CONSTRAINT-DRIVEN OPEN-WORLD SCENE GENERATION

A Thesis

presented to

the Faculty of California Polytechnic State University,

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Computer Science

by

Hunter Borlik

June 2023
COMMITTEE MEMBERSHIP

TITLE: Constraint-Driven Open-World Scene Generation

AUTHOR: Hunter Borlik

DATE SUBMITTED: June 2023

COMMITTEE CHAIR: Zoë Wood, Ph.D.
Professor of Computer Science

COMMITTEE MEMBER: Phoenix (Dongfeng) Fang, Ph.D.
Assistant Professor of Computer Science

COMMITTEE MEMBER: Rodrigo Canaan, Ph.D.
Assistant Professor of Computer Science
ABSTRACT

Constraint-Driven Open-World Scene Generation

Hunter Borlik

We introduce an alternative method for open-world scene generation. In this thesis, Graph-based Wave Function Collapse (GWFC) is integrated with Space Colonization Algorithm (SCA) and used to place objects in an unstructured 3D environment. This combined algorithm, Space Colonization Graph-based Wave Function Collapse (SC-GWFC), leverages the constraint-based capabilities of GWFC and the ability of SCA to populate arbitrary 3D volumes. We demonstrate that objects of variable scale can be successfully used with SC-GWFC. Since this algorithm is run in an interactive environment, we demonstrate iterative modifications to a partially complete scene and incorporate PCG into a scene editing process. As part of the implementation, we also introduce our Scene Modeling Application for rendering and editing 3D scenes. This modeling application allows for editing and viewing constraints for our SC-GWFC scene generator. We evaluate the performance characteristics of SC-GWFC in the Scene Modeling Application to demonstrate that SC-GWFC can be used interactively. Through the application, users can specify adjacency requirements for objects, and SC-GWFC will attempt to place objects in patterns that respect these rules. We demonstrate the ability to place up to 5000 items on a terrain using our proposed SC-GWFC technique.
ACKNOWLEDGMENTS

Special thanks to:

• My advisor Zoë Wood, for your patience through this journey of computer graphics and procedural content generation, and your endless support and encouragement to make this project the best that I could.

• My committee members, Phoenix Fang, and Rodrigo Canaan for your valuable time and feedback.

