Procedural Images Generator

CS-C2120 - Ohjelmointistudio 2

Denis Kuznetsov, 867272

Teknistieteellinen kandidaatti ja diplomi-insinööri,

Tietotekniikka

First year of studies

14.4.2023

General description

In this project, I developed the program to facilitate the creation of 2D images through the utilisation of the advanced wave function collapse algorithm for procedural generation. The program provides a minimalistic graphical user interface for selecting the folder containing the tiles to be used in the image generation, selecting a file with the rules, and adjusting various settings, such as image width or height and chunks ratios. Once the user has made the necessary selections, he can initiate the image generation by pressing a "Generate" button. The generated image will be displayed on the screen and can be regenerated as many times as necessary until the user is satisfied with the result. Finally, the user has the option to export the image in PNG format.

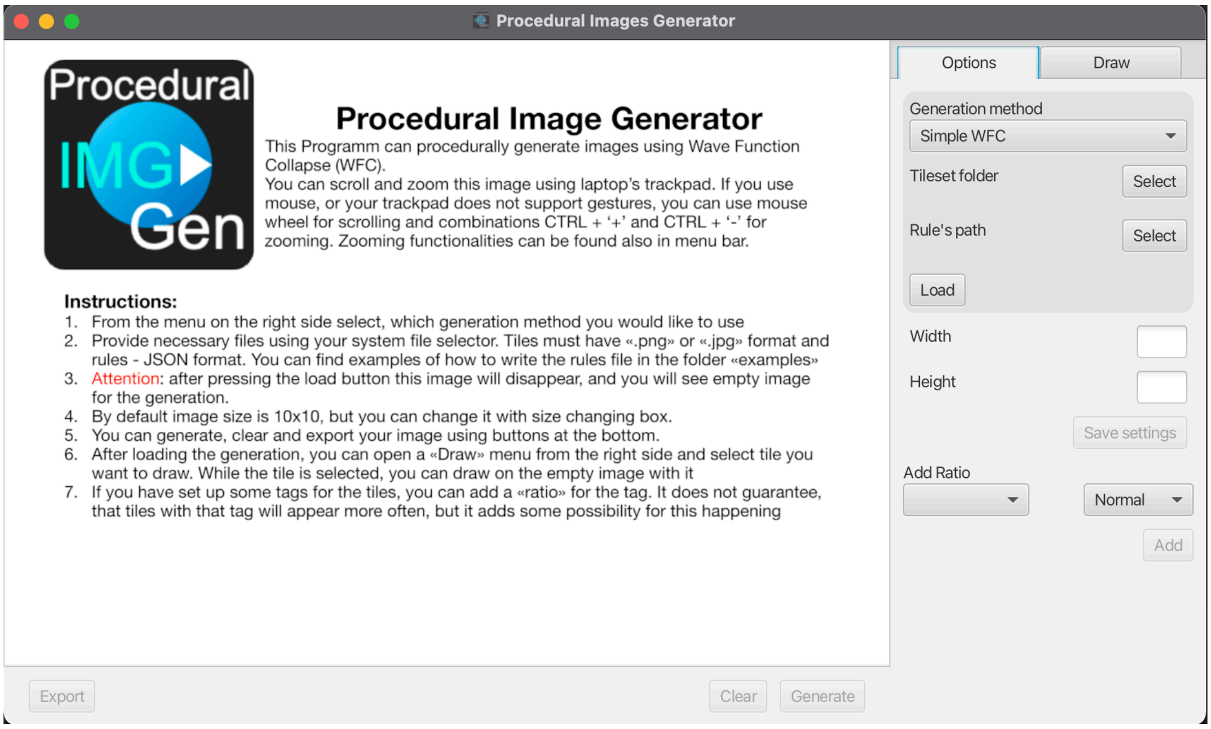
Additionally, the program includes an "auto generation" option, which allows the user to upload a sample image consisting of tiles. The program automatically generates the tile sets and rules based on the sample, producing similar images. The user can specify various parameters, such as such as mentioned above, and manually "draw" specific chunks using a drawing menu, when tiles are loaded

This project may sound very similar to the projects I made in a previous Programming Studio 1 course, but it is not true, because a lot of improvements have been made. Firstly, the user interface has been significantly improved to make it more intuitive and user-friendly. Additionally, the old program required the user to manipulate file names to set the rules, whereas the new program utilises a simple-to-read JSON file with easily understandable language. Secondly, the drawing system has been stabilised and improved, eliminating the bugs and instability that were present in the old program. It is also important to mention that auto generation is a new feature in this project.

Finally, the program also has a backtracking system, which solves the main problem of the old project where it would occasionally result in a situation where there were no chunks left to place according to the rules.

