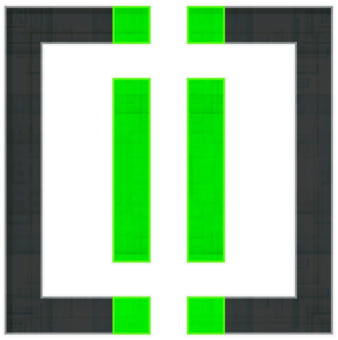
Fractal Auto Tile Format
For Destructible Worlds

www.twitch.com/kanjicoder
kanjicoder@Gmail.com

May 27, 2021



1 Introduction

EASILY AND QUICKLY Create 100% destructible environments by painting with materials that have a mind of their own.

2 Classic Auto Tiling : Overview & Benefits

Wang style auto tiling typically uses a strip of 16 sub graphics within an "auto tile strip" to make it possible to quickly define aesthetically pleasing delineated chunks of material within your world map.

The two major benefits are:

1. Streamlines asset creation pipeline

   Rather than meticulously picking and matching 16 sub graphics while creating your tile map environment, you can make one swift brush stroke that does all of this for you. Which is at LEAST a 16X speedup in terms of level designer productivity.

2. Solid foundation for DECENT destructible geometry games

   When a tile is removed from the game, it doesn't look like someone used the eraser brush in MS-Paint on your tilemap and just deleted a chunk of pixels. This really helps maintain the illusion of a solid material and enables what I call "destroying in an aesthetically pleasing way". For example, notice how in doom eternal the cacodemon is sliced in half. Someone had to define that beautiful internal detail. Otherwise you would just have 2 flat textureless polygons after cutting the model in half. When you model with auto tiles, you get that internal detail FOR FREE!

   `[    https://www.youtube.com/watch?v=vTqvjORXvwI&t=1m35s    ]`

3 Classic Auto Tiling : PROBLEM SUMMARY

1. It is bad for your mood [1]
2. It is bad for your productivity[2]
3. It is bad for your engine performance [3]
4. Levels oddly "self heal" [4]
5. Yields a CHILDISH design [5]
6. Pixillated graphics are severely limiting [6]

4 Classic Auto Tiling : SOLUTION INDEX

1. It is bad for your mood
   Wang style auto tile sets are very redundant, and thus a displeasure to create.

   solution_tag(s)

   (a) FOURSET [15]

2. It is bad for your productivity
   Wang style auto tile sets are very redundant, and thus waste your human brain cycles.

   solution_tag(s)

   (a) FOURSET [15]

3. It is bad for your engine performance
   Wang style auto tile sets are very redundant, and thus waste RAM on both CPU & GPU.

   solution_tag(s)

   (a) FOURSET [15]

4. Levels oddly "self heal"
   It is better than the alternative of just removing a tile and having the void left behind look like someone went into MS-Paint and manually deleted the pixels, but it still looks a bit weird and jarring.

   solution_tag(s)

   (a) FRACTAL_AUTO_TILING [6]
   (b) AUTO_LATCHING (AUTO_MIXING) [7]
   (c) DAMAGED_SPACE_TILES [8]

5. Yields a CHILDISH design
   Building a levelmap on a uniform grid leads to a childish design. We need a size hierarchy and areas of low and high detail in order to get a a visual look that looks SERIOUS rather than CHILDISH.

   solution_tag(s)

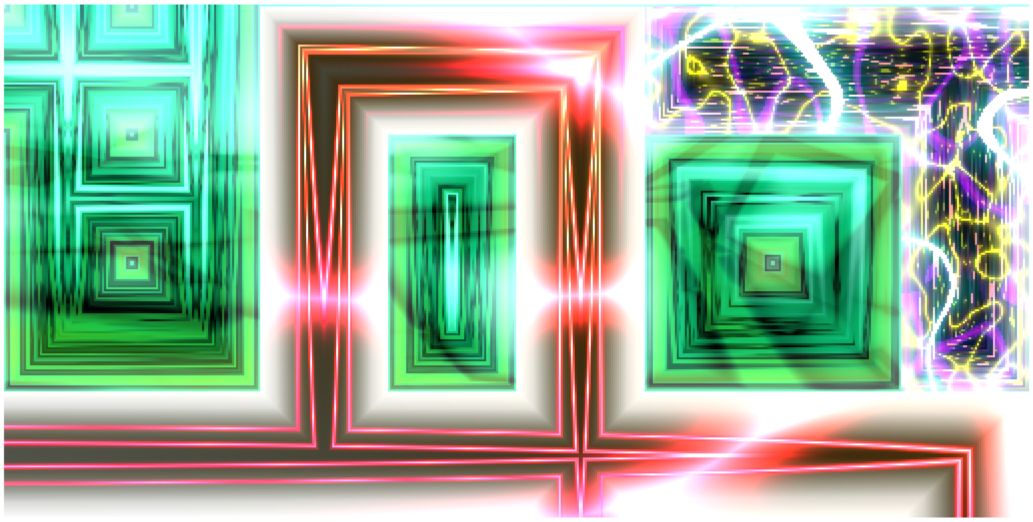
   (a) MULTI_LAYERED_TILEMAP [9]
   (b) DIFFERENT_TILE_SIZES [10]
   (c) DECAL_LAYER [11]
   (d) ENVIORNMENT_PREFABS [12]

6. Pixillated graphics are severely limiting
   Pixillated retro looking games have some charm. But we want to make something badass looking. If things get pixillated when you zoom in that severely limits what you can do with your camera.

   solution_tag(s)

   (a) SHADER_AUTO_TILING [5]
   (b) FRACTAL_AUTO_TILING [6]
   (c) NORMAL_MAP_COMPOSITING [13]

5 SHADER_AUTO_TILING



The implementation shown in the screenshot is pretty simple. Procedurally generate wang-tile sub-tile gradients on the fly and use these sub-tile gradients to warp the space of any material shader to create tiles with well delineated edges.

Here is an interactive demo I made hosted on my heroku server. The server is FREE so please wait up to 30 seconds when you load this URL into your browser. A refresh after 30 seconds may also be required.

```
[    https://d3m0.herokuapp.com/demo/nexient.html    ]
```

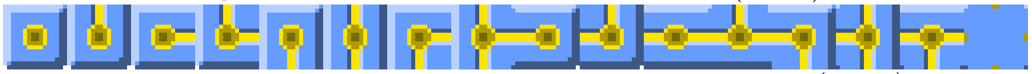
While fun to play with, that was a first draft proof of concept and the code is very ugly. Here are isolated examples explaining the math I created on shadertoy.com. (NOTE: I am DEKTEN on shadertoy.com)

```
Auto Tile Basics:
[    https://www.shadertoy.com/view/7sSSWm    ]

Auto Tile Basics used to re-map an interesting shader:
[    https://www.shadertoy.com/view/ssSXWm    ]
```

6 FRACTAL_AUTO_TILING

The concept is dead simple.

Take a wang tile auto tile set like this: (T_0)



And use it to draw ANOTHER auto tile set. (T_1)

You have now created a T_1 Auto Tile Set (Auset).

- T_0 Auto Tile Sets (Ausets) are TERMINAL and drawn as solid colors.
- T_1 Auto Tile Sets (Ausets) are drawn with T_0 auto tiles.
- T_2 Auto Tile Sets (Ausets) are drawn with T_1 auto tiles.
- T_3 Auto Tile Sets (Ausets) are drawn with T_2 auto tiles.

  And so on and so forth.

The higher the T value of the auto tile set, the more intricate emergent behavior. The hope is that with enough nesting levels (T values) auto tiles will not just simply "snap into alignment" but will rather intricately fuse together in exciting and unpredictable (yet deterministic) ways.

For example, when laying down T_3 auto tiles on a map, the outer T_3 auto tiling must be resolved before it is possible to solve for the T_2 auto tiling. And so on for each next nesting level down all the way to T_0.

This constraint means the consequences of auto tiling at each level cascades into the next level down. This may likely yield a chaotic deterministic system.

Because the amounts of checks to render each pixel explodes geometrically with each new T level, I will be targeting only T_2 auto tiles for this first attempt. If successful, I will take the plunge into implementing a Vulkan implementation that will use a ping-pong buffer to make the complexity linear rather than geometric.

```
        +---=---+
        |       |
        |       |
        |       |
+---=---+---=---+---=---+  To render TIL we must make
|       |       |       |  a total of 5 checks to.
|       |  TIL  |       |  resolve the auto tiling.
|       |       |       |
+---=---+---=---+---=---+
        |       |
        |       |
        |       |
        +---=---+

2x2 buffer for depth_0 (T_0)
```

```
5 checks per tile to resolve auto tiling.
+---+---+
|   |   |
+---+---+
|   |   |
+---+---+

8x8 buffer for depth_1
1. Outer tile requires 5 checks.
2. Each sub-tile of outer tile requires 5 more checks.

So to render a T_1 tile you need:
5*5 checks to resolve auto tiling for a given pixel.
25 checks.
+---+---+ +---+---+
|   |   | |   |   |
+---+---+ +---+---+
|   |   | |   |   |
+---+---+ +---+---+
+---+---+ +---+---+
|   |   | |   |   |
+---+---+ +---+---+
|   |   | |   |   |
+---+---+ +---+---+

16x16 buffer for depth_2 (T_2):
Requires 5*5*5 to resolve auto tiling for any
given pixel. 125 checks!
+---------------------+ +---------------------+
| +---+---+ +---+---+ | | +---+---+ +---+---+ |
| |   |   | |   |   | | | |   |   | |   |   | |
| +---+---+ +---+---+ | | +---+---+ +---+---+ |
| |   |   | |   |   | | | |   |   | |   |   | |
| +---+---+ +---+---+ | | +---+---+ +---+---+ |
| +---+---+ +---+---+ | | +---+---+ +---+---+ |
| |   |   | |   |   | | | |   |   | |   |   | |
| +---+---+ +---+---+ | | +---+---+ +---+---+ |
| |   |   | |   |   | | | |   |   | |   |   | |
| +---+---+ +---+---+ | | +---+---+ +---+---+ |
+---------------------+ +---------------------+
+---------------------+ +---------------------+
| +---+---+ +---+---+ | | +---+---+ +---+---+ |
| |   |   | |   |   | | | |   |   | |   |   | |
| +---+---+ +---+---+ | | +---+---+ +---+---+ |
| |   |   | |   |   | | | |   |   | |   |   | |
| +---+---+ +---+---+ | | +---+---+ +---+---+ |
| +---+---+ +---+---+ | | +---+---+ +---+---+ |
| |   |   | |   |   | | | |   |   | |   |   | |
| +---+---+ +---+---+ | | +---+---+ +---+---+ |
| |   |   | |   |   | | | |   |   | |   |   | |
| +---+---+ +---+---+ | | +---+---+ +---+---+ |
+---------------------+ +---------------------+
```

