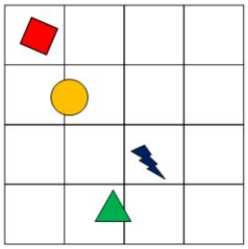
Uniform Grids

jodavis42@gmail.com

Uniform Grid

Partition space into grids



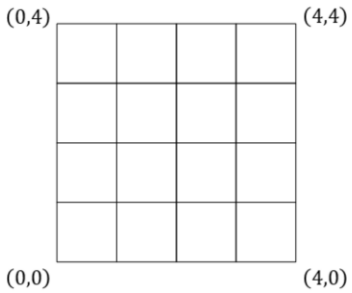
We know only nearby objects can overlap

A uniform grid is a fairly simple spatial partition (ignoring all of the special cases…) that divides space into uniform sized chunks. You've probably already implemented one at some point. They work by assuming that an object can only overlap with objects in the same cell.

When used correctly, or in the correct situation, uniform grids can be incredibly powerful.

Position Discretization

Can turn any point into a cell position



$$i_x = floor\left(\frac{p_x}{size_x}\right)$$
$$i_y = floor\left(\frac{p_y}{size_y}\right)$$

The first thing to understand in a uniform grid (and why it's so powerful) is how you turn an arbitrary point into a cell index. We can immediately turn a point to its index by discretizing it to the grid. An index from a coordinate is simply found by dividing out the cell size and taking the floor of the result.

Point Discretization

This discretization implies a “bottom left” fill rule

Grid also positioned at bottom left at origin

It’s important to realize what the point discretization we’ve used implies. First of all we have a “bottom left fill-rule”. What I mean by this is that points get categorized as:

$$\vec{\mathrm{p}}(0,0) \rightarrow \vec{\imath}(0,0)$$
$$\vec{p}(0.9,0.9) \rightarrow \vec{\imath}(0,0)$$
$$\vec{p}(1,1) \rightarrow i(1,1)$$

This is to show that the edge between two cells belongs to the cell to the upper-right of it. In particular, this is important because when we turn a cell index into a position by: $(p_x, p_y) = (i_x, i_y) * (size_x, size_y)$ that the resultant position is the bottom left of the cell. So if we want any edge or the center of the cell we need to offset this position accordingly. In particular, this is important later when dealing with ray-casting.

The other important thing to realize is that we’ve positioned the origin at the position of cell $(0,0)$ in order to make the math nice. There are occasions where it would be nice if the cell center was at the origin instead of the bottom-left but for efficiency we are not doing it this way.

Initial Grid Structure

```
class Grid
{
  size_t mSizeX;
  size_t mSizeY;
  Array<Cell> mCells;
};
```

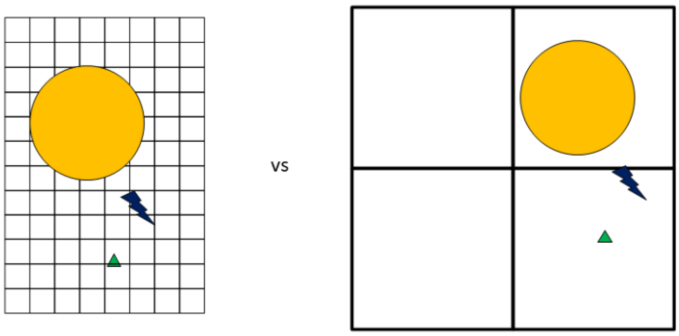
```
class Cell
{
  Array<void*> mObjects;
};
```

The simple first structure to use in a uniform grid is having a fixed sized number of cells in an array. Typically we represent a 2-d array as a 1-d array with indexing $i = x + size_x * y$. Each cell needs to be able to store all objects that it contains so for simplicity we'll store an array of void pointers which is the client data we were handed.

We could use some other container instead of an array such as a list or even a map to make it easier to find objects but that's not something I will worry about now. The main reason I pick an array is that it has a very small memory overhead.

Cell Size

How do we determine what cell size to use?