Regrettably, the implementation of the "two rounds generation" was not incorporated into this project due to time constraints. However, in my opinion, it can be readily simulated through the use of simple WFC, rendering it unnecessary. Despite this, it would have been intriguing to observe its effects.

User Interface



Procedural Image Generator

This Programm can procedurally generate images using Wave Function Collapse (WFC).
You can scroll and zoom this image using laptop's trackpad. If you use mouse, or your trackpad does not support gestures, you can use mouse wheel for scrolling and combinations CTRL + '+' and CTRL + '-' for zooming. Zooming functionalities can be found also in menu bar.

Instructions:

1. From the menu on the right side select, which generation method you would like to use
2. Provide necessary files using your system file selector. Tiles must have «.png» or «.jpg» format and rules - JSON format. You can find examples of how to write the rules file in the folder «examples»
3. Attention: after pressing the load button this image will disappear, and you will see empty image for the generation.
4. By default image size is 10x10, but you can change it with size changing box.
5. You can generate, clear and export your image using buttons at the bottom.
6. After loading the generation, you can open a «Draw» menu from the right side and select tile you want to draw. While the tile is selected, you can draw on the empty image with it
7. If you have set up some tags for the tiles, you can add a «ratio» for the tag. It does not guarantee, that tiles with that tag will appear more often, but it adds some possibility for this happening

To launch the program, users must download the project's source code and execute it by entering the command "sbt run" in the terminal window within the appropriate folder (NOTE: sbt should be installed). Once launched, the program will display a brief overview of its functions and usage. However, for the sake of clarity, the relevant information is also provided here, with additional details.

To begin using the program, users should select a generation method from the right-hand side menu by clicking on the "Option" tab, which is selected by default. Depending on the selected method, users may be prompted to provide different files or folders. For example, for the Simple WFC method, users must provide a folder containing tile sets and a rule file. All tiles must be in either PNG or JPG format and accompanied by a corresponding rule written in a file. The rule file has a JSON format, which will be further discussed later. Once all the necessary files have been selected, users should press the "Load" button to enable the program to read and load the generation.

Once the generation is loaded, users can set the size of the image in tiles using the size selection section below the method selection box. Additionally, users can add ratios to the tags if they wish to see certain tiles appear more frequently. After making these selections, users can generate an image by pressing the "Generate" button, clear it using the "Clear" button, or export the content to a PNG file using the "Export" button.

Under the "Draw" menu tab, users can manually draw a tile on the image, which will be automatically incorporated during the generation process. This menu will only display images for which rules have been provided, and users are prohibited from drawing tiles that may lead to a contradiction.

It is worth noting that users can scroll and zoom in on the image using the trackpad or mouse with the keyboard. The combinations "CTRL +'+'" and "CTRL +'-' " on the keyboard can also be used to zoom in and out, respectively. Furthermore, zooming functions can be accessed through the menu bar, which also provides options to close the program and change to a dark theme.

Program structure

The classes of the program can be classified into two major categories: GUI classes and Generation classes.

The primary purpose of GUI classes is to display information on the screen and to receive input from the user. ScalaFX, a JavaFX analog for Scala, is utilised for rendering purposes. These classes use ScalaFX functionalities to display items such as the image, menu buttons, options, and others.

