WFCpp Design Document

Authors:

- Jose A. Ramos (jar2333)
- Jiayang Hu (jh4192)
- Ricardo Garcia (rg3112)

Reviewers:

- Bjarne Stroustrup
- Ivy Marre Basseches
- Navjot Singh

Functional description

The library provides classes for accomplishing image synthesis using Max Gumin's Wave Function Collapse[1] (WFC) algorithm. This is an algorithm which takes in a set of discrete square tiles, a set of adjacency constraints which determine which tiles can be next to other tiles, and then randomly generates a new image. This choice of algorithm constrains the kinds of images which can be generated to those composed of discrete square tiles, arranged in a grid to form a bigger square image. This library aims to provide an easily understood API for loading an image file from disk, extraction of tiles and constraints from the image, and for generating new images using that data. The core functionality of the WFC algorithm is encapsulated in its own class (the **Solver**), with a public API available to the end user or the other components of the library. These other components (the **Extractor** and **Synthesizer**) focus on image synthesis specifically, offering their own API to encapsulate manually dealing with the Solver's API and image manipulation libraries. That way, the user still has the ability to utilize the standalone WFC algorithm through the Solver, but can easily utilize it for image synthesis specifically through the Extractor and Synthesizer APIs.

[1] https://github.com/mxgmn/WaveFunctionCollapse

Solver

The Solver implements the WFC algorithm itself, providing a high-level interface to set its initial constraints, to register callback functions to be triggered at certain WFC events, and to run the algorithm on a provided Grid. The Solver—despite the encapsulation of the algorithm itself—remains the low-level component of the library, using integer keys to represent tiles in its API.

Extractor

The Extractor provides a high-level interface for specifying an image file to be split into square tiles of a given pixel length, for deriving adjacency information from these extracted tiles using color matching, and for adding these constraints to a given Solver (using the Solver's own API).

Synthesizer

The Synthesizer provides a high-level interface for exporting a Grid of tiles as an bitmap image object which may be stored into disk in multiple formats. In addition, it also supports step-by-step image generation.

Project Design

Library Design

The library is designed to maximize separation of concerns, driven by the three major components of the system described earlier: Solver, Extractor, Synthesizer. Ideally, these fully abstract away the WFC algorithm itself and any image manipulation code. To this end, we decided to not make a single-header library, as that would mean that all code would be located in the same file. This breaks separation of concerns at two layers:

- Any external libraries would need to be provided to the user, giving them unnecessarily low-level access to image manipulation libraries, and bloating the code that they would need to include in their projects.

- All of our own classes would be located in the same file, and furthermore, be included in every compilation unit that the single-header library is included in. This would unnecessarily increase compilation times for the end-user, and make navigating the codebase very difficult in the case of extension or modification.

Instead, the project was organized in a directory structure, and a build system was utilized to build all necessary targets. CMake[2] was chosen as it was the build system the authors were most familiar with. The locations of header files for all external libraries were set as private in the CMake configuration, while our own library's include directory was set to public. This gives any end-user utilizing CMake access to every header in WFCpp's include directory, but not those of other libraries we utilized. That way, the implementation details of our library are encapsulated. Additionally, the major classes each have their own compilation unit. Thus, WFCpp is compiled as a static library, which means the end-user only links against it, instead of recompiling the entire library in every one of their own compilation units.

Using CMake allows us to easily add a tester program as a build target, as well as documentation generation with Doxygen[3] and coverage report generation with LCov[4]. The external libraries used were lodepng[5] and EasyBMP[6] for loading, manipulating, and saving images of .png and .bmp formats, respectively. These libraries were added as build targets with CMake, compiled statically and linked with WFCpp. This keeps each library logically separate from WFCpp.

In the future, C++20 modules would be used to further limit the amount of unnecessary compilation from header inclusion.

