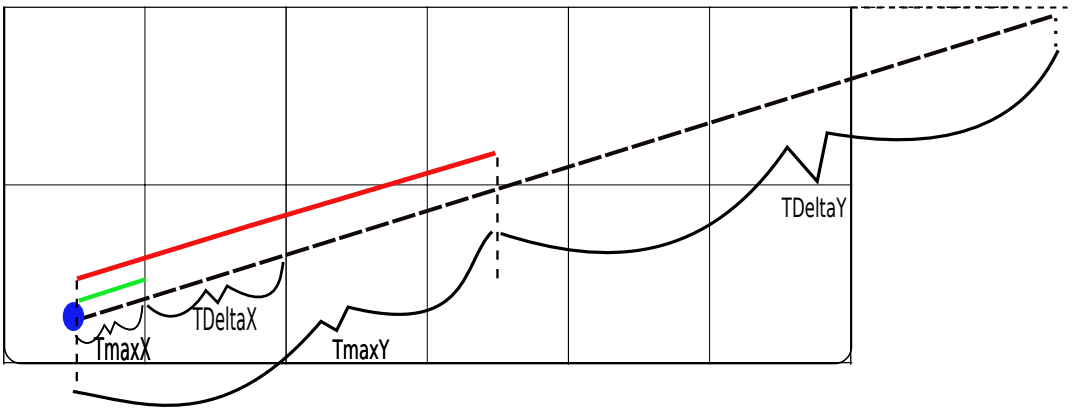
1 Ray Marching Algorithm

Cubic Cells

This section demonstrates the procedure of the ray marching algorithm in determining the piecewise path taken by a ray as it moves through the computational domain. For simplicity, this description will be carried out in two dimensions, although the same principles are applied in the three dimensional cases.

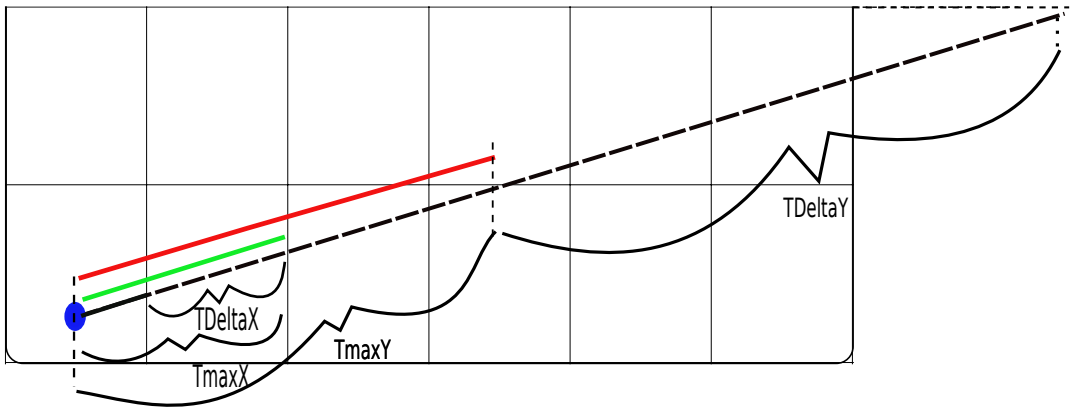
Let us begin with a small computational domain with two rows of cells, and six columns of cells, indexed as (i,j), where i represents the row number starting from the bottom of the domain, and j represents the column number starting from the left of the domain. Let the vertical lines represent x planes, and the horizontal lines represent y planes. We wish to trace a ray within cell (1,1) from the ray origin indicated by the blue circle in the below figure. Assume that a ray direction has already been determined, and will be represented by the long dashed line below.



