5-2018

TAMScript - High Level Programming Interface for the abstract Tile Assembly Model

Perry Mills
University of Arkansas, Fayetteville

TAMScript:
High Level Programming Interface for the abstract Tile Assembly Model

An Undergraduate Honors College Thesis

in the

Department of Computer Science and Computer Engineering
College of Engineering
University of Arkansas
Fayetteville, AR

by

Perry William Mills

TAMScript

High Level Programming Interface for the abstract Tile Assembly Model

Perry Mills * Matthew J. Patitz †

Abstract

This paper describes a programming interface, TAMScript, for use with the PyTAS simulator. The interface allows for the dynamic generation of tile types as the simulation progresses, with the goal of reducing complexity for researchers. This paper begins with an introduction to the PyTAS software and a description of the 3D model which it simulates. Next, the changes made to support a dynamic generation scheme are detailed, and some of the potential benefits of this scheme are outlined. Then several of the example scripts which have been written using the TAMScript interface are reviewed. Finally, the potential for future research is discussed, laying out one intended approach for use with the interface.

1 Introduction

PyTAS is a Python-based Tile Assembly Model simulator currently under development. It is intended to serve as a successor to the ISU TAS software which was developed in C++ [1] and most recently released in August 2013. PyTAS currently simulates the abstract Tile Assembly Model in two or three dimensions.

The PyTAS simulator is designed to be used with the purpose-built .tds and .tdp input file formats. The tile types to be used in an assembly are described in the former. The latter defines the position and type of tiles in the seed assembly. These files are written in plain text with a strict format to allow the user to define, read, and modify tile sets with a simple text editor. Below is an example definition of a single tile type under this format.

```
1  TILENAME MainCounterSeed0
2  LABEL
3  NORTHBIND 1
4  NORTHLABEL 1_l_a
5  EASTBIND 0
6  EASTLABEL
7  SOUTHBIND 0
8  SOUTHLABEL
9  WESTBIND 2
10 WESTLABEL Left1
11 UPBIND 2
12 UPLABEL L1TurnCommand
13 DOWNBIND 0
14 DOWNLABEL
15 TILECOLOR Blue
16 CREATE
```

While this is useful for defining a simple assembly which requires only a few tile types, it becomes unwieldy as the user defines more complex operations. Assemblies which perform calculations or simulations

*Department of Computer Science and Computer Engineering, University of Arkansas, Fayetteville, AR, USA pwm001@email.uark.edu

†Department of Computer Science and Computer Engineering, University of Arkansas, Fayetteville, AR, USA patitz@uark.edu This author's research was supported in part by National Science Foundation Grant CCF-1422152 and CAREER-1553166.

can require hundreds of tile types. At that point it becomes difficult to maintain a coherent understanding of the relationships between tile types and infeasible to navigate the resulting definition file. To manage this complexity, a user can write scripts (using the programming language of his choice) which create these .tds and .tdp files as output.

It is important to note that design of an assembly under this process is constructed only as an input to the simulator. The simulator does not produce information to be used in the assembly descriptions. Consequently, if an assembly conducts a simulation, the description must account for all possible states - this can result in an explosion of tile types which must be designed. Further, if an assembly would include any nondeterministic behavior, the description must similarly include tile types to handle all possible outcomes.

1.1 Formal description of the 3D abstract Tile Assembly Model

This section gives a formal definition of the 3-dimensional version of the abstract Tile Assembly Model (3DaTAM), which is the natural extension of the (2-dimensional) abstract Tile Assembly Model (aTAM) [4]. For readers unfamiliar with the aTAM, [3] gives an excellent introduction to the model, here we provide a brief introduction.

Fix an alphabet Σ. Σ^* is the set of finite strings over Σ. $\mathbb{Z}$, $\mathbb{Z}^+$, and $\mathbb{N}$ denote the set of integers, positive integers, and nonnegative integers, respectively. Given $V \subseteq \mathbb{Z}^3$, the *full grid graph* of V is the undirected graph $G_V^{\text{f}} = (V, E)$, and for all $\vec{v_1} = (x_1, y_1, z_1), \vec{v_2} = (x_2, y_2, z_2) \in V$, $\{\vec{v_1}, \vec{v_2}\} \in E \iff \|\vec{v_1} - \vec{v_2}\| = 1$; i.e., if and only if $\vec{v_1}$ and $\vec{v_2}$ are adjacent in the 3-dimensional integer Cartesian space.

A *tile type* is a tuple $t \in (\Sigma^* \times \mathbb{N})^6$; e.g., a unit cube with six sides listed in some standardized order, each side having a *glue* $g \in \Sigma^* \times \mathbb{N}$ consisting of a finite string *label* and nonnegative integer *strength*. We assume a finite set of tile types, but an infinite number of copies of each tile type, each copy referred to as a *tile*. A tile set T is a set of tile types.

A *configuration* is a (possibly empty) arrangement of tiles on the integer lattice $\mathbb{Z}^3$, i.e., a partial function $\alpha : \mathbb{Z}^3 \dashrightarrow T$. Two adjacent tiles in a configuration *interact*, or are *attached*, if the glues on their abutting sides are equal (in both label and strength) and have positive strength. Each configuration α induces a *binding graph* G_α^{b}, a grid graph whose vertices are positions occupied by tiles, according to α, with an edge between two vertices if the tiles at those vertices interact. An *assembly* is a connected non-empty configuration, i.e., a partial function $\alpha : \mathbb{Z}^3 \dashrightarrow T$ such that $G_{\text{dom }\alpha}^{\text{f}}$ is connected and $\text{dom } \alpha \neq \varnothing$. The *shape* $S_\alpha \subseteq \mathbb{Z}^3$ of α is $\text{dom } \alpha$.

