Simulating Digital Circuits Using Wang Cubes

Mathew Kuthur James[1]

[1]David Cheriton School of CS, University of Waterloo

Abstract

This report describes a method by which Wang cubes tiling 3D space can be used to simulate digital circuits, and presents a web-app to run such simulations. It also describes an efficient algorithm used to compute adjacent planes along the z-axis, along with the software architecture and technology used to create the web-app. Common elements of digital circuits (such as clocks, wires, logic gates, *et cetera*), along with the challenges of instantiating them using Wang cubes, are discussed here.

Introduction

The primary goal of this project is to develop software to design and tile R^3 with Wang cubes, which are the 3D analogue for Wang tiles. Since 2D Wang tiles are Turing-complete, 3D Wang tiles possesses the expressive power required to simulate a Von Neumann machine built out of simple circuit elements like clocks, wires, logic-gates and flip-flops.

Movement along the z-axis is treated as equivalent to stepping forward and backward in time, which is treated as a distinct quantity. At each stage, the current plane contains a finite number of tiles, which are used to compute the preceding and succeeding plane in real time. The algorithm to do so is described in this report. A user manual [1], the web-app [2], its source [3] and files describing common circuit elements [4] are available online.

Architecture

A ”tile type” refers to an entity that describes the adjacency rules which determine the legality of a tiling. A tile is a combination of a tile type with a position vector (a vector being a 2-tuple of real numbers). All tiles have positions which are aligned with the grid - each number in the 2-tuple is an integer.

A tiling is a collection of tiles in the x-y plane. The function mapping the tiles in a plane tiling to the set of tile types is surjective, but not necessarily injective. The function mapping the tiles in a tiling to the set of all possible positions is surjective, and not injective (since there are an infinite possible number of positions, and the number of tiles is finite).

Instead of using colors, the adjacency rules for tiles are enforced using strings. A significant advantage of using unicode strings instead of RGB 3-tuples is that a string is self-documenting. Another advantage is that strings are easy to tell apart from each other, unlike close shades of colors. Furthermore, the system is designed such that one can use it even if completely color-blind, albeit not as efficiently as if one can perceive colors perfectly.

A Wang file consists of a global set of tile types, a main tiling (on which most of the work is done), a cached tiling (used to implement resets while editing), a collection of sub tilings (which can be copied from and pasted into the main plane tiling) and a global color map that maps strings to colors.

The entire state of the data is stored in the Wang file, and the Wang file can be exported and imported via JSON [15] serialization. This process presents the following challenges:

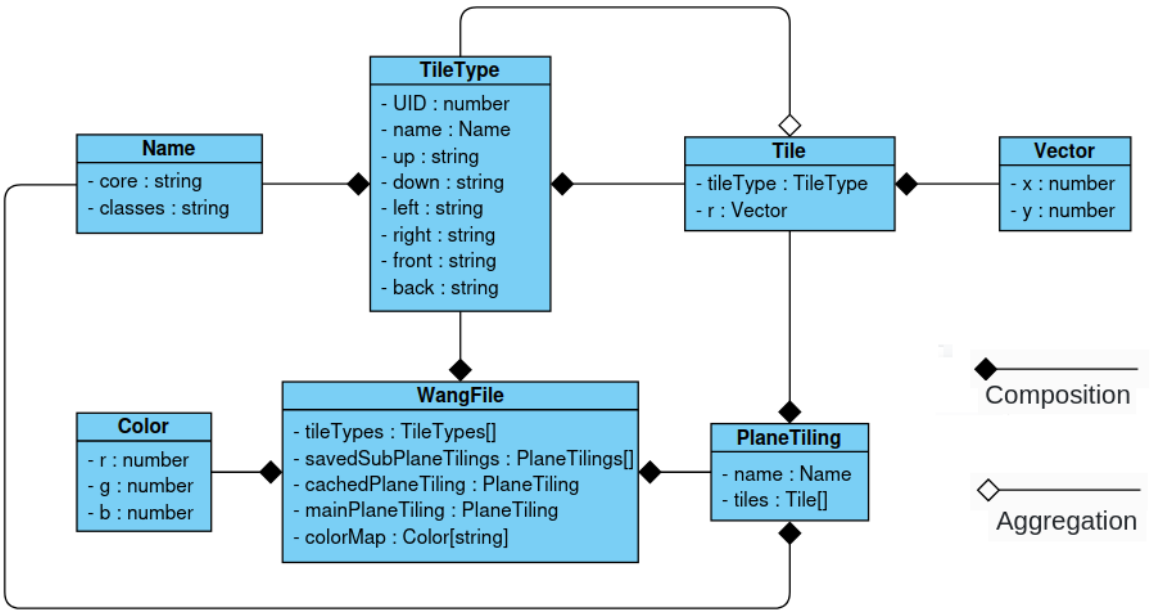


Figure 1: *A UML class diagram describing the ontology underlying the system*

- JSON objects do not preserve references, so if multiple tiles have references to the same tile type object, deserialization creates two distinct but identical tile type objects. This is resolved using a readonly static UID associated with the tile type class, which is used to collapse references to identical tile types into a single object.
- JSON objects carry only data and do not record non-static functions. This is resolved by replacing all non-static functions with static functions bound to the class, that takes an additional argument in place of ”this” that the non-static function operates on.

Algorithm

Given n tiles and m tile-types, a naïve backtracking algorithm operates with a time complexity of $O(m^n)$, which is unsuitable for Wang files which use many different tile types. Thus, a more efficient algorithm is used.

The computation of the neighboring planes is done using PRUNE (Propagation and Reduction Using Neighborhood Elimination), which is a modification of the wave-function collapse algorithm [5]. PRUNE requires that the current plane can have exactly one possible adjacent plane. Tilings that are non-deterministic are not supported, and the exact point of where the non-determinism occurs is pointed out by PRUNE.

The high-level idea behind PRUNE is:

- Each tile has a set of possible tile types that its neighbor in the next plane can have.
- Given a pair of tiles, the set of possible tile types of one tile can be reduced by considering the set of possible tile types of the other tile.
- A pass consists of applying a reduction to every pair of adjacent tiles in the tiling.
- PRUNE makes multiple passes, and terminates when a reduction-free pass is made.

- If a pass does not allow any reductions to be made, no further passes result in any reductions (since none of the sets of possible tile types are changed, making the result of every subsequent pass the same)

An analysis of the algorithm's correctness and efficiency is presented below:

- The maximum cardinality of the set of possible tile types is m.
- Let I be the sum of cardinalities of the set of possible tile types of every tile. Since there are n tiles, $I \leq mn$
- Let I_k be the value of I at the k^{th} pass.
- Since no pass expands the possible tile types, $I_{k+1} \leq I_k$
- If a pass does result in a reduction, at least one set of possible tile-types is reduced in size. Thus, $I_{k+1} < I_k$
- Thus, the maximum number of passes before termination is mn. Since each pass takes $O(n)$ time, the overall complexity of the algorithm is $O(mn^2)$

Thus, PRUNE uses repeated passes and computes the plane in $O(mn^2)$ time, which is significantly faster than the naïve backtracking approach. However, since the data structures that store tiles do not store the neighbors of the tiles, generating a neighbor-graph requires $O(n^2)$ preprocessing time and $O(n)$ space, which is the overhead for the current implementation of PRUNE. Maintaining a live neighbor-graph with each edit allows the preprocessing time to disappear entirely, when amortization is accounted for.

Technology

The software was designed to be a web-app, since this would allow it to be used by anyone with a browser. This also saves a lot of developer effort that would otherwise be spent in packaging the software for various operating systems and platforms.

The following technology was used in the web-app:

- HTML5 [8], which is the industry standard for web development, used for the high-level structure of the web-app
- NPM [13] (Node package manager), used to install and maintain dependencies
- SASS [9] (Syntactically Awesome StyleSheets), which is a superset of CSS with additional syntactic sugar, used for styling the elements of the web-app
- TypeScript [10], which is a superset of JavaScript with a typing system, used for the describing the algorithms, data structures and UI interactions of the web-app
- three.js [17], a package used to render objects in 3D using a WebGL context, without having to write any WebGL code directly
- Vite [11], used to manage the compilation to CSS/JavaScript, minification [7], tree-shaking [6] and bundling the web-app into a small, portable form
- Vitepress [16], a static site generator [12] used to generate the user manual from markdown [14] source files
- GitHub actions, used as a CI/CD solution to build and publish the web-app incrementally
- GitHub pages, used to host the web-app online

In addition, the icons for the web-app were sourced from Google Material [18] icons, downloaded as SVG images.