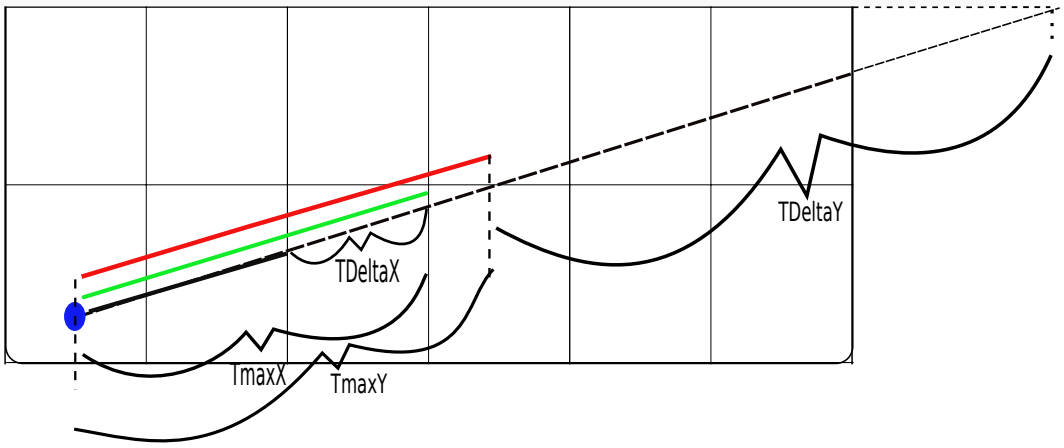
We must first determine the length of the ray segment from the origin to the first cell wall that is breached by the ray. But at this time, we do not know if it will be an x plane or a y plane that will first be breached by the ray. To determine which plane will be breached (and subsequently determine the next cell that the ray will enter) we use a method proposed by [**?**] and compare the distance from the origin to the first x plane, TmaxX (green), to the distance from the origin to the first y plane, TmaxY (red) in the direction of the ray. In two dimensions, this is accomplished by a simple "if/else" statement (additional comparisons are necessary in three dimensions).

Once the shortest of the two distances is determined, the current cell is updated by use of the step variable. In this case, the shortest direction is TmaxX, so we step in the x direction. The distance traveled (in this case the green line above) is stored as disMin, and will be used later in an algorithm that determines ray attenuation. Because the x component of the direction vector is positive, the cell index is incremented by 1 in the x direction, and the current

cell becomes (1,2). The ray has progressed, and the scenario is now represented by the following figure.

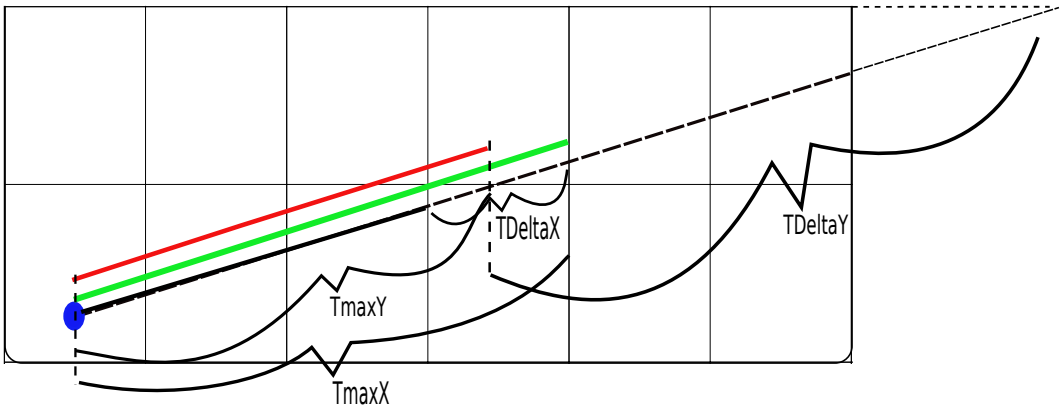


Note that the first segment of the dashed ray has now become solid. The distance from the origin to the first y plane has not changed, so the red line representing TmaxY, remains unchanged. However, the distance from the current location to the next x plane has changed, and its length has been increased by the distance TDeltaX. TDeltaX represents the distance required to traverse one cell length in the x direction. With our updated value for TmaxX, we again compare the green line to the red line. Because TmaxX is still shorter than TmaxY, we again step in the x direction, incrementing i. The current cell then becomes (1,3), and after storing the TDeltaX as disMin for later use, we advance to the following figure.

