



GAME DEVELOPMENT > TILE-BASED GAMES

How to Use Tile Bitmasking to Auto-Tile Your Level Layouts

by [Sonny Bone](#) 3 Feb 2016

Difficulty: Intermediate Length: Long Languages: English ▼

Tile-Based Games

Pixel Art

Game Art

Spritesheets

Programming

Algorithms

Bitmasking

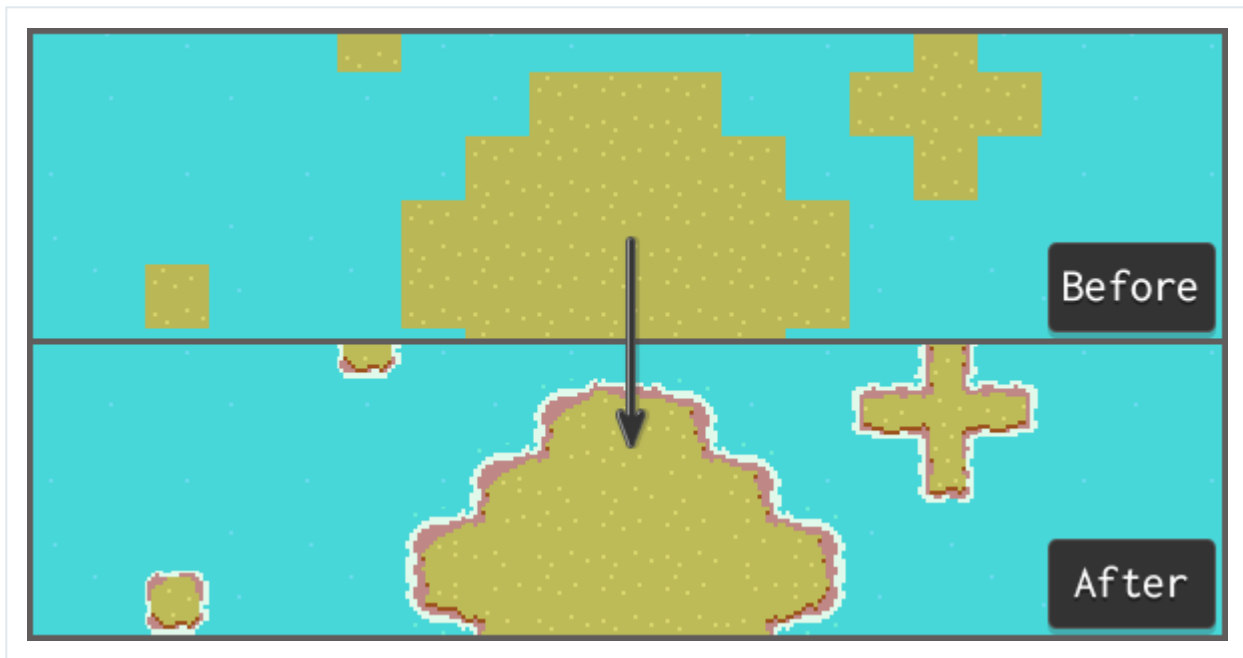
2D Games



Crafting a visually appealing and varied tileset is a time consuming process, but the results are often worth it. However, even after creating the art, you still have to piece it all together within your level!

You can place each tile, one by one, by hand—or, you can automate the process by using *bitmasking*, so you only need to draw the shape of the terrain.

What is Tile Bitmasking?



Tile bitmasking is a method for automatically selecting the appropriate sprite from a defined tileset. This allows you to place a generic placeholder tile everywhere you want a particular type of terrain to appear instead of hand placing a potentially enormous selection of various tiles.

See this video for a demonstration:

Tile Bitmasking: An Auto-Tiling Solution for Level Design



(You can [download the demos and source files from the GitHub repo.](#))

When dealing with multiple types of terrain, the number of different variations can exceed 300 or more tiles. Drawing this many different sprites is definitely a time-consuming process, but tile bitmasking ensures that the act of placing these tiles is quick and efficient.