Given $\tau \in \mathbb{Z}^+$, α is *τ-stable* if every cut of G_α^{b} has weight at least τ, where the weight of an edge is the strength of the glue it represents. When τ is clear from context, we say α is *stable*. Given two assemblies α, β, we say α is a *subassembly* of β, and we write $\alpha \sqsubseteq \beta$, if $S_\alpha \subseteq S_\beta$ and, for all points $p \in S_\alpha$, $\alpha(p) = \beta(p)$.

A *tile assembly system* (TAS) is a triple $\mathcal{T} = (T, \sigma, \tau)$, where T is a finite set of tile types, $\sigma : \mathbb{Z}^3 \dashrightarrow T$ is the finite, τ-stable, *seed assembly*, and $\tau \in \mathbb{Z}^+$ is the *temperature*. Given two τ-stable assemblies α, β, we write $\alpha \to_1^{\mathcal{T}} \beta$ if $\alpha \sqsubseteq \beta$ and $|S_\beta \setminus S_\alpha| = 1$. In this case we say α *produces* β *in one step*. If $\alpha \to_1^{\mathcal{T}} \beta$, $S_\beta \setminus S_\alpha = \{p\}$, and $t = \beta(p)$, we write $\beta = \alpha + (p \mapsto t)$. The *frontier* of α is the set $\partial^{\mathcal{T}}\alpha = \bigcup_{\alpha \to_1^{\mathcal{T}} \beta} S_\beta \setminus S_\alpha$, the set of empty locations at which a tile could stably attach to α. The *t-frontier* $\partial_t^{\mathcal{T}}\alpha \subseteq \partial^{\mathcal{T}}\alpha$ of α is the set $\left\{ p \in \partial^{\mathcal{T}}\alpha \;\middle|\; \alpha \to_1^{\mathcal{T}} \beta \text{ and } \beta(p) = t \right\}$.

Let $\mathcal{A}^T$ denote the set of all assemblies of tiles from T, and let $\mathcal{A}_{<\infty}^T$ denote the set of finite assemblies of tiles from T. A sequence of $k \in \mathbb{Z}^+ \cup \{\infty\}$ assemblies $\alpha_0, \alpha_1, \ldots$ over $\mathcal{A}^T$ is an *assembly sequence* if, for all $1 \leq i < k$, $\alpha_{i-1} \to_1^{\mathcal{T}} \alpha_i$. The *result* of an assembly sequence is the unique limiting assembly (for a finite sequence, this is the final assembly in the sequence).

We write $\alpha \to^{\mathcal{T}} \beta$, and we say α *produces* β (in 0 or more steps) if there is an assembly sequence $\alpha_0, \alpha_1, \ldots, \alpha_{k-1}$ of length $k = |S_\beta \setminus S_\alpha| + 1$ such that (1) $\alpha = \alpha_0$, (2) $S_\beta = \bigcup_{0 \leq i < k} S_{\alpha_i}$, and (3) for all $0 \leq i < k$, $\alpha_i \sqsubseteq \beta$. If k is finite then it is routine to verify that $\beta = \alpha_{k-1}$. We say α is *producible* if $\sigma \to^{\mathcal{T}} \alpha$, and we write $\mathcal{A}[\mathcal{T}]$ to denote the set of producible assemblies in $\mathcal{T}$. The relation $\to^{\mathcal{T}}$ is a partial order on $\mathcal{A}[\mathcal{T}]$. An assembly α is *terminal* if α is τ-stable and $\partial^{\mathcal{T}}\alpha = \varnothing$. We write $\mathcal{A}_\square[\mathcal{T}] \subseteq \mathcal{A}[\mathcal{T}]$ to denote the set of producible, terminal assemblies. If $|\mathcal{A}_\square[\mathcal{T}]| = 1$ then $\mathcal{T}$ is said to be *directed*. When $\mathcal{T}$ is clear from context, we may omit $\mathcal{T}$ from the notation above and instead write $\to_1$, $\to$, $\partial\alpha$, etc.

2 Dynamic Generation

TAMScript ties the design of tile types more closely to the simulation. The simulator can make calls to the user's script as the assembly grows. At each new frontier location, the script is queried for potential tile types which could be placed there. The user defines the script to respond to these queries, based on the glues of the tiles which are adjacent to the frontier location in question. Assemblies designed in this manner can take a more interactive stance — tile types are generated dynamically at run-time based on previous tile placements.

2.1 Reducing memory requirements

One potential benefit of this system is in reducing the number of tile types which must be tracked. Current hardware is capable of simulating very large assemblies using hundreds or thousands of tile types, instantiating several orders of magnitude more individual tiles. There is some benefit to tracking only tile types which will be used, reducing memory requirements; however, the primary advantage of such a reduction is in reducing complexity for the user.

For a project to use two large libraries of tiles cooperatively, the resulting tile set must include connective tiles which make use of glues from each combination of tiles which can interact between the two systems. As the size and number of cooperating libraries increases, the total number of unique tile types explodes. Each of these unique tile types might conceivably be used. It is often the case, however, that only a small portion are required by a simulation up to some arbitrary number of steps. As a result, it is far more space-efficient to construct tile types as they are needed.

In non-deterministic systems, this effect is amplified. In such a system, resulting assemblies can differ greatly in structure and complexity based on arbitrary decisions. The static scheme needs to account for all possible outcomes prior to simulation, and in any given run many of those tile types will not be used. If a decision made early in a simulation would rule out the instantiation of some tile types at any point in the future, the dynamic generation scheme simply does not need to generate and track those tiles.