The general idea of the math in this diagram (above) is correct. However the actual tiles have a lot more complexity to them than this, so I am uncertain about the exact implementation.
This geometric explosion of calculations could be made linear with the help of a compute shader and ping pong buffer. For example, if we had a map drawn in T_2 tiles we would:

1. Solve T_2 auto tiling for all tiles on screen.
2. Save results to buffer A
3. Read from buffer A and write T_1 auto tile solutions to buffer B.
4. Read from buffer B and write T_0 auto tile solutions to buffer A. (Yes, overwrite buffer A)

Each time the buffers ping pong between each other, the dataset at least **DOUBLES** in size. So make sure that buffers "A" and "B" can accommodate the largest tilemap cache.

We want something that is feasible with processing power of fragment shader, and is also satisfying to work with. And gets good looking results. After messing around with numbers I have decided on:

- T_0 : 128x128 : Terminal Auto Tile Set (Already Coded)
- T_1 : 32x32 damaged edge effects happen here.
- T_2 : 16x16 auto latching happens here.

I keep going over the math for the computational complexity of the rendering algorithm to support T_0 -to- T_2 ausets, but I do not have a firm grasp on it. I will attempt rendering up to T_2 level, and if it is not possible due to performance reasons, I will scale back my ambitions.

7 AUTO_LATCHING (AUTO_MIXING)

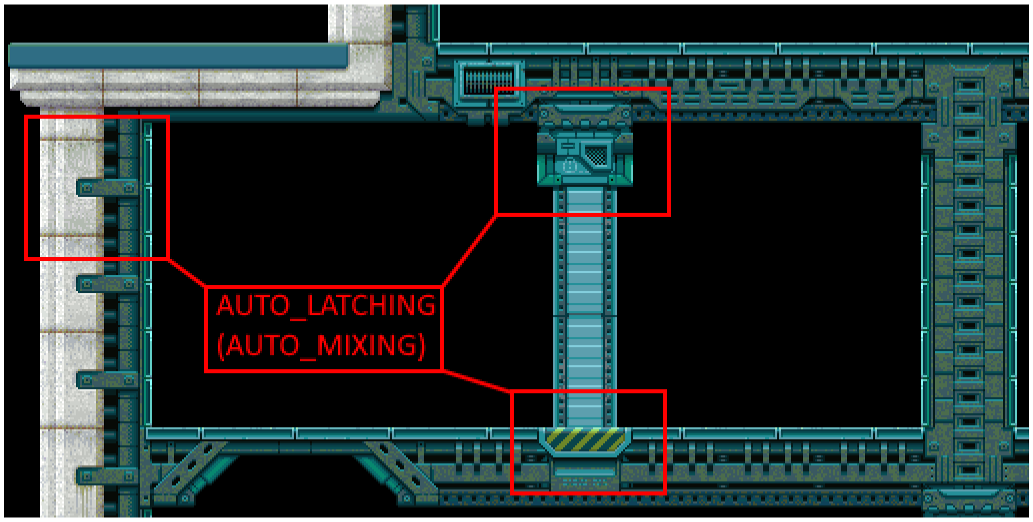


Image Credit: https://www.deviantart.com/rpwhitaker/art/Tilex-449645894

Autolatching (aka: automixing) defines how two auto tile sets interact when they are either overlapping or tangenting each other. The relevant examples have been circled (with red squares) in the picture above.

Here is a most basic truth table of a simple implementation I would like to use on the T_2 auto tiles:

```
--------------------------------------------------------
AYE & BEE have same settings:
--------------------------------------------------------
    +---+---+---+---+---+---+
    |           |           |
    +   +   +   +   +   +   +  AYE == No Action
    |    AYE    |    BEE    |  BEE == No Action
    +   +   +   +   +   +   +
    |           |           |
    +---+---+---+---+---+---+

    +---+---+--+ +--+---+---+
    |       ___| |___       |
    +   +  |         |  +   +  AYE == AUTOMIX==DIVOTS
    |    AYE         BEE    |  BEE == AUTOMIX==DIVOTS
    +   +  |         |  +   +
    |       ---| |---       |
    +---+---+--+ +--+---+---+
    +---+---+---+---+---+---+
    |           |           |
    +   +   +===+===+   +   +  AYE == AUTOMIX==PRONGS
    |    AYE|||||||||BEE    |  BEE == AUTOMIX==PRONGS
    +   +   +===+===+   +   +  ( 50/50 alpha blend overlap )
    |           |           |  @VID_IID[0275]T[02:47:32]
    +---+---+---+---+---+---+
--------------------------------------------------------
AYE or BEE overlap the other:
--------------------------------------------------------
    +---+---+---+---+---+---+
    |           |           |
    +   +   +   +===+   +   +  AYE == AUTOMIX==PRONGS
    |    AYE        |BEE    |  BEE == No Action
    +   +   +   +===+   +   +
    |           |           |
    +---+---+---+---+---+---+

    +---+---+---+---+---+---+
    |           |           |
    +   +   +===+   +   +   +  AYE == No Action
    |    AYE|        BEE    |  BEE == AUTOMIX==PRONGS
    +   +   +===+   +   +   +
    |           |           |
    +---+---+---+---+---+---+
--------------------------------------------------------
AYE & BEE Create Male-Female Sockets:
--------------------------------------------------------
    +---+---+---++--+---+---+
    |           ||___       |
    +   +   +   +===+|  +   +  AYE == AUTOMIX==PRONGS
    |    AYE        |BEE    |  BEE == AUTOMIX==DIVOTS
    +   +   +   +===+|  +   +
    |           ||‾‾‾       |
    +---+---+---++--+---+---+

    +---+---+--++---+---+---+
    |       ___||           |
    +   +  |+===+   +   +   +  AYE == AUTOMIX==DIVOTS
    |    AYE|       |BEE    |  BEE == AUTOMIX==PRONGS
    +   +  |+===+   +   +   +
    |       ‾‾‾||           |
    +---+---+--++---+---+---+
    ^
    |
    +--[ This last example is what is happening with   ]
```

```
        [ the CONCRETE and METAL tiles in the image at  ]
        [ the beginning of this paper's current section.]
    --------------------------------------------------------
```

I have more complex autolatching (automixing) schemes, but will start with this in my first draft.

8 DAMAGED_SPACE_TILES

No screenshot for this, I am not that far along.
Instead of having only ONE type of [ZERO]tile... We could have a whole map of different kinds of [ZERO] tiles.

These tiles are used as a mask to tell us[HOW] parts of the level were destroyed.

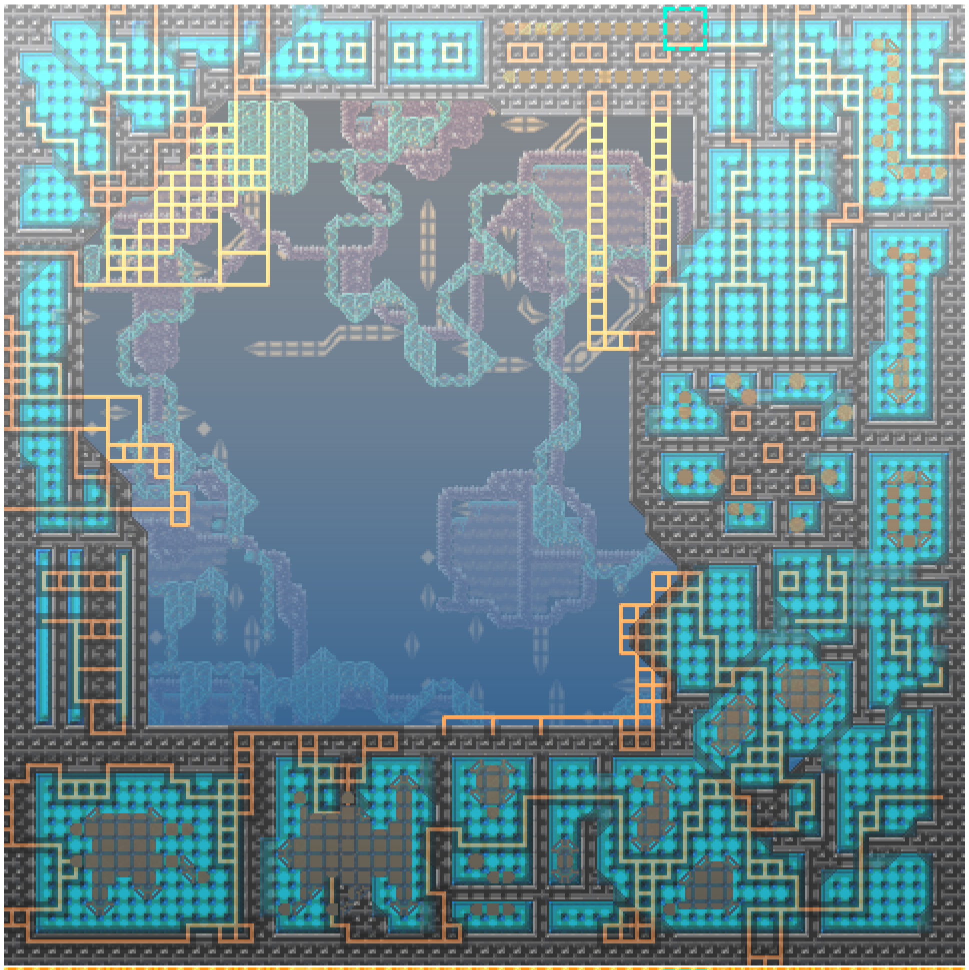
For example: A tile[deleted/destroyed] by a [FIRE_BOMB] may leave a different type of graphic behind than a tile[deleted/destroyed]by an[ELECTRIC_BOMB].