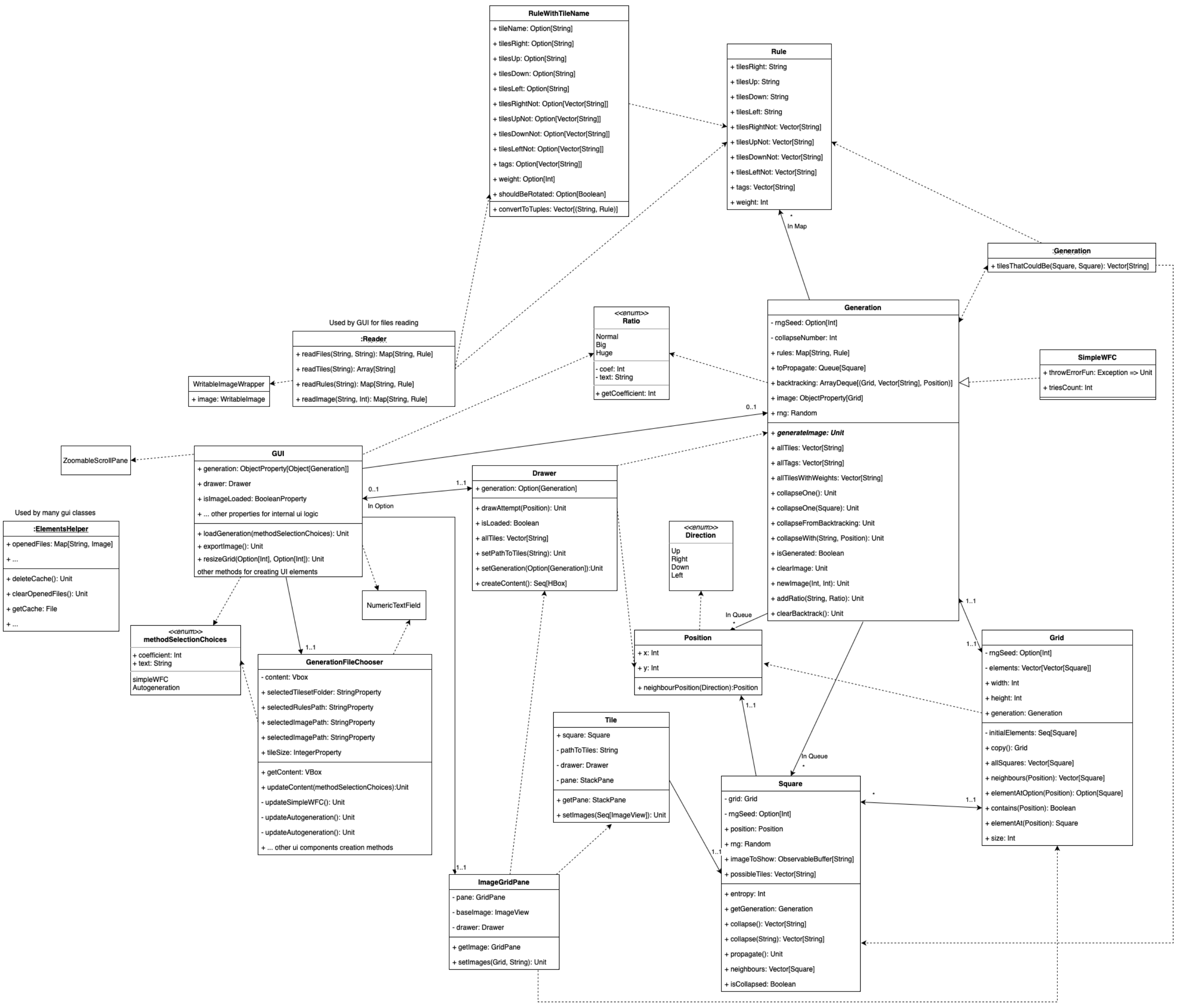
The GUI input classes validate user input and display error messages through pop-up windows in the case of invalid input. A separate object, referred to as the Reader, has been created for reading and validation purposes. Its responsibility is to scan files and validate them, throwing exceptions in case of errors to prevent the program from crashing. The Reader object returns the map of image name to image rules, ensuring that the program remains functional even if there is an imbalance between the number of images and rules. Importantly, the object is used instead of a class, as it does not contain any information in its body and is only called within the GUI class.

The GUI classes also include the Drawer class, which allows users to manually draw chunks and is responsible for creating the content of the «Draw» tab. In its constructor, this class receives a function to be used in case of errors during drawing, which allows the drawer to display an alert pop-up on the user's screen. In the event that the user attempts to paste a chunk in an incorrect location, the Drawer class provides the necessary information to the main class, which determines the outcome.

During the project's creation, it was decided to add the "GenerationFileChooser" class, which contains a VBox object to be shown to the user and changes its content to match the selected generation method. It also stores information about paths selected by the user, which is given to the Reader object by the GUI.

To handle image rendering, two additional classes were implemented: "ImageGridPane" and "Tile." They contain a ScalaFX object (GridPane in the first and StackPane in the second) and have methods for changing their content. The Tile class listens to changes in the Square it is responsible for and changes its content to match the square. Using this structure, the program avoids re-rendering all images every tick. It would have been possible to achieve the same functionality without these classes, but the decision was made to encapsulate the program to add more abstraction and hide the internal logic from the user.

Some of the GUI classes are dependent on the Generation classes. For example, the image canvas chooses an image to display in a given chunk based on the "imageToShow" property of the Grid's Square, but this dependency is minimal and is used only to avoid the re-rendering of the whole canvas.