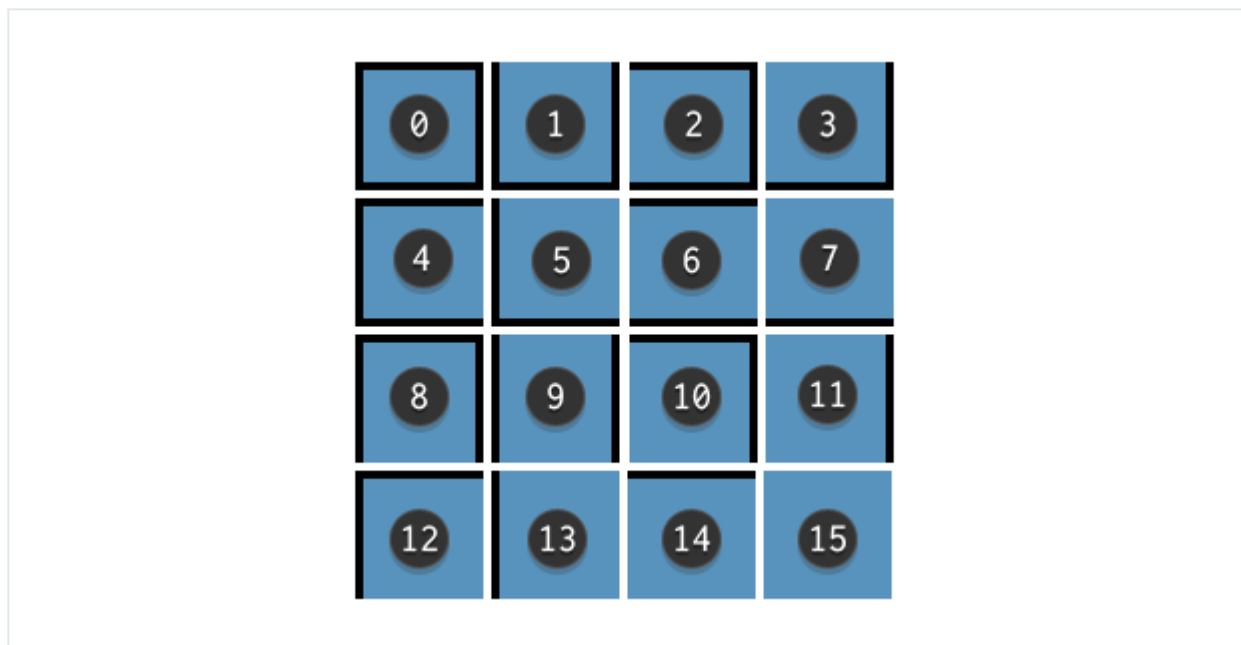
With a static implementation of bitmasking, maps are generated at runtime. With a few small tweaks, you can expand bitmasking to allow for dynamic tiles that change during gameplay. In this tutorial, we will cover the basics of tile bitmasking while working our way towards more complicated implementations that use corner tiles and multiple terrain types.

How Tile Bitmasking Works

Overview

Tile bitmasking is all about calculating a numerical value and assigning a specific sprite based on that value. Each tile looks at its neighboring tiles to determine which sprite from the set to assign to itself.

Every sprite in a tileset is numbered, and the bitmasking process returns a number corresponding to the position of a sprite in the tileset. At runtime, the bitmasking procedure is performed, and every tile is updated with the appropriate sprite.



The sprite sheet above consists of terrain tiles with all of the possible border configurations. The numbers on each tile represent the bitmasking value, which we will learn how to calculate in the next section. For now, it's important to understand how the bitmasking value relates to the terrain tileset. The sprites are ordered sequentially so that a bitmasking value of `0` returns the first sprite, all the way to a value of `15` which returns the 16th sprite.

Calculating the Bitmasking Value

Calculating this value is relatively simple. In this example, we are assuming a single terrain type with no corner pieces.

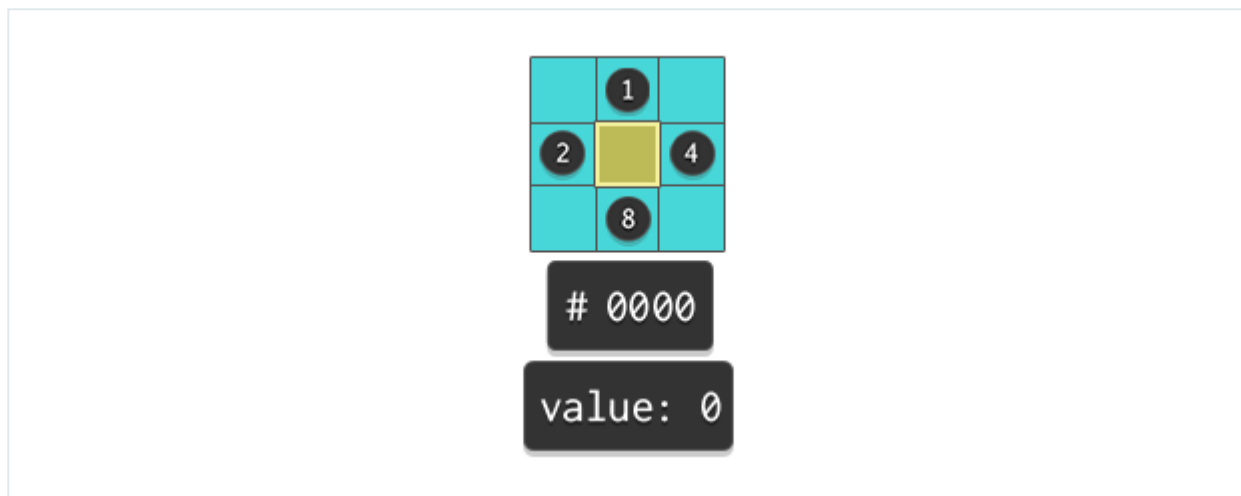
Each tile checks for the existence of tiles to the North, West, East, and South, and each check returns a Boolean, where `0` represents an empty space and `1` signifies the presence of another terrain tile.

This Boolean result is then multiplied by the binary directional value and added to the running total of the bitmasking value—it's easier to understand with some examples:

4-bit Directional Values

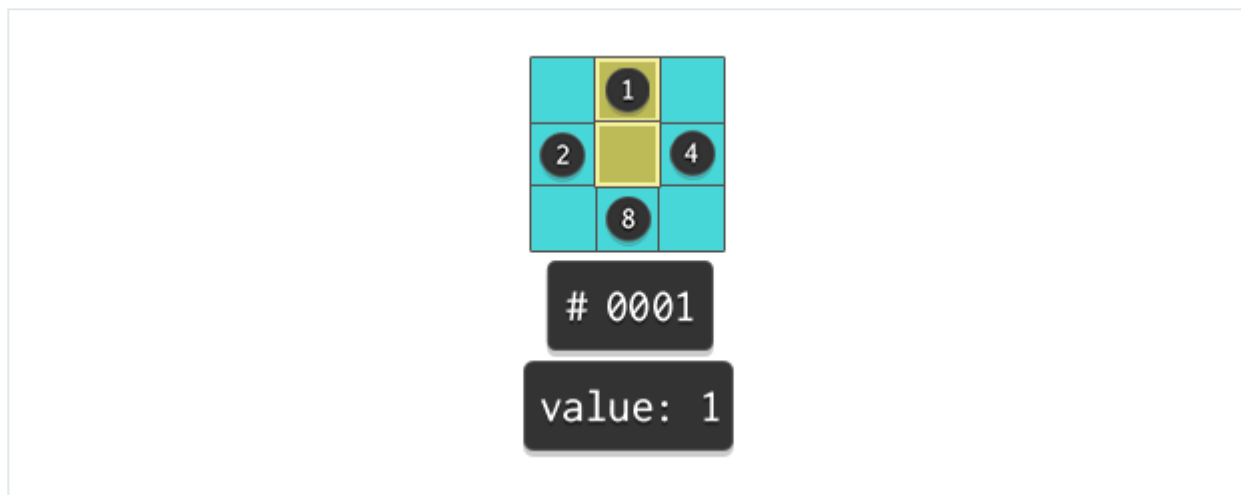
- North = $2^0 = 1$
- West = $2^1 = 2$

- East = $2^2 = 4$
- South = $2^3 = 8$

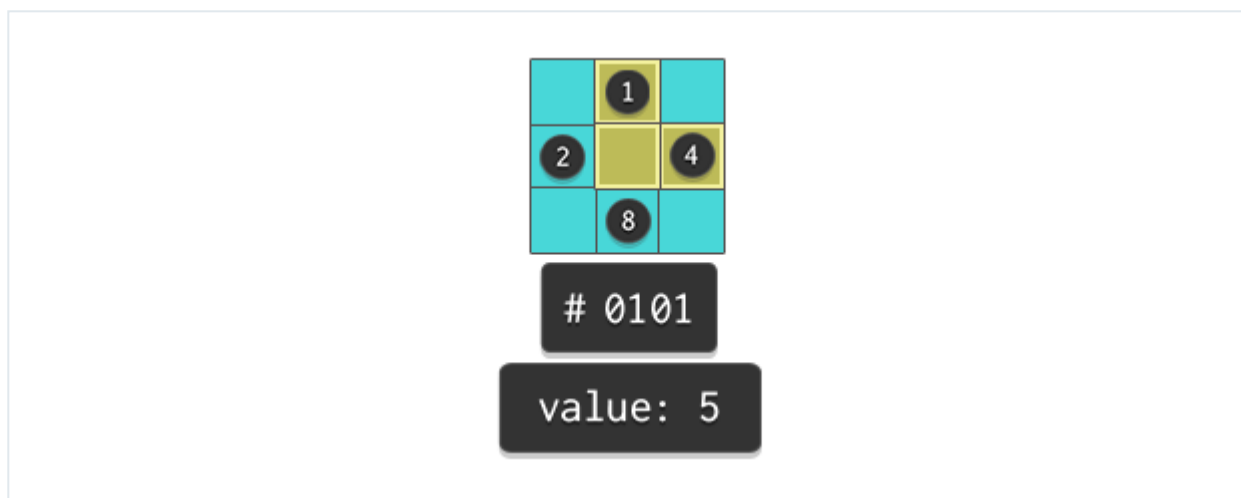


The green square in the figure above represents the terrain tile we are calculating. We start by checking for a tile to the North. There is no tile to the North, so the Boolean check returns a value of `0`. We multiply 0 by the directional value for North, $2^0 = 1$, giving us `1*0 = 0`.

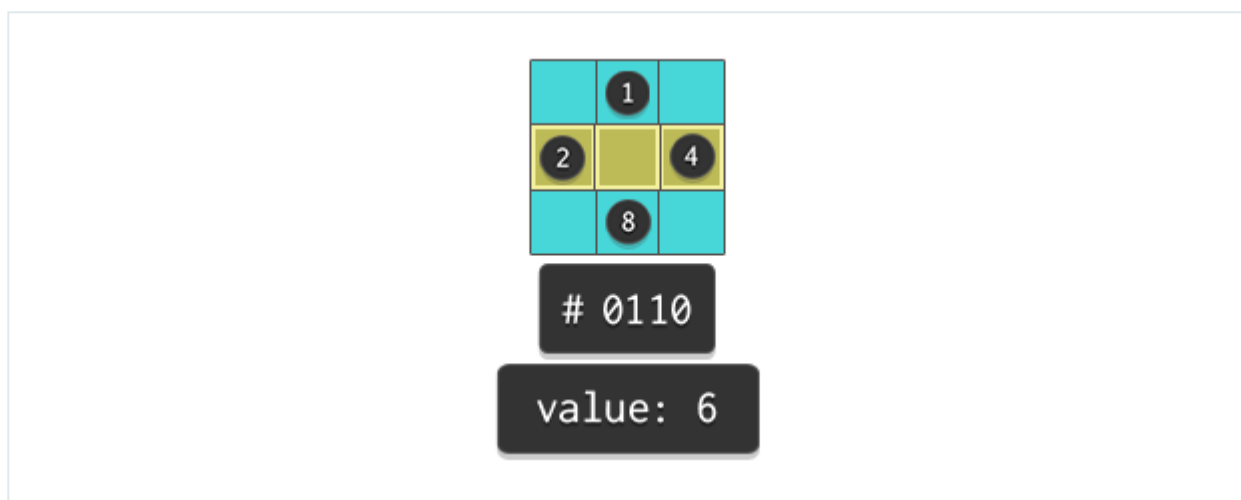
For a terrain tile surrounded entirely by empty space, every Boolean check returns `0`, resulting in the 4-bit binary number `0000` or `1*0 + 2*0 + 4*0 + 8*0 = 0`. There are 16 total possible combinations, from 0 to 15, so the 1st sprite in the tileset will be used to represent this type of terrain tile with a value of `0`.