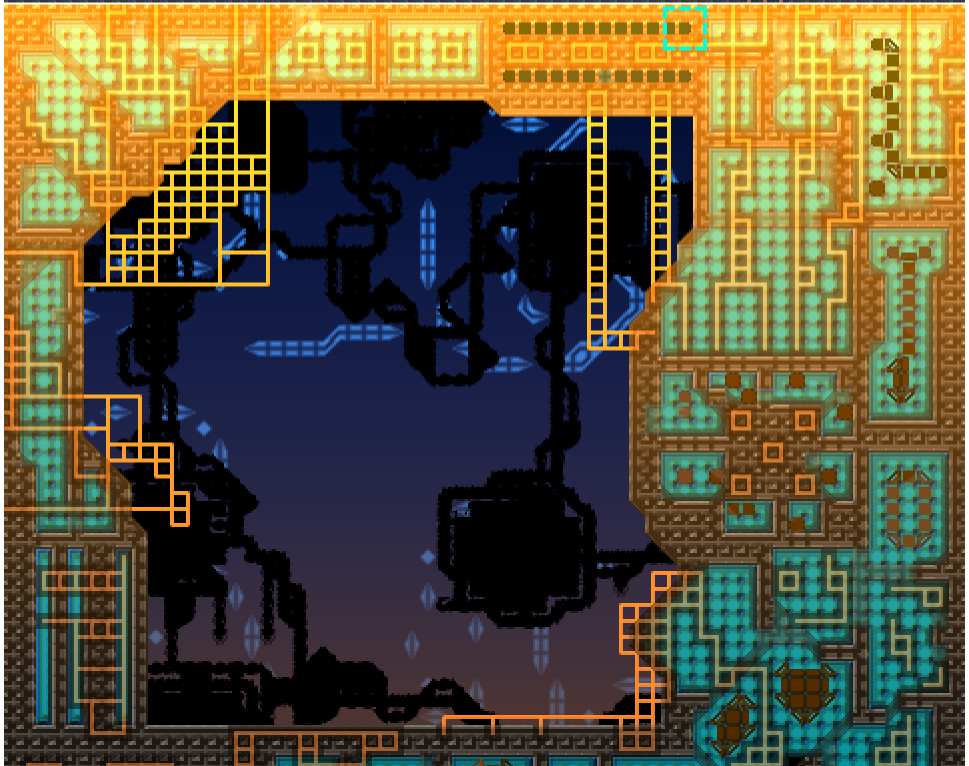
- ELECTRIC_DAMAGE_TILE:
  Might leave MELTED EDGE EFFECTS on tiles adjacent to it's overlap site.
- FIRE_DAMAGE_TILE:
  Might leave CHARRED BURNT COAL EFFECTS on tiles adjacent to it's overlap site.

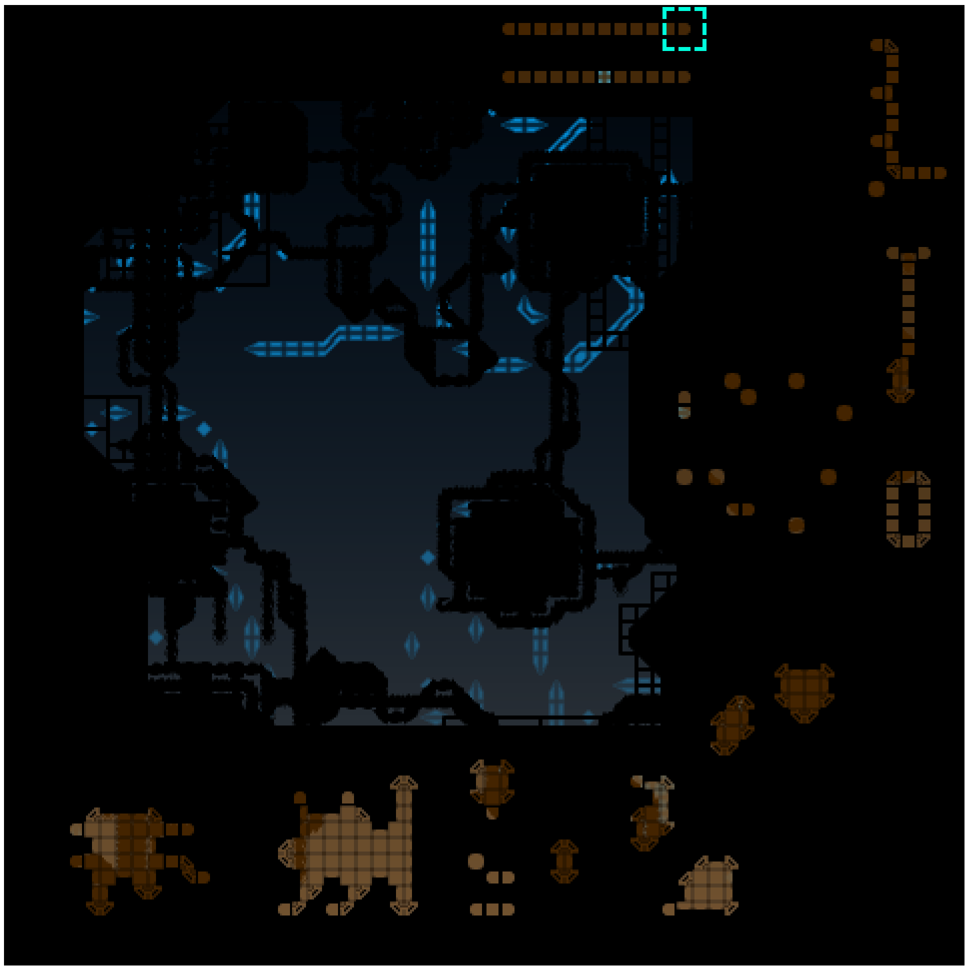
```
  Two bombs explode with a square 3X3
  blast radius on 7x7 chunk of tiles:

    0   1   2   3   4   5   6
  +---+---+---+---+---+---+---+ <---+
0 |   |   |ELE|ELE|ELE|   |   |     |
  +---+---+---+---+---+---+---+     |    overlay
1 |   |   |ELE|ELE|ELE|   |   |     +--- electrical
  +---+---+---+---+---+---+---+     |    damage tiles
2 |   |   |ELE|ELE|ELE|   |   |     |
  +---+---+---+---+---+---+---+ <---+
3 |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+ <---+
4 |   |   |FIR|FIR|FIR|   |   |     |
  +---+---+---+---+---+---+---+     |    overlay
5 |   |   |FIR|FIR|FIR|   |   |     +--- fire damage
  +---+---+---+---+---+---+---+     |    tiles
6 |   |   |FIR|FIR|FIR|   |   |     |
  +---+---+---+---+---+---+---+ <---+

  The two 3X3 blocks(s) of tiles
  STILL_EXIST below, but are _NOT_ visible
  to the player of the game. The DAMAGE_MAP
  of [ELE] & [FIR] tiles has masked out the
  tiles on the GEOMETRY_MAP.

  +---+---+           +---+---+
  |   |_M_|           |_M_|   | _M_ : Melted effects
  +---+---+           +---+---+       on these tiles.
  |   |_M_|           |_M_|   |
  +---+---+           +---+---+
  |   |_M_|           |_M_|   |
  +---+---+---+---+---+---+---+
  |   |   |DAM|DAM|DAM|   |   | DAM : Combined
  +---+---+---+---+---+---+---+       damage effects.
  |   |_C_|           |_C_|   |
  +---+---+           +---+---+
  |   |_C_|           |_C_|   | _C_ : Chared effects
  +---+---+           +---+---+       on these tiles.
  |   |_C_|           |_C_|   |
  +---+---+           +---+---+
```

Much like AUTOLATCHING , the _M_ tiles will render with melted edges because they detect that they are tangenting [ELE] damage tiles. The _C_ tiles will render with [burnt / charred] edges because they are tangenting [FIR] damage tiles. The tiles labeled [DAM] are [burnt / charred] on one edge and melted on the other.