RuleWithTileName
+ tileName: Option[String]
+ tilesRight: Option[String]
+ tilesUp: Option[String]
+ tilesDown: Option[String]
+ tilesLeft: Option[String]
+ tilesRightNot: Option[Vector[String]]
+ tilesUpNot: Option[Vector[String]]
+ tilesDownNot: Option[Vector[String]]
+ tilesLeftNot: Option[Vector[String]]
+ tags: Option[Vector[String]]
+ weight: Option[Int]
+ shouldBeRotated: Option[Boolean]
+ convertToTuples: Vector[(String, Rule)]
Rule
+ tilesRight: String
+ tilesUp: String
+ tilesDown: String
+ tilesLeft: String
+ tilesRightNot: Vector[String]
+ tilesUpNot: Vector[String]
+ tilesDownNot: Vector[String]
+ tilesLeftNot: Vector[String]
+ tags: Vector[String]
+ weight: Int
*
In Map
:Generation
+ tilesThatCouldBe(Square, Square): Vector[String]
Generation
- rngSeed: Option[Int]
- collapseNumber: Int
+ rules: Map[String, Rule]
+ toPropagate: Queue[Square]
+ backtracking: ArrayDeque[(Grid, Vector[String], Position)]
+ image: ObjectProperty[Grid]
+ rng: Random
+ generateImage: Unit
+ allTiles: Vector[String]
+ allTags: Vector[String]
+ allTilesWithWeights: Vector[String]
+ collapseOne(): Unit
+ collapseOne(Square): Unit
+ collapseFromBacktracking: Unit
+ collapseWith(String, Position): Unit
+ isGenerated: Boolean
+ clearImage: Unit
+ newImage(Int, Int): Unit
+ addRatio(String, Ratio): Unit
+ clearBacktrack(): Unit
SimpleWFC
+ throwErrorFun: Exception => Unit
+ triesCount: Int
Used by GUI for files reading
:Reader
+ readFiles(String, String): Map[String, Rule]
+ readTiles(String): Array[String]
+ readRules(String): Map[String, Rule]
+ readImage(String, Int): Map[String, Rule]
WritableImageWrapper
+ image: WritableImage
<<enum>>
Ratio
Normal
Big
Huge
- coef: Int
- text: String
+ getCoefficient: Int
0..1
ZoomableScrollPane
GUI
+ generation: ObjectProperty[Object[Generation]]
+ drawer: Drawer
+ isImageLoaded: BooleanProperty
+ ... other properties for internal ui logic
+ loadGeneration(methodSelectionChoices): Unit
+ exportImage(): Unit
+ resizeGrid(Option[Int], Option[Int]): Unit
other methods for creating UI elements
0..1
In Option
1..1
Drawer
+ generation: Option[Generation]
+ drawAttempt(Position): Unit
+ isLoaded: Boolean
+ allTiles: Vector[String]
+ setPathToTiles(String): Unit
+ setGeneration(Option[Generation]):Unit
+ createContent(): Seq[HBox]
Used by many gui classes
:ElementsHelper
+ openedFiles: Map[String, Image]
+ ...
+ deleteCache(): Unit
+ clearOpenedFiles(): Unit
+ getCache: File
+ ...
NumericTextField
<<enum>>
Direction
Up
Right
Down
Left
In Queue
*
Position
+ x: Int
+ y: Int
+ neighbourPosition(Direction):Position
1..1
<<enum>>
methodSelectionChoices
+ coefficient: Int
+ text: String
simpleWFC
Autogeneration
1..1
GenerationFileChooser
- content: Vbox
+ selectedTilesetFolder: StringProperty
+ selectedRulesPath: StringProperty
+ selectedImagePath: StringProperty
+ selectedImagePath: StringProperty
+ tileSize: IntegerProperty
+ getContent: VBox
+ updateContent(methodSelectionChoices):Unit
- updateSimpleWFC(): Unit
- updateAutogeneration(): Unit
- updateAutogeneration(): Unit
+ ... other ui components creation methods
Tile
+ square: Square
- pathToTiles: String
- drawer: Drawer
- pane: StackPane
+ getPane: StackPane
+ setImages(Seq[ImageView]): Unit
1..1
Grid
- rngSeed: Option[Int]
- elements: Vector[Vector[Square]]
+ width: Int
+ height: Int
+ generation: Generation
- initialElements: Seq[Square]
+ copy(): Grid
+ allSquares: Vector[Square]
+ neighbours(Position): Vector[Square]
+ elementAtOption(Position): Option[Square]
+ contains(Position): Boolean
+ elementAt(Position): Square
+ size: Int
1..1
1..1
In Queue
*
Square
- grid: Grid
- rngSeed: Option[Int]
+ position: Position
+ rng: Random
+ imageToShow: ObservableBuffer[String]
+ possibleTiles: Vector[String]
+ entropy: Int
+ getGeneration: Generation
+ collapse(): Vector[String]
+ collapse(String): Vector[String]
+ propagate(): Unit
+ neighbours: Vector[Square]
+ isCollapsed: Boolean
*
1..1
1..1
1..1
ImageGridPane
- pane: GridPane
- baseImage: ImageView
- drawer: Drawer
+ getImage: GridPane
+ setImages(Grid, String): Unit

