Game Development > Phaser

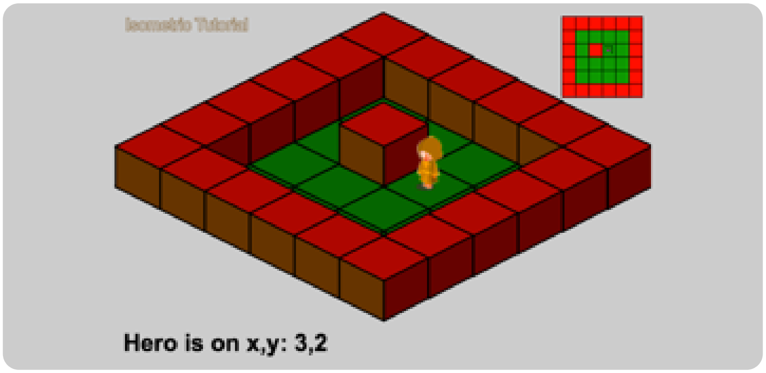
An Updated Primer for Creating Isometric Worlds, Part 1

Juwal Bose May 11, 2017

16 mins | English

This post is part of a series called **Primer for Creating Isometric Worlds**.

Updated Primer for Creating Isometric Worlds, Part 2



What You'll Be Creating

We have all played our fair share of amazing *isometric games*, be it the original Diablo, or Age of Empires or Commandos. The first time you came across an isometric game, you may have wondered if it was a *2D game* or a *3D game* or something completely different. The world of isometric games has its mystical attraction for game developers as well. Let us try to unravel the mystery of isometric projection and try to create a simple isometric world in this tutorial.

This tutorial is an updated version of my **existing tutorial on creating isometric worlds**. The original tutorial used Flash with ActionScript and is still relevant for Flash or **OpenFL** developers. In this new tutorial I have decided to use **Phaser** with JS code, thereby creating interactive HTML5 output instead of SWF output.

Please be advised that this is not a Phaser development tutorial, but we are just using Phaser to easily communicate the core concepts of creating an isometric scene. Besides, there are much better and easier ways to create isometric content in Phaser, such as the **Phaser Isometric Plugin**.

For the sake of simplicity, we will use the tile-based approach to create our isometric scene.

1. Tile-Based Games

In 2D games using the tile-based approach, each visual element is broken down into smaller pieces, called tiles, of a standard size. These tiles will be arranged to form the game world according to pre-determined level data—usually a two-dimensional array.

Related Posts

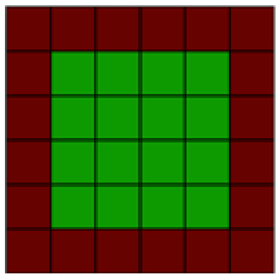
- **Tony Pa's tile-based tutorials**

Usually tile-based games use either a *top-down* view or a *side view* for the game scene. Let us consider a standard top-down 2D view with two tiles—a *grass tile* and a *wall tile*—as shown here:

Both of these tiles are square images of the same size, hence the *tile height* and *tile width* are the same. Let us consider a game level which is a grassland enclosed on all sides by walls. In such a case, the level data represented with a two-dimensional array will look like this:

```
[
 [1,1,1,1,1,1],
 [1,0,0,0,0,1],
 [1,0,0,0,0,1],
 [1,0,0,0,0,1],
 [1,0,0,0,0,1],
 [1,1,1,1,1,1]
 ]
```

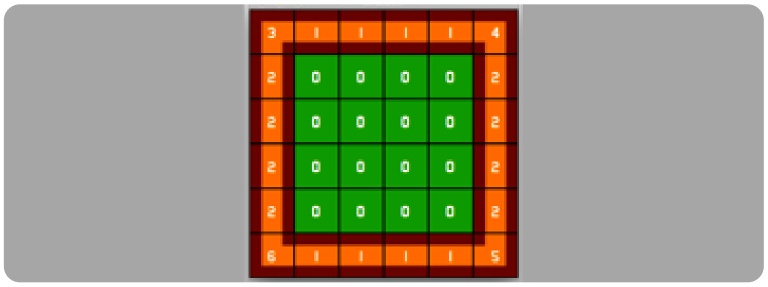
Here, `0` denotes a grass tile and `1` denotes a wall tile. Arranging the tiles according to the level data will produce our walled grassland as shown in the image below:



We can go a bit further by adding corner tiles and separate vertical and horizontal wall tiles, requiring five additional tiles, which leads us to our updated level data:

```
[
 [3,1,1,1,1,4],
 [2,0,0,0,0,2],
 [2,0,0,0,0,2],
 [2,0,0,0,0,2],
 [2,0,0,0,0,2],
 [6,1,1,1,1,5]
]
```

Check out the image below, where I have marked the tiles with their corresponding tile numbers in the level data:



Now that we have understood the concept of the tile-based approach, let me show you how we can use a straightforward 2D grid pseudo code to render our level:

```
for (i, loop through rows)
    for (j, loop through columns)
        x = j * tile width
        y = i * tile height
        tileType = levelData[i][j]
        placetile(tileType, x, y)
```

If we use the above tile images then the tile width and tile height are equal (and the same for all tiles), and will match the tile images' dimensions. So the tile width and tile height for this example are both 50 px, which makes up the total level size of 300 x 300 px—that is, six rows and six columns of tiles measuring 50 x 50 px each.

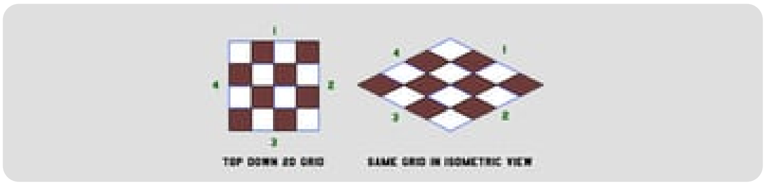
As discussed earlier, in a normal tile-based approach, we either implement a top-down view or a side view; for an isometric view, we need to implement the *isometric projection*.

2. Isometric Projection

The best technical explanation of what *isometric projection* means, as far as I'm aware, is from **this article by Clint Bellanger**:

> *We angle our camera along two axes (swing the camera 45 degrees to one side, then 30 degrees down). This creates a diamond (rhombus) shaped grid where the grid spaces are twice as wide as they are tall. This style was popularized by strategy games and action RPGs. If we look at a cube in this view, three sides are visible (top and two facing sides).*

Although it sounds a bit complicated, actually implementing this view is very easy. What we need to understand is the relation between 2D space and the isometric space—that is, the relation between the level data and the view; the transformation from top-down *Cartesian* coordinates to isometric coordinates. The image below shows the visual transformation:



Placing Isometric Tiles

Let me try to simplify the relationship between level data stored as a 2D array and the isometric view—that is, how we transform Cartesian coordinates into isometric coordinates. We will try to create the isometric view for our now-famous walled grassland. The 2D view implementation of the level was a straightforward iteration with two loops, placing square tiles offsetting each with the fixed tile height and tile width values. For the isometric view, the pseudo code remains the same, but the `placeTile()` function changes.

The original function just draws the tile images at the provided coordinates `x` and `y`, but for an isometric view we need to calculate the corresponding isometric coordinates. The equations to do this are as follows, where `isoX` and `isoY` represent isometric x- and y-coordinates, and `cartX` and `cartY` represent Cartesian x- and y-coordinates:

```
//Cartesian to isometric:

isoX = cartX - cartY;
isoY = (cartX + cartY) / 2;
```

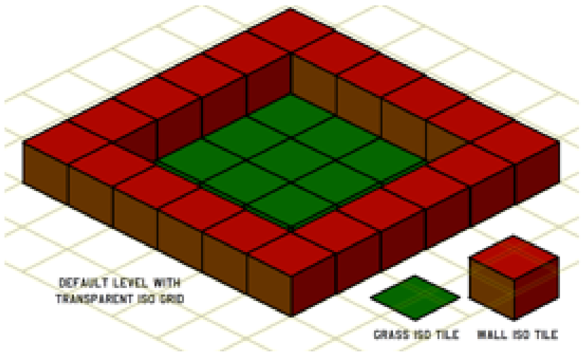
```
//Isometric to Cartesian:

cartX = (2 * isoY + isoX) / 2;
cartY = (2 * isoY - isoX) / 2;
```

Yes, that is it. These simple equations are the magic behind isometric projection. Here are Phaser helper functions which can be used to convert from one system to another using the very convenient `Point` class:

```
function cartesianToIsometric(cartPt){
    var tempPt=new Phaser.Point();
    tempPt.x=cartPt.x-cartPt.y;
    tempPt.y=(cartPt.x+cartPt.y)/2;
    return (tempPt);
}
```

```
function isometricToCartesian(isoPt){
    var tempPt=new Phaser.Point();
    tempPt.x=(2*isoPt.y+isoPt.x)/2;
    tempPt.y=(2*isoPt.y-isoPt.x)/2;
    return (tempPt);
}
```