• My mom, dad, brother, and friends for supporting me throughout this research!
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LIST OF TABLES</td>
<td>ix</td>
</tr>
<tr>
<td></td>
<td>LIST OF FIGURES</td>
<td>x</td>
</tr>
<tr>
<td></td>
<td>LIST OF ALGORITHMS</td>
<td>xi</td>
</tr>
<tr>
<td></td>
<td>LIST OF NOMENCLATURE</td>
<td>xiii</td>
</tr>
<tr>
<td>1</td>
<td>Motivation and Background</td>
<td>1</td>
</tr>
<tr>
<td>1.1</td>
<td>Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.2</td>
<td>Motivation</td>
<td>2</td>
</tr>
<tr>
<td>1.3</td>
<td>Contributions</td>
<td>3</td>
</tr>
<tr>
<td>1.4</td>
<td>PCG Search Spaces</td>
<td>4</td>
</tr>
<tr>
<td>1.5</td>
<td>PCG Challenges</td>
<td>6</td>
</tr>
<tr>
<td>1.5.1</td>
<td>Representing Style</td>
<td>6</td>
</tr>
<tr>
<td>1.5.2</td>
<td>General Content Generators</td>
<td>7</td>
</tr>
<tr>
<td>1.5.3</td>
<td>Interfaces and Controllability for PCG systems</td>
<td>7</td>
</tr>
<tr>
<td>1.6</td>
<td>Constraint Satisfaction Problems</td>
<td>8</td>
</tr>
<tr>
<td>1.7</td>
<td>Wave Function Collapse</td>
<td>9</td>
</tr>
<tr>
<td>1.8</td>
<td>Space Colonization Algorithm</td>
<td>11</td>
</tr>
<tr>
<td>2</td>
<td>Related Work</td>
<td>14</td>
</tr>
<tr>
<td>2.1</td>
<td>Answer Set Programming</td>
<td>14</td>
</tr>
<tr>
<td>2.2</td>
<td>Graph-Based Wave Function Collapse</td>
<td>15</td>
</tr>
<tr>
<td>2.3</td>
<td>Space Colonization for Procedural Road Generation</td>
<td>17</td>
</tr>
<tr>
<td>2.4</td>
<td>City Generation</td>
<td>18</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
<td></td>
</tr>
<tr>
<td>----------------------------</td>
<td>------</td>
<td></td>
</tr>
<tr>
<td>6.2 GWFC Performance</td>
<td>62</td>
<td></td>
</tr>
<tr>
<td>7 Conclusion</td>
<td>66</td>
<td></td>
</tr>
<tr>
<td>7.1 Limitations</td>
<td>66</td>
<td></td>
</tr>
<tr>
<td>7.2 Future Work</td>
<td>67</td>
<td></td>
</tr>
<tr>
<td>8 Bibliography</td>
<td>69</td>
<td></td>
</tr>
</tbody>
</table>
## LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>52</td>
</tr>
<tr>
<td>5.2</td>
<td>53</td>
</tr>
<tr>
<td>5.3</td>
<td>54</td>
</tr>
<tr>
<td>5.4</td>
<td>55</td>
</tr>
<tr>
<td>5.5</td>
<td>56</td>
</tr>
<tr>
<td>5.6</td>
<td>57</td>
</tr>
<tr>
<td>5.7</td>
<td>58</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Wave Function Collapse with partially complete output. Karth and Smith [10].</td>
<td>11</td>
</tr>
<tr>
<td>1.2</td>
<td>SCA initial stage of tree skeleton generation example from Runions, Lane, and Prusinkiewicz [15].</td>
<td>12</td>
</tr>
<tr>
<td>2.1</td>
<td>Graph Based WFC neighbor examples</td>
<td>16</td>
</tr>
<tr>
<td>2.2</td>
<td>Graph Based WFC Adjacency Rules examples and the corresponding output results from Kim et al. [12].</td>
<td>16</td>
</tr>
<tr>
<td>2.3</td>
<td>Citygen roads</td>
<td>18</td>
</tr>
<tr>
<td>2.4</td>
<td>Citygen Down-town and Suburban lots</td>
<td>19</td>
</tr>
<tr>
<td>3.1</td>
<td>Geometry-Buffer example for Deferred Rendering</td>
<td>22</td>
</tr>
<tr>
<td>3.2</td>
<td>Final output of Deferred Rendering.</td>
<td>24</td>
</tr>
<tr>
<td>3.3</td>
<td>Scene Editor Window interface</td>
<td>28</td>
</tr>
<tr>
<td>4.1</td>
<td>Illustration of a single step of SCA graph expansion</td>
<td>32</td>
</tr>
<tr>
<td>4.2</td>
<td>Object Propagation Pattern editor</td>
<td>34</td>
</tr>
<tr>
<td>4.3</td>
<td>Labelled diagram for node bounding sphere properties.</td>
<td>35</td>
</tr>
<tr>
<td>4.4</td>
<td>Example of a pattern that a user can supply to the Graph-Based WFC (GWFC) algorithm</td>
<td>40</td>
</tr>
<tr>
<td>4.5</td>
<td>Propagation stage of GWFC solver step</td>
<td>46</td>
</tr>
<tr>
<td>4.6</td>
<td>Initial Boundary Example</td>
<td>47</td>
</tr>
<tr>
<td>4.7</td>
<td>Expansion of boundary for a single GWFC solver step</td>
<td>47</td>
</tr>
<tr>
<td>4.8</td>
<td>Pattern and Object Database editor</td>
<td>48</td>
</tr>
<tr>
<td>4.9</td>
<td>Adjacency requirements example</td>
<td>48</td>
</tr>
<tr>
<td>Algorithm</td>
<td>Page</td>
<td></td>
</tr>
<tr>
<td>--------------------------------------------------------------------------</td>
<td>------</td>
<td></td>
</tr>
<tr>
<td>1  Algorithm for finding a new node domain based on pattern $\mathbf{x} \in D_{\text{prop}}$ for node $N_{\text{prop}}$</td>
<td>37</td>
<td></td>
</tr>
<tr>
<td>2  Algorithm for SCA propagating of $n$ nodes around node $N_{\text{prop}}$</td>
<td>38</td>
<td></td>
</tr>
</tbody>
</table>
LIST OF NOMENCLATURE