2.2 Accessibility

Rather than writing to a text file which is read into the simulator, TAMScript allows the user to instantiate and operate on the Python classes which are directly used by PyTAS. PyTAS implements data structures for tiles and glues, with support for two or three dimensions. These built-in types provide a common foundation from which user scripts can build, enabling script libraries to be more easily shared or expanded.

The internal data structures of use to TAMScript include:

- Constants used to enumerate the cardinal directions within PyTAS. For example, the eastern direction can be indicated using the variable **E**, the variable **East**, or simply the integer **1**.
- Arrays used to convert between the above constants and their respective string equivalents ("East").
- **Glue** class. A glue requires a `strength` value (a nonnegative integer) and a `label` (a string). The built-in methods for copying and comparing objects of this class are routinely useful in the design of concise scripts.
- **NeighborGlues** class. A NeighborGlues instance represents a frontier location. It contains Glue instances to represent each of the faces which potentially expose glues to the location. An instance of this class is passed as a parameter to the user script's *GetTileTypes()* function. This is the primary avenue through which information can pass from the simulator to the user's scripts.
- **PyTASTileType** class. A PyTASTileType instance contains Glue instances to represent each of its faces. In two dimensions, there are four of these, while three-dimensional tiles have 6 glues. Scripts may routinely access these glues through the **sides**[] dictionary attribute. In conjunction with the directional constants, each glue can be referenced directly. For example, the eastern glue of a tile

type named `MyTile` can be referenced at `MyTile.sides[E]`. The class also includes attributes used to visually represent the tile, such as color, which are not relevant to the aTAM.

2.3 Interacting with the user script

Several changes were made to the PyTAS simulation software in order to accommodate a dynamic generation scheme. Once a user has written a script for use with the simulator, it can be loaded from PyTAS' normal "Load system" dialog. The dialog allows the user to select an arbitrary file from the file system, with support for .tdp or .py files. If the file loaded is a Python script, it is imported by the interpreter and the dynamic scheme is used to determine the tile set.

Thereafter, the simulator queries the user script when new information is required. First, the system temperature and seed assembly are read from the script. Then each time the simulator encounters a new distinct frontier location, it polls the script for tile types.

User scripts must adhere to an interface in order to cooperate with PyTAS. There are two required functions the user script must implement:

- **GetSeed()** This is the function that is called by the simulator to get the definitions and locations of tiles in the seed assembly. The user script uses this function to determine the starting configuration of the simulation, with control over the position and orientation. This function returns a set of tuples, where each tuple defines a tile type and the location at which a tile of that type should be placed.
- **GetTileTypes()** This is the function that is called by the simulator as it encounters new frontier locations. It expects an instance of the NeighborGlues class as input, which the function can use to make decisions about potential candidates for the location. The function should return a set of tile types, if any, that can be placed in this frontier location. The simulator uses the set in this case, adding the tile types to its pool. If the same NeighborGlue configuration is encountered later, GetTileTypes() is not called again. Instead, the simulator draws from the stored results. For this reason, it is important that the user script return all possible candidates for a location rather than merely the first.

3 Example Usage

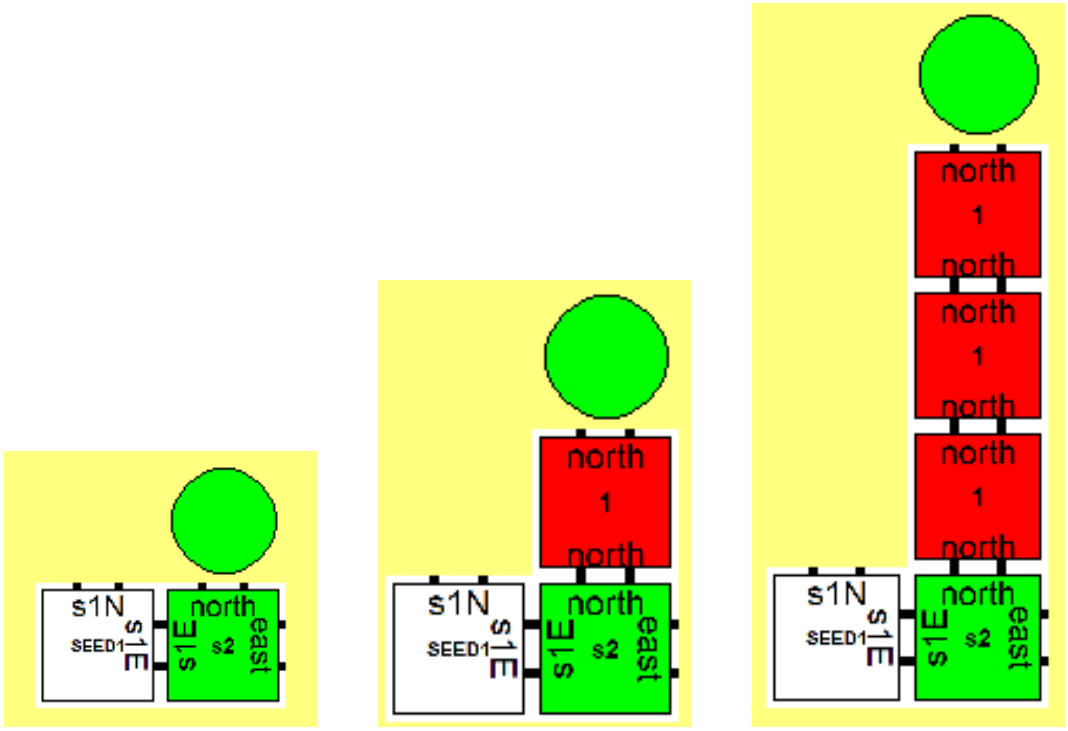
This section details several examples which make use of the simulator's new dynamic generation functionality. Appendix A lists the name and purpose of each script written as part of this research. While PyTAS is operating in two-dimensional mode, green circles are rendered to represent frontier locations. When new tiles are added to the assembly (including the seed), the simulator calls **GetTileTypes()** for each adjacent empty space. When a circle is rendered, it indicates therefore that the user script has returned one or more tile types which could fit at that position.