A terrain tile bordered by a single tile to the North returns a binary value of `0001`, or $1*1 + 2*0 + 4*0 + 8*0 = 1$. The 2nd sprite in the tileset will be used to represent this type of terrain with a value of `1`.



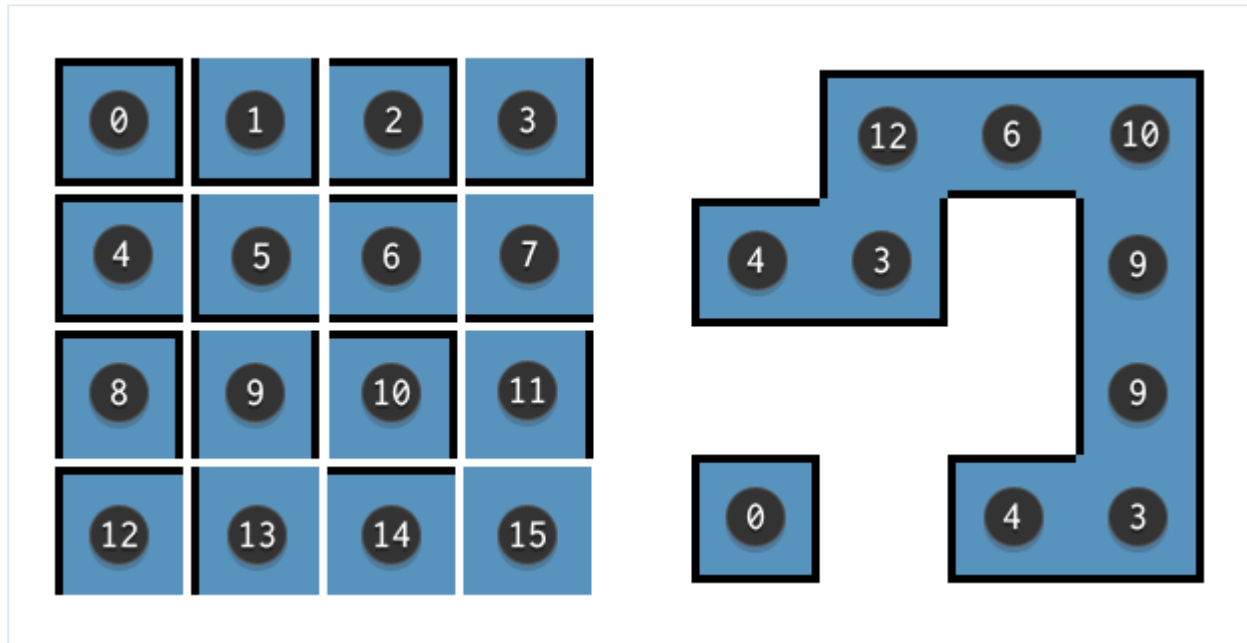
A terrain tile bordered by a tile to the North and a tile to the East returns a binary value of `0101`, or $1*1 + 2*0 + 4*1 + 8*0 = 5$. The 6th sprite in the tileset will be used to represent this type of terrain with a value of `5`.



A terrain tile bordered by a tile to the East and a tile to the West returns a binary value of `0110`, or $1*0 + 2*1 + 4*1 + 8*0 = 6$. The 7th sprite in the tileset will be used to represent this type of terrain with a value of `6`.

Assigning Sprites to Tiles

After calculating a tile's bitmasking value, we assign the appropriate sprite from the tileset. This final step can be performed in real time as the map loads, or the result can be saved and loaded into your tile editor of choice for further editing.



The figure on the left represents a 4-bit, single-terrain tileset as it would appear sequentially on a tile sheet. The figure on the right depicts how the tiles look in-game after they are placed using the bitmasking procedure. Each tile is marked with its bitmasking value to show the relationship between a tile's order on the tile sheet and its position in the game.

As an example, let's examine the tile in the upper-right corner of the figure on the right. This tile is bordered by tiles to the West and South. The Boolean check returns a binary value of `1010`, or $1*0 + 2*1 + 4*0 + 8*1 = 10$. This value corresponds to the 11th sprite in the tile sheet.

Tileset Complexity

The number of required directional Boolean checks depends on the intended complexity of your tileset. By ignoring corner pieces, you can use this simplified 4-bit solution that only requires four directional binary checks.