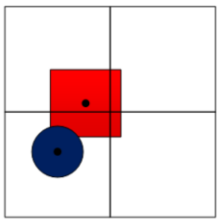
One of the most important considerations in a uniform grid is how big each cell is. If grids are too big then we'll have a lot of objects resolve to the same cell and not gain the benefits of the grid. If the cells are too small then an object will be put in a lot of cells and we have a lot of extra overhead maintaining that. Ideally we make the grid just large enough to contain any object at any rotation. However, if we have varying sized objects we have to choose some cell size that will be un-optimal for some of the objects. It's for this reason that uniform grids work best when all objects are the same rough size. There is an addition that'll be discussed later to deal with varying sized objects.

Object Insertion

Object can overlap up to 4 cells in 2d

Which cells do we insert into?

- Center's cell
- Bottom left's cell
- All overlapped?



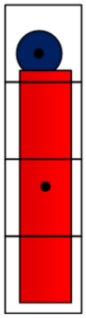
Assuming we can choose a good cell size, when inserting an object it'll overlap at most 4 cells (in 2d). This begs the question, how many cells and which ones should we insert the object into?

The two options are to insert into only one cell or into all overlapped cells. One cell would be the most optimal, but if so which one would we insert into? We could use some fixed position such as the object center. If we do this then an object can collide with a neighbor's cell so we'll have to check extra cells for pair queries.

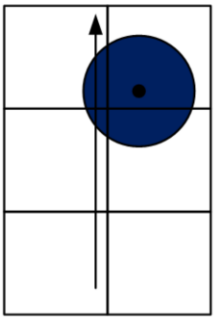
Unfortunately there's a few problems with this method…
The first major problem is that pair queries need to check neighboring cells. Shown above the two objects are clearly overlapping but if we insert based upon their center's then they won't check against each other.

Object Insertion

One-point insertion breaks with:



Large objects



Ray-casts

There are two main reasons inserting at only 1 position breaks down.

The first and most obvious case when it breaks is if an object is more than one cell in size. If so we have to check more than just the neighboring cells and we don't really have a good method of determining how many to check.

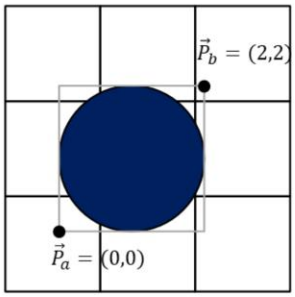
The other case is when casting another shape, such as a ray. To do an efficient ray test we only check what cells the ray goes through. However there's a chance that we'd have to essentially check a thick ray in order to find objects.

Because of these reasons I don't recommend inserting into only one cell, but rather all cells.

Object Insertion

To insert into all cells:

- Discretize Aabb min and max
- Insert into all cells in range



To insert into all cells is fairly simple, but to make life a bit easier we'll insert based upon an object's aabb. In that case we can just discretize the aabb's min and max position and then iterate through all cells that are overlapped, inserting into each one.

Update and Removal

Need to know where an object was to remove it

2 Methods:

- Rebuild from scratch each frame
- Store old information

There is one main problem with having chosen to insert into all overlapped cells: removal (and hence update) becomes harder. We can no longer just remove from whatever cell we were in, but rather we have to remove from all cells we were in.

It's actually a little tricky to figure out which cells to remove from. We could always discretize the current aabb to try to remove the object but this assumes that the object hasn't moved, so we'd have to pass in the old aabb. We'd like to avoid storing this data and having to modify the grid's interface to take this if possible so we'll ditch the idea of passing in the old aabb. We'll come back to the idea of using the old aabb later, just in a different way.

The other method is to just rebuild the grid every frame.

Rebuild From Scratch

Total re-insertion: $O(n)$

Doesn't work well with static objects
Wastes a lot of calculations
Different interface required

As insertion is $O(1)$ per object we can just clear the old grid and re-insert all of the objects again to avoid the removal/update problem.

There are two big issues with this though. The first is that to re-insert everything is fairly expensive, especially if there's a lot of non-moving objects. This is one reason to split static and dynamic objects into separate spatial partitions (they can both be grids). Even then, there can be a lot of slow moving objects so clearing the whole thing every frame is not cheap.