The Generation classes handle all of the logic behind the user interface, including generation, regeneration, and export. The key element for this group is the Generation trait, which allows the program to collapse the wave function for simple generation, draw and regenerate the image. The Generation trait has an instance of the Grid class, which serves as the main array containing all Squares. The Square class contains information about its position on the grid and possible images to render, as well as the "collapse" method, which serves as the foundation of the algorithm. To propagate changes in possible tiles of the square to the neighbours, the queue property "toPropagate" in Generation is used, and all squares that need to be checked for changes are added to this queue. The Generation trait also has a backtracking system, for which a double queue is used. (technical aspect will be explained later in the text)

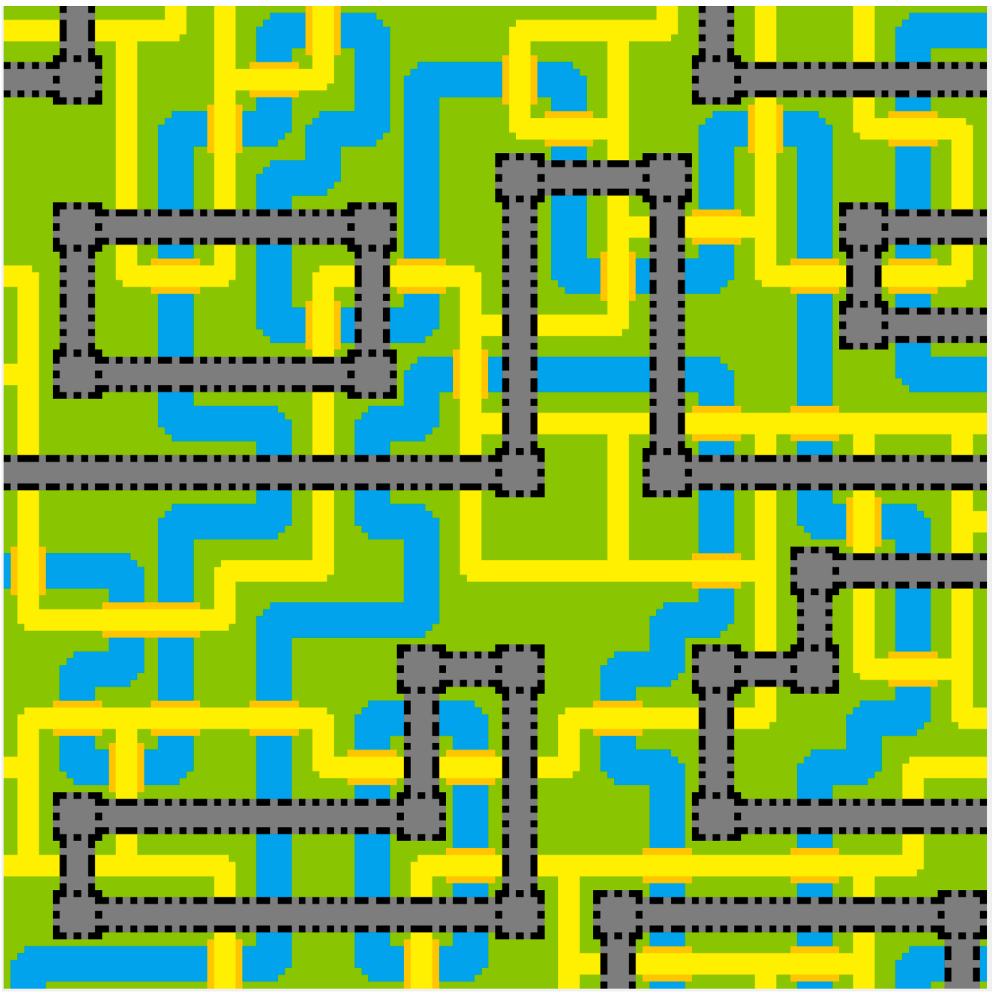
This structure of GUI and Generation classes is well-suited for the project as it separates the user interface and graphics from the program's logic. This makes it easier to change the UI or program logic classes without heavily affecting each other, which becomes particularly useful in further development. For example, adding new functions or changing the UI will be much easier, and testing the program will be facilitated by this structure