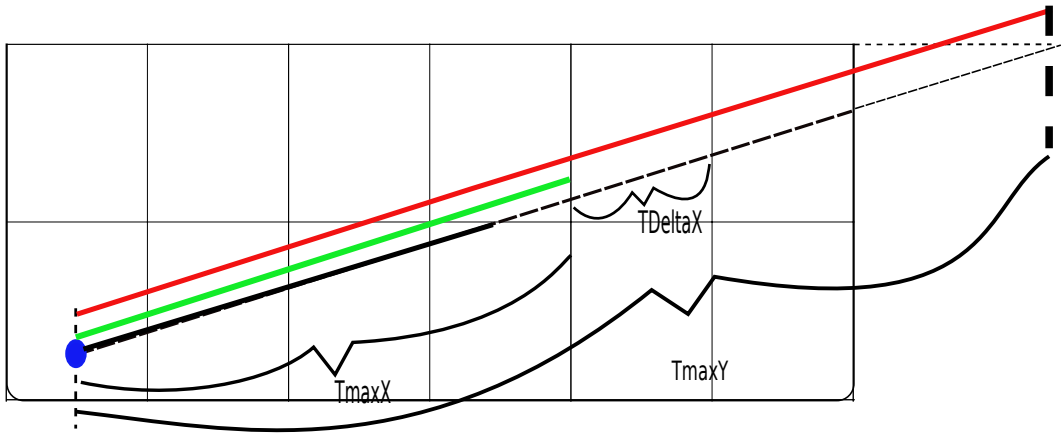


Now, the second segment of the dashed ray has become solid, and we have a new value for TmaxX, which was increased by TDeltaX. Note that for a given ray in a uniform mesh, TDeltaX and TDeltaY do not change, as the distance

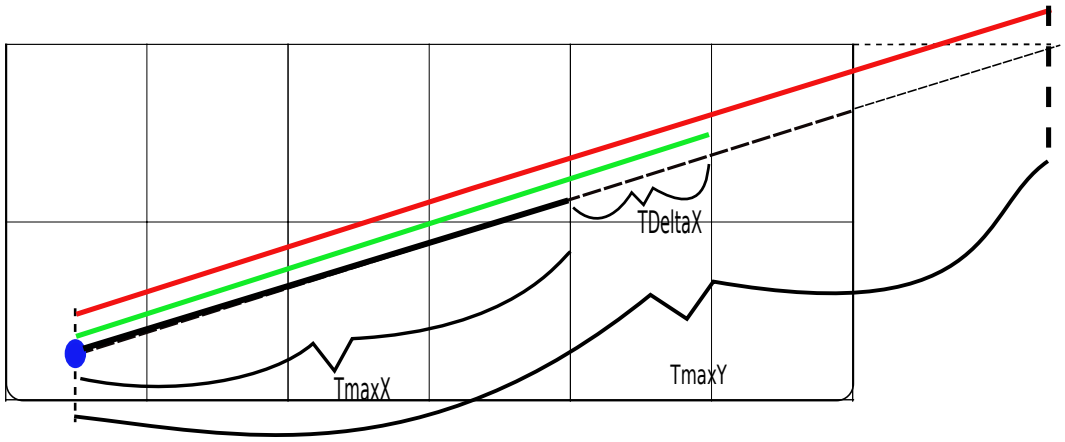
required to traverse a cell in the x or y direction is independent of the current cell. We again compare TDeltaX to TDeltaY. The green line is still shorter than the red, so we again step in the positive x direction, store TDeltaX as disMin, and reach cell (1,4) as shown below.



The third segment of the ray has now become solid. We increase the value of TmaxX by TDeltaX, and for the first time in this example, reach the scenario where TmaxX exceeds the length of TmaxY. We therefore step in the y direction and store TDeltaY as disMin. Because the y component of the direction vector is positive, we increment j and enter into cell (2,4) as shown below.



The fourth segment of the ray is shown as solid, and TmaxY has been increased by the distance TDeltaY. It is visually apparent that TmaxX is much shorter than TmaxY, and therefore the comparison in the algorithm would lead to a subsequent step in the x direction into cell (2,5) as shown below.



The fifth segment has become solid, and TmaxX has been increased by TDeltaX. At this time, the reader should be familiar enough with the algorithm to predict that the next two steps will be in the positive x direction, at which point the ray would either terminate if the wall is black, or reflect based on the reflection algorithm which will be discussed in later sections. Also in later sections, the reader will find a discussion of the attenuation of radiation from each of the cells along the ray path back to the origin, and the importance of disMin will become apparent.

Non-cubic Cells

For domains which contain cells that have non-unity aspect ratios, additional considerations become necessary in the ray marching algorithm. Throughout the above described algorithm, distances are handled in units of cell width, which can then be converted to physical units simply by multiplying by the cell size of Dx. However, when Dx is not equal to Dy or Dz, this conversion becomes non-trivial, and requires additional computations. By normalizing the lengths of Dy and Dz by Dx, we are able to minimize the number of additional computations, such that only 6 lines of code require modification. Explanation of this procedure is as follows.

