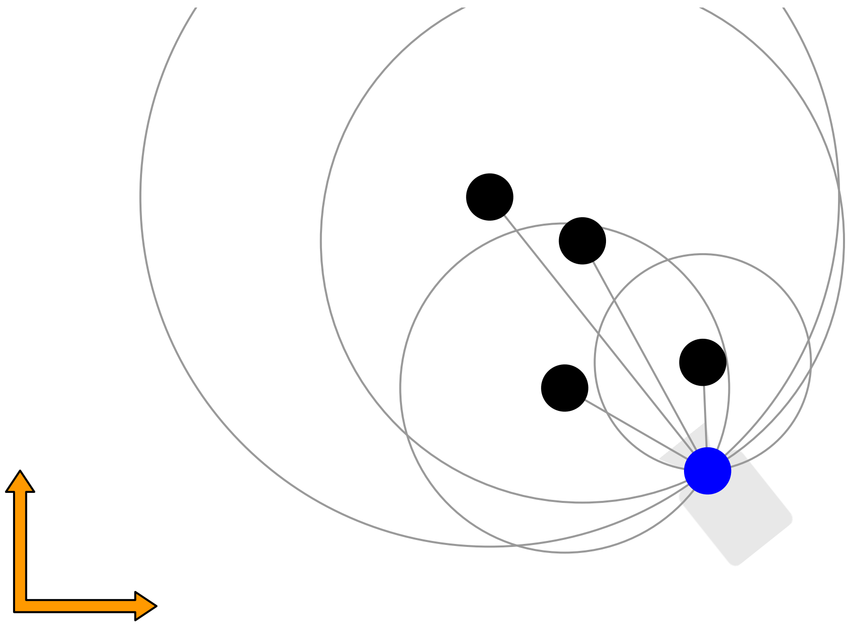
Let the positions of four points and their distances to point p be known, but let the position of point p be unknown. These four points are all that is needed to locate p. In fact, this is how GPS works: a device receives broadcasts from at least four satellites, each of which says the time at which the broadcast was sent and the position of the satellite at that time. The device calculates how long it must have taken for each signal to reach it, revealing how far away the satellite is. Now knowing the distances and positions of all four satellites, the device quadrilaterates its own position.

Let's find our position p using one point at a time. For any of the points, p must lie somewhere on the sphere that is centered on that point, whose radius is equal to the distance between p and the point. In the figure below, the region in which p could eligibly be located is shown as blue. As more points and their surrounding spheres are brought into consideration, this region shrinks. With only one sphere, p could lie anywhere on its surface (the figure shows a sphere, not a circle, surrounding each black point). However, after adding just one more sphere, p is constrained to lying on the circle where the two spheres intersect. Adding a third sphere, p is limited to being one of two points, and the last sphere identifies exactly where p is.

If multiple sets of four points are given, p could be located once for each of them, and then each estimation of p could be averaged together to yield a more accurate singular p.