9 MULTI_LAYERED_TILEMAP



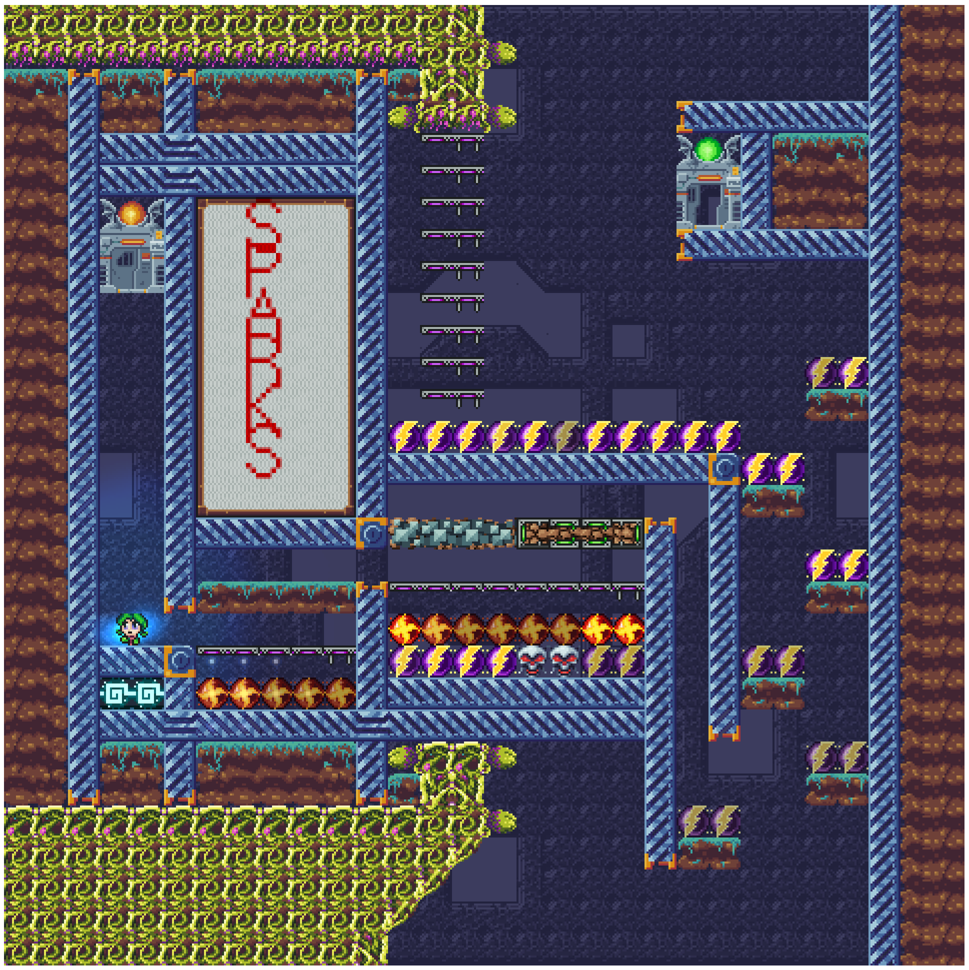




Pictured above is a multi-layered tilemap engine I coded 10 years ago in action-script 3.0 as a proof of concept. The basic idea is to create a more interesting tilemap by compositing and blending together multiple layers of auto tiles. As you can see, there are two distinct groups of auto tiles that have been composited together. One group in the background, and another in the foreground.

The purpose of this proof of concept was to create an asset pipeline capable of creating richer tile-based environments with more size and detail variation than typical tile map.

Compare this to a screenshot of Atomic Alice #1, and we can see that without multi-layering and compositing, the grid is very obvious and there is a lack of visual detail hierarchy that leads to the end product looking rather childish despite my artist's best efforts.



Without areas of high and low visual detail, the level design looks overly clunky. Which makes it feel like a toy for a toddler. Multi layering and compositing will help give a more [serious / mature] look to the resulting visual aesthetics without requiring the skills of a professional artist. As long as the designer of the level has aesthetic sense, they need not have[top notch]art skills to produce professional quality graphics. Because of the ease of use with this workflow, you can output content with ease and do it quickly, which yields and addictive and rewarding experience for the users of this engine.

10 DIFFERENT_TILE_SIZES

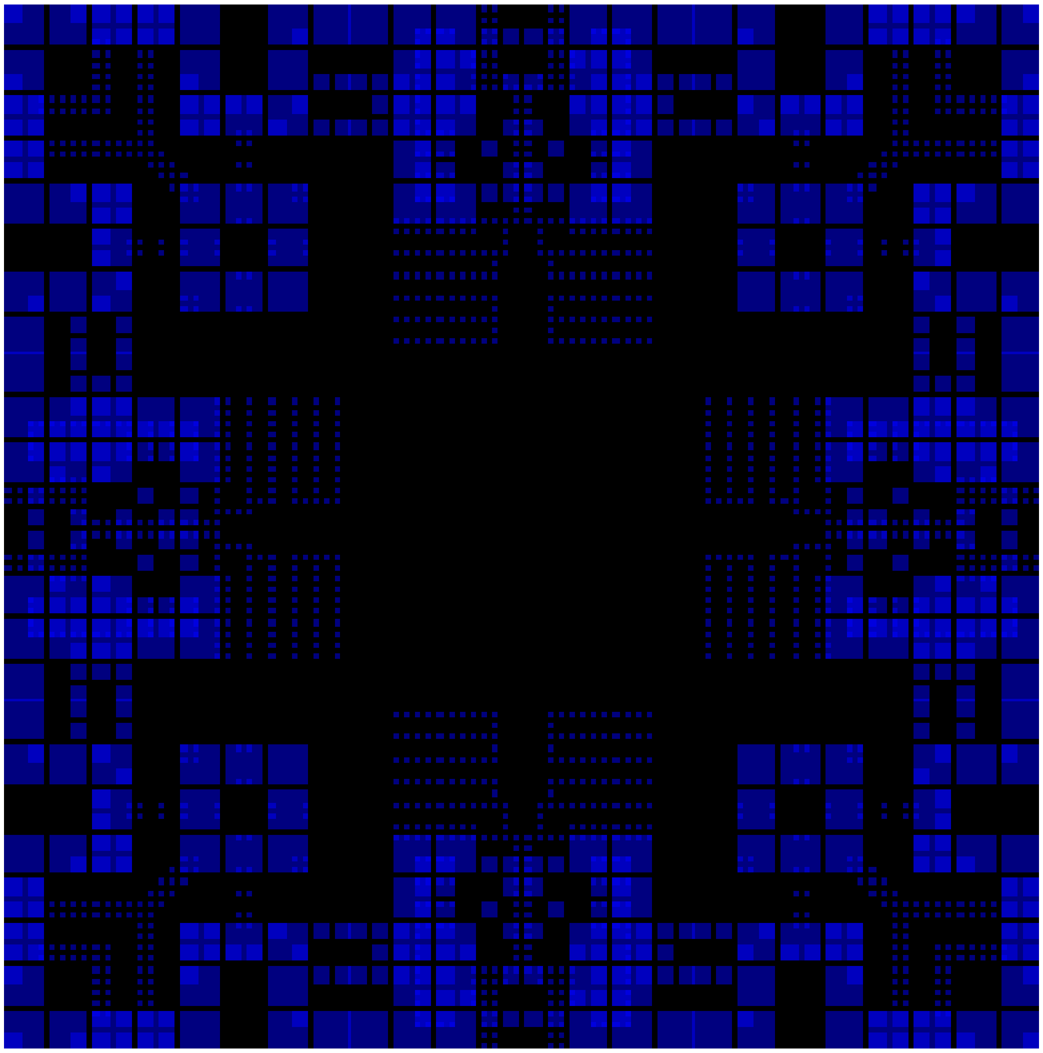
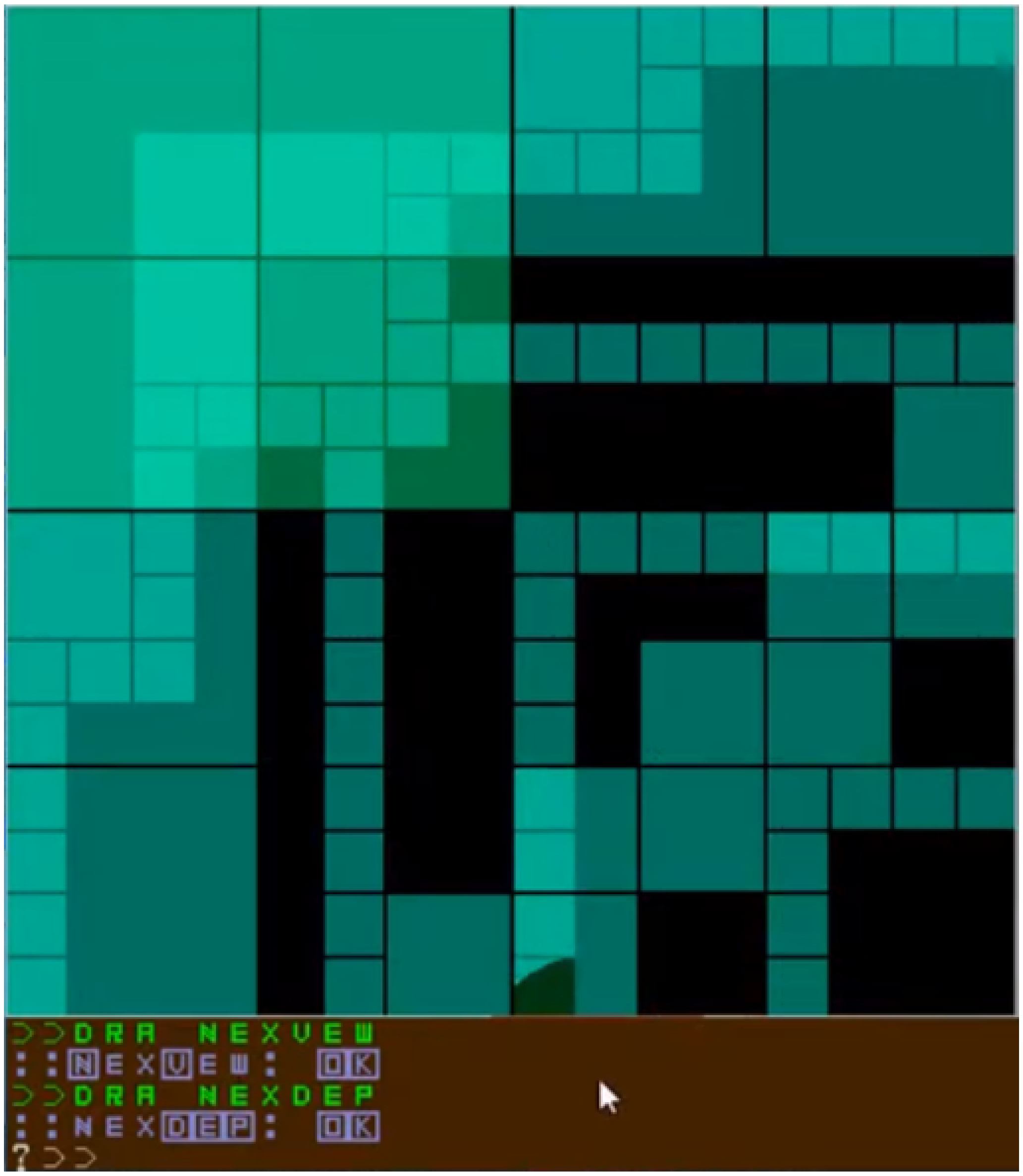