Take the distance Dx to be of unit length. Then Dy and Dz have normalized lengths of $\frac{Dy}{Dx}$ and $\frac{Dz}{Dx}$, respectively. The first section of the algorithm that requires modification is then the determination of ray origins. Previously, for cubic cells, we assumed that the origin is located at $(i+rand(),\ j+rand(),\ k+rand())$ where $rand()$ represents a function call to the random number generator which returns a random number distributed between 0 and 1. For non-cubic cells, we must scale the random numbers for the y and z directions by $\frac{Dy}{Dx}$ and $\frac{Dz}{Dx}$, such that the origin of a ray becomes $(i+rand(),\ j+\frac{Dy}{Dx}rand(),\ k+\frac{Dz}{Dx},rand())$. In this manner, we ensure that the origins are randomly distributed throughout the cell, and not simply throughout a cube.

The second portion of the algorithm in need of modification is the determina-

tion of the original TMax values. Recall that the initial TMax values represent the distance from the origin to each of the respective x,y, and z planes. For example, recall that for cubic cells, TMaxY is calculated as follows

$$TMaxY = (j + sign[1] - rayLocation[1]) * invDirVector[1] \tag{1}$$

where $sign[1]$ is a boolean with a value of 1 if the y component of the direction vector is positive, and zero otherwise. $invDirVector[1]$ represents one divided by the y component of the direction vector. For instance, in figure 1, the origin is located at 17.343, 8.617. The direction vector has components of 0.7071, and 0.7071, such that the sum of their squares is equal to 1. Because the sign of the y component of the direction vector is positive, we add one to 17 to represent the location of a y breach, and subtract that value from the y value of the origin, then multiply by the inverse of the y direction vector. Implementing equation (1) we find TMaxY as follows

$$TmaxY = (8 + 1 - 8.617) * \frac{1}{.7071} = 0.5416$$

This value is smaller than that of TmaxX which is calculated as follows

$$TmaxX = (17 + 1 - 17.343) * \frac{1}{.7071} = 0.9291$$

For non-cubic cells, however, when a given component of the direction vector is positive, we need to subtract the origin value not from 1 + j, but from $1 * \frac{Dy}{Dx}$ +j. When the component of the direction vectore is negative, this multiplication of $\frac{Dy}{Dx}$ becomes unnecessary. This is because the negative face value (8 in figure 1) is independent of the skewness ratio. To elegantly handle the condition of multiplying by the ratio $\frac{Dy}{Dx}$ the following formulation is used for the more general case of non-cubic cells.

$$TMaxY = (j + sign[1] * \frac{Dy}{Dx} - rayLocation[1]) * invDirVector[1]$$

To illustrate, see figure 2 where a cell with $\frac{Dy}{Dx} = 2$ is superimposed onto the same setup as illistrated in figure 1. Here, TMaxX is still equal to 0.9291, but TMaxY is solved as follows

$$TMaxY = (8 + sign[1] * \frac{2}{1} - 8.617) * \frac{1}{.7071} = 1.959.$$

Therefore, the ray will not breach the y face during the first step, but will instead breach the x face and enter into the cell to the right. TDeltaY and TDeltaZ are solved in a similar manner, such that for non-cubic cells, the following formula holds.