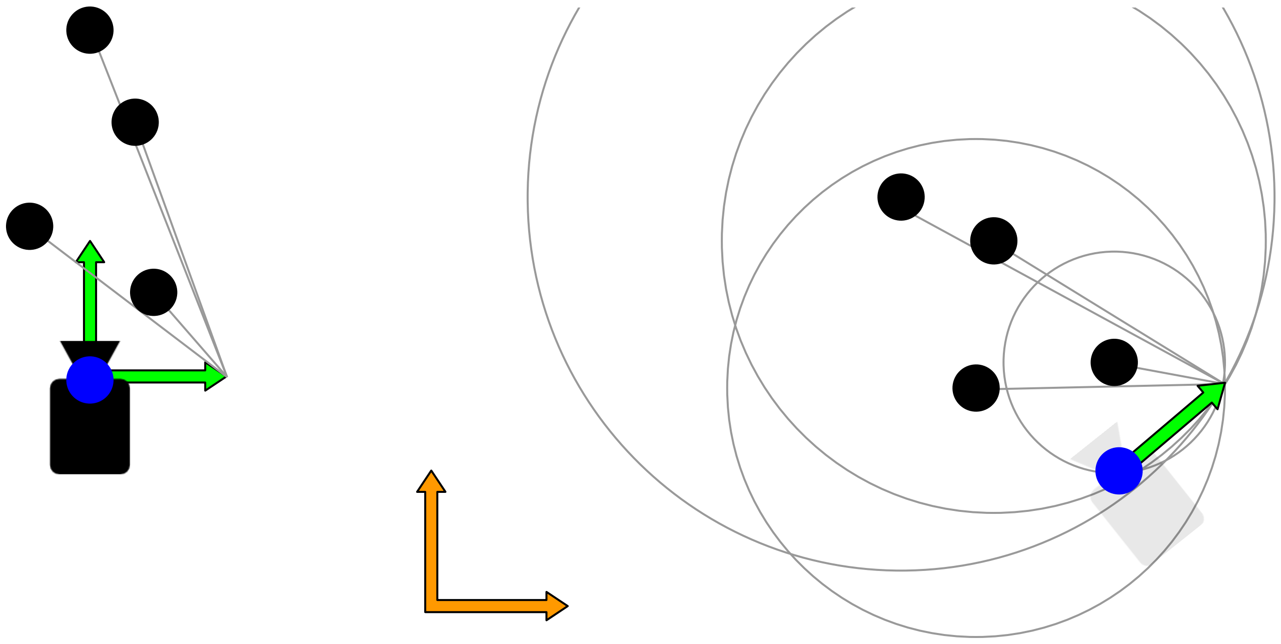
The Camera Placement Update updates the camera's placement (position and orientation) in the space of Plane Map such that the virtual camera's perspective in the Plane Map is the same as the physical camera's perspective in real life. To do this, a Face Match Search must be performed using the Face Map as Scene 1 and the list of perceived faces, which would be compiled off of the Perceived Quads, for Scene 2. The camera placement is fed in as the Expected Camera Placement, and an arbitrary multiple of 4 is passed in as the requested return quantity. The list of matches M is returned. M[i] is the i^{th} match, M_M[i] is the Face Map face corresponding to M[i], and M_P[i] is the perceived face corresponding to M[i].

The faces themselves aren't what's important, but their center-points are. Any time a face is referenced in this section, it will really be referring to its center point. The distances from the camera to M_M[i] and to M_P[i] should be equal. Since M_P[i] is already in camera space, the distance between it and the origin is M[i]'s distance to the camera, and since M_M[i] is already in Plane Map space, its position is M[i]'s position. Defining the matches to have these distances and positions allows them to be grouped into sets of four and used to quadrilaterate the camera's position in Plane Map space.

The upper left figure illustrates M_P, the points of the matched perceived faces, in camera space, as well as their distances from the camera. In this figure, the camera space basis is shown by green arrows. The upper right figure illustrates M_M, the points of the matched Face Map faces, as well as their spheres and distances. The basis of Plane Map space is shown by orange arrows in the second figure. In both figures, the origin of camera space is shown by a blue point. At this time, only the camera's position is known (not its orientation).

Finding the camera's location isn't the only application of quadrilateration; it can also find the camera's orientation.