So we can use the `cartesianToIsometric` helper method to convert the incoming 2D coordinates into isometric coordinates inside the `placeTile` method. Apart from this, the rendering code remains the same, but we need to have new images for the tiles. We cannot use the old square tiles used for our top-down rendering. The image below shows the new isometric grass and wall tiles along with the rendered isometric level:



Unbelievable, isn't it? Let's see how a typical 2D position gets converted to an isometric position:

```
2D point = [100, 100];
// isometric point will be calculated as below
isoX = 100 - 100; // = 0
isoY = (100 + 100) / 2;  // = 100
Iso point == [0, 100];
```

Similarly, an input of `[0, 0]` will result in `[0, 0]`, and `[10, 5]` will give `[5, 7.5]`.

For our walled grassland, we can determine a walkable area by checking whether the array element is `0` at that coordinate, thereby indicating grass. For this we need to determine the array coordinates. We can find the tile's coordinates in the level data from its Cartesian coordinates using this function:

```
function getTileCoordinates(cartPt, tileHeight){
    var tempPt=new Phaser.Point();
    tempPt.x=Math.floor(cartPt.x/tileHeight);
    tempPt.y=Math.floor(cartPt.y/tileHeight);
    return(tempPt);
}
```

(Here, we essentially assume that tile height and tile width are equal, as in most cases.)

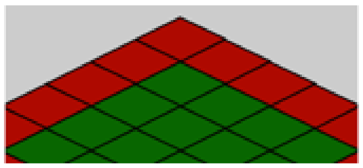
Hence, from a pair of screen (isometric) coordinates, we can find tile coordinates by calling:

```
getTileCoordinates(isometricToCartesian(screen point), tile height);
```

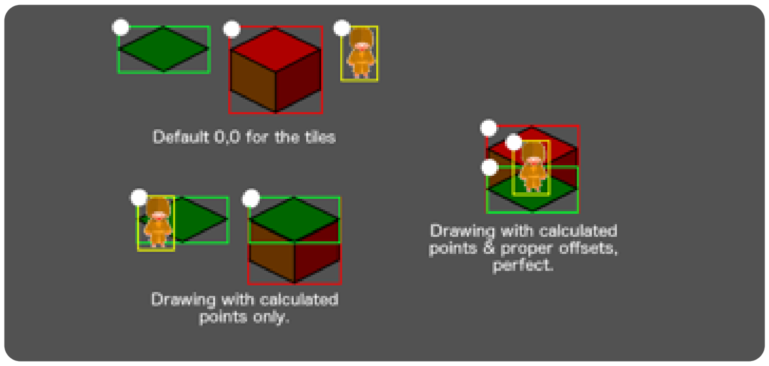
This screen point could be, say, a mouse click position or a pick-up position.

Registration Points

In Flash, we could set arbitrary points for a graphic as its centre point or `[0,0]`. The Phaser equivalent is `Pivot`. When you place the graphic at say `[10,20]`, then this `Pivot` point will get aligned with `[10,20]`. By default, the top left corner of a graphic is considered its `[0,0]` or `Pivot`. If you try to create the above level using the code provided, then you will not get the displayed result. Instead, you will get a flat land without the walls, like below:



This is because the tile images are of different sizes and we are not addressing the height attribute of the wall tile. The below image shows the different tile images that we use with their bounding boxes and a white circle where their default [0,0] is:


Default 0,0 for the tiles
Drawing with calculated points only.
Drawing with calculated points & proper offsets, perfect.

See how the hero gets misaligned when drawing using the default pivots. Also notice how we lose the height of the wall tile if drawn using default pivots. The image on the right shows how they need to be properly aligned so that the wall tile gets its height and the hero gets placed in the middle of the grass tile. This issue can be solved in different ways.

1. Make all tiles in the same image size with the graphic aligned properly within the image. This creates a lot of empty areas within each tile graphic.
2. Set pivot points manually for each tile so that they align properly.
3. Draw tiles with specific offsets so that they align properly.

For this tutorial, I have chosen to use the third method so that this works even with a framework without the ability to set pivot points.