$$TDeltaY = invDirVector[1] * \frac{Dy}{Dx}. \tag{2}$$

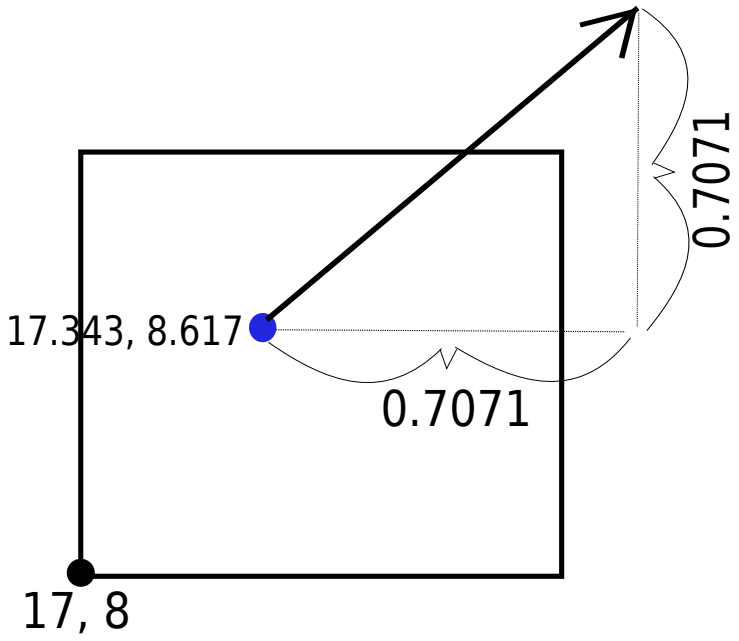


Figure 1: First step in a cubic cell

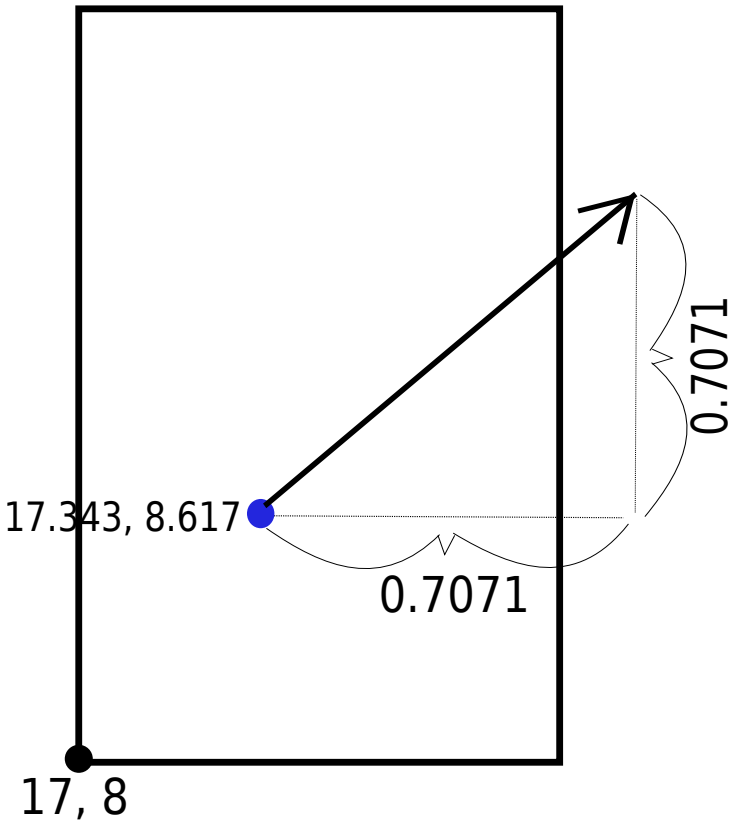


Figure 2: First step in a non-cubic cell with $\frac{Dy}{Dx} = 2$

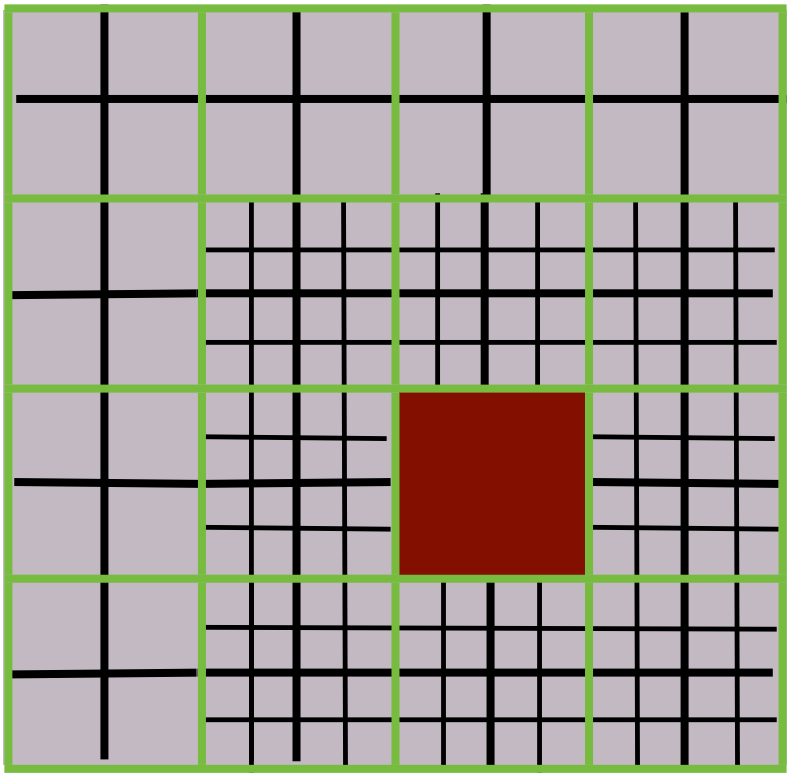


Figure 3: Multi-level mesh. The processor owns the fine mesh information indicated by the red square, and is passed coarsened versions of the mesh for regions outside of the local extents.