Position Update



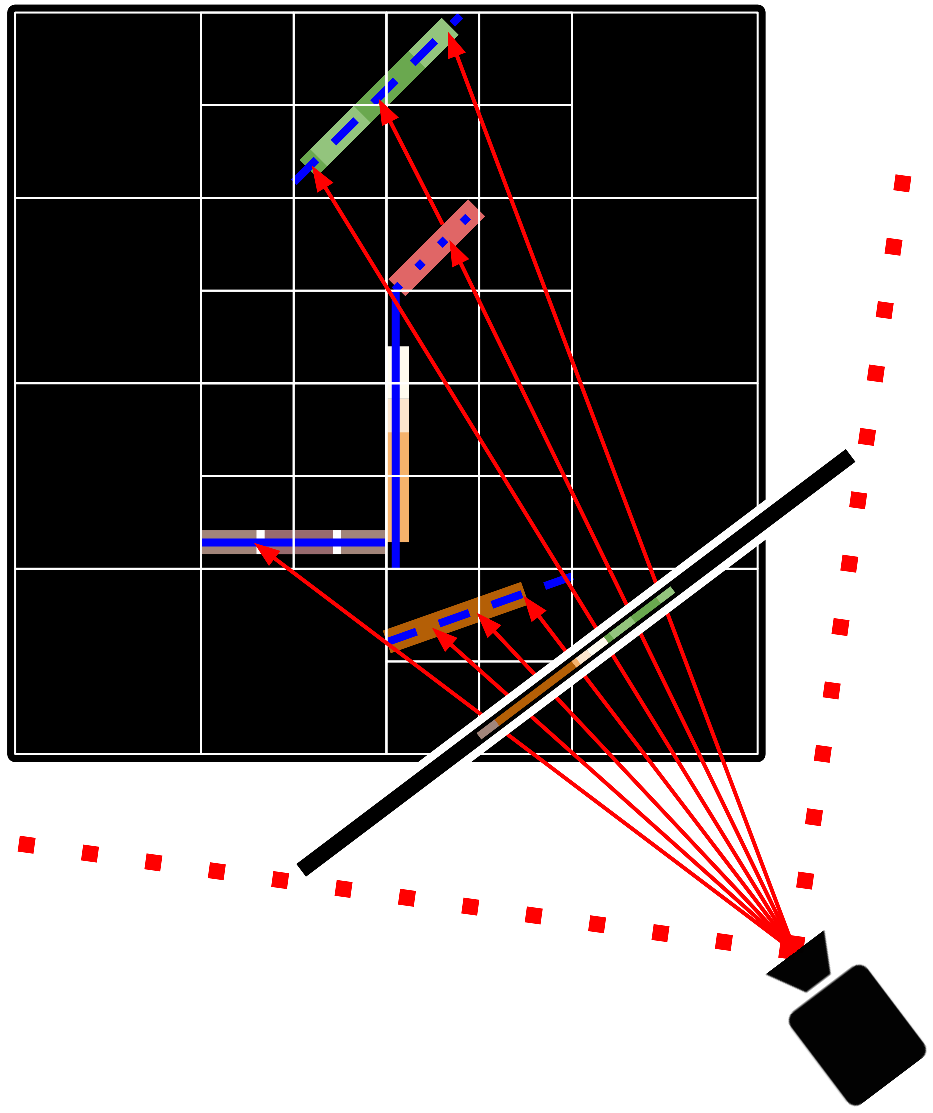
Although it is perhaps one of the least eloquent way to find the camera's orientation, this is also a fairly simple one. If each point's recorded distance is between its position and the end of a basis vector of camera space, the position of the basis vector's endpoint could be quadrilaterated into Plane Map space. This would be done in a similar manner to the quadrilateration of the camera space origin. The vector itself is the displacement from the camera space origin to the vector's endpoint.

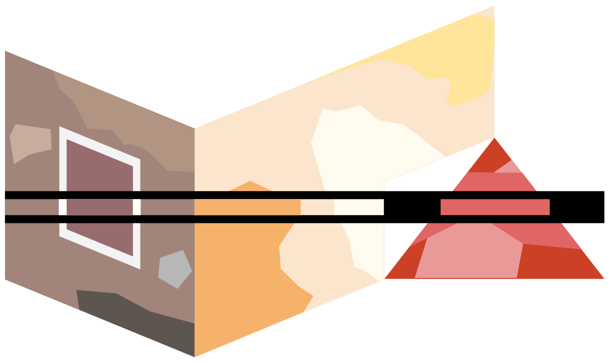
This process would only need to be performed for two basis vectors, and the third could be found as their cross product.