3. Moving in Isometric Coordinates

We will never try to move our character or projectile in isometric coordinates directly. Instead, we will manipulate our game world data in Cartesian coordinates and just use the above functions for updating those on the screen. For example, if you want to move a character forward in the positive y-direction, you can simply increment its `y` property in 2D coordinates and then convert the resulting position to isometric coordinates:

```
y = y + speed;
placetile(cartesianToIsometric(new Phaser.Point(x, y)))
```

This will be a good time to review all the new concepts that we have learned so far and to try and create a working example of something moving in an isometric world. You can find the necessary image assets in the `assets` folder of the source git repository.

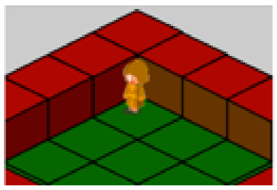
Depth Sorting

If you tried to move the ball image in our walled garden then you would come across the problems with *depth sorting*. In addition to normal placement, we will need to take care of *depth sorting* for drawing the isometric world, if there are moving elements. Proper depth sorting makes sure that items closer to the screen are drawn on top of items farther away.

The simplest depth sorting method is simply to use the Cartesian y-coordinate value, as mentioned in **this Quick Tip**: the further up the screen the object is, the earlier it should be drawn. This may work well for very simple isometric scenes, but a better way will be to redraw the isometric scene once a movement happens, according to the tile's array coordinates. Let me explain this concept in detail with our pseudo code for level drawing:

```
for (i, loop through rows)
    for (j, loop through columns)
        x = j * tile width
        y = i * tile height
        tileType = levelData[i][j]
        placetile(tileType, x, y)
```

Imagine our item or character is on the tile `[1,1]`—that is, the topmost green tile in the isometric view. In order to properly draw the level, the character needs to be drawn after drawing the corner wall tile, both the left and right wall tiles, and the ground tile, like below:



If we follow our draw loop as per the pseudo code above, we will draw the middle corner wall first, and then will continue to draw all the walls in the top right section until it reaches the right corner.

Then, in the next loop, it will draw the wall on the left of the character, and then the grass tile on which the character is standing. Once we determine this is the tile which occupies our character, we will draw the character *after* drawing the grass tile. This way, if there were walls on the three free grass tiles connected to the one on which the character is standing, those walls will overlap the character, resulting in proper depth sorted rendering.

4. Creating the Art

Isometric art can be pixel art, but it doesn't have to be. When dealing with isometric pixel art, **RhysD's guide** tells you almost everything you need to know. Some theory can be found **on Wikipedia** as well.

When creating isometric art, the general rules are:

- Start with a blank isometric grid and adhere to pixel-perfect precision.

- Try to break art into single isometric tile images.
- Try to make sure that each tile is either *walkable* or *non-walkable*. It will be complicated if we need to accommodate a single tile that contains both walkable and non-walkable areas.
- Most tiles will need to seamlessly tile in one or more directions.
- Shadows can be tricky to implement, unless we use a layered approach where we draw shadows on the ground layer and then draw the hero (or trees, or other objects) on the top layer. If the approach you use is not multi-layered, make sure shadows fall to the front so that they won't fall on, say, the hero when he stands behind a tree.
- In case you need to use a tile image larger than the standard isometric tile size, try to use a dimension which is a multiple of the iso tile size. It is better to have a layered approach in such cases, where we can split the art into different pieces based on its height. For example, a tree can be split into three pieces: the root, the trunk, and the foliage. This makes it easier to sort depths as we can draw pieces in corresponding layers which correspond with their heights.

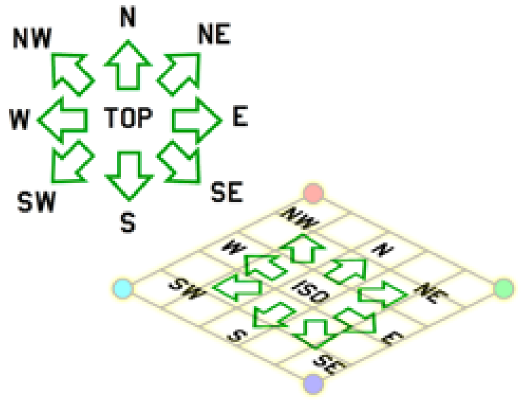
Isometric tiles that are larger than the single tile dimensions will create issues with depth sorting. Some of the issues are discussed in these links:

Related Posts

- Bigger tiles
- Splitting and Painter's algorithm
- OpenSpace's post on effective ways of splitting up larger tiles

5. Isometric Characters

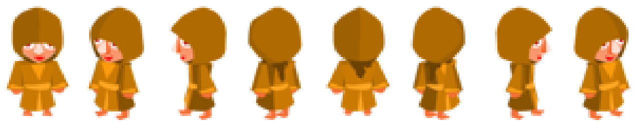
First we will need to fix how many directions of motion are permitted in our game—usually, games will provide four-way movement or eight-way movement. Check out the image below to understand the correlation between the 2D space and isometric space:



Please note that a character would be moving vertically up when we press the **up arrow** key in a top-down game, but for an isometric game the character will move at a 45-degree angle towards the top right corner.