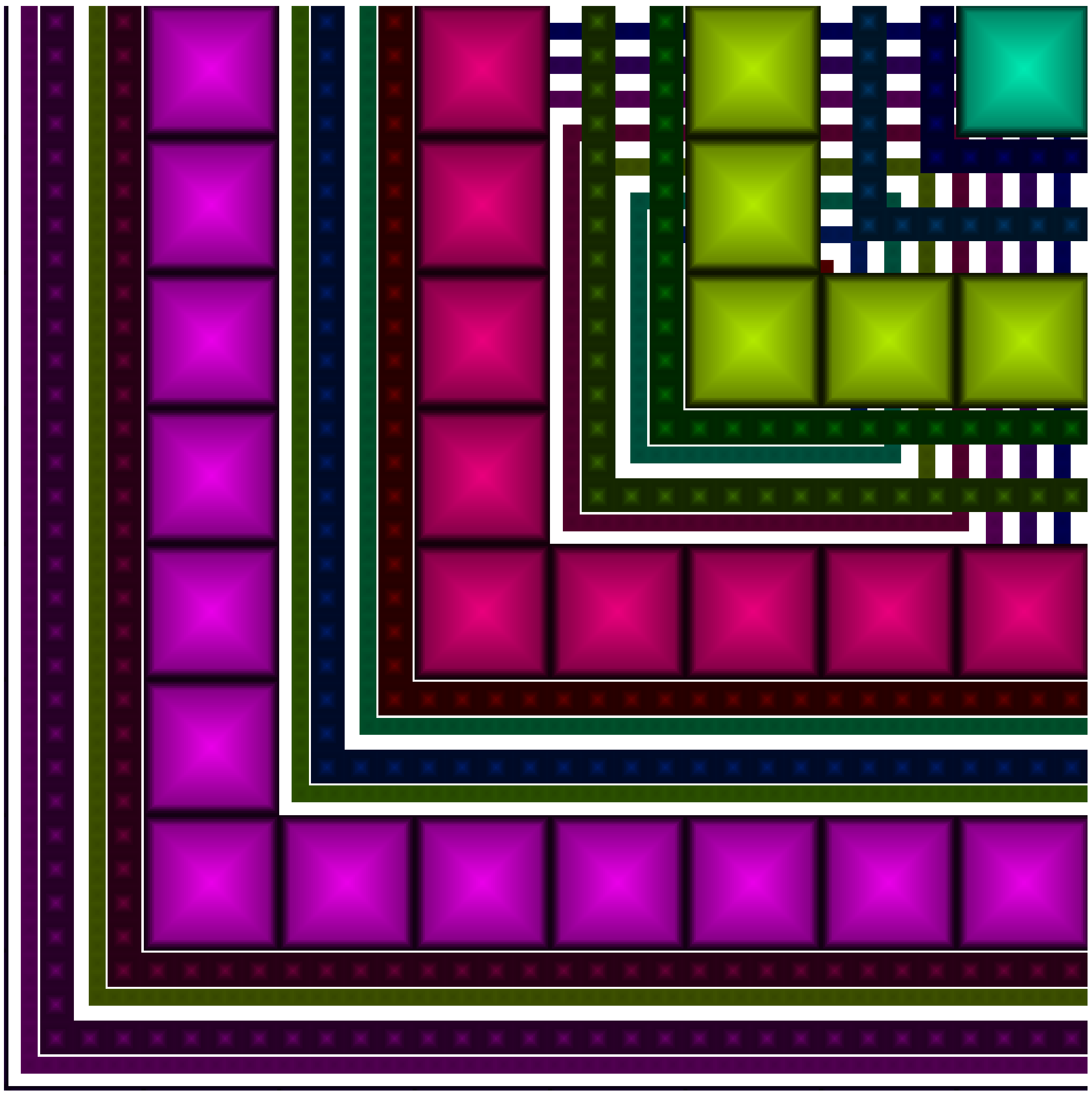
Image above was created using my 2017 golang+SFML prototype. While the results look nice, the underlying data structure is a sparse heap-allocated quad tree, while the current AAC2020 tech uses a dense GPU-allocated texture to store the quadtree information.



```
>>DRA NEXVEW
::NEXVEW: OK
>>DRA NEXDEP
::NEXDEP: OK
?>>
```

This is a screenshot of the 2017 dragon editor. I also coded the terminal below because I am NOT a UI person (despite all my UI industry experience) and a command line tends to have a much better return on investment than a UI. The command line works by typing the name of the subsystem first, and then issuing a command. That command is then passed to the correct sub-system and interpreted by the sub system. Here is the video of the editor in action.

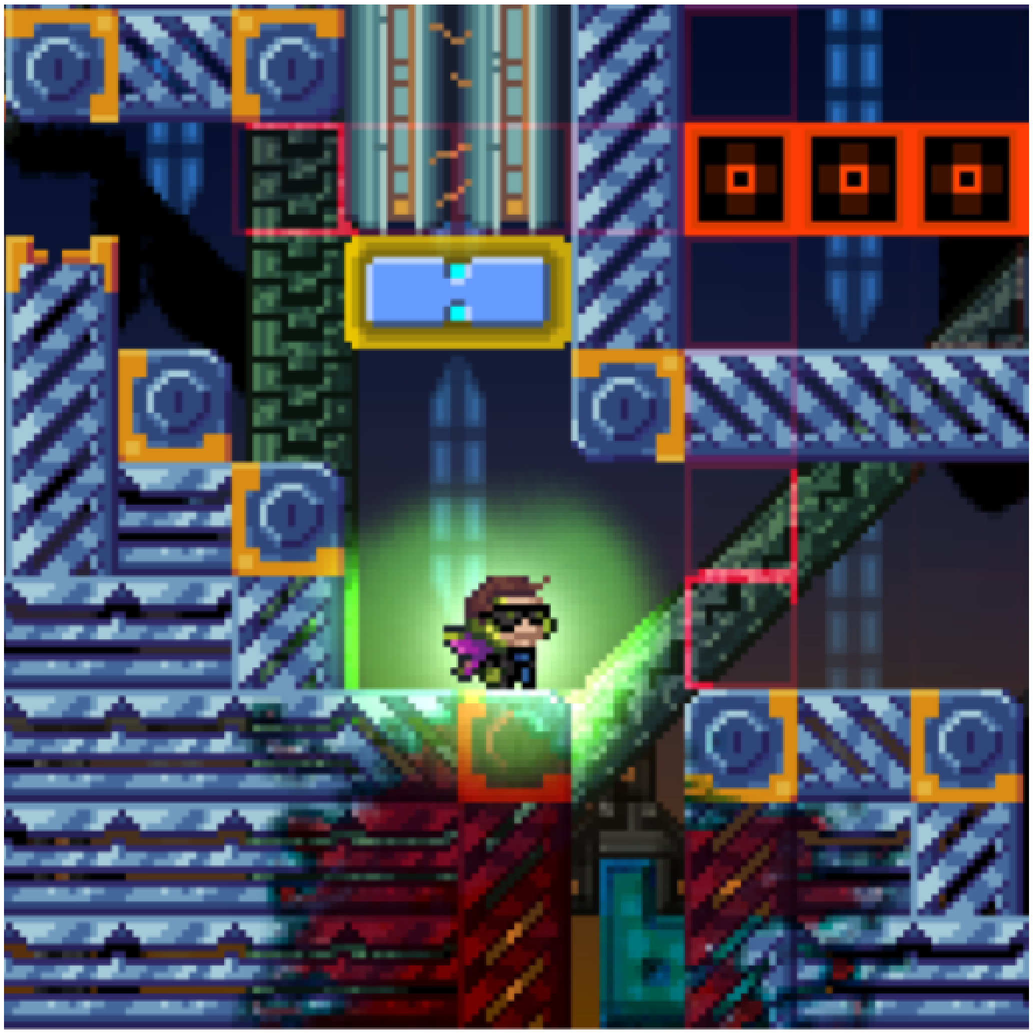
[https://www.youtube.com/watch?v=uv601dRt56Q]