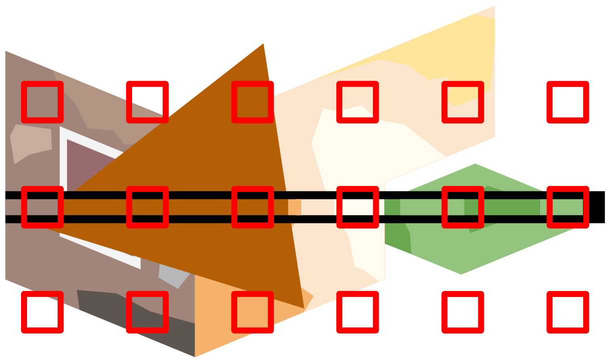
The figures above show the 0th basis vector's endpoint going through the same quadrilateration process that the camera space origin went through to be located.

Orientation Update

Memory Update




Expected Scene
(Memorized)

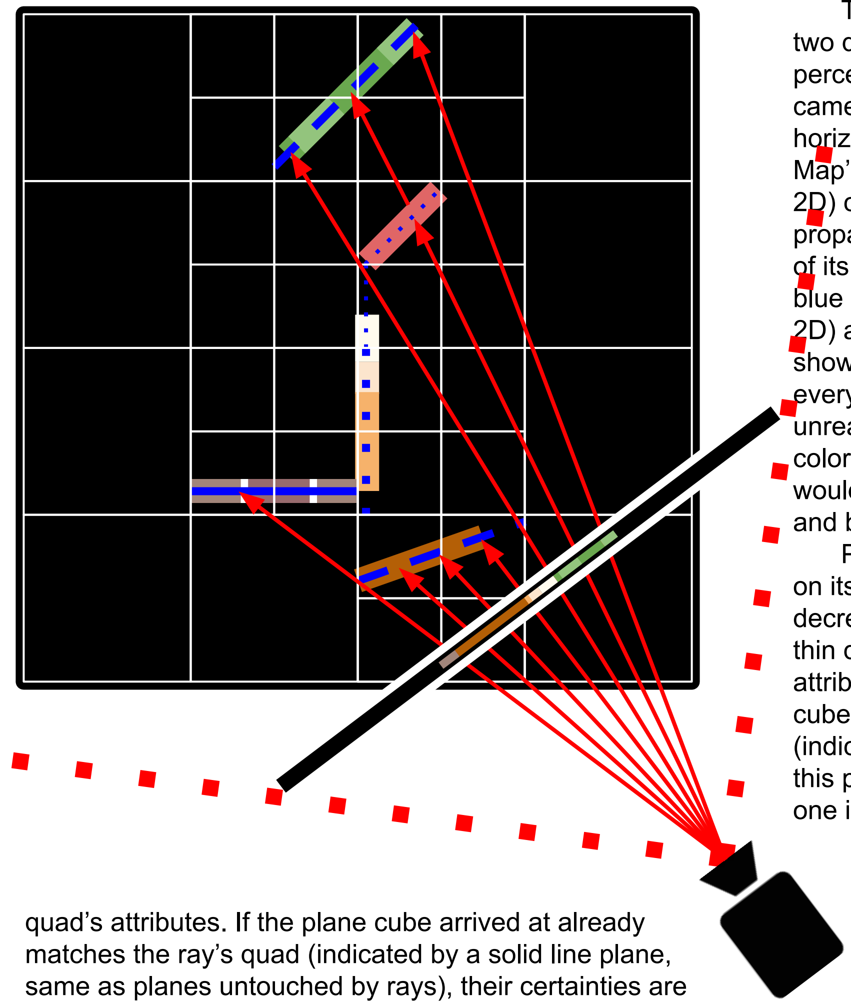

Real Scene
(Perceived)

The figure above on the left shows a scene held by the Plane Map. The figure to the right shows the scene modeled by the Perceived Quads. Both scenes are taken from the same camera placement. Cut out from each image is a horizontal strip that will be used in the 2D representation on the next page. The center of each quad is marked by one of the red squares in the right figure. The Plane Map Update partially reconciles discrepancies between these scenes, making more progress every time it runs.