3.1 Base Example

The first example, **tileGeneratorBaseExample**, serves as a proof of concept as the dynamic scheme is used to define a seed of two tiles and generate a third tile. Note that the generated tile binds to itself in addition to the seed, allowing the assembly to grow to the north indefinitely. Figure 1 depicts the growth of the assembly defined by this script.

The **tileGeneratorBaseExample** assembly is generated from the following script. The script implements the two functions required by the TAMScript interface (discussed in subsection 2.3). Users intending to design assemblies using TAMScript may look to this file as a simple example of correct syntax.

```
1 from PyTASTileType import *
2
3 TEMPERATURE = 2
4
5 ## The label of the glues which will extend infinitely
6 ClimbingLabel = "north"
```



(a) Seed assembly (b) After one tile is placed (c) More tiles are placed

Figure 1: tileGeneratorBaseExample

```
7
8  #  GetSeed is the function called by the simulator to get the definitions and
9  #  locations of tiles in the seed
10 def GetSeed():
11     ## Left seed tile
12     t = PyTASTileType()
13     t.sides[N] = Glue(2, "s1N")
14     t.sides[E] = Glue(2, "s1E")
15     t.label = "SEED1"
16     t.name = "LeftSeed"
17
18     ## Right seed tile
19     r = PyTASTileType()
20     r.label = "s2"
21     r.name = "Right_Seed"
22     r.sides[E].strength = r.sides[N].strength = 2
23     r.sides[E].label = "east"
24     r.sides[N].label = ClimbingLabel
25     r.sides[W] = t.sides[E].Copy()
26     r.color = "GREEN"
27
28     return ((t,(0,0,0)),(r,(1,0,0)))
29
30 #  GetTileTypes is the function called by the simulator as it encounters
31 #  frontier locations
32 def GetTileTypes(nbrGlues):
33     t = PyTASTileType()
```

```
34      if nbrGlues.sides[S] != None:
35          if nbrGlues.sides[S].label == ClimbingLabel:
36              ## The tile to the south has the glue we want
37              t.name = "NorthClimber"
38              t.sides[N].strength = 2
39              t.sides[N].label = ClimbingLabel
40              t.sides[S] = nbrGlues.sides[S].Copy() # Match to the south
41              t.color = "RED"
42              t.label = "1"
43
44              returnSet = set()
45              returnSet.add(t)
46              return returnSet
47      else:
48          return None
```

The import statement on line 1 allows the script to make use of the data structures used internally by PyTAS, as detailed in subsection 2.2.

On line 3, the script defines the temperature at which the simulation should run. In this case, the assembly is a temperature-2 system. This means that tiles will be placed on the frontier only if a connection can be formed to adjacent tiles with a 2-strength glue or two 1-strength glues. All of the tiles placed in this example use 2-strength glues to bind to their neighbors.

In the first function, **GetSeed()**, the script defines the seed tiles (white and green). The left seed tile is defined by assigning instances of the Glue class to the sides of the tile. In this case, the tile will expose glues of strength 2 to its northern and eastern sides. The right seed tile is defined by individually assigning the strengths and labels of the glues, similar to the format used in .tds input files. To ensure the two seed tiles will bind to each other, the green tile's adjoining glue is assigned by copying the white tile's existing glue.

In the second function, **GetTileTypes()**, the script defines the dynamically generated tile. The tile is intended to be placed only to the north of of tiles which expose the "north" glue, so checks are made on lines 34 and 35 to ensure that this condition is met. If it is, the tile type is defined using the "north" glue on its north face and copying the glue on the south face (in this case, also the "north" glue).

While this example script has approximately the same length as the equivalent .tds file, the level of abstraction helps clarify the tile set's intended purpose and makes the project less brittle to changes. The relationships between tiles are more readily apparent in the TAMScript format. In this example, the label which is shared between several tile types is defined only once, as `ClimbingLabel`. As new tile types are defined, they can use this name, rather than a "magic" string, where a glue should be shared. Further, the glue copy method used in line 40 ensures that the tile type will construct a glue to match adjacent tiles, regardless of the name used for that glue.

3.2 Structures

To facilitate the writing of cooperative libraries, users may write classes which adhere to the interface described in subsection 2.3. To do so, users are free to use Python's duck typing, define their own classes, or inherit from the TAMScriptModule abstract class. Objects which inherit from the abstract class (i.e., "modules") can be verified to have the required methods via Python's **issubclass()** built-in function.

Structures are classes which implement the interface and have a shutoff switch. The shutoff switch is a Boolean which can be used to signal to other scripts when a structure has exhausted its possible outputs. Structures can define this switch to occur when all possible tile types have been generated, when some finite size has been reached, or on any other relevant condition.