But what happens when you want to create more visually appealing terrain? You will need to deal with the existence of corner tiles, which increases the amount of sprites from 16 to

48. The following 8-bit bitmasking example requires eight Boolean directional checks per tile.

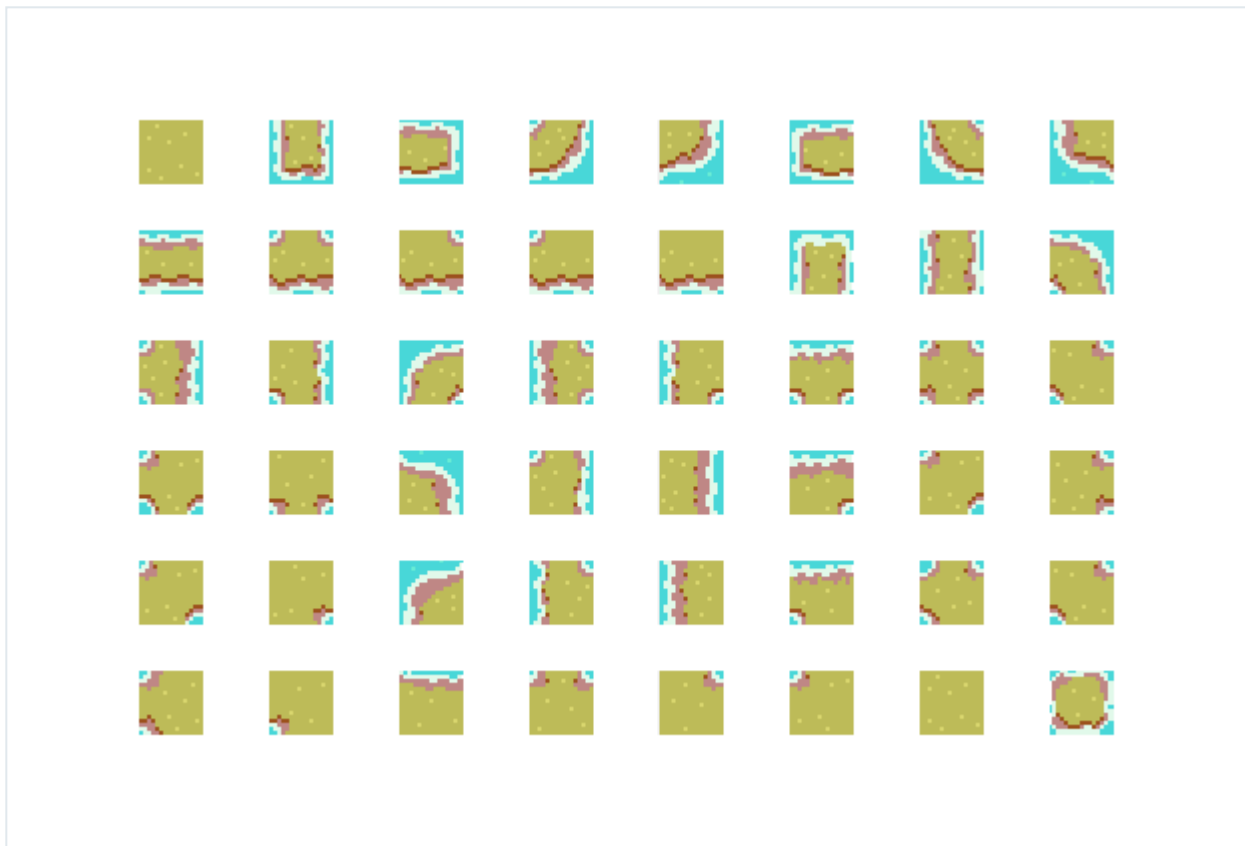
8-Bit Bitmasking with Corner Tiles

For this example, we are creating a top-down tileset that depicts grassy terrain near the ocean. In this case, our ocean exists on a layer underneath the terrain tiles. This allows us to use a single-terrain solution, while still maintaining the illusion that two terrain types are colliding.

Once the game is running and the bitmasking procedure is complete, the sprites will never change. This is a seamless, static implementation of bitmasking where everything takes place before the player ever sees the tiles.

Introducing Corner Tiles

We want the terrain to be more visually interesting than the previous 4-bit solution, so corner pieces are required. This extra bit of visual complexity requires an exponential amount of additional work for the artist, programmer, and the game itself. By expanding on what we learned from the 4-bit solution, we can quickly understand how to approach the 8-bit solution.



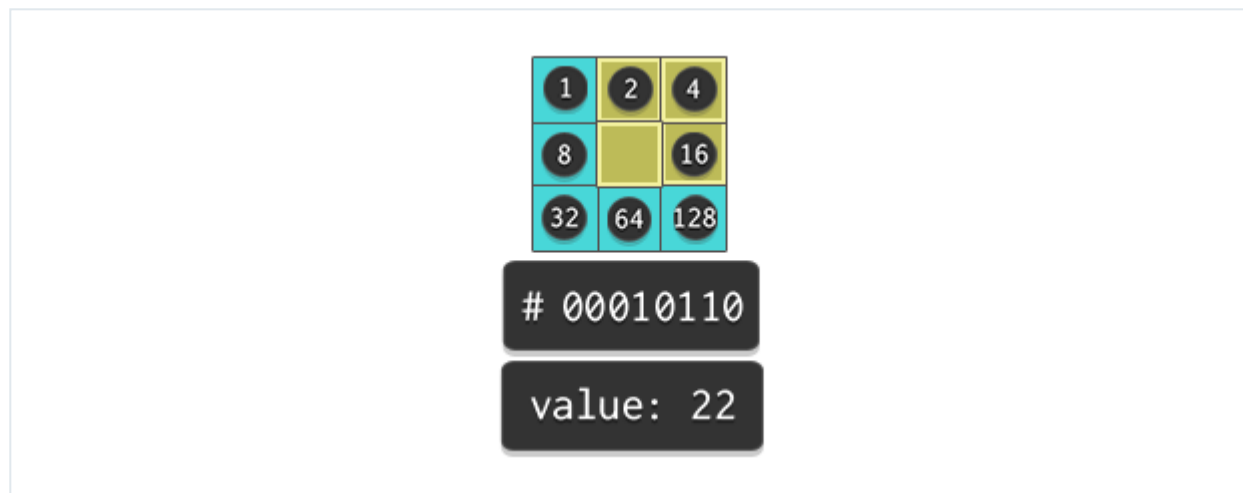
Here is the complete sprite sheet for our ocean-side terrain tiles. Do you notice anything peculiar about the number of tiles? The 4-bit example from earlier resulted in $2^4 = 16$ tiles, so this 8-bit example should surely result in $2^8 = 256$ tiles, yet there are clearly fewer than that there.