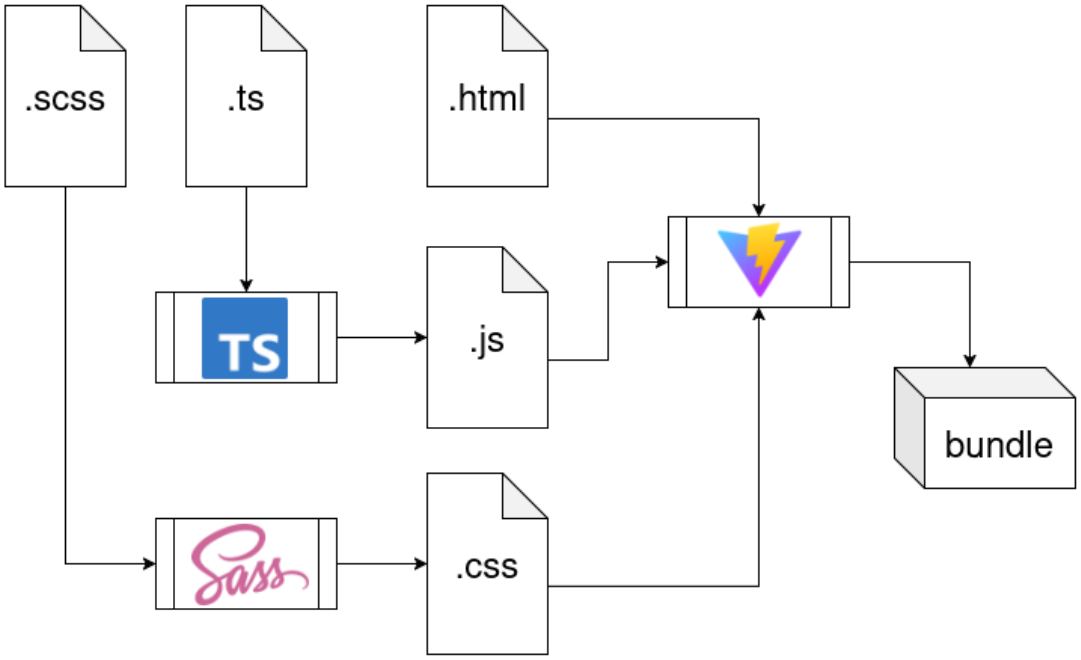


Figure 2: *The build pipeline, from the source files to the bundle*

Results

A reference implementation for a two-state standalone clock is described below:

Table 1: *Adjacency rules for standalone-clock-0*

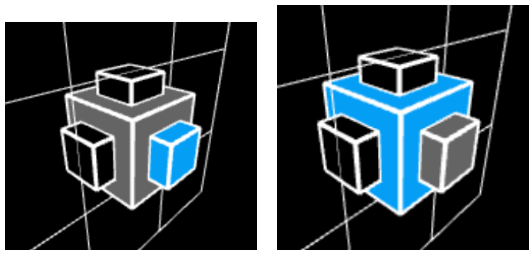
string	value
up	anything
down	anything
left	anything
right	anything
front	clock-1
back	clock-0

Table 2: *Adjacency rules for standalone-clock-1*

string	value
up	anything
down	anything
left	anything
right	anything
front	clock-0
back	clock-1

The standalone clock consists of two states - clock-0 and clock-1. Each state only allows the other state to be placed next to it in the plane behind and in front of it along the z-axis. When moving along the z-axis,

the tile at that location alternates between clock-0 and clock-1, which simulates a clock pulse.



standalone-clock-0 *standalone-clock-1*

Figure 3: *standalone-clock: "clock-0" is mapped to grey and "clock-1" is mapped to cyan*

Clocks that have different frequencies are implemented by adding more states - for instance, a clock that runs at half the base frequency has 4 states - clock-00, clock-01, clock-10 and clock-11. Other elements that don't have internal states that depend directly on time are given adjacency rules such that they propagate themselves.

