Quantum Collapse Algortihm

Kerlin Michel

2022

List of Algorithms

1 Introduction

Creating works of art can be a laborious and tedious task. In some cases may be impossible to do so. The 3D game *Minecraft: Java Edition* is a game where a world is created using different procedural generation algorithms. These worlds are so large that creating a world by hand would be not feasible.

The Wave Function Collapse algorithm[1], a texture synthesis algorithm, was made by Gumin and is heavily based on Merrell's Model Synthesis thesis paper[2]. In a high level description the models synthesis algorithm creates an randomly generate output, image or 3D model, that locally looks similar to some input. The algorithm interprets the input and output as being locally partitioned into parts. When the algorithm finishes succesfully each output part will be equal to some input part. The model synthesis initializes all output parts to be in state where each input part is a possible assignment. The algorithm then loops through each output part and randomly assign it to an input part and then possible output assignments for unassigned output parts are updated based on an adjaceny contraint. The adjaency constaint is built by analyzing the adjaceny of parts in the input. The Wave Function Collapse works similarly with the following differences:

1. Instead of iterating through the output in a scanline manner like in Model Synthesis the output is iterated through by choosing the output with the lowest entropy, the output with least number of possible input parts that can be assigned to it

2. Wave Function Collapse doesn't implement a feature of Model Synthesis where the output is modified in sections rather than as a whole.

Merrell examines the difference between Model Synthesis and Wave Function Collapse in Comparing Model Synthesis and Wave Function Collapse[3].

This project is meant to examine and generalize both algorithms. Implementing the algorithm using quantum computing will also be explored.

2 Procedural Generation

Creating data manually can be extremely time consuming. For example in the game *Minecraft* the 3D cube voxel world map is separated into chunks that are $16 \times 16 \times 384$ in size. If a human working at a pace of setting 4 blocks, a voxel cube, a second to a chunk then the it would take more than 6 hours to set all the blocks in a chunk. The game contains a large number of chunks so creating a *Minecraft* world would be very time consuming and costly if a human builder was used. *Minecraft* uses procedural generation to create chunks so that chunks. Procedural generation can be described as a computational function that takes pre constructed data and probability distribution as input and gives data as the output.

A common procedural generation technique is to generate an image with Perlin noise, a type of gradient noise, and use the values at each pixel to represent the height for 3D terrain. *Minecraft* randomly generates structures, such as villages where village buildings are randomly placed at the location of the village. *Minecraft* generates structures atomically meaning that the input pre constructed data, the building model, is placed entirely in the map and not a part of it. Villages may have unique layouts but this leads to seeing the same buildings in each village which can make the world feel more artificial. A possible improvement would be for the village generation process to make it so that buildings are made to look similar to the pre constructed buildings but allowing unique differences.

2.1 Example Based Generation

An example based procedural generation algorithm creates an output that is similar to the pre constructed input data. Similarly is define as a small part of the output having the same value or near equal value of some part of the input.

3 Quantum Collapse Algorithhm

The Quantum Collapse Algorithhm randomly generates an output that is composed of parts that are locally similar to some input. The algorithm will assign a part in the input to a part in the output. The input parts are described by the set $S = \{s_1, \ldots, s_{k-1}\}$ where k is the number of possible states, $k = |S|$.

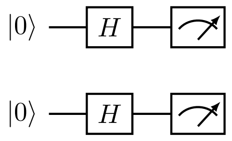
The output O is represented by a $d_0 \times \cdots \times d_{D-1} \times k$ array where D is the number of dimensions of the output and $d_i \in \mathbb{Z}^+$. For images $D = 2$ and for videos and 3D models $D = 3$. Each output part is described by $o_{d_0,\ldots,d_{D-1}}$ where is an array: $O_{d_0,\ldots,d_{D-1}} = [O_{d_0,\ldots,d_{D-1},0},\ \ldots,\ O_{d_0,\ldots,d_{D-1},k-1}]$. Each entry in a o represents the possible states of an output part where each index of corresponds to a state in S. If an entry in some o is > 0 then the the state in S that corresponds to the index of said entry is a possible assignment to the output part. The algorithm initializes each entrpy O to 1 so that all output parts, $o_{d_0,\ldots,d_{D-1}}$, can be assigned any state in S.

