Regular Grids

When we discussed BVHs, we discussed that while they have the ability to adapt to the geometry and their distribution, they did have their downsides. One of the more significant problems as far as ray tracing is concerned is the following: suppose we shoot a ray into the scene that contains a BVH. We traverse the hierarchy as explained before, and we find that a geometry intersects the ray. As part of our calculations, we now need to determine whether that particular point on the surface is in shadow. To accomplish this, we would have to shoot a ray towards each of the light sources and determine whether any objects intersect the ray. The issue with this is the following: the starting point for the ray is within the BVH itself. The reason why this is problematic is because we can't just start intersecting the ray against the BVH as we did before. Since the origin of the ray is in the BVH, it will always return true on every node until it returns to the surface that shot the ray.

The problem outlined above is definitely solvable: all that we would have to do is climb our way back up the hierarchy and determine if there are any other child nodes above us that do intersect the ray and then continue normally. The problem is that this does require special handling, which may not be ideal. What we would like is then another type of data structure that would continue to work *regardless* of the origin of the ray. At the same time, we would also like to have the same advantages that the BVH gives us: the ability to segment space into different regions so we can cull unneeded objects, and the ability to determine which object(s) we need to intersect against instead of checking all of them.

A data structure that can accomplish this is a *regular grid*. Essentially, the idea is to split space into a series of adjacent cells, creating a grid pattern that sits on top of the objects. Note that for a regular grid, all cells are the same size and (ideally) cubical.

The way the grid accomplishes these objectives is by storing a reference to which object(s) lie within each of the cells. The cells of the grid are then traversed in the same order that they are intersected against. What this means is that for areas where the objects are sparse, we would have to traverse the majority of the grid, which is part of the reason why they don't work very well for sparse scenes. The situation improves dramatically if the objects are tightly packed. In this situation, we would only need to traverse the grid until the first hit, as we know that any cells behind the nearest hit will not be reached by the primary ray. The same logic can then apply to secondary rays. For a scene with a large number of objects n, a grid will have $O(\sqrt[3]{n})$ cells in each direction, and rays will therefore be intersected against $O(\sqrt[3]{n})$ objects. This case applies when the distribution of objects is sparse. However, in the case that objects are tightly packed, the number of objects intersected is closer to $O(\log n)$.

Constructing a Grid

For the sake of simplicity, we will make the following assumptions:

- Every object in our scene has an AABB.
- The grid will be cubical.

In reality it isn't that hard to get rid of these assumptions and the underlying logic for constructing the grid doesn't change.

With these in mind, the first stage in constructing a grid will be to determine the volume that the grid will span. A very straight-forward way of doing this is by taking all of the bounding boxes of the objects in the scene and merging them together into a single AABB. The dimensions of the grid will now be determined by this maximal AABB. Another option would be to manually specify the dimensions of the grid, which can be helpful if we have a particular sub-region that we wish to focus on.

Once the dimensions have been specified, the next part is to define the resolution of the grid. Let n_x, n_y, n_z be the number of cells in the x, y, z directions and w_x, w_y, w_z be the dimensions of the grid. We can use the following to calculate n_x, n_y, n_z such that the cells are roughly cubical:

$$
\begin{aligned}
s &= \sqrt[3]{\frac{w_x w_y w_z}{n}} \\
n_x &= \text{trunc}(m w_x / s) + 1 \\
n_y &= \text{trunc}(m w_y / s) + 1 \\
n_z &= \text{trunc}(m w_z / s) + 1
\end{aligned}
$$

Here, m is a factor that allows us to vary the number of cells, while the $+1$ ensures that we can never have 0 cells in any direction. From these equations, we can derive that the total number of cells is mn^3, provided that $n_x \gg 1, n_y \gg 1, n_z \gg 1$ which is usually the case. If $m = 1$, then the number of cells is approximately equal to the number of objects in the scene. In general, it is not ideal to have too few cells, as we would end up having to intersect against most (if not all at the limit) objects in the scene. On the other extreme, if we have too many cells, we would end up wasting time having to traverse empty cells. In general, grids tend tend to be at their best when they are 8 to 10 times more than n. As a result of this, we usually use $m = 2$ as a factor, but it is important to experiment with different cell numbers to see which one has the greatest benefit for a given scene.

Once the cells are determined, we can now start assigning objects to each cell. This process is straight-forward since we assumed that all the objects have their own AABB. All that we would have to do is determine the range of cells that each AABB occupies, which can be easily computed using the two corners of

the box. For each cell that the box covers, we would assign a reference to that object in each cell. At the end, we would end up with cells containing no objects, or one or more objects. This concludes the creation of the grid.

Traversal

Traversing a grid is relatively straight-forward. In this case, we will focus on ray traversals, though it is easy to see how this extends to any object intersections. The general algorithm is the following:

1. If the ray does not intersect the grid's bounding box, return immediately.
2. If the ray starts inside the grid, find the cell that contains the origin of the ray. Otherwise, find the first cell that the ray hits.
3. Walk the ray along the grid.

The first measure is mostly an optimization, though it can be used to obtain useful information regarding the maximal values of t the ray can have. The step to compute the first cell to use is straight-forward. If the ray starts inside the cell, we can quickly determine which cell it occupies by performing

$$\frac{(o_x - x_0)n_x}{x_1 - x_0}$$

For each of the x, y, z axes. Here o_x is the x-coordinate of the origin of the ray, x_0, x_1 are the x-coordinates of the lower-left and upper-right corners of the grid's box, respectively. Once we have this number, we can clamp it in the range $[0, n_x - 1]$ to obtain the final index of the cell.

For the case when the ray starts outside the grid we can do something very similar. The only change is that we have to find the point of intersection between the ray and the bounding box of the grid. Once we have that, we essentially have a point inside the grid and so the process is the same.

Once we have determined the starting cell, the next step will be to determine which cell the ray will hit next. To compute this, we will note that although the distribution of hit points along the ray will be uneven, if we focus on the same distribution over a single axis, it will be even. What this essentially means is that we can advance along each axis by just adding a fixed amount to the initial value of t of the ray. This delta can be computed by taking the difference between the maximal and minimal values of t along each axis and dividing this by the number of cells in that direction.

So, we have $\delta_x, \delta_y, \delta_z$ for t. What we do next is we add each δ to the value of t that was computed for the given cell and we obtain a new value of t on each axis. From there, the only question is to determine which cell will be chosen next if the ray is angled in such a way that it will hit two (or more cells) at once. This can be determined by always following the smallest value of t from among t_x, t_y, t_z. This strategy ensures that we don't skip cells by accident.

Once we have determined the next cell, we advance to it and check if it has any objects in it. If it does, we then intersect the ray against those objects. If any objects are hit by the ray, the process stops. By construction of the grid, we know that this intersection will be the closest one along the direction of the ray, and therefore all other objects that are behind will not be hit.

Beyond Grids...

Grids are one of the simplest forms of space-partitioning algorithms, and lend themselves very well to tightly packed scenes. They can also be extended to allow *nested* grids, where each cell can be subdivided into another grid. This structure is called an *oct-tree* and allows further flexibility than the grid by allowing finer resolutions in areas that are more densely packed, while keeping coarser grids for sparse areas. Note that an oct-tree still has a regular structure as all the grid cells are still proportional to each other. If we drop the restriction that all cells need to be the same size (and cubical), we obtain a kD-tree. This can provide more flexibility and precision when dividing space since it allows tighter cells, but at the cost of update times. If you are interested in reading more about these types of structures, you can check out Real Time Collision Detection by Christer Ericson.