The structure **StructureLine** was written as a simple example application of this interface. An object of the type **StructureLine** can be defined in a user script with a temperature, length, and direction. Additionally, the object requires the user hand off a tile which will be used to end the line (internally called finalTile). The structure will generate tiles dynamically up to one less than the length. Then, the structure

will generate the tile which completes the line from the finalTile, replacing one glue as necessary to make the connection. Finally, the structure flags itself as complete using the shutoff switch.

generateStructureLine makes use of the **StructureLine** structure as an example of this interaction, depicted in figure 2. Two lines are constructed. The first grows to the west, is two tiles long and uses glues of strength 3. The second grows from the final tile of the first, extending to the north for 5 tiles with glues of strength 2. The script defines the red, blue, and green tiles, while the structures generate the white tiles.

Figure 2: generateStructureLine

3.3 Rotation Class

When designing a set of tile types as part of a library or larger project, it is important to account for the orientation of tiles. Because individual instances of tiles cannot rotate in the simulation, often a set of tile types can construct an assembly only in one fixed orientation. If a set of tile types is intended to be used in multiple directions, rotated copies of each tile type should be constructed to account for the various other rotated orientations.

For example, a single 3-dimensional tile with 6 unique faces can be rotated to 24 distinct orientations. However, a tile with fewer unique faces has fewer distinct orientations. In order for a set of tile types to behave as intended regardless of the direction in which it assembles, the resulting set must account for all possible orientations of all tile types. This superset is often difficult to accurately construct, and the increased complexity is a burden on the user.

There is a fixed relationship between the faces of a tile. For example, the west face is always 90 degrees from the north face and the up face. Using this knowledge, a class can be constructed which can define a tile type which has been rotated from any orientation to any orientation. The **Rotate** script defines an **Orientation** class which performs this operation on arbitrary tiles. The script utilizes two approaches to performing rotations within a dynamic scheme:

- Rotate the tiles. For each of the tile types which could potentially be rotated to fit a frontier location, rotate it to each other orientation and test whether it fits. Complexity, given n tile types is $\Omega(24n) = \Omega(n)$.
- Rotate the frontier location. Construct a rotated version of the frontier location for each possible orientation. Then test each against each of the tile types in question. If a match is found, apply the reverse of the frontier location's rotation to the tiletype. Complexity, given n tile types is $\Omega(24) = \Omega(1)$.

The former method, while simpler, requires a linear number of rotations in the best case. The latter is more complex but requires only a constant number rotations in the best case. In practice, scripts tend toward the best case, making the second approach more useful for most applications.

3.4 Sierpinski Triangle example

The Sierpinski triangle has been used as a test of the potential offered by the aTAM, due to the shape's infinite, aperiodic nature. The discrete form of the shape can be weakly self-assembled using only 7 tile types [2]. These tile types consist of one seed tile, two tiles which extend along the x and y axis, and four tiles which compute the exclusive-or operation from two single-bit inputs.

To demonstrate the flexibility of the dynamic generation scheme, **generateSierpinskiTriangle** constructs the discrete triangle fractal shape. A portion of the assembly generated by the script is shown in figure 3. As in the static scheme, the seed tile and axis tiles are defined manually. However, the remaining four tiles are generated using Python's exclusive-or operator itself. As in this case, using code to define tile behavior provides a higher level of abstraction than the definition file format allows, reducing complexity for the user.

Additionally, generating tiles from a script allows higher reusability. Functions which are used throughout a project need only be written once, and they can make use of more complex logic than is feasible for the definition file format. For example, the following code governs the operation performed in the dynamic tiles. Using the below configuration, the tiles are generated to create the discrete Sierpinski triangle shown in figure 3.

```
1  # Given two bool inputs, define an output bool
2  def performOperation(inLeft, inBottom):
3      return (inLeft ^ inBottom)
4
5  # The name that will be assigned to the tile
6  def buildLabel(inLeftLabel, inBottomLabel):
7      return (inLeftLabel + " xor " + inBottomLabel)
```

If the user wished to construct tiles using another operator than exclusive-or, the change could be made in a single line. Figure 5 depicts three outcomes of this change. The change to use the negative-and operation is made below.

```
5  # Construct dynamic tiles using NAND operation
6  def performOperation(inLeft, inBottom):
7      return not(inLeft and inBottom)
```

The tiles generated by this operation produce an assembly with a checkerboard appearance, as shown in figure 4.

Similarly, the function $\neg(P \rightarrow Q)$, where P and Q are the neighboring tiles used as inputs, can be represented using its logical equivalent in Python. This produces an assembly with a horizontally striped appearance. The resulting assembly is shown in figure 5c.

```
5  # Construct dynamic tiles using !(P -> Q) function
6  def performOperation(inLeft, inBottom):
7      return not(not(inLeft) or inBottom)
```

Given the modular nature of programming languages like Python, decisions like this one can be divided into more complex functions, such as those which produce random outcomes or which make decisions based on