Firstly, in order for the Plane Map to look more like the Perceived Quads, any plane cube obstructing the path of a ray from the camera to the location of a perceived quad (i.e. any plane cube that doesn't appear to be there anymore, since we can apparently see through it now) should have its certainty reduced. Secondly and more obviously, any plane cube containing a perceived quad's point should have its surface be made to more closely resemble the quad in color, position, orientation and certainty. Lastly, certainties of plane cubes that were expected to be visible but are not, perhaps because they are obstructed, should be reduced.

Plane Map Update Summary

Plane Map Update

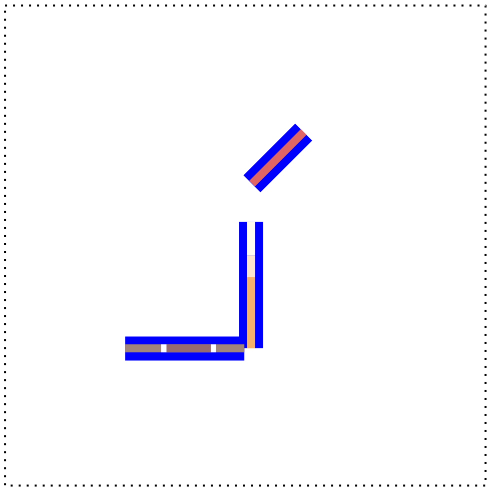


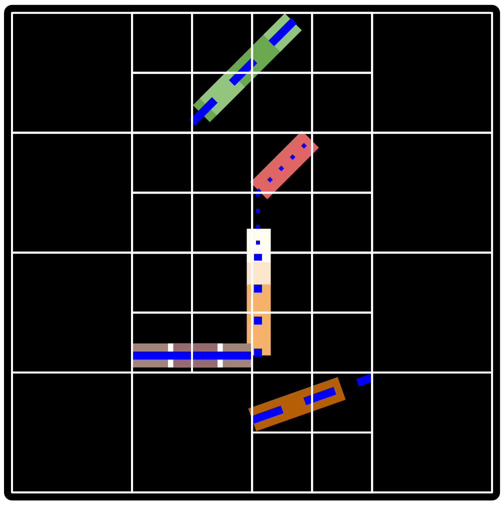
The figure on the left shows, in two dimensions, a ray for each perceived quad being traced from the camera out through the image (the horizontal strip) and into the Plane Map's hierarchy of cubic (square in 2D) containers, through which it will propagate until it reaches the location of its quad. In these containers, the blue lines represent planes (lines in 2D) and each wider colored line shows the interpolated colors for every point along its plane. This is an unrealistic representation of how the color interpolation would look; it would really be much more gradient and blurry.

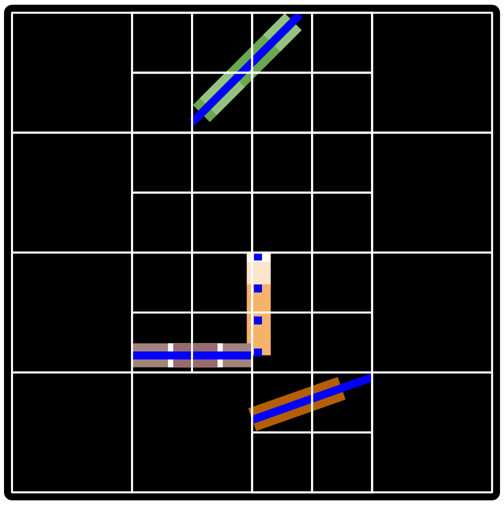
Plane cubes that a ray intersects on its way to its quad's location are decreased in certainty (indicated by a thin dotted line plane). Each quad's attributes are averaged into the plane cube at which its ray finally arrives (indicated by dashed line plane). If this plane cube doesn't exist, a new one is created with the quad's attributes. If the plane cube arrived at already matches the ray's quad (indicated by a solid line plane, same as planes untouched by rays), their certainties are still averaged.