While it's true that this 8-bit bitmasking procedure results in 256 possible binary values, not every combination requires an entirely unique tile. The following example will help explain how 256 combinations can be represented by only 48 tiles.

8-bit Directional Values

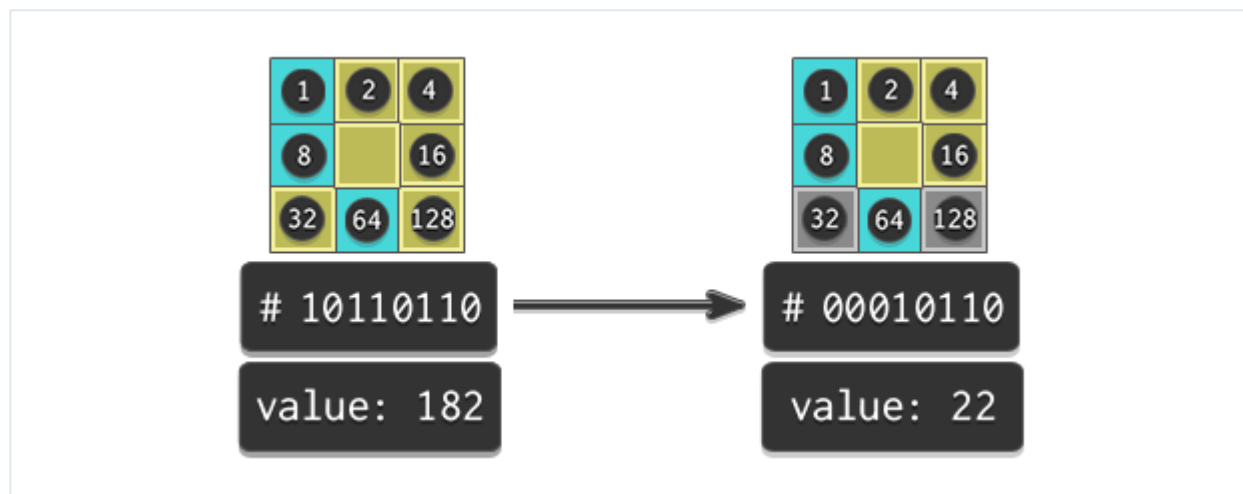
- North West = $2^0 = 1$
- North = $2^1 = 2$
- North East = $2^2 = 4$
- West = $2^3 = 8$
- East = $2^4 = 16$
- South West = $2^5 = 32$

- South = $2^6 = 64$
- South East = $2^7 = 128$



Now we're making *eight* Boolean directional checks. The center tile above is bordered by tiles to the North, North-East, and East, so this Boolean check returns a binary value of

00010110 or $1*0 + 2*1 + 4*1 + 8*0 + 16*1 + 32*0 + 64*0 + 128*0 = 22$.



The tile on the left above is similar to the previous tile, but now it is also bordered by tiles to the South West and South East. This Boolean directional check *should* return a binary value of 10110110, or $1*0 + 2*1 + 4*1 + 8*0 + 16*1 + 32*1 + 64*0 + 128*1 = 182$.

This value is different from the previous tile, but both tiles would actually be visually identical, so it becomes redundant.

To eliminate the redundancies, we add an extra condition to our Boolean directional check: when checking for the presence of bordering *corner* tiles, we also have to check for neighboring tiles in the four cardinal directions (directly North, East, South, or West).

For example, the tile to the North-East is neighbored by existing tiles, whereas the tiles to the South-West and South-East are not. This means that the South-West and South-East tiles are not included in the bitmasking calculation.

With this new condition, this Boolean check returns a binary value of `00010110` or `1*0 + 2*1 + 4*1 + 8*0 + 16*1 + 32*0 + 64*0 + 128*0 = 22` just like before. Now you can see how the 256 combinations can be represented by only 48 tiles.

Tile Order

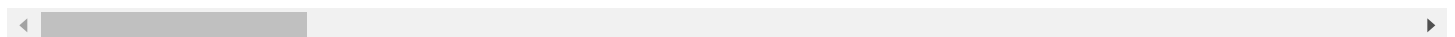
Another problem you may notice is that the values calculated by the 8-bit bitmasking procedure no longer correlate to the sequential order of the tiles in the sprite sheet. There are only 48 tiles, but our possible calculated values range from 0 to 255, so we can no longer use the calculated value as a direct reference when grabbing the appropriate sprite.

What we need, therefore, is a data structure to contain the list of calculated values and their corresponding tile values. How you want to implement this is up to you, but remember that the order in which you check for surrounding tiles dictates the order in which your tiles should be placed in the sprite sheet.

For this example, we check for bordering tiles in the following order: North-West, North, North-East, West, East, South-West, South, South-East.