The other big problem is that this has to fit into the design of the engine using it. Now we have to make the user pass in all objects every frame when inserting, not just the ones being inserted or updated. This works fundamentally different from other spatial partitions which makes it harder to swap out dynamically.

Intrusive Information

Store range in the grid (per object)

```
class Grid
{
  size_t mSizeX;
  size_t mSizeY;
  Array<GridBox> mBoxes;
  Array<Cell> mCells;
};
```

```
class GridBox
{
  int mMinX, mMaxX;
  int mMinY, mMaxY;
  void* mClientData;
};
```

```
class Cell
{
  Array<size_t> mObjectIndices;
};
```

Maybe we can go back to the old idea of storing the range but find a better way to do it. The biggest problem with this was changing the interface and forcing the user to intrusively store this information on each object. Instead we can store this information intrusively in the grid.

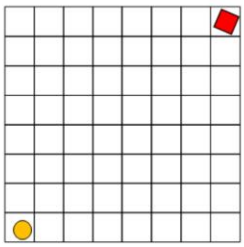
We can change our grid to be a unique list of objects. This unique list can store the void* for the client data as well as the current overlap range of the object. We could also change the grid to store an index into this array instead of the client data itself.

The only two problems are that we have to store extra information per object, which is not bad, and we have to find the object in the array when updating/removing it. The second problem is easily mitigated by the proxy method described in class. The information we make the user intrusively store can just be the index into the object array making lookup time constant.

Grid Size

Current grid is fixed size (m x n)

Memory requirements based upon space



Needs 64 cells for
2 objects

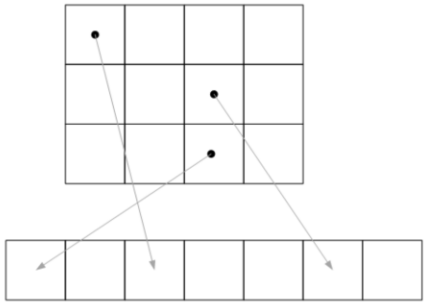
Now one large problem with the current grid is that it is of a fixed size. If we have a world that is limited then this works out just fine, but what if we have an unbounded world? We could dynamically grow the grid but this is extremely costly as we need to allocate a new buffer and copy everything over.

A more fundamental issue than growing is how much space we need is not dependent on the number of objects, but on the space we're representing. If we have 2 objects really far apart we need a lot of memory to represent the empty cells between them. We could use a larger grid size but this would cause other expenses. This problem is even worse in 3d and makes any reasonably sized 3d grid unreasonable. A grid of size 1000 in 3d would require 3MB of cells (times the size of a cell…).

Because of these reasons, we have to look into a representation of a uniform grid that retains the previous benefits but can support growth and not waste a lot of memory.

Cell Hashing

Hash into an array based upon cell position



The simplest solution to this problem is implicitly represent a grid via a hash function. You should already know that a hashmap takes some key and turns it into an index into an array. In the same way we can create a hash for a cell coordinate to map it into an array. In doing so the amount of memory we need is only based upon the number of objects (or rather, the number of cells overlapped). This method also implicitly grows and is unbounded.

The main thing to keep in mind is hash collisions. There's two main methods of resolving collisions: chaining and probing. For most part it doesn't matter what method is used as this is more of a container issue, but the important thing to make sure of is that you don't put two objects into the same cell just because they have a hash collision.

Cell Hashing

Many possible hash functions

```
size_t ComputeHash(int x, int y, int z, size_t bucketCount)
{
  //Arbitrary large primes
  const int h1 = 0x8da6b343;
  const int h2 = 0xd8163841;
  const int h3 = 0xcb1ab31f;
  size_t n = h1 * x + h2 * y + h3 * z;
  n = n % bucketCount;
  return n;
}
```

There's a lot of different hashing functions that you could use. Here I present a simple one, but I do recommend search for various other ones.