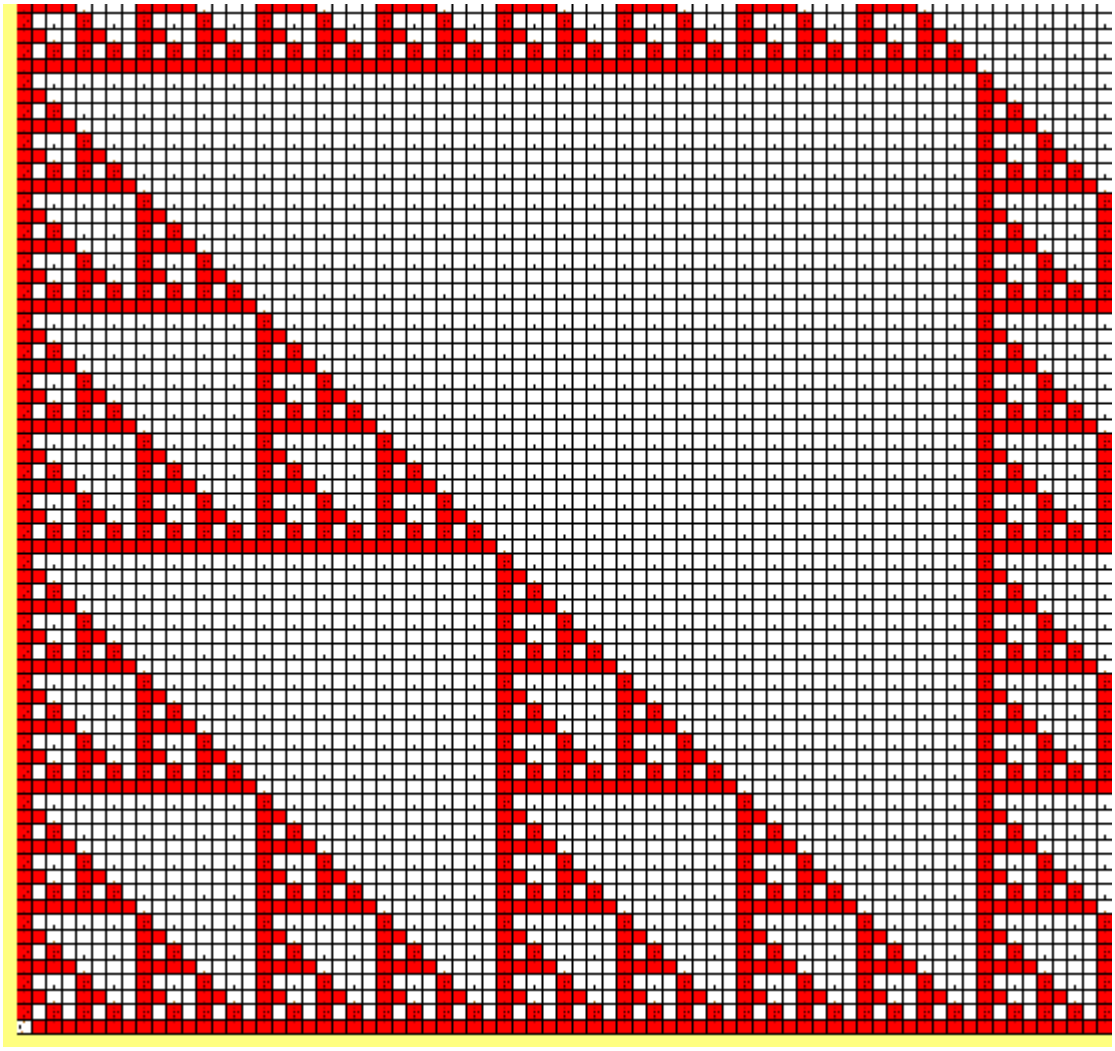


Figure 3: Bottom corner of the resulting assembly of generateSierpinskiTriangle (XOR)

previous nondeterministic tile placements. The use of high-level programming syntax to define tile behavior does allow increasingly complicated assemblies to be constructed with relative ease. The dynamic generation scheme allows the user greater flexibility in design. It should be noted, however, that this approach does not increase the power of the underlying aTAM simulation. Each of the above examples over time produces a finite number of static tiles. This tile set can be exported and used in later simulations with the exact same result.

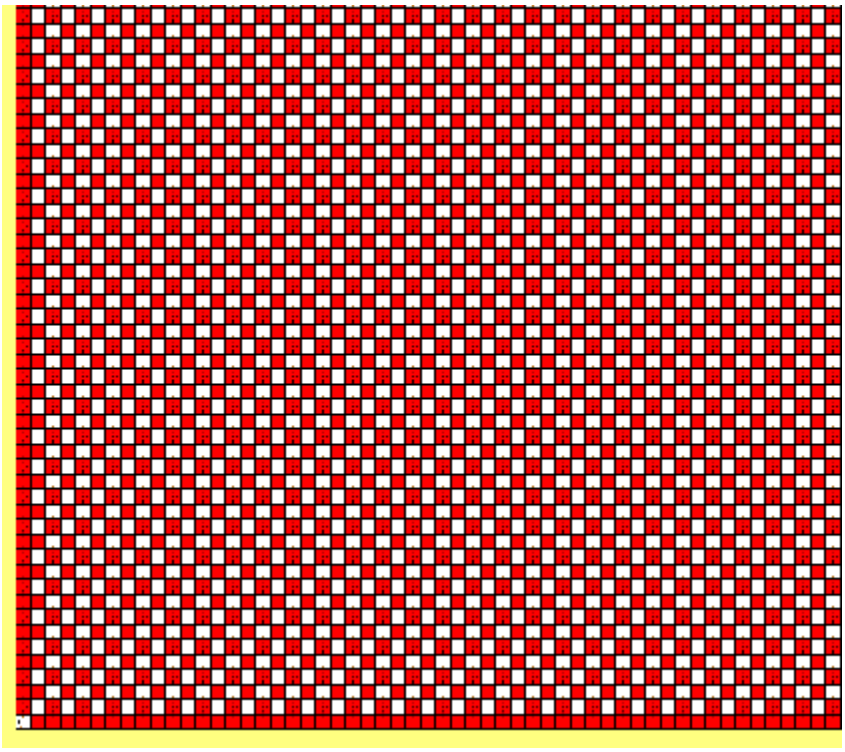
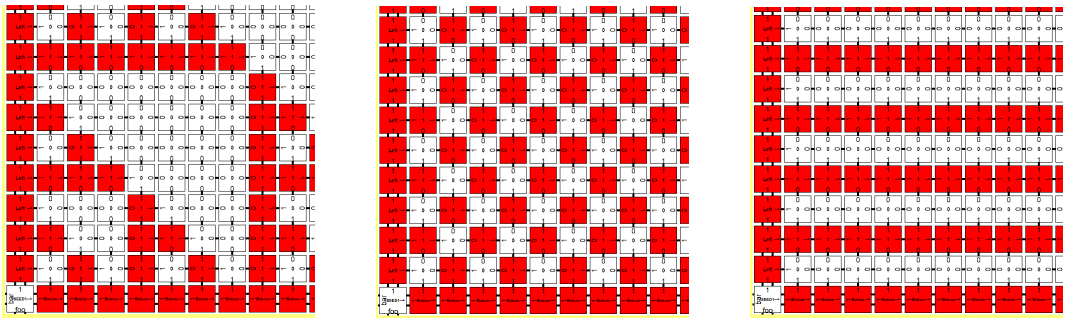