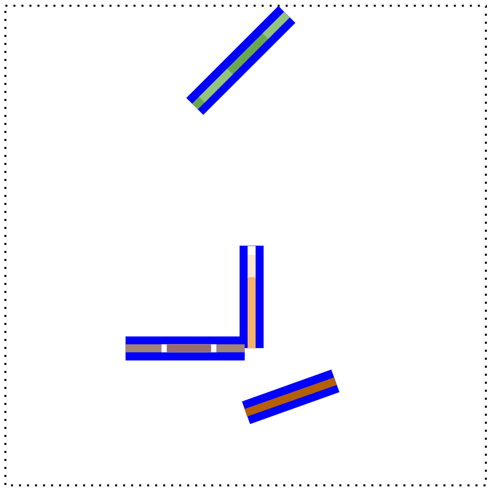
This average is weighted by certainty, so the less certain the plane cube is and the more certain the quad is, the more the plane cube will change. This sense of averaging makes sense for the attributes like position, normal, and certainty, but it isn't as straightforward for color. Let C(p) be the color within a plane cube at point p, linearly interpolated between each corner's color, and the nth corner's color be C_n. If $c_n(p)$ is a coefficient between 0 and 1 representing how much of C_n is in C(p), $C(p) = \Sigma_n c_n(p) C_n$. Q is the color of the quad, who is located at point q. A loss function L, given by the equation $L = ½(C(q) - Q)^2$, could represent the difference between the colors C(q) and Q. Ideally, L would be minimized and C(q) and Q would be equal. Subtracting an infinitesimally small amount of $\partial L/\partial C_n$ from each C_n is sure to decrease L by an infinitesimally small amount, since each derivative is constant over an infinitesimally small interval. However, this would be impractically small, so $\alpha * \partial L/\partial C_n = \alpha(C(q) - Q) * \partial C(q)/\partial C_n = \alpha(C(q) - Q) * c_n(p) = \alpha c_n(p)(C(q) - Q)$ is subtracted from each C_n instead, where α is either a constant or a function of certainties, which in both cases should be much less than 1.

The certainty of every plane cube that is located in the camera's view frustum but wasn't touched by a ray (indicated by thicker dotted line plane) is decremented by some arbitrarily small value. Finally, any plane cube whose certainty is ≤ 0 (or some other minimum certainty value) is deleted from the Plane Map. This may occur at any stage of the Plane Map Update in which such a plane cube is detected.









After the Plane Map is altered, the Face Map needs to be updated. However, updating faces every single time a change is made would be extremely inefficient because most changes are very small, and recompiling faces is tedious. For this reason, the Face Map is recompiled and its feature spacial sets are reextracted after a certain number of Plane Map Updates have occurred.

The top figure on the left shows what the original Face Map looked like (thick purple lines are planes of faces and the thin colored lines are facial textures). The next figure down illustrates the Plane Map from the previous page, showing what changes are being made. The figure below that shows what the Plane Map might look like if these changes continue to be made over a long period of time. This illustration assumes that the partially obstructed wall behind the floating triangle is not fully erased yet. The final figure shows the Face Map after having been recompiled from the new Plane Map.

Face Map Update

Anomaly Detection

Anomalies, or unrecognized objects in the operating environment, are stored in the Anomaly Map. Each anomaly is described by a face (not from the Face Map, since the Face Map isn't updated often enough) and velocity.