Algorithms

The primary function of this program is the "generateImage" method of the Generation trait. Its implementation varies, but all implementations utilise the Wave Function Collapse (WFC) algorithm as their foundation. The Markov-Chain model proposed by Pronay Peddiraju has been chosen for the WFC implementation. According to Peddiraju's master's thesis, there are two main WFC models: Overlapping and Tiling. However, the Markov-Chain model has several advantages over the other models. For example, the user does not need to manually write a list of tile names in the rules for each tile. Instead, the user can use "sockets" to filter neighbours. In this project’s implementation, each tile has 12 constraints: three for each of the four sides. Each side has a three-word rule consisting of English uppercase letters, such as "ABC". Only tiles with the same constraints on neighbouring sides can be placed near each other. This implementation varies significantly from the Overlapping model, where the sample image is divided into small squares and overlaps the images with the same pixels. In general, the "socket" model suits the project well, as it is much easier to write rules for the tiles.

During the creation of the project, it was revealed that the "contradiction error," which was the main problem to solve, can appear even with a well-written algorithm for propagating changes. During the generation of the image, the program can place a tile, that leads to a contradiction somewhere on the image, in other words, it is not possible to place any tiles at all. Paul C. Merrell discusses this issue in more detail in his PhD dissertation. According to him, computing the right tile to place, which will not produce any contradiction, is an NP-hard problem. That means it would be extremely slow to construct such an algorithm, so a workaround method had to be used. There are a few ready solutions for this problem. For this project, it was decided to use a "backtracking system." Briefly speaking, before collapsing a tile, the grid state is saved to the stack, and if a contradiction is found, the program pops the state from the stack and tries to collapse the tile again without the problematic tile name. To mitigate performance issues, the grid state is not saved after every tile collapse. Instead, a predetermined rate is used, which is calculated based on the maximum size of the grid's dimension divided by three. For instance, in a 100x100 grid, the state is saved after every 33 tile collapses. This approach strikes a balance between performance and state preservation, as it reduces the number of times the program saves the grid state without sacrificing the integrity of the generation process. Also the size of the backtracking is limited to 20 states.

In the Autogeneration option, the overlapping and Markov-Chain models are combined. The user provides the program with a sample image created using a tile-set and with a tile size used in the image. The program then automatically generates tiles and rules based on this sample. The algorithm divides the image into squares with the provided size and generates constraints based on the pixel colours of the image. It is worth mentioning that the sample image cannot be very complex, in other words, it cannot have more than 26 different colours on the borders, as there are only 26 letters in the English alphabet.


Possible image generated by the program

Data structures

In this project, a decision was made to utilise some of the data structures from the Scala standard library for storing the backtracking and squares to propagate, as well as a custom structure known as Grid and Square for storing the relevant information pertaining to the squares.

To store the backtracking information, the Double Queue "Array Deque" was chosen, as it offers the functionality of both a queue and a stack. This structure was preferred over a standard Stack, as it permits the removal of elements in a similar manner to a queue. Due to the fact that a backtracking stack can consume a large amount of memory for a large grid, the last element is removed when the size of the stack exceeds 20. This ensures that the stack is prevented from becoming too large and overburdening the system.

For propagation purposes, the Queue is employed, which functions on a First-in, First-out (FIFO) principle. This order of propagation is advantageous, as it reduces the probability of encountering contradictions that were previously experienced when using the stack.

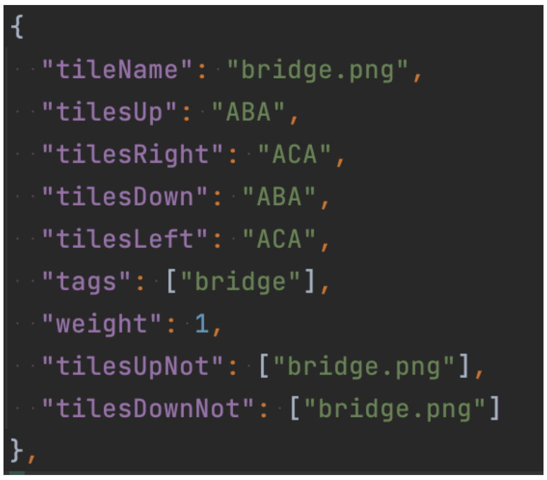
The Grid structure is an immutable structure that contains all the squares and provides some methods for taking neighbours. Since the grid itself cannot be changed, a new grid must be created when a new size is required. The Grid employs the double dimension Vector for storing the squares. Although it was possible to avoid using an additional grid class, doing so would make the code less readable and more difficult to work with.

The square is a structure that holds the information about the tile's state, and thus contains two main vector arrays: a Vector of possible tiles this square can be collapsed with, and an observable buffer that contains the tile names this square should display.

