# README! v1.0

This large document explains all the concepts and tricks that you need to know in order to fully understand my isometric engine and expand on it further. You can read it first, or just dive into the engine's MMF source right away. But you still need to read this if you are really interested in developing a cool isometric game and have fun (or not fall into depression) while doing it.

Another good reason to read it is because I spent a whole bloody truckload of my time to write it, just for you.

Enjoy…

## Environment and moving characters

One particularity about most isometric games I've played is that moving characters can go behind and under parts of the environment. Some games were made of small rooms and others were large scrolling areas. In any case, they had lots of places where moving characters could go behind static things.

In MMF, the environment is usually made of backdrop objects and moving characters are done with active objects. But active objects can't be layered behind backdrop objects, unless you move them to a lower frame layer. Using frame layers for this could be extremely hard to manage dynamically if you have many moving characters that can go behind many different parts of the environment.

So using some active objects as static environment elements is the way to go. Their alterable values are also necessary for a good dynamic layering process. Backdrop objects are still needed for walls, ground and other environment elements that are sure to always be behind everything else.

Equinox, on the SNES, published by Sony Imagesoft in 1994

## A 3D array for collisions

In the MMF engine, backdrop objects and active object have a collision map for detecting precise collision between them. Non-precise box collision can also be used, but they are still only 2D overlap detection features.

An isometric game is a 3D environment rendered in 2D. Objects that overlap visually don't necessarily touch each other in the 3D space. So collisions need to be done by comparing 3D volumes and detecting if they intersect or not. Volumetric information for active objects can be defined and stored within their alterable values. But what about the static environment made of backdrop objects? Those don't have any alterable values that we can work with. And covering the whole frame area with active objects would swell up memory and slow down the game to a crawl, especially if you intend to do some scrolling.

So the 3D collision volumes of the static environment need to be stored and managed somewhere else, like in an array. A 3D array is a good idea for this. Think of it as a large space for boxes to be neatly stacked around. Each coordinate in the array can represent a space of fixed size. And the value stored in that space represents its content. We can simulate collision within this array by letting moving object pass through empty boxes and stop when hitting a filled box. This is the *Collision array* of my isometric engine.

If you've played isometric action-adventure games or search the internet for images of such games, you will probably notice how blocky and cube-like the environments were. That is a good indication of how those worlds were also made with 3D arrays of cubes for collisions.

Landstalker, on the Sega Genesis, published by SEGA of America in 1993


**Subcategorizing active objects: Static or Moving**

I've explained how active objects are needed as static environment elements, so that the moving characters can visibly go behind and under things. I've also described how the 3D collision information of the environment elements should be managed as a 3D array of volumes. This brings up the concept of *Static objects* in my isometric engine.

"Static object" is a name I use to define active objects that act as environment elements. This type of object should never move. The alterable values it has are mostly used for the layering process; not for movement and collision. The collision information for the *Static objects* is part of the *Collision array*. Visually, the *Static object* is shaped and shaded to fit with a cluster of filled cube cells in the *Collision array*.

A good example for a *Static object* is a tree stump. It's well rooted into the ground and can't move. Little forest creatures can hide behind it. And those little forest creatures are the *Mover objects* in my isometric engine.

"Mover object" is what I call active objects that move around in the 3D space. *Mover objects* are often layered above or below other objects that they overlap visually, depending on how their 3D position and size values relate to the values of those other objects. The collision information of *Mover objects* is stored in their own alterable values, forming a box volume around the object. This volume can be of any size; Large, flat, thin, long, etc. It is used to detect when a *Mover object* hits the *Collision array* or other *Mover objects* during movement.

Another thing to note about isometric games is that objects don't rotate. Though an object might be animated to look like it's facing in different directions, it only does so visually. The actual volume that the object occupies in the 3D can not turn or tilt. All objects are aligned next to each other along all tree axis of the 3D world. This is to simplify all the collision and layering math.