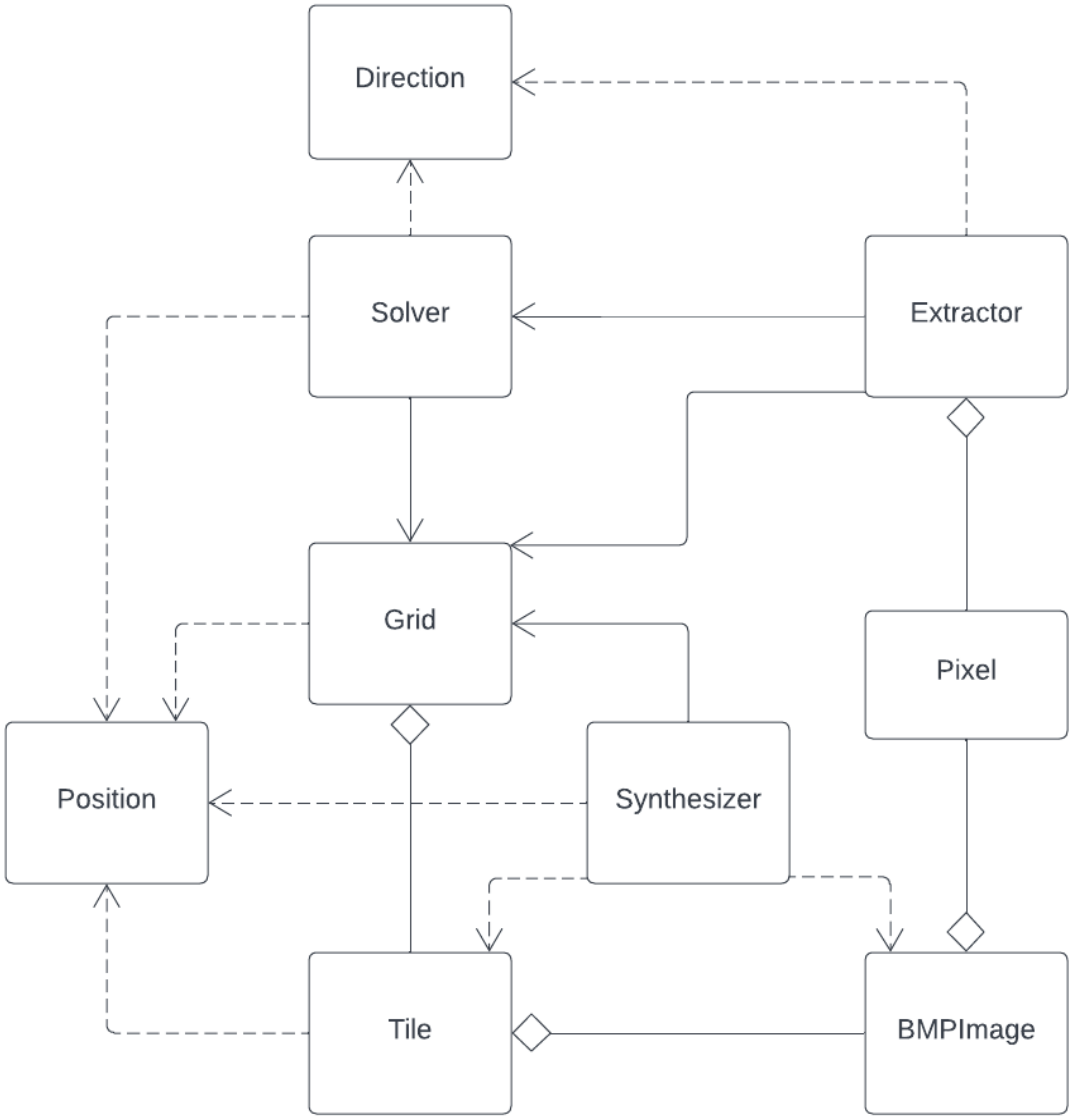
Class Design

WFCpp opts for an object oriented approach, where the main functionality is encapsulated in classes, stateful instances of which are utilized by the user. Nevertheless, we do not make extensive use of object oriented features such as inheritance. For extending functionality, we prefer the more functional approach of passing anonymous functions to classes, over virtual method override. The benefit of

[2] https://cmake.org/
[3] https://doxygen.nl/
[4] https://github.com/linux-test-project/lcov
[5] https://github.com/lvandeve/lodepng
[6] https://github.com/aburgh/EasyBMP

this approach is that modern C++ has very robust support for function objects, which allows one to store them in STL containers and invoke them effortlessly. The experience for the user is also augmented, as they only worry about providing a lambda as an argument, instead of having to extend the codebase. This lambda may capture objects in the current scope by reference, making them a very powerful mechanism for adding functionality to existing classes (in the form of callbacks).

In the future, some inheritance-based polymorphism may be useful to generalize algorithms, but for now this is unexplored. Provided is a class diagram of all current classes and enums available to the user:



API Design

Here, we go over each of the publicly available classes, their public API, and how the user is expected to handle instances of them.

FileType

An enum to denote the image file types supported. Can be extended as file type support is increased.

Position

A simple struct specifying 2d nonnegative integer coordinates. Utilized throughout the code to signify a grid position, instead of always requiring two non-descriptive integer arguments every time.

Pixel

A simple struct specifying the RGBA color values of a pixel in an image, in all image-manipulation classes. Helps in clarifying data structures which manipulate or store image data.

Direction

This enum denotes a direction. Used to specify adjacencies between tiles across the codebase. Extensible for when the algorithm is generalized to higher dimensions or different grid types (hexagonal, triangular).

BMPImage

A class which represents a bitmap image. Acts as a resource handle for a file on disk. Offers interface for getting the pixel width and pixel height of the image, getting/setting the pixel at some position, and exporting it to a file (.png and .bmp formats).

Tile

A wrapper class specifying a WFC tile, with an interface for getting the tile pixel size, all pixel positions, and the image data (a BMPImage). Helps separate uses of images and tiles.

Grid

Grid offers an interface for getting the current size, resizing it, setting the tile key at some position, getting the tile at some position, and setting the mapping between tile keys and tiles. The common object which is utilized to communicate between Extractor, Solver, and Synthesizer.

Solver

Solver offers an interface for invoking the WFC algorithm, for adding constraints to the algorithm, for registering callbacks, and for changing the collapse behavior. The method which runs the WFC algorithm modifies a Grid passed by reference, setting the tile key at a given grid position.

Extractor

Extractor offers an interface for opening a .bmp or .png image from disk to extract square tiles of a given pixel length, and for adding this information to an existing Grid. Additionally, provides methods for deriving the tile adjacency data, and adding it to an existing Solver.