Table 3: *Adjacency rules for horizontal-wire-0*

string	value
up	anything
down	anything
left	0
right	0
front	wire
back	wire

Table 4: *Adjacency rules for horizontal-wire-1*

string	value
up	anything
down	anything
left	anything
right	anything
front	wire
back	wire

For all such elements, rotations and reflections require different tile types, though the relationships between the adjacency rules of rotated and reflected tile types enables the web app to generate them automatically, without requiring the user to manually enter all the rules.

Future work

The following circuits can be added to the project:

- D-flip-flops, constructed either as a standalone tile or from two D-latches. This requires a consistent translation of the notion of rising and falling edges to a system with discrete time, which requires research into variants of flip-flops
- shift-registers, built from T-flip-flops. This allows one to build clocks of arbitrary frequencies without creating a large number of clock tiles to represent each state

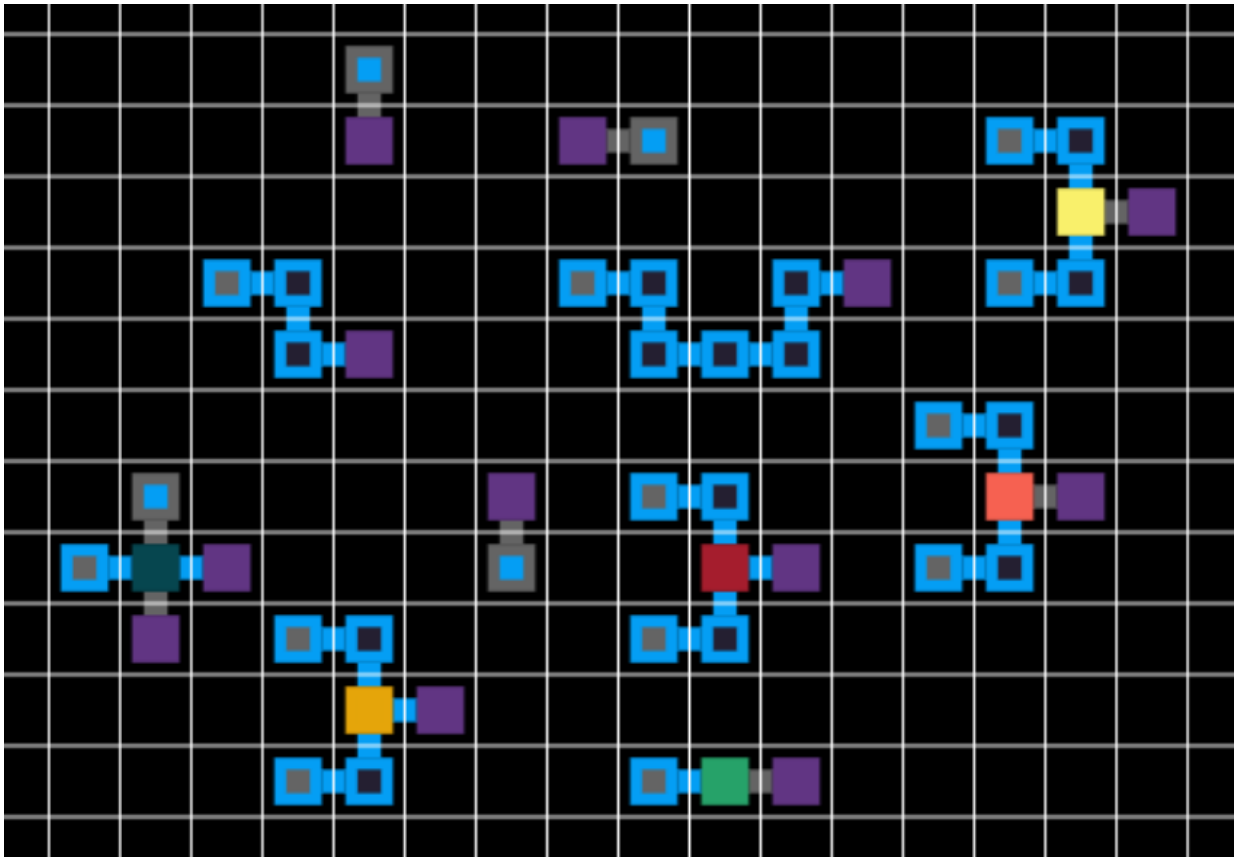


Figure 4: *Logic gates: AND (red), OR (deep-yellow) NOT (green), NAND (orange), NOR (light-yellow), wires (grey/cyan), junctions (deep cyan), clocks, and sinks (purple)*