* Remember while reading this document that *Static objects* and *Mover objects* aren't actual object types in MMF, but specific roles given to active objects. In my isometric engine's source code, I simply use qualifiers to separate the two;

Group1 and Group2.

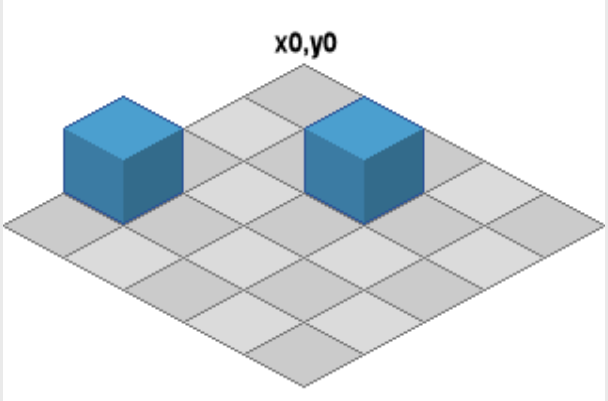
**The rules to sort above or below**

The isometric world is a 3D environment that expands along the X, Y and Z coordinates. We see that world with an orthographic view and at a constant angle. Objects in that world may have volumetric properties, but they are still displayed in MMF as flat 2D images cleverly drawn to look as if they had depth and form. So depending on how these objects relate to each other in position and size, they need to be layered properly to maintain this illusion of depth. The *Static objects* need to be layered only at edit-time, while the *Mover objects* need to be layered dynamically at runtime.

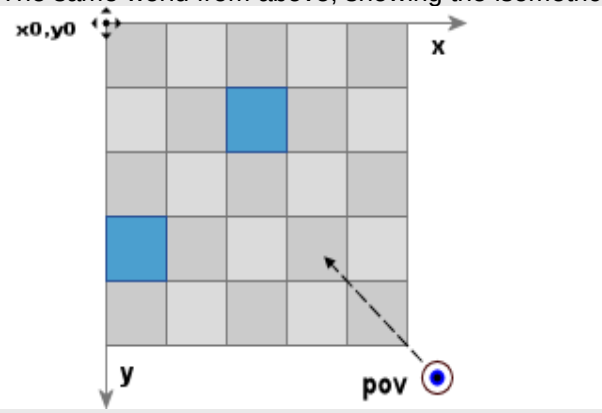
The rules that are part of the dynamic layer sorting process can be really complicated to understand if we keep thinking about them in all three axis of space. But they can be simplified to 2D thinking.

Let's say that in our 3D space, all objects can move and expand along the X and Y axis only. As for the Z axis, all objects sit on the flat surface of coordinate 0. And they are all of the same fixed height. This cancels out all math along the Z axis, leaving our thinking to only X and Y.

Our constrained 3D world from an isometric point-of-view:

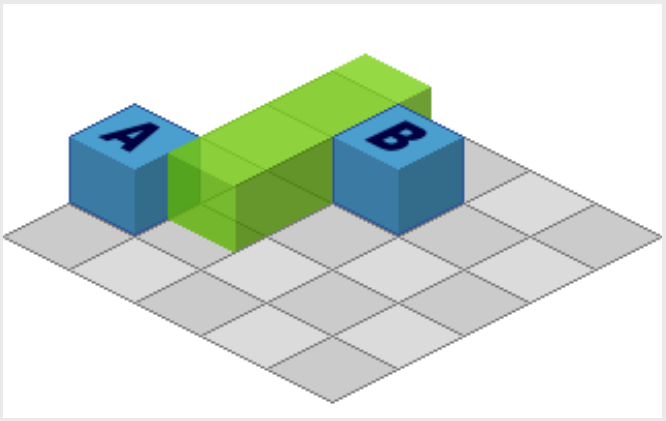



The same world from above, showing the isometric point-of-view as an eye icon:

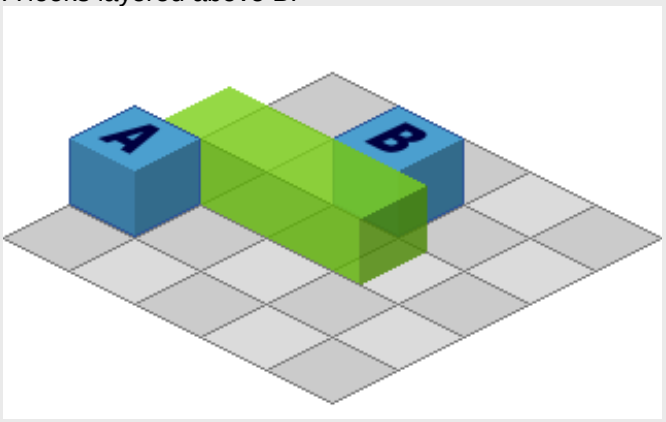



In the setup above, it's actually impossible to know how the two cubes should be layered. Not because they don't visually overlap in the isometric point of view, but because of the possible placement of other objects between them.

A looks layered below B:



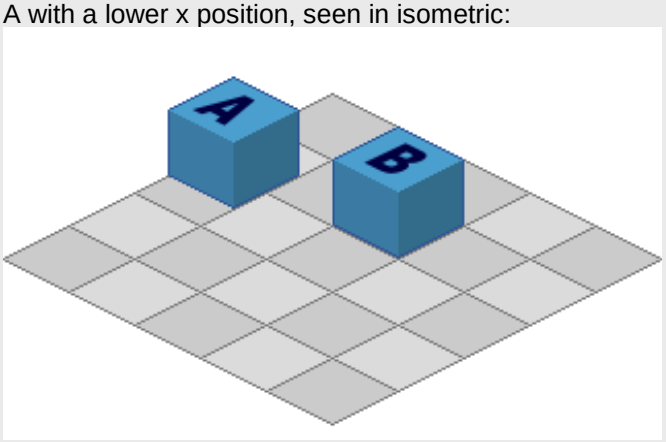

A looks layered above B:



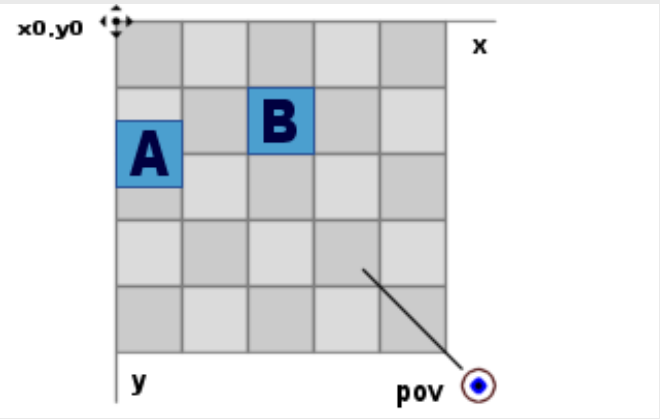

So comparing A vs B should simply be ignored in this case. If there are other objects around them, then A and B should be compared with those other objects to sort out their layer order.

Let's try a slightly different setup that is sure to put one cube behind the other.

A with a lower x position, seen in isometric:

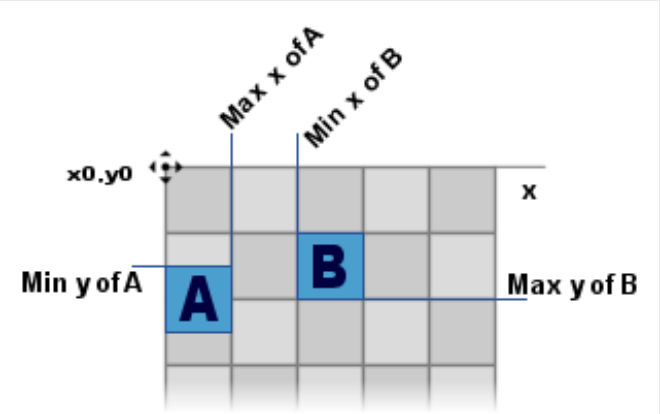



Same thing, seen from above:




If you think that A looks like it's behind B, you're right. No matter what objects could be placed between them, A will always be behind B. And there is a specific logical rule for that:

The **maximum x** of cube A is lower than the **minimum x** of cube B
AND
The **minimum y** of cube A is lower than the **maximum y** if cube B




There's even a second similar rule that puts A behind B:

The **minimum x** of cube A is lower than the **maximum x** of cube B
AND
The **maximum y** of cube A is lower than the **minimum y** of cube B

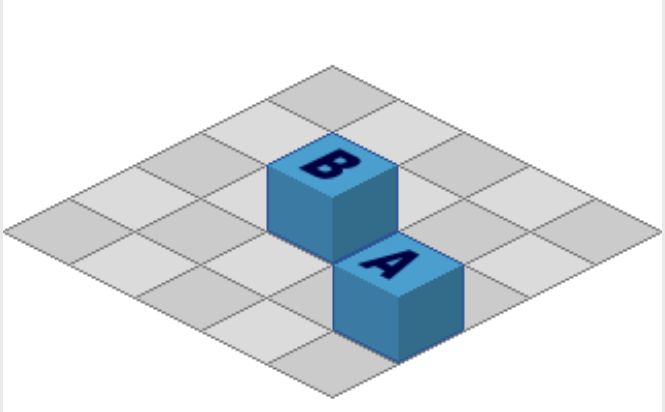
What about the rule that is sure to always put cube A in front of cube B? There is a specific one for that, but it goes with a slightly different logic that considers the relationship between *Mover objects* and *Static objects.*

I've explained earlier that a *Static object* doesn't have a collision volume, but represents a cluster of filled cells in the *Collision array*. Depending on the shape that this cluster of filled cells can take, a *Mover object* will often be able to enter the overall volume of a *Static object* (more on that subject later in this document). So the next rule needs to apply to *Mover objects* that can intersect with *Static objects*. We don't have to worry about *Mover objects* intersecting with other *Mover objects*, but the movement and collision process will need to make sure that it never happens. Otherwise the layering would fail depending on the order in which the *Mover objects* are compared.

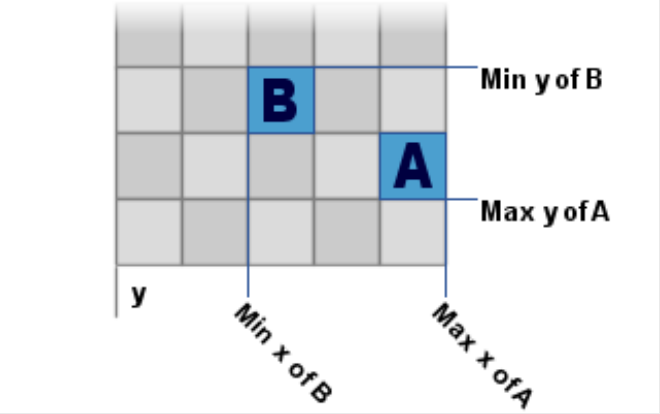
So here is the rule that puts A in front of B:

The **maximum x** of cube A is greater than the **minimum x** of cube B
AND
The **maximum y** of cube A is greater than the **minimum y** of cube B

Cube A and cube B, at it again:




Comparing cube A to cube B with the above rule:




Now let's bring back the Z axis into this whole logic. It's actually very simple since all the same principles apply for the X, Y and Z axis of the isometric 3D world.

