

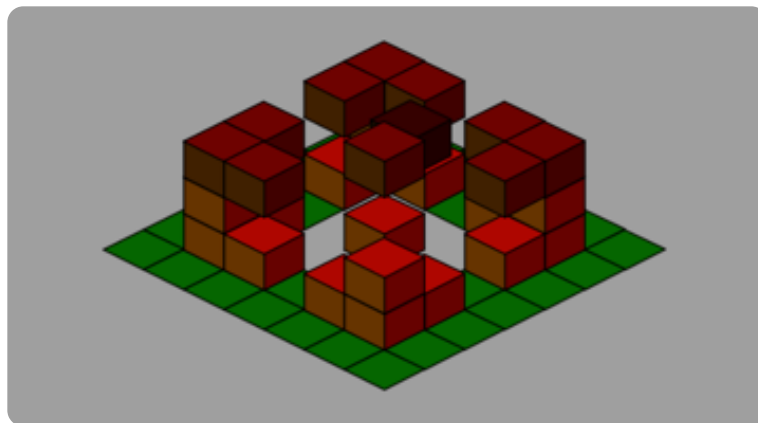


Game Development > Isometric

Isometric Depth Sorting for Moving Platforms

Juwal Bose Feb 1, 2018

🕒 11 mins | 💬 English



What You'll Be Creating

Depth sorting can be explained in simple terms as a way of figuring out which element is nearer to the camera and which is farther away, thereby determining the order in which they need to be arranged in order to convey the right depth in the scene.

In this tutorial, we will dig deeper into depth sorting for isometric levels as we try to add moving platforms. This is not a beginner tutorial on isometric theory and is not about the code. The focus is to understand the logic and theory rather than to dissect the code. The tool of choice for the tutorial is Unity, and hence depth sorting essentially is changing the

<https://gamedevelopment.tutsplus.com/tutorials/isometric-depth-sorting-for-moving-platforms--cms-30226>

tool of choice for the tutorial is Unity, and hence depth sorting essentially is changing the `sortingOrder` of the sprites involved. For other frameworks, it may be a changing of the z order or the sequence of drawing order.

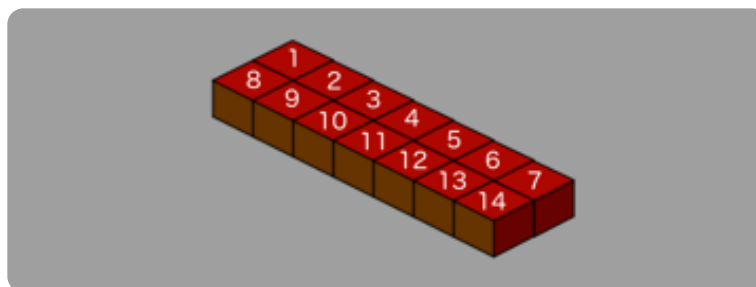
In order to get started on isometric theory, please refer to [this tutorial series](#). The code and scene structure follow my [previous isometric tutorial](#). Please refer to these if you find the tutorial hard to follow as I will be focusing only on logic in this tutorial.

1. Levels Without Movement

If your isometric level does not have any moving elements or just has a few characters walking over the level, the depth sorting is straightforward. In such cases, the characters occupying the isometric tiles would be smaller than the tiles themselves and can easily just use the same drawing order/depth as the tile they occupy.

Let's refer to such motionless levels as static levels. There are a few ways in which such levels can be drawn so that the right depth is conveyed. Typically, the level data will be a two-dimensional array where the rows and columns will correspond to the rows and columns of the level.

Consider the following isometric level with just two rows and seven columns.



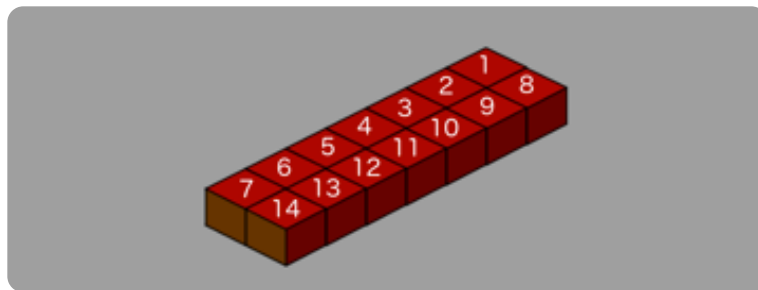
The numbers on the tiles indicate their `sortingOrder` or depth or z order, i.e. the order in which they need to be drawn. In this method, we are drawing all the columns in the first row, starting with the first column with a `sortingOrder` of 1.

row, starting with the first column with a `sortingOrder` of 1.