Abbreviations

API  Application Programming Interface
CSP  Constraint Satisfaction Problem
GPU  Graphics Processing Unit
GWFC Graph based Wave Function Collapse
PCG  Procedural Content Generation. We also use PCG to refer to Procedural Content Generators for brevity.
SC-GWFC Space Colonization Graph-Based Wave Function Collapse
SCA  Space Colonization Algorithm
UI   User Interface
WFC  Wave Function Collapse

Other

node    is an object in the scene graph and directly represents an object that is visible to the user. Nodes can either be solved or unsolved. Solved nodes have a known type and only a single value in their domain $D$. Unsolved nodes still have uncertainty of their type, and have a domain larger than one.

pattern is used in a similar way to the original WFC algorithm [10]. Patterns are adjacency rules that specify what type of node needs to be near the current node for it to be valid. Figure 4.4 shows an example of these rules. A pattern
represents only a single type, but a single type may have multiple patterns representing it.

propagation node is the node SCA is using to expand the existing GWFC graph by placing new unsolved nodes within the scene. It is the node SCA expands the scene from in any single step. Figure 4.1 shows the propagation node and the resulting SCA propagation.

propagation pattern is associated with every type. The propagation pattern supplies information to SC-GWFC about how objects like trees and buildings appear in relation to their neighboring objects. They are used only during the SCA phase of SC-GWFC. Figure 4.1 shows an example propagation pattern.

type of an object describes its visual appearance and its propagation pattern. Object types are created by the user and can represent any collection of assets. These can be things like trees, houses, and rocks. The types exist as an organisational tool and are what the pattern solved for by GWFC represents.

Symbols

D Node domain. The set of types an unsolved node could be assigned.

N Node in the 3D scene. Visible to the user.

S Set of adjacency patterns, supplied by user.

T Set of object types patterns, supplied by user. These are 3D assets and models that can be rendered on scene. T is the set of types of object a user wants their scene populated with.
This section provides an overview of Procedural Content Generation (PCG) and its applications. We also discuss, at a high level, the limitations of current systems that motivated this work. We also briefly discuss some of the challenges faced when attempting to design and implement general PCG algorithms. The end of this section provides background on Space Colonization Algorithms (SCA), Constraint Satisfaction Problems (CSP), and Wave Function Collapse (WFC), each of which are essential ideas in our proposed method, SC-GWFC.

1.1 Introduction

Procedural Content Generation (PCG) refers to the algorithmic construction of content with limited human contribution. In PCG, “content” refers to the assets, levels, environments, or any other game-related elements that are generated algorithmically rather than manually. This includes things like terrain, textures, objects, characters, animations, music, sound effects, and more. PCG creates game content automatically, which can save substantial time and resources when compared to creating that same content by hand. These resources are both system resources, as that content may not need to be saved on disk; and money, through time saved during content authoring. Additionally, PCG can introduce variety and unpredictability in game content, making games more engaging and replayable.

Building general PCG systems, that work with multiple kinds of content, present several challenges due to the variety and complexity of content required by games.
Terrain, textures, objects, and characters all require different domain-specific methods to generate plausible outputs that align with the intent and design goals of the user. The PCG algorithms used to generate these elements must often be tailored to the specific needs of each content type. General PCG systems must be flexible enough to meet the user’s expectations and allow the user to modify the generated content. For example: This thesis focuses on an area of PCG called “Levels and Scenes”, so the final output should have features that align with expectations of placement, scale, and location of objects. However, this is often a non-trivial task due to the unstructured nature of 3D scenes.

1.2 Motivation

The motivation behind this project stemmed from an interest in both 3D rendering and in exploring the field of procedural content generation (PCG). The objective was to create a PCG solution that not only offered a more general solution for creating diverse content, but also prioritized usability across a wide range of domains. It has been observed that the adoption of complex PCG algorithms remains limited due to the inherent difficulties in utilizing them effectively [21]. There is a lack in availability of general PCG systems that can cater to the needs of various games without requiring extensive custom implementation. Furthermore, the challenge of generating open-world scenes, which lack regular spatial subdivisions like patterns or grids, presented an intriguing problem. The ability to place objects anywhere within such scenes, with varying orientations, adds an additional layer of complexity to any solution. These factors drove the development of this project. We wanted to design a specifically tailored solution to address the challenges of generating content for open-world scenes and environments.
1.3 Contributions