These are all the logical rules that dictate how an object is layered against another object according to their position and size:

IF…
The **maximum x** of A is lower than the **minimum x** of B
AND
The **minimum y** of A is lower than the **maximum y** of B
AND
The **minimum z** of A is lower than the **maximum z** of B
OR…
The **minimum x** of A is lower than the **maximum x** of B
AND
The **maximum y** of A is lower than the **minimum y** of B
AND
The **minimum z** of A is lower than the **maximum z** of B
OR…
The **minimum x** of A is lower than the **maximum x** of B
AND
The **minimum y** of A is lower than the **maximum y** of B
AND
The **maximum z** of A is lower than the **minimum z** of B

THEN, put A below B

IF…

The **maximum x** of A is greater than the **minimum x** of B
AND
The **maximum y** of A is greater than the **minimum y** of B
AND
The **maximum z** of A is greater than the **minimum z** of B

THEN, put A above B

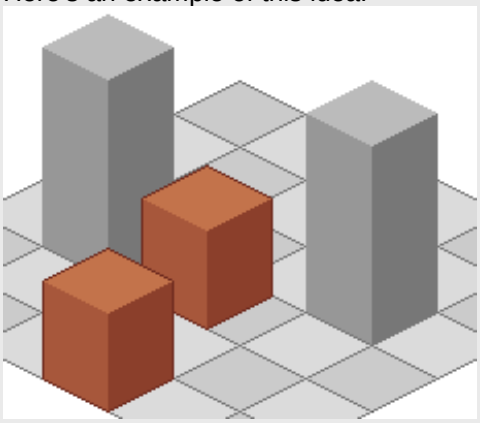
ELSE, do nothing

This set of rules will manage the order of layers, as long as *Static objects* are always compared as the second object (object B) and that *Mover objects* don't intersect with other *Mover objects*.

**Restricting layer sorting & comparing objects in pairs**

Since my isometric engine uses the *Layer Object* extension, it's best to have a layer sorting process that reorganizes only a minimum amount of objects. Categorizing objects is useful for this, so we can limit the layer sorting process to *Mover objects*. But we should also narrow down the sorting only to *Mover objects* that visually overlap *Static objects* and other *Mover objects*. Once those objects are properly isolated, we can compare them in pairs; each isolated *Mover objects* VS each isolated *Static objects* and other isolated *Mover objects*.

Here's an example of this idea:



We have two grey *Static objects* and two brown *Mover objects*: S1, S2, M1 and M2.
M1 overlaps M2 and S1
M2 also overlaps S1
S2 is not overlapped by any *Mover objects*.

Only M1, M2 and S1 should be part of the layer sorting process.
Here is what the comparison sets should ideally look like.
- M1 vs M1 : ignore
- M1 vs M2 : M1 is set above
- M1 vs S1 : M1 is set above
- M2 vs M1 : M2 is set below
- M2 vs M2 : ignore
- M2 vs S1 : M2 is set above

The first of a pair is the one to be sorted in the layer stack. It is always a Mover object, never a *Static object.*
The second of a pair is only used for comparison with the first. It can be another *Mover object* or a *Static object.*

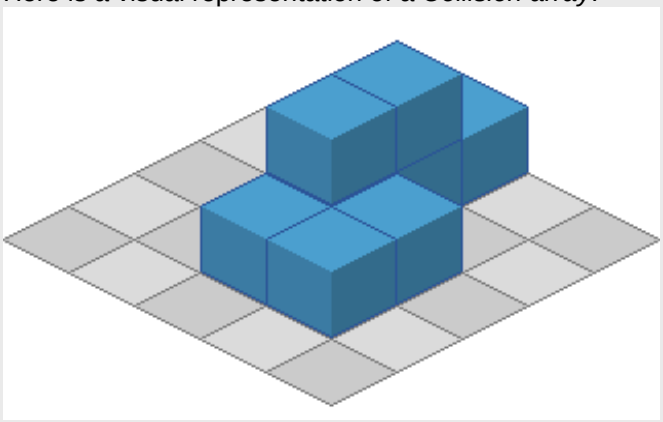
**How to organize Static objects with the Collision array**

*Static objects* need to display the exact content of the *Collision array*. This is more of an artistic and performance issue. If you have a structure in your isometric environment where *Mover objects* can go behind, you will have to make one or multiple *Static objects* to fit with the shape of the Collision grid. This can be tricky.