Once all columns are drawn in the first row, the nearest column to the camera has a `sortingOrder` of 7, and we proceed to the next row. So any element in the second row will have a higher `sortingOrder` than any element of the first row.

This is exactly how the tiles need to be arranged to convey the correct depth as a sprite with a higher `sortingOrder` will get overlaid over any other sprites with lower `sortingOrder`.

As for the code, this is just a matter of looping through the rows and columns of the level array and assigning `sortingOrder` sequentially in an increasing order. It would not break, even if we swap rows and columns, as can be seen in the image below.



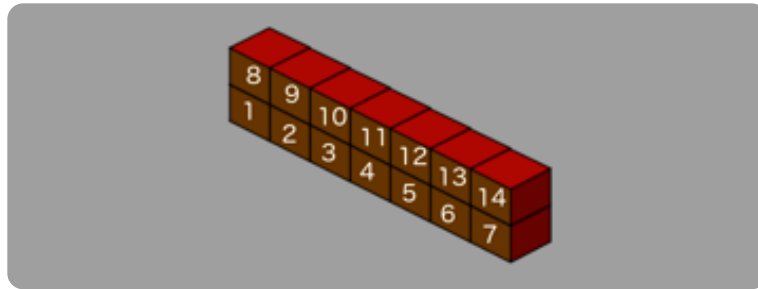
Here we draw a complete column first before moving to the next row. The depth perception stays intact. So the logic for a static level is to draw either a complete row or complete column and then proceed to the next while assigning `sortingOrder` sequentially in an increasing order.

Adding Height

If we consider the level as a building, we are currently drawing the ground floor. If we need to add a new floor to our building, all we need to do is to wait till we draw the whole ground floor first and follow the same method for the next floor.

For proper depth, we waited till the full row was complete before we moved to the next

row, and similarly we wait till all the rows are complete before we move to the next floor. So for a level with only a single row and two floors, it would look like the image below.



Essentially, any tile on the higher floor will have a higher `sortingOrder` than any tile on the lower floor. As for the code for adding higher floors, we just need to offset the `y` value of the screen coordinates for the tile, depending on which floor it occupies.

```
float floorHeight=tileSize/2.2f;
float currentFloorHeight=floorHeight*floorLevel;
//
tmpPos=GetScreenPointFromLevelIndices(i,j);
tmpPos.y+=currentFloorHeight;
tile.transform.position=tmpPos;
```

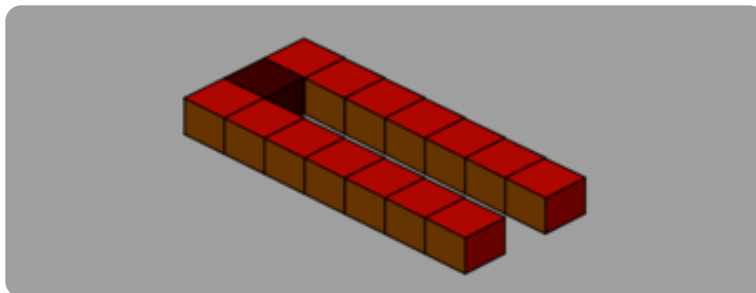
The `floorHeight` value indicates the perceived height of the isometric block tile image, whereas `floorLevel` indicates which floor the tile occupies.

Advertisement

2. Moving Tiles on the X Axis

Depth sorting on a static isometric level was not complicated, right? Moving on, let us decide to follow the row first method, where we assign `sortingOrder` to the first row completely and then proceed to the next. Let's consider our first moving tile or platform which moves on a single axis, the x axis.

When I say that the motion is on the x axis, you need to realize that we are talking about the cartesian coordinate system and not the isometric coordinate system. Let's consider a level with only a ground floor of three rows and seven columns. Let's also consider that the second row only has a single tile, which is our moving tile. The level will look like the image below.