Cell Hashing

Pros:

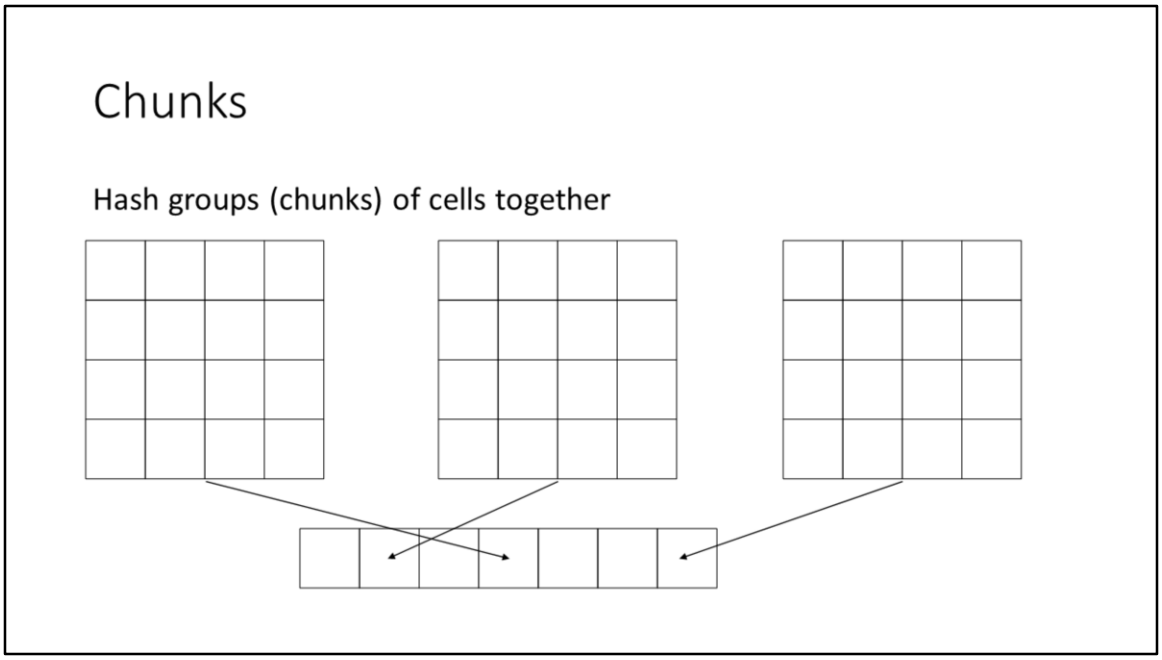
- Less memory
- Unbounded world

Cons:

- Hashing is slower
- Hash collisions slow things down

While hashing cells does fix our initial two problems it does slightly slow down our uniform grid. The extra cost from hashing isn't too bad (although it can definitely add up) but we also have to worry about the cost of resolving hash collisions.

These downsides aren't too bad, but there is another interesting idea to combine fixed arrays and hashes that give a few other benefits: Chunks.



The idea of a chunk is quite simple, instead of hashing each cell we can hash a group of cells, or a chunk. Each chunk contains a mini-uniform grid of fixed size (say 16x16). While this will waste a bit of extra space as each chunk takes a fixed amount of memory even if it only has one active cell it also has a few neat benefits.

First of all, we can get some extra speed-ups by not having to hash so often. We do now have a double look up (chunk indices and cell indices) but we can cache the last active chunk to avoid re-hashing an index. The memory of a chunk is also going to be local so cache performance should increase.

The other major benefit of chunks is the ability to stream portions of the world. Chunks very nicely fit this paradigm and games like minecraft use it. Simply take any chunk that is too far away from the player and save it to disk. So with this we gain the benefit of being able to have a much larger world than we can have loaded into memory by dynamically loading/unloading portions of it.

Object Casting

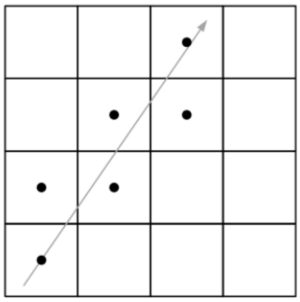
Discretize Aabb of object

Check all overlapping cells

Object casting is the easiest of the intersection tests to perform. Essentially we want to discretize the object's boundaries to grid coordinates and walk over all of those cells. The simplest method of doing this is to take the aabb of the object and then walk all of the relevant cells, just as we did when inserting an object. This will check extra cells but it is much simpler than walking an arbitrary object's boundary.