You can make a cube shaped *Static object* and place copies on each filled cell of the *Collision array* where it's needed. Reusing instances of an object at many places is a good way to save memory and minimize the file size of your MMF game. But this can create a lot of active objects in your frame, depending on the complexity of your environment. Too many active objects can slow down your game. But since *Static objects* don't move, this helps on the speed.

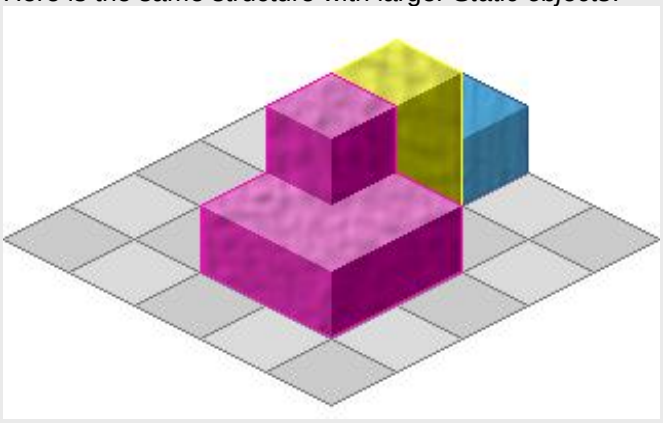
You could also make some larger *Static objects* that represent multiple adjacent collision cells. This will make larger objects but can lower the number of objects in the frame. These compound *Static objects* can still be copied at multiple places where they can fit with the *Collision array*. But all this is up to you and how you want you game to look like.

Here is a visual representation of a *Collision array*:



That one was simply done with a bunch of the same *Static object* cube. You get the idea of what the shape of the structure is like.

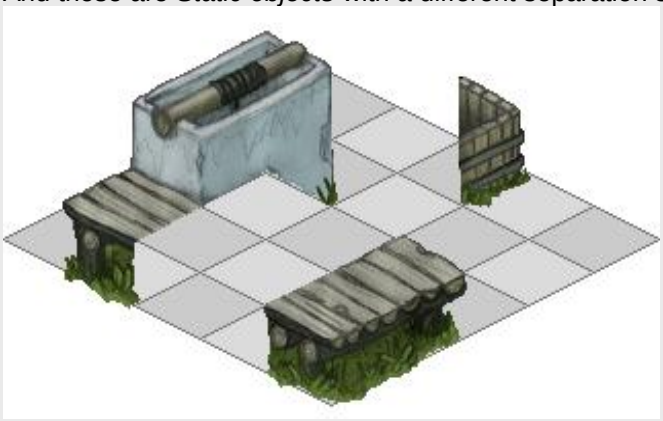
Here is the same structure with larger *Static objects*:



You can see three different *Static objects*, and two of them fit with multiple collision cells. The separation is done in the corners that aren't fully visible and where we can expect *Mover objects* to be partially hidden. Also, the large purple *Static object* has a convex shape. *Mover objects* can go into and intersect the general volume of the *Static object*. The rules of the layering process support this intersection only with *Static objects*.

With pretty hand drawn graphics:

And those are *Static objects* with a different separation setup:



Notice on how these last *Static objects* are a close fit, but not perfect. That's quite all right. Objects don't have to be flush with the edges of the collision's shape. They can have some pixels slightly outside or inside of the shape. Not too much though, or it could result in visible flickering with the layer sorting of *Mover objects*.

The most important thing to manage with *Static objects* is the precise placement of their action spot in MMF and their alterable values. You will need to set them yourself at edit time:
- The position of the action spot must be where the minimal 3D coordinates are located. This is also the 3D "origin" of the Static object's volume.
- The 3D position of the *Static object* is converted at runtime from its 2D position in the MMF frame. And for multiple instances of the same *Static object* placed at different elevations, some special markers can be placed over them to assign the proper Z elevation.
- You will also need to specify the *Static object*'s size in its alterable values. This size is the number of collision cells that the *Static object* occupies in X, Y and Z.