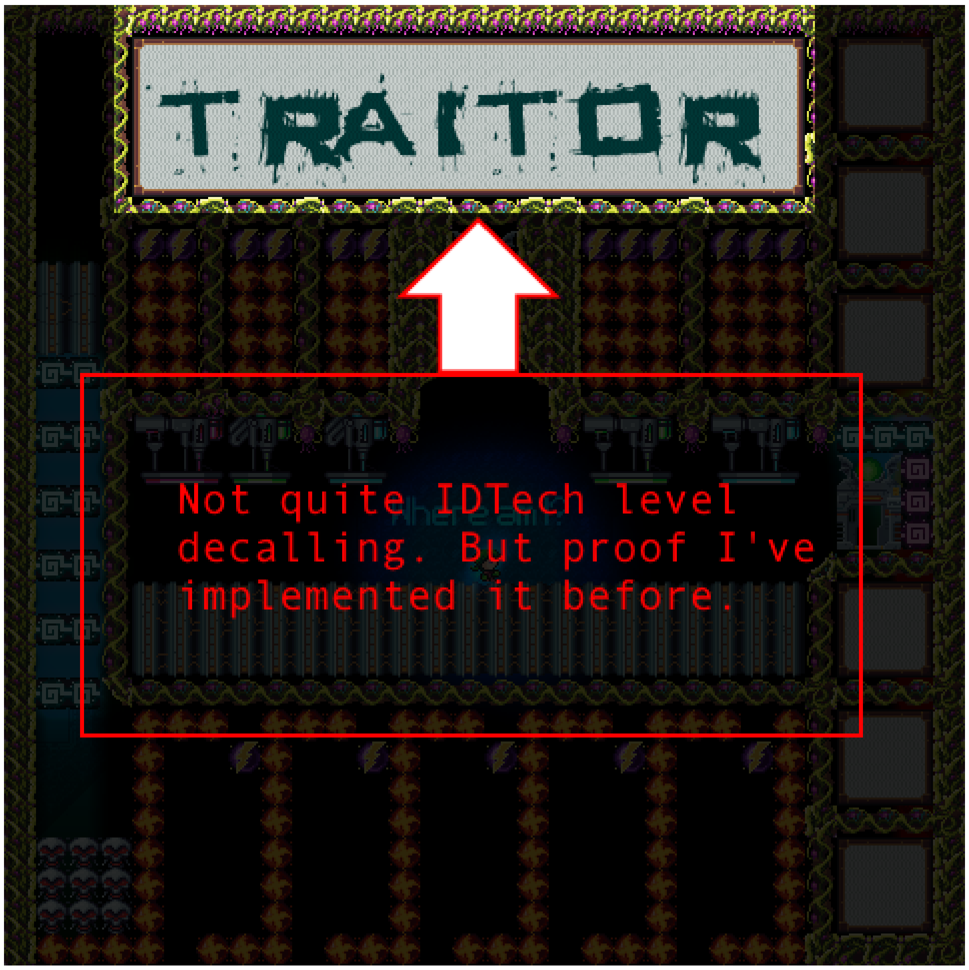
Multiple tile sizes can be used both with an auset (auto tile set) and within the design of a level map. The levelmap data can also easily be packed into a dense GPU-allocated quad-tree structure for maximum memory efficiency. The image seen above is a parallax test doing said thing. You can try it out yourself. It's a proof of concept, so the code is pretty messy, and the UI sucks. But at least I have proof that I know what I am talking about. :)

`[    https://d3m0.herokuapp.com/for_andrew/FRACTAL_STORAGE.html    ]`

11 DECAL_LAYER (See The Blood ?)



Decals can be part of the implementation of the DAMAGE_MAP effects. For example, instead of blood splatter decals, there can be charring decals for fire damaged tiles.


TRAITOR
Not quite IDTech level decalling. But proof I've implemented it before.

— Example from Atomic Alice 1 stampped onto "billboard" tiles —

The decal system is something I eventually want to implement, but for the time being, is outside of the scope of this iteration. I haven't yet contemplated what the underlying GPU-friendly data structure would look like for the decal information in a level-set or tile-set. But hopefully some ideas will come to me as I implement the parts of the auto-tile format that I do have clear ideas on.

I would like decals to also be in a tile format. This particular tile format storing normal map information among other things to alter the topology of the map.

12 ENVIRONMENT_PREFABS

Basically a mini-tilemap that can be clone stamped into the level design. Nothing ultra unique here. Just an aspect of polish that could boost productivity.

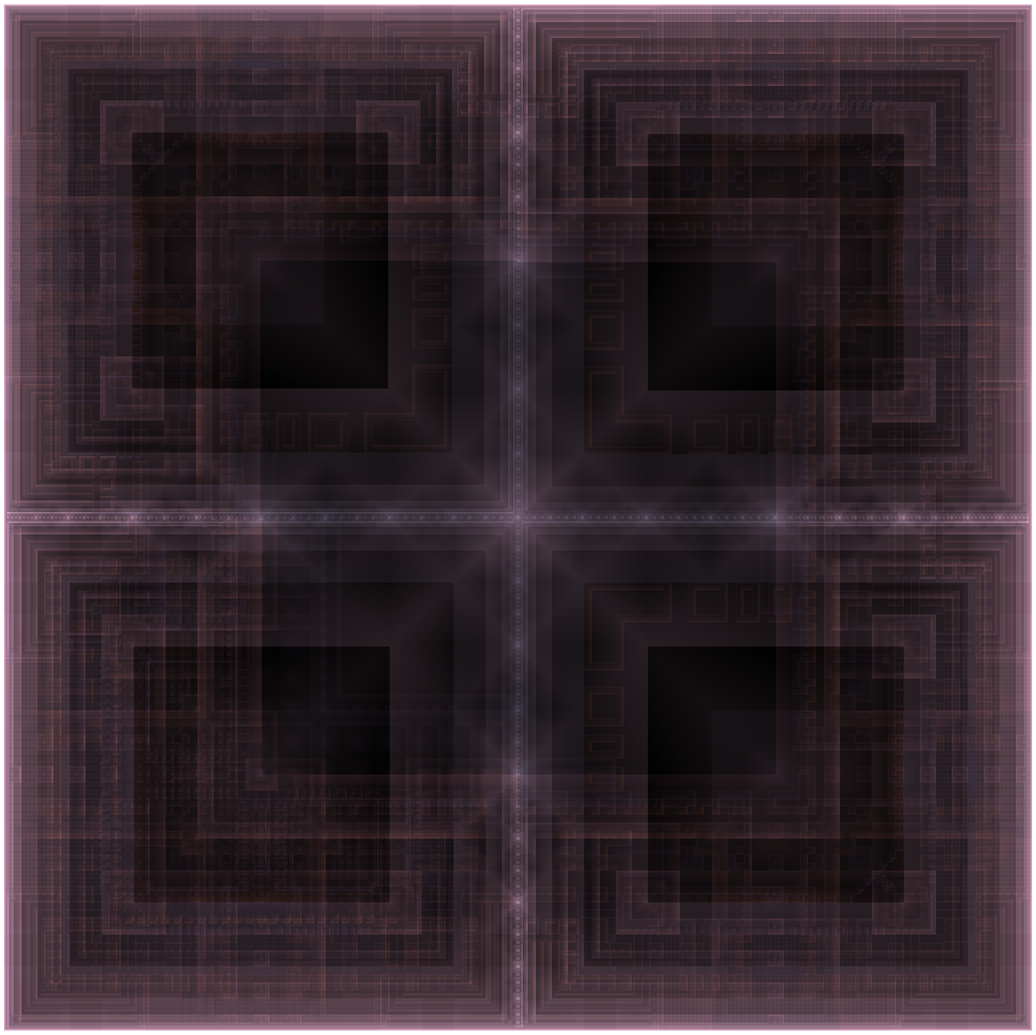
13 NORMAL_MAP_COMPOSITING

The hyperpixels of terminal auto tiles could contain normal mapping information. When using a multi-layered tilemap, we could composite those normals together to get interesting high resolution 2.5 dimensional topology.

14 PING_PONG_BUFFERS

By taking two GPU-allocated buffers, we can cache auto-tiling solution maps for each nesting depth. We only need two buffers because the solution to the next depth always comes from the solution in the previous depth. Most of the details of how I think I can pull this off are already in the 6 section.

15 FOURSET



A FOURSET is a terminal auto tile set composed of only 4 sub tiles.
It consists of four corner designs:

- T_L / TOP_LEF : Top Left Corner Sub-Tile Design
- T_R / TOP_RIG : Top Right Corner Sub-Tile Design
- B_L / BOT_LEF : Bottom Left Corner Sub-Tile Design
- B_R / BOT_RIG : Bottom Right Corner Sub-Tile Design