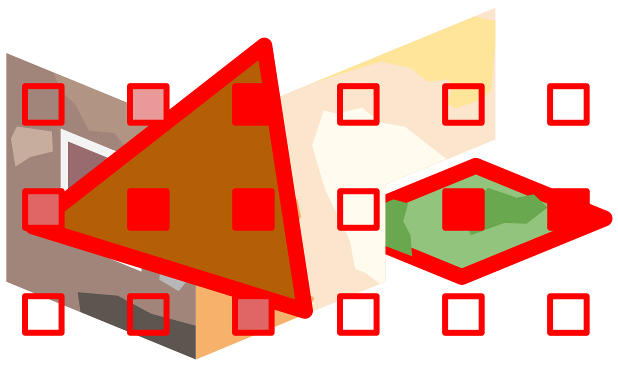
In the Plane Map Update, each perceived quad will be assigned a discrepancy value that generally reflects how much it changed the Plane Map (and how different it was from the Plane Map's contents). The figure in the bottom right shows the perceived scene from the Plane Map Update section. Here, the discrepant (relative to the memorized scene from that same section) quads are distinguished as having red-filled squares. Perceived faces that contain a certain level of discrepancy throughout their constituent quads are labeld as anomalous (outlined in red in the figure).

The Face Match Search is performed on the Anomaly Map faces for all of these perceived anomalous faces. If one matches an Anomaly Map face, the Anomaly Map face and its velocity are updated. Any unmatched anomalous faces are entered into the Anomaly Map with a velocity of 0.

The certainties of any Anomaly Map faces that reside within the camera's view frustum but aren't matched to a perceived face are reduced. Similar to the Plane Map, Anomaly Map faces whose certainties fall below 0 or some other minimum threshold are deleted.

Each anomaly has a risk potential, which is some arbitrary function of its certainty, size, velocity, and position relative to the camera. This risk potential is updated whenever the anomaly's other attributes are updated, and can be used to determine to what extent the anomaly should be avoided, making it particularly useful in Route Planning.

If an anomalous face remains in place for long enough, the Plane Map will slowly conform to the new perceived scene by the processes already described in the Plane Map Update section, such that the anomalous face's constituent perceived quads will become less discrepant with the Plane Map. Because the face will eventually be found no longer anomalous, the Anomaly Map's anomaly will slowly lose certainty until deletion.



Route Planning

(For Mowing)