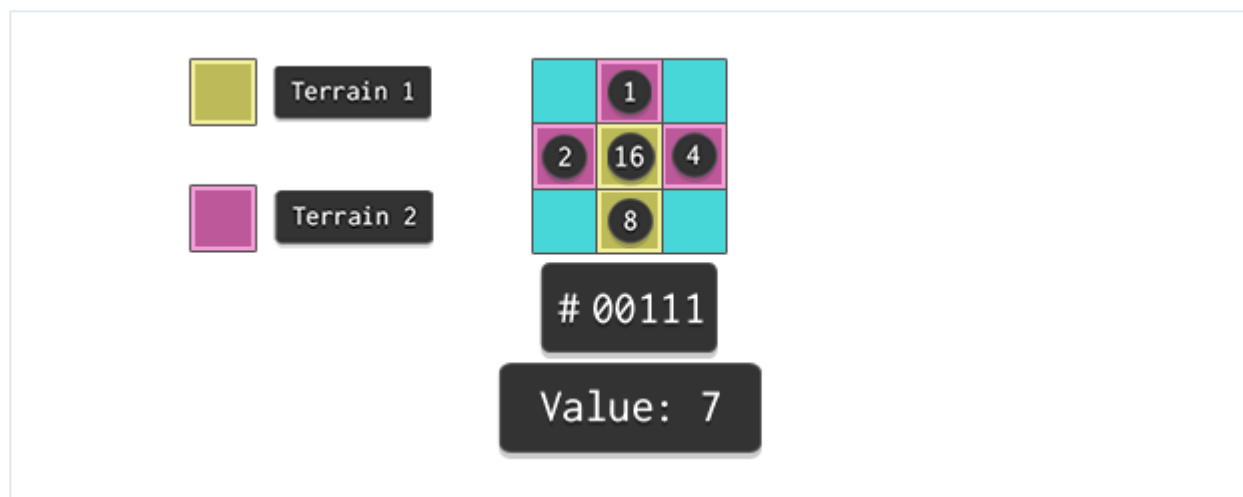
Below is the complete set of bitmasking values as they relate to the positions of tiles in our sprite sheet (feel free to use these values in your project to save time):

1 | { 2 = 1, 8 = 2, 10 = 3, 11 = 4, 16 = 5, 18 = 6, 22 = 7, 24 = 8, 26 = 9, 27 = 10,



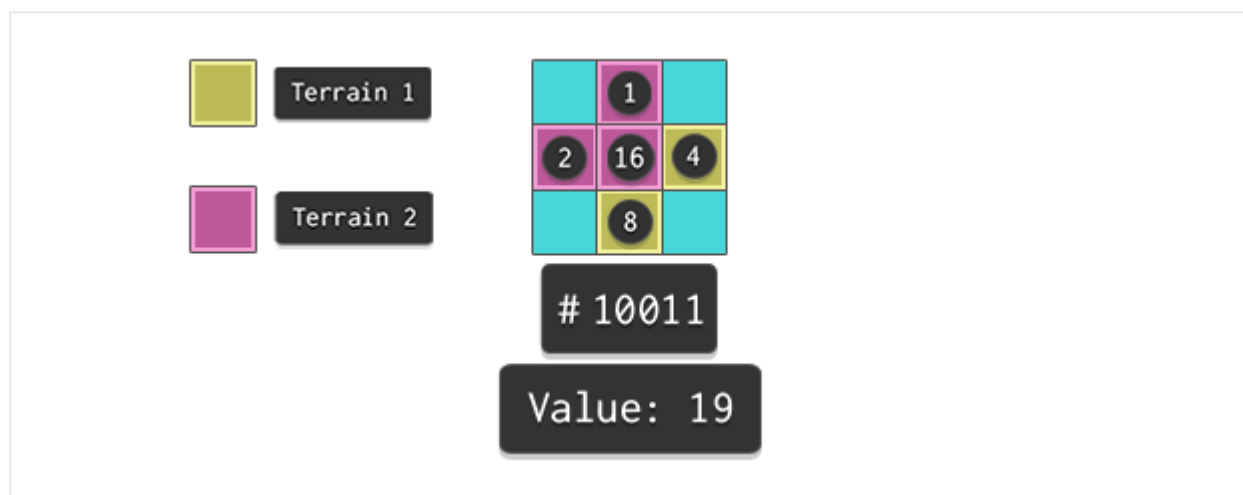
Multiple Terrain Types

All of our previous examples assume a single terrain type, but what if we introduce a second terrain to the equation? We need a *5-bit* bitmasking solution, and we need to define our two terrain types. We also need to assign a value to the center tile that is only counted under specific conditions. Remember that we are no longer accounting for "empty space" as in the previous examples; tiles must now be surrounded by another tile on all sides.



The above figure shows an example with two terrain types and no corner tiles. Type 1 *always* returns a value of 0 whenever it is detected during the directional check; the center tile value is calculated and used *only* if it is terrain type 2.

The center tile in the above example is surrounded by terrain type 2 to the North, West, and East, and by terrain type 1 to the South. The center tile is terrain type 1, so it is not counted. This Boolean check returns a binary value of 00111, or $1*1 + 2*1 + 4*1 + 8*0 + 16*0 = 7$.



In this example, our center tile is terrain type 2, so it will be counted in the calculation. The center tile is surrounded by terrain type 2 to the North and West. It is also surrounded by terrain type 1 to the East and South. This Boolean check returns a binary value of `10011`, or `1*1 + 2*1 + 4*0 + 8*0 + 16*1 = 19` .