Ray Casting

Basically line rasterization (DDA)



Uniform grids are very efficient to ray-cast through. They are essentially a line rasterization algorithm, however we have to be careful which technique we use. Unlike Bresenham's algorithm, we need to make sure that we traverse every cell the ray goes through.

The technique to use here is DDA or the digital difference algorithm.

Ray Casting (DDA)

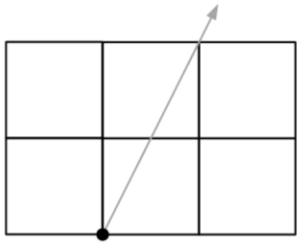
Check for x and y intersection times and choose smaller

$p(x, y) = (1,0)$

$p(x + 1, y) = (2,0)$

$p(x, y + 1) = (1,1)$

$\vec{d} = (1,2)$



$$t_x = \frac{2 - 1}{1} = 1$$

$$t_y = \frac{1 - 0}{2} = 0.5$$

The simplest implementation of DDA is as follows: At a given point $\vec{p}$ we know we are in cell $\vec{c}$ and the direction of the ray $\vec{d}$. Assuming we can compute the position of the next edge's intersection $\vec{e}$ (the x position and the y position of the next edge) we can perform two ray vs. plane intersections to get the t values at which we intersect each edge. We can then walk the ray to the closer of the two t values and increment the indices accordingly.

There are a few complications to deal with here though. First of all we need to be careful of which direction the ray is going, namely we need to check the x and y direction to determine if the next cell index is +1 or -1. The other thing to be extra careful about is how to compute the next edge's value. The current method of computing cell indices is based upon the bottom-left of a cell. So depending on the direction the ray is traveling we have to make sure we get the correct edge. The simplest way to do this is to just compute the cell center (position + half cell size) and then add or subtract half the cell's size on each axis depending on the sign of the direction vector.

Ray Casting

How do we deal with infinite grids?

Infinite ray + Infinite grid = Infinite traversal…

Need to convert ray to segment

One weird problem that shows up with a ray to grid traversal now shows up if we have an infinite grid. We have to walk cells one at a time but we'll reach a lot of non-existant ones (chunks help this a bit). But since the grid and the ray have an infinite size how do we know when to stop walking cells?

The simplest answer is to turn the ray into a segment and only walk until it ends. But how do we do this? One way is to force the user to do this. There is one major problem with that though, what if the user doesn't know what range they want? They'll just pass in an arbitrarily large range which means we could walk cells forever. Either way we need some method of creating an upper limit on the ray.

The easiest way to do this is to compute the current scene's aabb. This can be done by iterating once over all objects every time an insertion or removal happens. While it is easy to expand an aabb every time we add an object there is no way to deflate an aabb when removing an object apart from re-computing it from scratch. Alternatively, we can walk over all active chunks and combine their aabb's. This will often only traverse a fraction of actual objects. Once we have the aabb's size then we can compute the max range of the ray as the max time we leave the aabb.

Pair Query

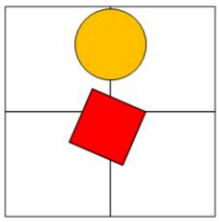
2 Methods:

- Incremental
- Batch

There's two main methods of compute pairs: Increment and Batch. The most straightforward one to think of is batching. Batching is just checking all existing objects in the grid to see what can overlap. Conversely incremental maintains a pair list that is updated any time an object is inserted or removed. In either case the logic is the same for checking for pairs.

Pair Query

Add pairs for every object in the same cell



Be careful of getting a pair twice

If we inserted objects into every cell they were overlapping then pair checks are simply iterating over all cells and adding pairs for every object that shares a cell. There is one main caveat here though: we have to be careful of creating duplicate pairs. If two objects share two cells that they overlap in then they'd be registered twice. The easiest method to deal with this is to hash the pairs based upon a lexicographic index. There exists other complicated methods of time-stamping pairs, but I won't address them as hashing should be sufficient.

If we had only inserted into one cell this test would be much easier as we'd only have to check the current cell and neighbors. In particular, if we inserted based upon the bottom left then by only checking the upper right 4 cells since other cells would check against us.