- Single-tile wires with more than two states, which can be used to transmit signals effectively, when coupled with multiplexers and demultiplexers
- A 7-segment display, using a $3x5$ grid of screen tiles, with each tile having 10 states, though making 150 manually using the current UI is tedious

The following UI improvements can be made to the project:

- An implementation of the pause/play functionality, which is currently left unimplemented due to difficulties involved with managing the animation framerate with the system clock (for security reasons, most browsers do not allow scripts to access the exact time on the system clock)
- A custom console, to display errors and warnings at different levels of verbosity (at present, all such messages show up in the console provided with the developer tools of the browser)
- An input console to allow the programmatic manipulation of the Wang file, to automate tedious tasks
- Collapsible classes in the search results, making a tree-like structure that's easier to navigate than scrolling over all the results
- A graphical representation of the illegality of the tiling (for instance, using red circles to point out areas where the adjacency rules are violated)
- Integrating the main editor with three.js, allowing the user to manipulate the main tiling using orbit controls (currently unimplemented due to problems with mixing dynamic resizing with the WebGL API of three.js)

LLM disclosure

No LLMs were used in the generation of this report or the user manual for the web-app. However, when writing the code for the web-app, the involvement of LLMs is difficult to quantify and disentangle from the code written by a human, for the following reasons:

- Development often involves copying and pasting code from online forums such as StackOverflow. The source of the content on such forums cannot be guaranteed to be written by a human, unless accompanied with a timestamp preceding the release of popular LLMs like ChatGPT.
- Since search engines integrate AI assistance with their service, searching for a topic results in the user being exposed to an AI-generated answer that appears before every search result. This could subtly influence the thought-process of the user, guiding them toward certain implementations and solutions.
- Popular code-editors like Copilot now integrate agentic coding assistants powered by LLMs, which autocomplete code snippets and generate functions from comments. Though no such editor has been used in the development of this web-app, it is often the case that the coder does not have direct control over what editor they are allowed to use.

Acknowledgements

The author would like to acknowledge the attendees of CS898 Spring 2025, for their feedback on features to implement in the web-app.

References

[1] User manual, GitHub pages.
https://mathewkj2048.github.io/Wang-cube-circuit-simulator-user-manual

[2] Web app, GitHub pages. https://mathewkj2048.github.io/Wang-cube-circuit-simulator/

[3] Web app source code, GitHub https://github.com/MathewKJ2048/Wang-cube-circuit-simulator

[4] Clocks, wires and logic gates, GitHub https://github.com/MathewKJ2048/Wang-cube-circuit-simulator/tree/main/Wang-files

[5] Merrel, Paul. "Example-based model synthesis" *Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games*, 2007, pp.105–112 https://doi.org/10.1145/1230100.1230119

[6] Tree-shaking, MDN web docs. https://developer.mozilla.org/en-US/docs/Glossary/Tree_shaking

[7] Minification, MDN web docs. https://developer.mozilla.org/en-US/docs/Glossary/Minification

[8] HTML5, MDN web docs. https://developer.mozilla.org/en-US/docs/Glossary/HTML5

[9] Documentation, sass-lang. https://sass-lang.com/documentation/

[10] TypeScript, MDN web docs. https://developer.mozilla.org/en-US/docs/Glossary/TypeScript

[11] Vite. https://vite.dev/

[12] Static Site Generator, MDN web docs. https://developer.mozilla.org/en-US/docs/Glossary/SSG

[13] Node Package Manager. https://docs.npmjs.com/about-npm

[14] How to write in Markdown, MDN web docs. https://developer.mozilla.org/en-US/docs/MDN/Writing_guidelines/Howto/Markdown_in_MDN

[15] JavaScript Object Notation. https://www.json.org/json-en.html

[16] Vitepress https://vitepress.dev/

[17] three.js https://threejs.org/

[18] Google Material icons. https://fonts.google.com/icons