The Static object's hotspot at its minimal 3D coordinates:

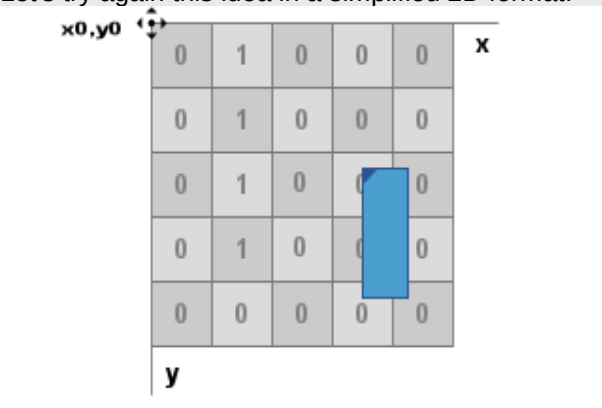
This object's size in cells is 2 in x, 2 in y and 3 in z.

If you misplace the hotspot, or the alterable values don't reflect the object's size properly, the layering process will fail.

**Objects that collide with a 3D array of data**

How can objects in the MMF frame collide with data from an array?
Well, *Mover objects* store data that represents their 3D position and volume. And I've mentioned how the data in the *Collision array* is like a large room filled with boxes where each box is a cell in the 3D array. So it's all a matter of comparing the values of *Mover objects* against the values placed in the *Collision array* during movement. Also, the movement process only manipulates 3D data that is converted to actual object positions after all movements are done.

Let's try again this idea in a simplified 2D format.

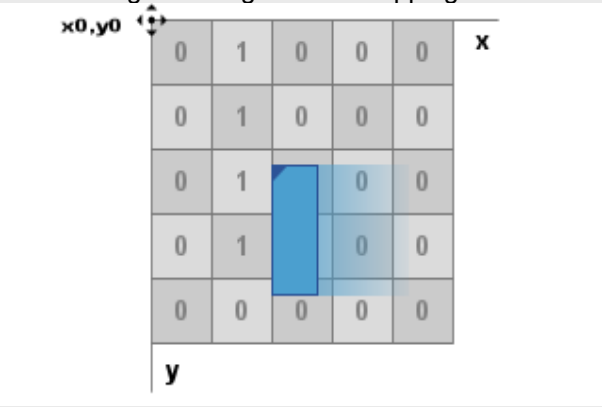



Here I have a 5x5 array that stores a value in each of its cells. Most of those values are 0, but four of them store 1. I also have a rectangular object over the array. The position of that object is defined by its top-left corner.

We want to translate that blue rectangle in a straight gradual movement towards the left, but have it stop as soon as it reaches the cells that contain the value 1. So those cells in the array will act as a wall.

Each step of the movement, we need to verify the content of the array at the left side of the rectangle. Since we move it in the X axis, we first need to know how many cells the rectangle occupies in Y. Looking at the array, we see that we will need to verify three cells at each step of the movement. If none of those cells contain 1 during the verification, then we can move the rectangle left for one step. If at least one cell is found to contain 1, than we leave the rectangle where it is and stop the movement process.

The rectangle moving left and stopping at the “1” cells:




The same kind of process is used for moving any *Mover object* along all three axis of the 3D space. Objects are given speed. Then a loop is started for the amount of speed. And at each step of the loop, the content of the array is verified. The 3D position values of the *Mover object* are changed or the loop is stopped, depending on the result of each scan of the array.

## Converting 3D position to 2D position

I've based my isometric engine on an important 2:1 ratio. A flat floor tile is 64x32 pixels in 2D size. A simple cube is 64x64 pixels in 2D size, and represents 3D cube of 16x16x16 units in volume. These are the fixed sizes of the cube cells in the *Collision array*, and they simplify the math that controls movements, collisions, positions and sizes.

But mostly, it simplifies a lot the conversion of 3D coordinates to the MMF 2D coordinates. Any other ratio I tried needed Sin() and Cos() calculations instead of the *2 and /2 I have now.

## The end

For any questions or comments, just write to redhades@yahoo.ca