Figure 4: Bottom corner of the resulting assembly of generateSierpinskiTriangle (NAND)



(a) P XOR Q (b) P NAND Q (c) $\neg(P \rightarrow Q)$

Figure 5: Variations on generateSierpinskiTriangle

4 Expanded Usage

4.1 Genericity

The dynamic generation scheme lends itself to the writing of generic libraries. Because tiles types can be defined as the simulation progresses based on the glues of previously placed tiles, a library can adapt to fit existing tile sets by making use of templates. For example, a library which is used to construct data paths for the conveyance of information from one position in space to another does not need to account for all possible output tiles. Rather, the library can be defined to accept any tile type as input in order to generate the required assembly.

4.2 Example project hierarchy

In the course of this work, a base-line library of example scripts for use with the dynamic generation scheme has been developed. These scripts can be used and expanded upon by future PyTAS users. In particular, the interface described in subsection 2.3 allows the construction of increasingly complex libraries of tiles. The following is an example project hierarchy:

The user writes the base script, which implements **GetSeed()** and **GetTileTypes()**. The base script may utilize structures or modules, which implement the interface. The script should maintain a list of the structures which are used, enabling the user to query GetTileTypes() from each structure. The script can loop through all active structures each pass in order to remove completed structures from the list, as shown below.

```
1  def GetTileTypes(nbrGlues):
2      returnSet = set()
3
4      ## Prune list if structure has generated all of its tiles
5      for structure in StructuresList:
6          if structure.Done:
7              StructuresList.remove(structure)
8
9      ## Add new structures as necessary
10     ...
11     StructuresList.append(anotherNewStructure)
12
13     ## Check for viable tiles from other sources
14     ...
15
16     ## Check all structures for viable tiles
17     for structure in StructuresList:
18         returnSet.update(structure.GetTileTypes(nbrGlues))
19     return returnSet
```

Additionally, structures and modules can make use of other structures and modules, allowing the user to define as many levels of abstraction as he wishes. This increases the reusability of scripts and better enables users to include scripts written by other users.

References

[1] Matthew J. Patitz, *Simulation of self-assembly in the abstract tile assembly model with isu tas*, Tech. Report 1101.5151, Computing Research Repository, 2011.

[2] Matthew J. Patitz, *An introduction to tile-based self-assembly and a survey of recent results*, Natural Computing **13(2)** (2014), 195–224 (English).

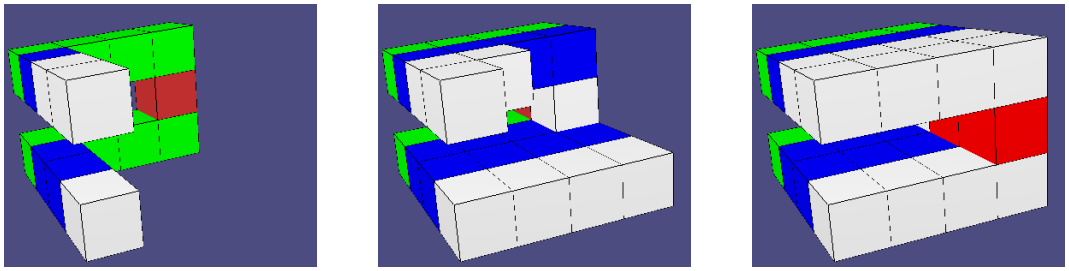
[3] Paul W. K. Rothemund and Erik Winfree, *The program-size complexity of self-assembled squares (extended abstract)*, STOC '00: Proceedings of the thirty-second annual ACM Symposium on Theory of Computing (Portland, Oregon, United States), ACM, 2000, pp. 459–468.

[4] Erik Winfree, *Algorithmic self-assembly of DNA*, Ph.D. thesis, California Institute of Technology, June 1998.

Appendix A Files

A.1 EqualsTest.py

(3D only) Example assembly which tests whether two bitstrings are equal. The two bitstrings propagate along parallel guide-rails (green), which are placed with one tile of space. When the strings reach a DifferenceBar tile (orange), the space between the bitstrings is filled with dynamic tiles in a line perpendicular to the direction of the bitstrings. When the comparison is finished, the end tile exposes a glue indicating the result of the operation. The tile is colored red, to indicate a difference has been detected, or white, to indicate the two bitstrings are equal.



(a) Initial state

(b) In Progress. No difference detected.

(c) Final state. Difference detected.

Figure 6: Comparison of "011" (top) with "001" (bottom)

A.2 generateLineSpiral.py

Example assembly which uses structureLines of increasing length to draw a square spiral.