These corner designs are used to create wang-style auto tiling. The missing 12 tiles are generated on the fly by taking cross sectional slices of the auto-tile set, the cross sectional slices are as follows:

```
Summary: All 16 sub-tiles of auset(auto_tile_set)
         can be created by selecting FOUR(4) of
         the CANVAS_USER_VIEW partitions and merging
         them together.

         The Obvious ones need no shuffling:

         TOP_LEF  |  TOP_RIG
        [00] [01] | [02] [03]
        [04] [05] | [06] [07]
        ----------|----------
        [08] [09] | [10] [11]
        [12] [03] | [14] [15]
         BOT_LEF  |  BOT_RIG


+-----------------------------------------------------+
|<---- CANVAS_USER_VIEW --->|      | NEIGHBOR_BITTABLE |
+--==--+--==--+--==--+--==--+ ---  +---------+---------+
|      |      |      |      |  ^   |   _X_   |  _Y_    |
|  00  |  01  |  02  |  03  |  |   |         |         |
|      |      |      |      |  |   | _A_ _B_ | _C_ _D_ |
+--==--+--==--+--==--+--==--+  |   | Min Max | Min Max |
|      |      |      |      |  |   | LEF RIG | TOP BOT |
|  04  |  05  |  06  |  07  |  |   +---------+---------|
|      |      |      |      | 256  | EXAMPLE:          |
+--==--+--==--+--==--+--==--+  |   | _0_ _1_ | _0_ _1_ |
|      |      |      |      | PEB  |       (TOP)       |
|  08  |  09  |  10  |  11  |  |   |        [0]        |
|      |      |      |      |  |   |(RIG)[0][T][1](LEF)|
+--==--+--==--+--==--+--==--+  |   |        [1]        |
|      |      |      |      |  |   |       (BOT)       |
|  12  |  13  |  14  |  15  |  |   |  T==( 0101 )==(5 )|
|      |      |      |      |  V   |  t==( T_L Corner )|
+--==--+--==--+--==--+--==--+ ---  +-------------------+
                 _T_ : Tile        |       (TOP)       |
  _____________  _A_ : 1000        |        [C]        |
 |    [_C_]    | _B_ : 0100        |(RIG)[A][T][B](LEF)|
 [_A_][_T_][_B_] _C_ : 0010        |        [D]        |
 |____[_D_]____| _D_ : 0001        |_______(BOT)_______|

  _____________       _______       _____________
 |    [   ]    |_00_ |       | _01_|    [   ]    |
 [   ][_T_][   ]0000 [NON|T_C] 0001[   ][_T_][   ]
 |____[___]____|ABCD |_______| ABCD|____[///]____|

  _____________       _______       _____________
 |    [///]    |_02_ |       | _03_|    [///]    |
 [   ][_T_][   ]0010 [B_C|V_P] 0011[   ][_T_][   ]
 |____[___]____|ABCD |_______| ABCD|____[///]____|

  -------------       -------      -------------
 |    [   ]    |_04_ |       | _05_|    [   ]    |
 [   ][_T_][///]0100 [L_C|T_L] 0101[   ][_T_][///]
 |____[___]____|ABCD |_______| ABCD|____[///]____|

  -------------       -------      -------------
 |    [///]    |_06_ |       | _07_|    [///]    |
 [   ][_T_][///]0110 [B_L|3_L] 0111[   ][_T_][///]
 |____[___]____|ABCD |_______| ABCD|____[///]____|

  -------------       -------      -------------
 |    [   ]    |_08_ |       | _09_|    [   ]    |
 [///][_T_][   ]1000 [R_C|T_R] 1001[///][_T_][   ]
 |____[___]____|ABCD |_______| ABCD|____[///]____|

  -------------       -------      -------------
 |    [///]    |_10_ |       | _11_|    [///]    |
 [///][_T_][   ]1010 [B_R|3_R] 1011[///][_T_][   ]
 |____[___]____|ABCD |_______| ABCD|____[///]____|

  -------------       -------      -------------
 |    [   ]    |_12_ |       | _13_|    [   ]    |
 [///][_T_][///]1100 [H_P|3_T] 1101[///][_T_][///]
 |____[___]____|ABCD |_______| ABCD|____[///]____|

  -------------       -------      -------------
 |    [///]    |_14_ |       | _15_|    [///]    |
 [///][_T_][///]1110 [3_B|ALL] 1111[///][_T_][///]
 |____[___]____|ABCD |_______| ABCD|____[///]____|

 NON: Non_Way (NONe Touching Tile     ) [00]
 T_C: Top_Cap (Top_Cap                ) [01]
 B_C: Bot_Cap (Bottom_Cap             ) [02]
 V_P: Ver_Pip (Vertical_Pipe          ) [03]
 L_C: Lef_Cap (Left_Cap               ) [04]
 T_L: Top_Lef (Top_Left(Corner)       ) [05]
 B_L: Bot_Lef (Bottom_Left(Corner)    ) [06]
 3_L: W_3_Lef ([3]way: [L]eft-edged   ) [07]
 R_C: Rig_Cap (Right_Cap              ) [08]
 T_R: Top_Rig (Top_Right(Corner)      ) [09]
 B_R: Bot_Rig (Bottom_Right(Corner)   ) [10]
 3_R: W_3_Rig ([3]way: [R]ight-edged  ) [11]
 H_P: Hor_Pip (Horizontal_Pipe        ) [12]
 3_T: W_3_Top ([3]way: [T]op-edged    ) [13]
 3_B: W_3_Bot ([3]way: [B]ottom-edged ) [14]
 ALL: All_Way (All_Ways(Touching)     ) [15]
```

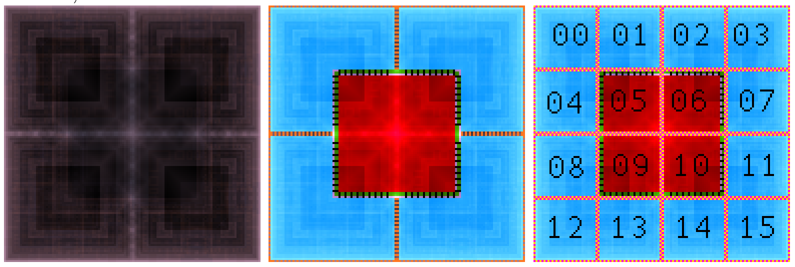
```
+-----------------------------------------------------+
| How all 16 sub-tiles of auset(auto_tile_set) are     |
| generated by merging sub-blocks of the                 |
| CANVAS_USER_VIEW diagram                                |
+----------------------------+----------------------------+
|#00               +--+--+   |#01               +--+--+   |
|          NON     |00|03|   |          T_C     |00|03|   |
|   [T]    =0000=  +--+--+   |   [T]    =0001=  +--+--+   |
|                  |12|15|   |   [ ]            |04|07|   |
|                  +--+--+   |                  +--+--+   |
+----------------------------+----------------------------+
|#02               +--+--+   |#03               +--+--+   |
|   [ ]    B_C     |08|11|   |   [ ]    V_P     |04|07|   |
|   [T]    =0010=  +--+--+   |   [T]    =0011=  +--+--+   |
|                  |12|15|   |   [ ]            |08|11|   |
|                  +--+--+   |                  +--+--+   |
+----------------------------+----------------------------+
|#04               +--+--+   |#05               +--+--+   |
|          L_C     |00|01|   |          T_L     |00|01|   |
|   [T][ ] =0100=  +--+--+   |   [T][ ] =0101=  +--+--+   |
|                  |12|13|   |   [ ]            |04|05|   |
|                  +--+--+   |                  +--+--+   |
+----------------------------+----------------------------+
|#06               +--+--+   |#07               +--+--+   |
|   [ ]    B_L     |08|09|   |   [ ]    3_L     |04|06|   |
|   [T][ ] =0110=  +--+--+   |   [T][ ] =0111=  +--+--+   |
|                  |12|13|   |   [ ]            |08|10|   |
|                  +--+--+   |                  +--+--+   |
+----------------------------+----------------------------+
|#08               +--+--+   |#09               +--+--+   |
|          R_C     |02|03|   |          T_R     |02|03|   |
|[ ][T]    =1000=  +--+--+   |[ ][T]    =1001=  +--+--+   |
|                  |14|15|   |   [ ]            |06|07|   |
|                  +--+--+   |                  +--+--+   |
+----------------------------+----------------------------+
|#10               +--+--+   |#11               +--+--+   |
|   [ ]    B_R     |10|11|   |   [ ]    3_R     |05|07|   |
|[ ][T]    =1010=  +--+--+   |[ ][T]    =1011=  +--+--+   |
|                  |14|15|   |   [ ]            |09|11|   |
|                  +--+--+   |                  +--+--+   |
+----------------------------+----------------------------+
|#12               +--+--+   |#13               +--+--+   |
|          H_P     |01|02|   |          3_T     |01|02|   |
|[ ][T][ ] =1100=  +--+--+   |[ ][T][ ] =1101=  +--+--+   |
|                  |13|14|   |   [ ]            |09|10|   |
|                  +--+--+   |                  +--+--+   |
+----------------------------+----------------------------+
|#14               +--+--+   |#15               +--+--+   |
|    [ ]      3_B    |05|06|     |    [ ]      ALL    |05|06|     |
|[ ][T][ ] =1110= +--+--+     |[ ][T][ ] =1111= +--+--+     |
|                   |13|14|     |    [ ]            |09|10|     |
|                   +--+--+     |                   +--+--+     |
+-----------------------------+-----------------------------+
```

A terminal auto tile set (auset) is an auset who's hyperpixels cannot be rendered using auto tiles. It is the deepest nested tileset within a fractal auto tile. The terminal auto tile set is much more high resolution than the non-terminal auto tile sets, so that the terminal auto tile sets can behave like a [base-material / base-texture]. These base textures can come pre-loaded into the engine, and allow users to get started quickly without having to create one of these information dense tile types themselves. Because of the information density of these tiles, I would eventually like to implement a mirrored drawing tool to help automate the process. SEE[refmirroringtool]