For a top-down view, we could create one set of character animations facing in one direction, and simply rotate them for all the others. For isometric character art, we need to re-render each animation in each of the permitted directions—so for eight-way motion, we need to create eight animations for each action.

For ease of understanding, we usually denote the directions as North, North-West, West, South-West, South, South-East, East, and North-East. The character frames below show idle frames starting from South-East and going clockwise:



We will place characters in the same way that we placed tiles. The movement of a character is accomplished by calculating the movement in Cartesian coordinates and then converting to isometric coordinates. Let us assume we are using the keyboard to control the character.

We will set two variables, `dX` and `dY`, based on the directional keys pressed. By default, these variables will be `0` and will be updated as per the chart below, where `U`, `D`, `R`, and `L` denote the **Up**, **Down**, **Right**, and **Left** arrow keys, respectively. A value of `1` under a key represents that the key is being pressed; `0` implies that the key is not being pressed.

```
  Key       Pos
U D R L    dX dY
================
0 0 0 0     0  0
1 0 0 0     0  1
0 1 0 0     0 -1
0 0 1 0     1  0
0 0 0 1    -1  0
1 0 1 0     1  1
1 0 0 1    -1  1
0 1 1 0     1 -1
0 1 0 1    -1 -1
```

Now, using the values of `dX` and `dY`, we can update the Cartesian coordinates like so:

```
newX = currentX + (dX * speed);
newY = currentY + (dY * speed);
```

So `dX` and `dY` stand for the change in the x- and y-positions of the character, based on the keys pressed. We can easily calculate the new isometric coordinates, as we've already discussed:

```
Iso = cartesianToIsometric(new Phaser.Point(newX, newY))
```

Once we have the new isometric position, we need to *move* the character to this position. Based on the values we have for `dX` and `dY`, we can decide which direction the character is facing and use the corresponding character art. Once the character is moved, please don't forget to repaint the level with the proper depth sorting as the tile coordinates of the character may have changed.

Collision Detection

Collision detection is done by checking whether the tile at the newly calculated position is a non-walkable tile. So, once we find the new position, we don't immediately move the character there, but first check to see what tile occupies that space.

```
tile coordinate = getTileCoordinates(isometricToCartesian(current position), tile height
if (isWalkable(tile coordinate)) {
  moveCharacter();
} else {
  //do nothing;
}
```

In the function `isWalkable()`, we check whether the level data array value at the given coordinate is a walkable tile or not. We must take care to update the direction in which the character is facing—*even if he does not move*, as in the case of him hitting a non-walkable tile.

Now this may sound like a proper solution, but it will only work for items without volume. This is because we are only considering a single point, which is the midpoint of the character, to calculate collision. What we really need to do is to find all the four corners of the character from its available 2D midpoint coordinate and calculate collisions for all of those. If any corner is falling inside a non-walkable tile, then we should not move the character.

Depth Sorting With Characters

Consider a character and a tree tile in the isometric world, and they *both have the same image sizes*, however unrealistic that sounds.

To properly understand depth sorting, we must understand that whenever the character's x- and y-coordinates are less than those of the tree, the tree overlaps the character. Whenever the character's x- and y-coordinates are greater than that of the tree, the character overlaps the tree. When they have the same x-coordinate, then we decide based on the y-coordinate alone: whichever has the higher y-coordinate overlaps the other. When they have the same y-coordinate then we decide based on the x-coordinate alone: whichever has the higher x-coordinate overlaps the other.

As explained earlier, a simplified version of this is to just sequentially draw the levels starting from the farthest tile—that is, `tile[0][0]`—and then draw all the tiles in each row one by one. If a character occupies a tile, we draw the ground tile first and then render the character tile. This will work fine, because the character cannot occupy a wall tile.

6. Demo Time!

This is a demo in Phaser. Click to focus on the interactive area and use your arrow keys to move the character. You may use two arrow keys to move in the diagonal directions.

You can find the complete source for the demo in the source repository for this tutorial.

Phaser Isometric 2D Games Tile-Based Games Game Development

Did you find this post useful?

Yes No

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.

View Online Demo

View on GitHub