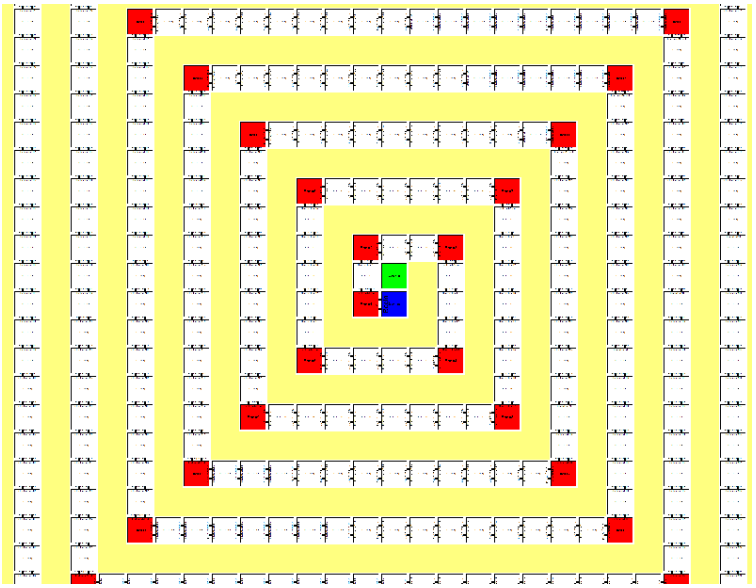


Figure 7: Center of resulting assembly of generateLineSpiral

A.3 generateSierpinskiTriangle.py

Example assembly which dynamically generates the discrete Sierpinski triangle structure using the XOR operator. The operator to be used, as well as the composition of the axis walls, can be changed easily to produce other structures.

A.4 generateStructureLineExample.py

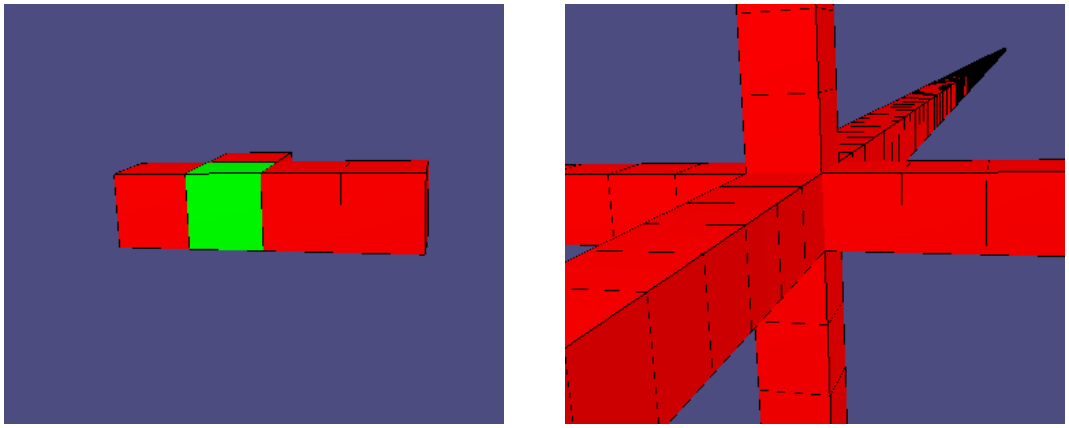
Example assembly which demonstrates use of two structureLines to place a user-defined tile at a position offset from the seed.

A.5 Rotate.py

Example assembly which defines and uses the Orientation class. The Orientation class can be used to rotate a tile or frontier location from any orientation to any other orientation.

The script generates a Knucklebone structure after defining only two tile types. The first is defined as the seed, which exposes copies of the same glue to all sides. The second tile (dynamically generated) exposes this glue in only one direction. Rotation is applied to produce each of the distinct orientations of the dynamic tile.

The script demonstrates two methods of rotating dynamic tiles. In the first, the script rotates the dynamic tile to each of its 6 distinct orientations to test against the frontier location. In the second, the frontier location is rotated to test against the dynamic tile.



(a) Assembly after 4 rotated tile placements (b) Assembly after many more tile placements

Figure 8: Knucklebone structure generated using Rotate script

A.6 StructureLine.py

A structure defined to generate a line one-tile wide which extends in a cardinal direction. The user specifies a tile to be placed in the last position.

Constructor:

```
__init__(self,temperature,direction,length,initialGlue,finalTile,is3d=False)
```

- `temperature` An integer determining the strength of glues connecting tiles internal to the structure.
- `direction` A cardinal direction constant determining the direction the line should grow.

- `length` A positive integer determining the final length of the line, including the finalTile placement. That is, the line is constructed using (`length` − 1) tiles, then the finalTile is placed.
- `initialGlue` A **Glue** instance determining the glue from which this structure should begin. A structure will grow from any and all positions where the exposed glue is compatible with the intended direction.
- `finalTile` A **PyTASTileType** instance which is placed in the final position of the structure.
- `is3d` A Boolean indicating whether the simulation is being conducted in 3 dimensions.

The number of unique tile types generated by the structure is based on the length of the line. For example, the third tile in a 5-long line is marked 3|5, distinguishing it from the third tile in a 6-long line. If multiple 5-long lines with the same characteristics are present, they use the same tile types.

A.7 TAMScriptModule.py

Abstract class which TAMScript Modules may implement. Subclasses must respond to *GetTileTypes()* calls (even if only with a *super()* call). The *GetSeed()* method is optional.

A.8 tileGeneratorBaseExample.py

Example assembly which serves as a proof of concept of the dynamic generation system. A set of seed tiles is loaded from the script. Thereafter, a single tile type is generated which grows north from the seed.