When a state s is assign to some o all entries in o become 0 except for the entry at the index that corresponds to the state s being assigned. When a o is assigned other o's states will be updated for which states are possible to assign to them based on a neighborhood constraint. The neighborhood constraint restricts which states can be assigned near each other and is created by examining the neighborhood of the input parts. To construct a neighborhood constraint the neighborhood needs to be defined. The neighborhood, N, is $\{(v_0^0,\dots,v_{D-1}^0),\ \dots,\ (v_1^{|N|-1},\dots,v_{D-1}^{|N|-1})\}$ where $v_j^i \in \mathbb{Z}$. Each v^i in N specifies that some $o_{d_0,\dots,d_{D-1}}$ is a neighbor to some other $o_{d_0+v_0^i,\dots,d_{D-1}+v_{|N|-1}^i}$. v^i is essentially an index offset. The neighborhood constraint, C_n, is then represented by a $k \times |N| \times k$ array. Every entry $c_{i,j,k}$ in C_n, where indexes i and k correspond to two states and j corresponds to some neighborhood offset is either equal to 1 if this neighborhood constraint is in the input and 0 if not.

4 Quantum Computing

4.1 Quantum Randomness

One application of quantum computing is the ability to produce truly random numbers. For example the following quantum circuit will randomly generate a 2 bit number where 00, 01, 10, 11 are all equally likely:



References

1. GUMIN, M., 2016. Wave function collapse, https://github.com/mxgmn/wavefunctioncollapse.

2. MERRELL, P. 2009. Model Synthesis. PhD thesis, University of North Carolina at Chapel Hill.

3. MERRELL, P. 2021. Comparing Model Synthesis and Wave Function Collapse, https://paulmerrell.org/wp-content/uploads/2021/07/comparison.pdf

Algorithm 1 Quantum Collapse Algorithhm

```
$S \leftarrow \{s_1, s_2, \ldots, s_k\}$
$N \leftarrow \{(v_0^0, \ldots, v_{D-1}^0),\ \ldots,\ (v_1^{|N|-1}, \ldots, v_{D-1}^{|N|-1})\}$
$O \leftarrow [d_0] \ldots [d_{D-1}][k]$
$P \leftarrow$ CreateStack()
$C_n \leftarrow [k][|N|][k]$

Fill($O, 1.0$)                    ▷ Set output parts to a superposition of all states

while $\exists O_{d_0,\ldots,d_{D-1}} \neq e_i$ do    ▷ $e_i$ = some standard basis vector, e.g.$[0, 1, 0]$
    $d_0, \ldots, d_{D-1} \leftarrow$ SelectUncollapsedOutputPart($O$)
    O$[d^0] \ldots [d^{D-1}] \leftarrow$ Collapse($d_0, \ldots, d_{D-1}$)
    push($P$, $(d_0, \ldots, d_{D-1})$)
    while $P$ is not empty do
        $v \leftarrow$ pop($P$)
        for $n \in N$ do
            shouldPropagate $\leftarrow$ false
            for $s = 0$ to $k - 1$ do
                if O$[(v + n)][s] > 0$ then        ▷ Loop allow states in $O[v + n]$
                    stateIsPossible $\leftarrow$ false
                    for $s^* = 0$ to $k - 1$ do
                        if $C_n[s^*][n][s] = 1$ and $O[v][s^*] = 1$ then
                            stateIsPossible $\leftarrow$ true
                        end if
                    end for
                    if stateIsPossible = false then
                        O$[(v + n)][s] \leftarrow 0$
                        shouldPropagate $\leftarrow$ true
                    end if
                end if
            end for
            if shouldPropagate then
                push($P$, $(v + n)$)
            end if
        end for
    end while
end while
```