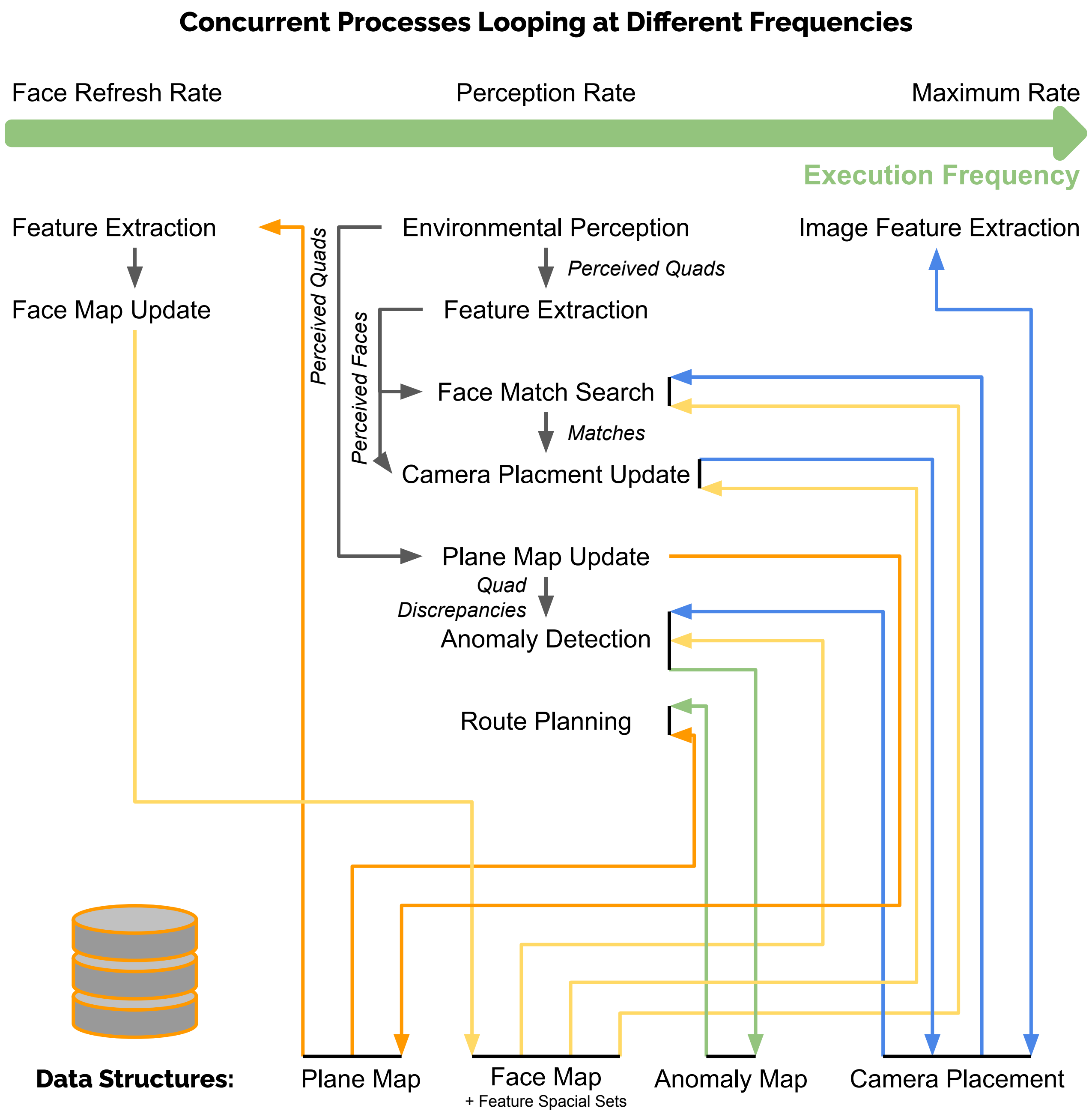
Route Planning uses a square 2D boolean array called the Navigability Map to construct its routes. The Navigability Map has the same width as the Plane Map, and each boolean value at a given 2D position measures whether the terrain of the plane cube at that top-down position of the Plane Map is navigable. A value on the Navigability Map is only true if the following are true about its corresponding plane cube:

- The plane cube is located within the pre-defined lawn borders.
- The plane's steepness is well below the maximum navigable steepness.
- The shapes of the plane and the planes surrounding it do not form a sizeable bump or divot.
- The location of the plane cube is at a distance greater than or equal to the avoidance radius for each anomaly of the Anomaly Map.

The Navigability Map is updated every time the Plane Map is updated, only in the places where the Plane Map is updated.

Before the mower takes off, a new 2D boolean array the same size as the Navigability Map is made and every value is initialized to false. This array represents whether the lawn has been mowed at each top-down position. The mower then generates one mini-route, travels the mini-route while mowing, generates the next, and so on and so forth. As mini-routes are completed, the appropriate positions on the new boolean array are set equal to true. A mini-route is generated by applying a mowing pattern (such as going back and forth in straight lines or using an inward-winding circular path) to a medium-large region that is marked true on the Navigability Map and marked false on the newer temporary boolean array.

Subsystem Integration



The arrows in this figure show the flow of data between the subsystems and the primary data structures. Each column represents one process in which multiple subsystems run in a sequence. The different processes run concurrently at different frequencies, meaning, for instance, that Image Feature Extraction might run for five iterations and the Face Map Update might only iterate once within the same period of time. Reading/writing to the permanent data structures would be all that is in need of cross-process synchronization.