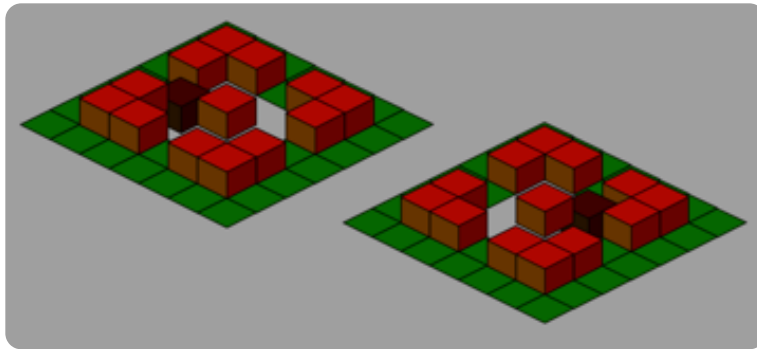


The dark tile is our moving tile, and the `sortingOrder` it would get assigned will be 8 as the first row has 7 tiles. If the tile moves on the cartesian x axis then it will move along the trench between the two rows. At all of the positions it may occupy along that path, the tiles in row 1 will have a lesser `sortingOrder`.

Similarly, all the tiles in row 2 will have a higher `sortingOrder`, irrespective of the position of the dark tile along said path. So as we follow a row first method of assigning `sortingOrder`, we do not need to do anything for motion on the x axis. Now, that was easy.

3. Moving Tiles on the Y Axis

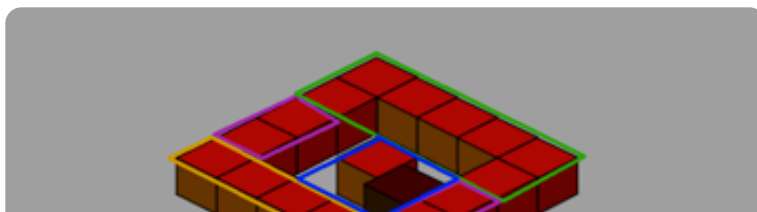
Problems start to arise when we start considering the y axis. Let's consider a level in which our dark tile is moving along a rectangular trench, as shown below. You can see the same in the `MovingSortingProblem` Unity scene in the source.

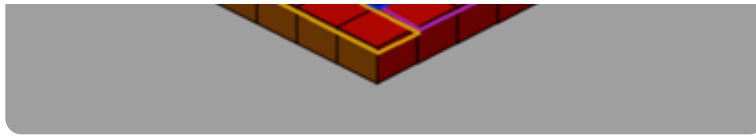


Using our row first approach, we can provide a `sortingOrder` for our moving tile based on the row it currently occupies. When the tile is between two rows, it would get assigned a `sortingOrder` based on the row it is moving from. In that case, it cannot follow the sequential `sortingOrder` in the row into which it is moving. This essentially breaks our depth sorting approach.

Sorting in Blocks

In order to solve this, we need to divide our level into different blocks, among which one is the problem block, which breaks under our row first approach, and the rest are blocks which can follow the row first approach without breaking. Consider the image below for a better understanding.



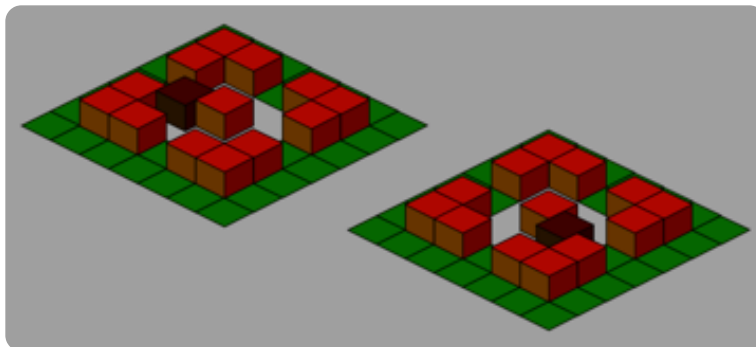


The 2x2 tile block represented by the blue area is our problem block. All the other blocks can still follow the row first approach. Please do not be confused by the image as it shows a level which is already properly sorted using our block approach. The blue block consists of the two column tiles in the rows between which our dark tile is currently moving and the tiles immediately to the left of them.

In order to solve the depth issue for the problem block, we can use the column first approach for this block alone. So for the green, pink, and yellow blocks, we use row first, and for the blue block, we use the column first approach.

Notice that we still need to sequentially assign the `sortingOrder`. First the green block, then the pink block to the left, then the blue block, now comes the pink block to the right, and finally the yellow block. We break the order only to switch to the column first approach while at the blue block.

Alternatively, we can also consider the 2x2 block to the right of the moving tile column. (The interesting thing is, you do not even need to switch approaches as breaking into blocks itself has already solved our problem in this case.) The solution can be seen in action in the `BlockSort` scene.