It is essential to highlight the use of ScalaFX observable structures, which were utilised extensively in the project. These structures mimic the standard library structures but offer the added benefit of allowing for a subscription model throughout the program. By enabling the attachment of listeners, it is possible to perform some actions whenever an observable object changes. These structures were primarily employed to update the GUI to reflect the current state of the logic classes.

Files

In this project, the program's standout feature is its ability to facilitate the creation of rules files with ease. The user can provide a JSON file through the program's UI, which contains an array of objects. Each object is composed of fields such as "tileName", "tilesUp" (and other directions), "tilesUpNot" (and other directions), "tags", "weight", and "shouldBeRotated". The illustration on the right-hand side provides an example of such a rule.



```
{
  "tileName": "bridge.png",
  "tilesUp": "ABA",
  "tilesRight": "ACA",
  "tilesDown": "ABA",
  "tilesLeft": "ACA",
  "tags": ["bridge"],
  "weight": 1,
  "tilesUpNot": ["bridge.png"],
  "tilesDownNot": ["bridge.png"]
},
```

The "tileName" field denotes the name of the picture for which the rule is intended. It must adhere to the image format, allowing for images with the same names but different formats to be used. The rule contains a string with 3 letters, indicating the constraints for each side, as well as a vector of tile names that should not be placed adjacent to the tile in question. For example, if the user wishes to prevent "tile1.png" from being placed next to "tile2.png", they can use the "tileUpNot" field and specify the name "tile2.png". Furthermore, the user can specify the rotation of the tile by adding the "@" sign followed by a number, such as "tile1.png@1", which means that the tile is rotated by 90 degrees clockwise.

It is noteworthy that certain fields, namely "weight", "tags", "tilesNot", and "shouldBeRotated", are optional as they have default values. The default behaviour is to allow any image to be rotated, resulting in the creation of four distinct rules for each rotation of the image. This means that all constraints and "tilesNot" rules will also be rotated accordingly. The weight of the tile is 1 by default.

For images, the program accepts any jpg or png image in the selected folder, and if a rule from the rules file is found for the image, it will be used.

If the user decides to use Autogeneration, they must provide the program with the tile size used in the sample image. If the image's size does not match the provided tile size, the file will not be accepted.

Testing

The testing of the program was held according to the plan with such difference, that it was decided to use unit testing also for the generation process. Two testing classes were made for unit testing: one for Reader object and one for the Generation, they can be found from the project src/test folder.

The testing of the program is conducted through a combination of unit and system testing methodologies. System testing were primarily used to evaluate the GUI classes, as it is deemed to be more practical and intuitive than unit testing in this scenario. The program's appearance and functionality was monitored after each addition, and the values received in input fields were logged in the console for verification purposes.

On the other hand, for the Reader class, a crucial component responsible for reading and parsing the necessary files such as the tileset and rules, unit testing was used. This methodology verified the correct functioning of the Reader class, allowing for the detection of any potential bugs during the early stages of development. To accomplish this, test cases were written for various scenarios, including reading a valid file, an invalid file, and a file with missing information. This ensures the robustness and reliability of the Reader class, allowing it to handle various inputs and exceptions in an appropriate manner. For example, a broken JSON file without a required "tileName" field is provided to the reader, and it is expected to throw the exception "tileName for the Rule not found". Similar procedures is performed for other exceptions.

Generation testing class focuses mainly on the contradictions and using the backtracking. Resizing of the image and drawing are also tested.

In conclusion, the combination of unit and system testing provides comprehensive coverage and aid in the timely and effective identification and resolution of any issues that may arise during the development of the program.

Bugs and missing features

Unfortunately, this program suffers from several significant bugs that may impact its functionality and overall performance. One of the most prominent issues is the presence of a memory leak during the generation process. The graphical user interface (GUI) structure employed in the program involves updating every GUI tile through a listener. As a result, when a contradiction occurs and the grid is replaced, all GUI tiles are also altered. Somewhere during the grid changing, a memory leak occurs, leading to the program consuming all available memory and eventually stopping working. In such cases, the only recourse is manual closing, which can be a frustrating experience for the user.

An attempt was made to address this problem by increasing the extended memory heap size to 4 GB. However, this measure only partially mitigates the problem and does not solve it altogether. The optimal image size for the program is 10x10 or 20x20, although it may vary depending on the computer's specifications and the tile set provided. One of the simplistic solutions to the issue is to update the entire grid after every collapsing event, but this approach comes with its own downsides. Thus, the issue remains unresolved.

The second problem of the program is related to how autogeneration works. The program generates rules for the tiles based on the pixel colors, which may result in inaccurate results for complex images. A good example of such an image can be found in the examples folder from the project root.