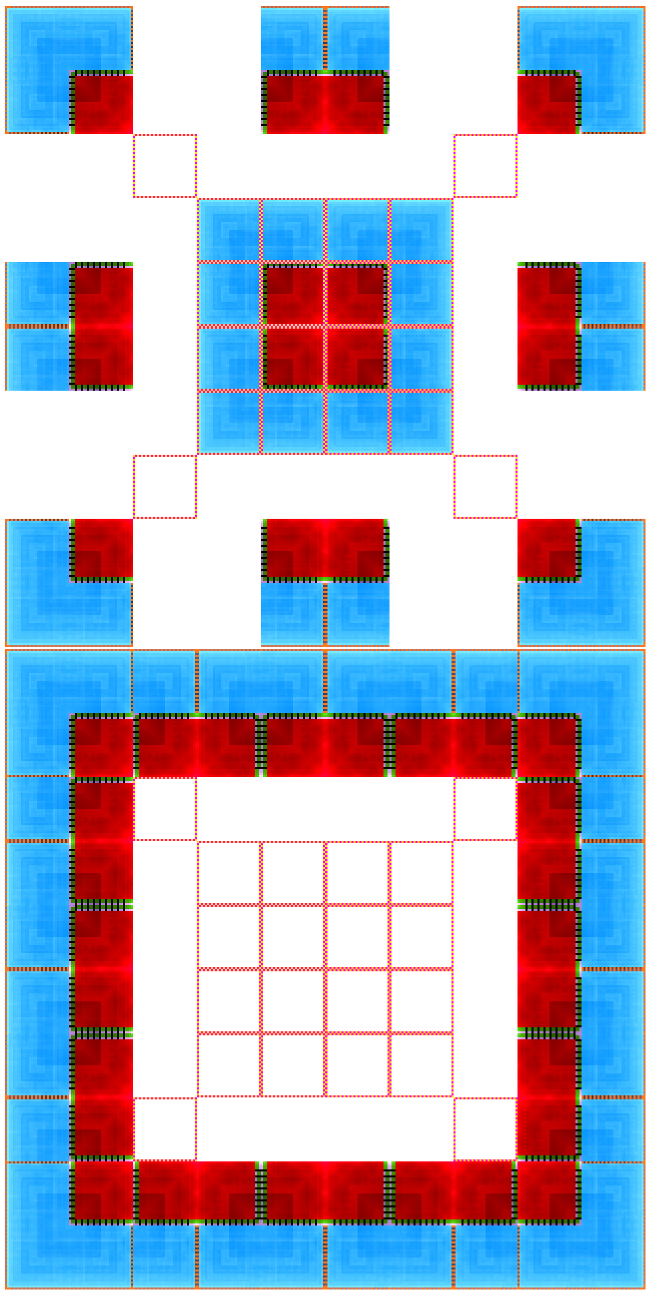
Because we only have 4 sub-tiles to an auto tile set instead of the typical 16, we remove much redundancy from the art asset creation pipeline, speed up artist productivity, and minimize the amount of memory needed to be stored on the GPU.

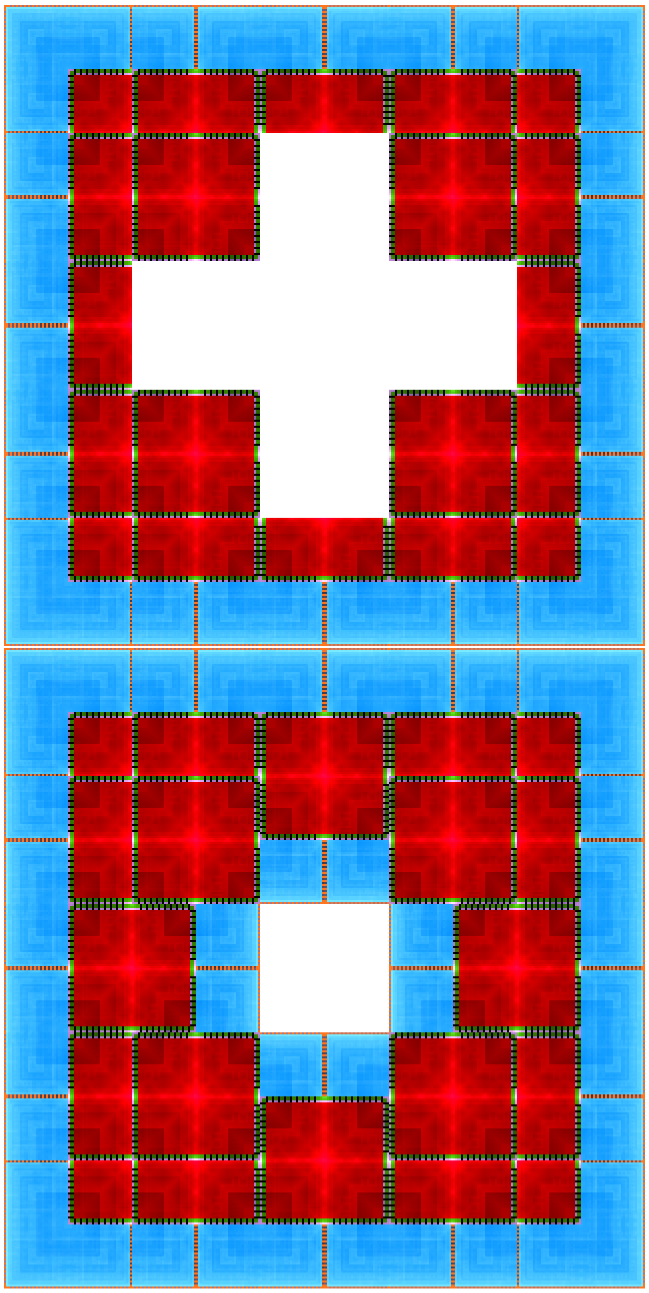
Let's do a visualization of the ascii diagram to use the forset to procedurally generate a 5x5 tilemap with a [donut / ring]shape drawn with the FOURSET data:

First, lets colorize and number to make this easier to understand:



Now lets animate through the steps of creating a 5x5 autotiled map with 1 tile missing in the center. (A donut shape)

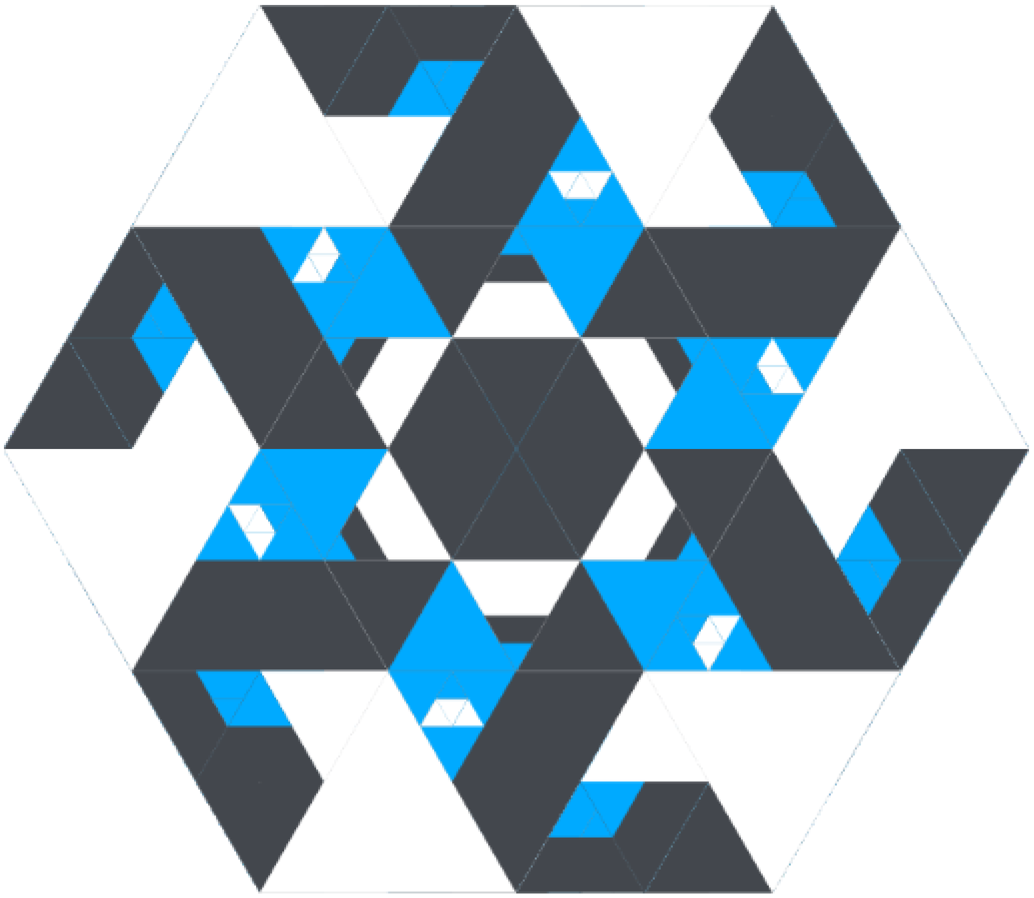




Another benifit of the choice of sub-tiles used in a FOURSET is that they are bilateally symmetric to each other on both the X & Y axis, allowing for the use of a mirroring brush tool to further boost productivity and make it FUN to create auto tile graphics that look AWESOME. For what I mean by this, check out my snowflake maker application, and imagine it being applied to a square, (rather than triangular+hexagonal) tilemap.

[https://d3m0.herokuapp.com/not_inf/nexif_app.html]

16 MIRRORING_TOOL



[https://d3m0.herokuapp.com/not_inf/nexif_app.html]

As alluded to earlier, I would like to implement a mirroring brush system to the final terminal auto tile set editor to enhance both productivity and enjoyment within the asset creation pipeline. Simply by randomizing the brush tile-size and color, and mirroring it on horizontal, vertical, and 45 degree axes, we can output rather interesting designs easily and enjoyably.

17 FOCUSED_FEATURE_SUBSET

"Better HALF a game than a HALF-ASSED game" -hell if I know who said this
And better a SMALL well polished tool than an over ambitious project.
I have a lot of goals for this system, but I want to keep the scope small.
It is better to create a small , compact, well-defined and well implemented fractal auto tiling spec than to try to do EVERYTHING I have listed in this document. I will be focusing on the FOURSET and rendering it on a tilemap. This way, even if I decide to do a re-write of the engine, the current software as it stands will be a good asset creation tool to create tiles in the fractal auto tiling format.

So basically, I'd like to make "MS Paint , but fractal"