THIS TRANSLATES TO CODE AS BELOW.

```
private void DepthSort(){
    Vector2 movingTilePos=GetLevelIndicesFromScreenPoint(movingGO.transfo

int blockColStart=(int)movingTilePos.y;
    int blockRowStart=(int)movingTilePos.x;
    int depth=1;

    //sort rows before block
    for (int i = 0; i < blockRowStart; i++) {
        for (int j = 0; j < cols; j++) {
            depth=AssignDepth(i,j,depth);
        }
    }
    //sort columns in same row before the block
    for (int i = blockRowStart; i < blockRowStart+2; i++) {
        for (int j = 0; j < blockColStart; j++) {
            depth=AssignDepth(i,j,depth);
        }
    }
    //sort block
    for (int i = blockRowStart; i < blockRowStart+2; i++) {
        for (int j = blockColStart; j < blockColStart+2; j++) {
            if(movingTilePos.x==i&&movingTilePos.y==j){
                SpriteRenderer sr=movingGO.GetComponent<S
                sr.sortingOrder=depth;//assign new depth
                depth++;//increment depth
            }else{
                depth=AssignDepth(i,j,depth);
            }
        }
    }
    //sort columns in same row after the block
    for (int i = blockRowStart; i < blockRowStart+2; i++) {
        for (int j = blockColStart+2; j < cols; j++) {
            depth=AssignDepth(i,i,depth);
```



```

        depth=AssignDepth(i,j,depth);
    }
}
//sort rows after block
for (int i = blockRowStart+2; i < rows; i++) {
    for (int j = 0; j < cols; j++) {
        depth=AssignDepth(i,j,depth);
    }
}
}

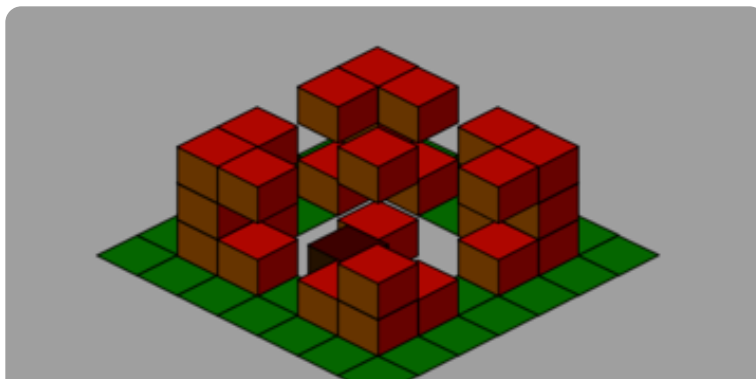
```

4. Moving Tiles on the Z Axis

A motion in the z axis is a fake motion on an isometric level. It essentially is just motion on the screen y axis. For a single-floor isometric level, there is nothing more to do in order to add motion on the z axis if you have already done the block sorting method described above. You can see this in action in the `SingleLayerWave` Unity scene, where I have added an additional wave motion on the z axis along with the lateral trench movement.

Z Movement on Levels With Multiple Floors

Adding an additional floor to your level is just a matter of offsetting the screen y coordinate, as explained before. If the tile does not move on the z axis then there is no need to do anything special for depth sorting. We can block sort the ground floor with motion and then apply row first sorting to each successive floor. You can see this in action in the `BlockSortWithHeight` Unity scene.



A very similar depth problem arises when the tile starts moving between floors. It can only satisfy the sequential order of one floor using our approach and would break the depth sorting of the other floor. We need to extend or modify our block sorting to three dimensions to deal with this depth problem with floors.

The problem essentially will be just the two floors between which the tile is currently moving. For all other floors, we can stick to our current sorting approach. Special needs apply to only these two floors, among which we can first determine the lower floor as below where `tileZOffset` is the amount of motion on the z axis for our moving tile.

```
float whichFloor=(tileZOffset/floorHeight);
float lower=Mathf.Floor(whichFloor);
```