It is also essential to mention that the program does not support two-round generation. There are two primary reasons for this. First, time constraints played a role in not implementing this feature. Second, the one-round replacement approach was deemed effective for the intended purposes of the program. It is worth noting that two-round generation can be simulated by creating well-written rule files using the simple WFC method. Hence, while the lack of support for two-round generation is a limitation, it is not a significant problem that compromises the program's core functionality.

Best sides and weaknesses

Strengths:

1. Easy to use: One of the main strengths of the program is its user-friendly interface, which allows users to easily select image files and create rule files. The program also provides default values for certain fields, making it more accessible to those with limited programming knowledge.

2. Versatility: The program accepts a wide range of image formats, including JPG and PNG files, and can generate tile sets based on pixel colors. This allows for a diverse range of images to be used as input and creates more flexibility in the output.

3. Customisability: The program allows users to create their own rule files, which can be tailored to specific image sets and user preferences. This provides a high degree of customisability and control over the generated output.

Weaknesses:

1. Memory issues: The program has a significant memory leak during generation, which can lead to memory consumption and program crashes. This issue is particularly problematic with larger image sizes and complex tile sets, limiting the program's ability to handle more demanding inputs.

2. Inaccurate autogeneration: The program's autogeneration feature generates tile sets based on pixel colors, which can lead to inaccurate results with complex images. This limits the program's ability to accurately generate tile sets.

3. Lack of two-round generation: The program lacks a two-round generation feature, which was noted in the plan

Overall, the program has some notable strengths in terms of user-friendliness, versatility, and customisability. However, the program also suffers from significant memory issues, inaccurate autogeneration, and lacks certain advanced features, limiting its potential for more complex applications.

Deviations from the plan, realised process and schedule

The process of creating the project can be divided into three parts: GUI development, logic implementation, and SimpleWFC implementation, and autogeneration. The initial two weeks were dedicated to the implementation of the graphic, reader, and tests for the reader class. The following two weeks were spent creating the base for the Draw option tab and the imageGridPane class. Subsequently, the focus shifted towards UX features, such as an icon at the dock, and a backtracking system was added. During the final two weeks, the autogeneration feature was added, and efforts were made to rectify the backtracking issue.

It is worth mentioning that the implementation process did not go as planned due to other university-related commitments. The time required for the other commitments was significantly underestimated, and this was the primary reason for the deviation from the initial plan during the first two weeks. Overall, the time invested in the project amounted to approximately 80 hours, which was in line with the initial estimate.

The project development process provided valuable insights and learning opportunities. I discovered the importance of accurately estimating the time required for a project and the need for proper time management.

Final evaluation

The final program has both strengths and weaknesses. One of its strengths is that it has a user-friendly interface, backtracking, and autogeneration. The program can handle small image sizes with ease and the logic for the tiles is implemented correctly. However, the program also has some significant shortcomings. The autogeneration function produces inaccurate results for complex images and there is a memory leak during generation that can lead to the program crashing.

To improve the program, a possible solution would be to implement a more efficient method for autogeneration that can handle complex images. Furthermore, the memory leak during generation needs to be addressed to prevent the program from crashing.

If I were to start the project again, I would allocate more time to planning and research before beginning the implementation. Additionally, I would prioritise implementing the autogeneration function earlier in the process to allow more time for refining and improving it. Finally, I would be more careful with the structures I use to avoid a memory leaks. .

In conclusion, I am satisfied with the program that was developed during this course. Its user interface is intuitive and user-friendly, which makes it an engaging tool to work with. Moreover, the program's ability to generate a vast number of visually appealing images within a short timeframe is impressive, so we can say, that the program fulfils the primary objective that was set for its development.

References and links

1. The Coding Train (2022, July 3). *Coding Challenge 171: Wave Function Collapse* [Video]. YouTube. https://www.youtube.com/watch?v=rI_y2GAlQFM

2. Robert Heaton (2018, December 17). *The Wave function Collapse Algorithm explained very clearly.* https://robertheaton.com/2018/12/17/wavefunction-collapse-algorithm/

3. Martin Donald (2022, July 31). Superpositions, Sudoku, the Wave Function Collapse algorithm [Video]. https://www.youtube.com/watch?v=2SuvO4Gi7uY

4. Pronay Peddiraju (2020). Markov-Chain based Wave Function Collapse [Master's thesis, Southern Methodist University – Guildhall] https://www.pronay.me/thesis-markov-chain-based-wave-function-collapse/

5. Paul C. Merrell (2009). Model Synthesis [PhD dissertation, University of North Carolina] https://paulmerrell.org/wp-content/uploads/2021/06/thesis.pdf