Multiple levels

Perhaps the most novel feature of our ray marching algorithm is that it allows for ray tracing on a multi-level adaptive-focus mesh. To accomodate for memory constraints and message-passing constraints, a mesh with multiple levels is passed to each processor. An example of one such mesh is shown in Fig. (3).

The ray tracing algorithm then needs to be able to accomodate this mesh. Because the first step of a ray on a new level is handled in a unique fashion, let us first address the general case of ray marching in the coarser regions. Recall from Eqn. 2 that the values of $TDelta$ are obtained from the direction vector and the normalized lengths of the cell in the x, y, and z direction. Therefore, for general ray marching in a coarsened domain, these values simply need to be scaled by the respective coarsening ratios in each of the Cartesian directions.

$$TDeltaY_c = invDirVector[1] * \frac{Dy}{Dx} * \frac{Dy_c}{Dy_f},$$

where $TDeltaY_c$ represents $TDeltaY$ on the coarser level, Dy_c represents the cell width in the y direction on the coarser level, and Dy_f represents the cell width in the y direction on the previous (fine) level. Similar expressions may be obtained for the x and z directions. Notice that this scheme can work with any arbitrary number of levels, by simply using the current level as the "coarse" level, and the previous level as the "fine" level.

First Step in a new level

The first step in a new level requires special attention because the values of $tMax$ cannot simply be incremented by $TDelta$ of the previous level, nor can it simply be incremented by $TDelta$ of the current level.. This is demonstrated by the four different segment lengths through cell L1:01 of Fig. (4). The four rays have equivelant direction vectors, but each enters the coarser cell through a different fine cell. With careful attention to the location of the fine cell relative to the coarser cell, we can deduce the four segment lengths of the rays. The cell indices of Fig. (4) are written as if the left and bottom edges represent the boundary of the domain. However, even if this is not the case, we can obtain equivelant indices by use of the modulus operator (%). Let *cur* represent a Uintah Vector that contains the non-modultated cell indices of the finer cell after a new level has been reached, but before the indices have been mapped to the new coarser level. Then, to get the cell indices of the fine cell relative to the coarser cell, we perform the following operation: *cur* % CR, where CR is also a Uintah Vector and represents the coarsening between the two levels in each of the three directions. For example, CR in the y direction is given as follows,

$$CR_y = \frac{Dy_c}{Dy_f}.$$

Notice that in Fig. (4), after the y face has been breached, but before *cur* is mapped to the coarser level, the indices of cur would become 0,2; 1,2; 2,2; and 3,2. Performing the modulus operation on these values relative to their respective coarsening ratios yields the following values: 0,0; 1,0; 2,0, 3,0. Now, in order to determine the segment length of each of the rays through cell L1:0,1, we need to subtract the value of $Tmax[dir]$ from $Tmax_{prev}$, where $Tmax_{prev}$ is the length from the ray origin to the level boundary breach, and is equivelant in all 4 rays, and where $Tmax[dir]$ is equal to the length from the origin of the ray to the location where the ray exits cell L1:0,1, and is different for each ray. We therefore must solve for the distance of $Tmax[dir]$, which is a function of the location of the finer cell from which the ray entered the coarser cell. Note that for the ray leaving cell L0:3,1, its $TDeltaX$ value for this first step in the new level is equivelant to $TDeltaX$ of the previous level. Therefore, the $TmaxX$ value correctly describes the next x breach of this ray, and needs no adjustment for this first step in the new level. The remaining cells L0:0,1, L0:1,1, and L0:2,1 however, will require an additional $3TDeltaX$, $2TDeltaX$, and $TDeltaX$ to be added to the current $TmaxX$ values in order to accurately represent the location at which the next x breach will occur in this new level.