This means that `lower` and `lower+1` are the floors which need a special approach. The trick is to assign `sortingOrder` for both these floors together, as shown in the code below. This fixes the sequence so that the depth issues are sorted out.

```
if(floor==lower){
    // we need to sort lower floor and the floor just above it together i
    depth=(floor*(rows*cols))+1;
    int nextFloor=floor+1;
    if(nextFloor>=totalFloors)nextFloor=floor;
    //sort rows before block
    for (int i = 0; i < blockRowStart; i++) {
        for (int j = 0; j < cols; j++) {
            depth=AssignDepth(i,j,depth,floor);
            depth=AssignDepth(i,j,depth,nextFloor);
        }
    }
    //sort columns in same row before the block
```

```

    for (int i = blockRowStart; i < blockRowStart+2; i++) {
        for (int j = 0; j < blockColStart; j++) {
            depth=AssignDepth(i,j,depth,floor);
            depth=AssignDepth(i,j,depth,nextFloor);

        }
    }
    //sort block

    for (int i = blockRowStart; i < blockRowStart+2; i++) {
        for (int j = blockColStart; j < blockColStart+2; j++) {
            if(movingTilePos.x==i&&movingTilePos.y==j){
                SpriteRenderer sr=movingGO.GetComponent<SpriteRen
                sr.sortingOrder=depth;//assign new depth
                depth++;//increment depth
            }else{
                depth=AssignDepth(i,j,depth,floor);
                depth=AssignDepth(i,j,depth,nextFloor);
            }
        }
    }

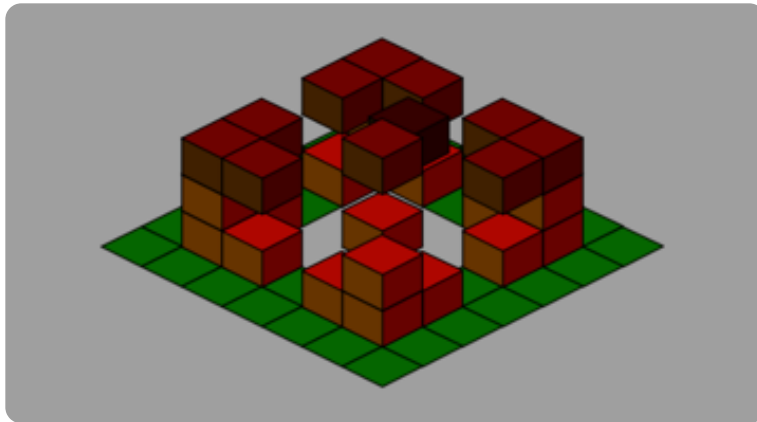
    //sort columns in same row after the block
    for (int i = blockRowStart; i < blockRowStart+2; i++) {
        for (int j = blockColStart+2; j < cols; j++) {
            depth=AssignDepth(i,j,depth,floor);
            depth=AssignDepth(i,j,depth,nextFloor);
        }
    }
    //sort rows after block
    for (int i = blockRowStart+2; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            depth=AssignDepth(i,j,depth,floor);
            depth=AssignDepth(i,j,depth,nextFloor);
        }
    }
}

```

J

Essentially, we are considering two floors as a single floor and doing a block sort on that single floor. Check out the code and action in the scene `BlockSortWithHeightMovement`. With

this approach, our tile is now free to move on any of the two axes without breaking the depth of the scene, as shown below.



Advertisement

Conclusion

The idea of this tutorial was to clarify the logic of the depth sorting approaches, and I

hope you have fairly understood this. It is evident that we are considering comparatively simple levels with only one moving tile.

There are no slopes either as including slopes would have made this a much longer tutorial. But once you have understood the sorting logic, then you can try to extend the two-dimensional slope logic to the isometric view.

Unity has an active economy. There are many other products that help you build out your project. The nature of the platform also makes it a great option from which you can better your skills. Whatever the case, you can see what we have available [in the Envato Market](#).

Advertisement

Isometric

Unity

Unity 2D

2D Games

Did you find this post useful?



Yes



No

Want a weekly email summary?

Subscribe below and we'll send you a weekly email summary of all new Game Development tutorials.
Never miss out on learning about the next big thing.

Sign up



Juwal Bose

Creative soul, Game Developer & Creative Head at Csharks Games & Solutions. R&D team lead for all game development verticals including Web, iOS & Android. Uses Cocos2D for iOS, LibGDX for Android, Flash AS3 for Web & Unity for 3D development. Very active in social media tech circles & active participant in tech events. Likes to read, watch & travel. On [Google+](#) and [Twitter](#).

 FEED  LIKE  FOLLOW

[View on GitHub](#)

Advertisement

QUICK LINKS - Explore popular categories

ENVATO TUTORIALS


+

JOIN OUR COMMUNITY

+

HELP

+

 **tuts+**
30,351 Tutorials 1,316 Courses 50,287 Translations



[Envato](#) [Envato Elements](#) [Envato Market](#) [Placeit by Envato](#) [Milkshake](#) [All products](#) [Careers](#) [Sitemap](#)

© 2022 Envato Pty Ltd. Trademarks and brands are the property of their respective owners.