Dynamic Implementation

The bitmasking calculation can also be performed during gameplay, allowing for real-time changes in tile placement and appearance. This is useful for destructible terrain as well as games that allow for crafting and building. The initial bitmasking procedure is mandatory for all tiles, but any additional dynamic calculations should only be performed when absolutely necessary. For example, a destroyed terrain tile would trigger the bitmasking calculation only for surrounding tiles.

Conclusion

Tile bitmasking is the perfect example of building a working system to aid you in game development. It is not something that directly impacts the player's experience; instead, this method of automating a time-consuming portion of level design provides a valuable benefit to the developer. To put it simply: tile bitmasking is a quick way to make the game do your dirty work, allowing you to focus on more important tasks.

Advertisement



Sonny Bone

Player, developer, and lover of video games. Ludum Dare game jammer and rad dude. God of Phantom Green Studios.

 [Phantom_Green](#)

 FEED  LIKE  FOLLOW  FOLLOW

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.

Update me weekly

Advertisement

[View on GitHub](#)

Translations

Envato Tuts+ tutorials are translated into other languages by our community members—you can be involved too!


Translate this post

Powered by  native

[13 Comments](#)[Tuts+ Hub](#)[Login](#) [Recommend](#) 5 [Share](#)[Sort by Best](#) [LOG IN WITH](#)[OR SIGN UP WITH DISQUS](#) 

M. Francis Stramaglia • 2 years ago

Really great post. I implemented this pretty much exactly in javascript/html5 for my game. Four-bit dynamic, with some tweaks for multiple terrain types. Took me forever to figure out an efficient (graphics production and code-wise) way to implement it, wish I had this to work from. My implementation also makes use of classes of tiles that work together, and a base "empty" tile type (grass) that all terrain snaps to. So for instance, water next to desert will have a bit of grass in between. Not always visually ideal, but a very helpful cheat. You can see it at work here <http://bludgeonsoft.itch.io...>

24   • [Reply](#) • [Share](#) ›

Michael James Williams [Mod](#) ➔ **M. Francis Stramaglia** • 2 years ago

Fantastic example!

1   • [Reply](#) • [Share](#) ›

Guy Walker • 2 years ago

These are called 'Blob' tilesets as each tile contains a central blob of similar terrain, except maybe for one 'all water' tile where it can be removed.

Note that there are 47 tiles, not 48. In your tileset you have two 'all green' tiles, the first and second from last.

I would recommend placing values sequentially in a clockwise (or anticlockwise) direction. This can reduce code as all tiles have rotational symmetry. So a lot of the time you're doing the same thing but in four different directions.

There is more information here, <http://www.cr31.co.uk/stage...>

including two methods of constructing random terrains. Method 1 removes 90% of corner tiles to create bigger rooms as 'Unwary' suggests.

9 ^ | v • Reply • Share ›

Powerslave • 2 years ago

Nice one! Turning the concepts inside-out, you get an excellent tilemap generator: The algorithm starts the map basically anywhere, and for each randomly picked tile, randomly picks one from the set of allowed neighbors in each direction, then moves on to do the same to all the neighbors. Enhancing this with some heuristics (usually game-specific) can yield brilliant maps.

2 ^ | v • Reply • Share ›

Unwary ➔ Powerslave • 2 years ago

That is smart... So for say a game with a more open feel, one could have the corner cases have a lower weight, and thus result in bigger rooms, as there are less corners to go around... Very nice idea.

^ | v • Reply • Share ›

Ciro Bolado • 2 years ago

D: very usefull, for tactics games!!
this will help me very much thankyou!!

2 ^ | v • Reply • Share ›

pjnovas • 2 years ago

Awesome post, thanks!

1 ^ | v • Reply • Share ›

Simon Ferguson • 22 days ago

This is a nice article, I have similar in the past but with three different terrain types. Its a very neat trick! Just wanted to point out that I think your corner tileset has some errors, there are a number of duplicate tiles and missing ones.

^ | v • Reply • Share ›

7bit • 3 months ago

Thanks for good tutorial.

But i can't understand 'Multiple Terrain Types' part.

What is it for?

5 bit masking... so do i need 32 sprites for this?

and corner side like tile no.4 can be Type2 and it's neighbor can be Type1..

why the tiles must be surrounded by another tile on all sides?

^ | v • Reply • Share ›

Aaron • 8 months ago

Just wanted to add that this article was the missing piece I needed in my tile generator for a planet surface terrain I was building. Amazing the solution was right under my nose the whole time...bit masking!

^ | v • Reply • Share ›

Patrick Traynor • 10 months ago

Thanks for the great writeup! This helped me a lot.

Minor errors in the 8-bit tiles: Tile 23 should have water in the bottom right too, and tile 30 should have water in the top right too.

^ | v • Reply • Share ›

Matthew Arnold • 2 years ago

Thanks for this write up.

I was using the 4way method for the longest of times and I hated looking at the corners lol. I spent so much time working on the art of the tiles to try and negate the issue as much as possible, but its so much better now with correct 8way checks :)

Advertisement

QUICK LINKS - Explore popular categories

ENVATO TUTORIALS+

JOIN OUR COMMUNITY

HELP

tuts+

25,789

Tutorials

1,118

Courses

22,795

Translations

Envato.com Our products Careers Sitemap

© 2018 Envato Pty Ltd. Trademarks and brands are the property of their respective owners.

Follow Envato Tuts+