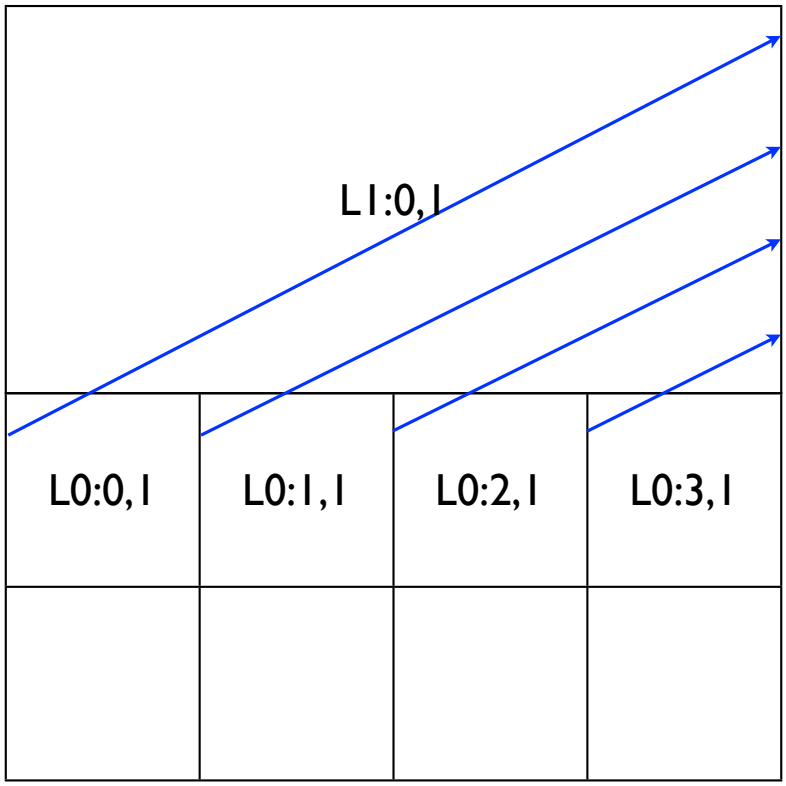


Figure 4: The segment length of the first step in a new level is a function of the location of the fine cell of interest relative to the coarser cell.

index	factor
0	3
1	2
2	1
3	0

Table 1: For positive components of the direction vector, mapping is shown of the modulus of the index to a factor that will scale the $TDelta$ values.

index	factor
0	0
1	1
2	2
3	3

Table 2: For negative components of the direction vector, mapping is shown of the modulus of the index to a factor that will scale the $TDelta$ values.

Looking at the x component of *cur* we can see that we need to map the indeces to the appropriate factor that will be used to become a multiple of $TDeltaX$ in determining the new $TmaxX$ as shown in Table (1). This table holds when the component of the direction vector that corresponds to the breached face is positive. When this component of the direction vector is negative, the mapping is trivial, and is shown in Table (2).

The mapping shown in these two tables is accomplished by the variable *lineup* which is then used to scale $TDelta$ and update $tMax$ as shown in Algorithm (1). Up to this point, we have primarily concerned ourselves with the x component of the locations and directions in the example of Fig. (4). Fortunately, the other component (or components, if in three dimensions) follow a similar procedure. If in three dimensions, the component normal to the page in Fig. (4) would be handled in an identical fashion to the x component. The y component is also handled identically, given that the usual incrementation of $tMaxY$ is handled prior to the executions of Algorithm (1). The modulus of the component of the index that corresponds to the breached face will always return 0, giving a *lineup* value of 0 for the positive cases and 1-CR for the negative cases. When *lineup* is scaled with $TDelta$ as shown in Algorithm (1), then $tMax$ in the first step of the new level will be assigned appropriately such that the subsequent breach in the new level will occur on the wall of the coarser cell, as indicated in Fig. (4).

Algorithm 1 This algorithm computes the tMax values for the first step in a new level. sign[ii] returns true if that component of the direction vector is positive. At this point in the algorithm, L has not yet been updated, and therefore represents the previous level.

```
for (int ii=0; ii<3; ii++){
 if (sign[ii]) {
 lineup[ii] = cur[ii] % coarsenRatio[ii] - (coarsenRatio[ii] - 1 );
 }
 else {
 lineup[ii] = cur[ii] % coarsenRatio[ii];
 }
}
tMax += lineup * tDelta[L];
```