This thesis focuses on an area of PCG called “Levels and Scenes”. We specifically focus on outdoor and open-world scene generation, so, we place objects by calculating their location and orientation in 3D space. The goals of this project were to build an algorithm capable of generating scenes in an unstructured 3D environment. Existing Wave Function Collapse (WFC) based methods rely on an existing subdivision of space [10, 12]. We wanted to take the constraint satisfaction capabilities of WFC and apply them to a 3D environment of multiple scales. That is, objects with variable scales so that we could generate scenes composed of small and large objects. Our proposed method integrates Space Colonization Algorithm (SCA) with Graph-Based Wave Function Collapse (GWFC) by running the two algorithms in independent but coupled stages. This combined procedural content generator, Space Colonization Graph-Based Wave Function Collapse (SC-GWFC), integrates the constraint based capabilities of WFC with the ability of SCA to populate arbitrary 3D volumes. This project explores the possibilities of using our proposed SC-GWFC algorithm in a 3D environment through our Scene Modeling Application. The Scene Modeling Application allows users to experiment with GWFC algorithms. Additionally, we run a number of experiments with different heuristic settings for SC-GWFC to find a set of settings that align with our project goals. For more details of SCA and GWFC see Sections 1.8 and 4.6 respectively. We describe our Scene Modeling Application in Chapter 3 and the implementation of SC-GWFC in Chapter 4.

Our contributions include:

- Our proposed SC-GWFC algorithm.
- Experiments with different heuristic settings for SC-GWFC.
Testing of SC-GWFC on several types of 3D objects. Our results were evaluated qualitatively and for performance to determine the effectiveness of the SC-GWFC algorithm for interactive 3D scene generation.

Our Scene Modeling Application, which is created to supply an environment for exploring procedural generation of 3D scenes. Here, experiments with SC-GWFC are performed. This modeling application also allows for real-time interaction and immediate feedback. Users can immediately observe the effects of different SC-GWFC configurations and make adjustments to those configurations.

Performance evaluation of SC-GWFC demonstrating that the algorithm can be used for interactive scene generation and editing.

Identification of the heuristic settings in SC-GWFC that result in the best integration of the properties of SCA and GWFC.

1.4 PCG Search Spaces

In procedural content generation (PCG), a “search space” refers to the range of outcomes, or solutions, that an algorithm must search through to generate content that meets the desired criteria. The search space for a PCG algorithm can vary depending on the type of content being generated and the specific requirements of the game it may be used in. Common PCG search spaces include:

- Terrain: In terrain generation, the search space could be several types of terrain features such as hills, mountains, valleys, lakes, and rivers. Terrains can be represented as both 2D and 3D data structures [18].
• Textures: In texture generation, the search space may consist of color combinations, patterns, and shapes that can be applied to various surfaces. For example: WFC was originally developed to mimic an input texture by generating textures with similar local features [10].

• Objects: In object generation, the search space might be types of objects such as trees, rocks, buildings, or vehicles. Object generation is usually limited to the geometry of the 3D model that represents the object. Specialized methods exist for trees [15, 23, 3], and include entire tree ecosystem simulations [7].

• Levels and Scenes: In level generation, the search space might be compositions of different rooms, corridors, and obstacles that are be combined to create a challenging and engaging experience. Often, levels are designed to be navigated from some start position to some end location, such as a maze, and scenes have a less strictly defined objective. There are many existing methods for generating urban scenes [14, 11, 8]. There are also numerous methods for level generation [13, 1, 5].

• Characters: In character generation, the search space could be attributes such as appearance, abilities, and behaviors.

• Puzzles: In puzzle generation, the search space might be different elements such as objects, enemies, and obstacles that must be solved to progress. For example: Kim et al. [12] generate Sudoku puzzles.

• Music and Sound Effects: In music and sound effect generation, there might be notes, rhythms, melodies, and sound effects used to enhance the game experience.

• Game rules and entire games [6, 4]
The search space for a PCG algorithm will depend on the specific needs and requirements of the game being developed and the type of content generated. This thesis focuses on “Levels and Scenes”, that is, we place objects in outdoor scenes by calculating their location and orientation in 3D space.

1.5 PCG Challenges

Over the past three decades, researchers have introduced new methods for generating various types of content, from textures to entire virtual cities. In the context of virtual world modeling, specific procedures have been proposed for different aspects and features in scenes. However, despite these advancements, there has not been a significant shift from manual to semi-automated virtual world modeling. Three primary factors cause this lack adoption: the complexity and lack of intuitive control over traditional procedural content generation methods, difficulties controlling the generated results, and challenges in integrating outputs from different procedural methods into a coherent and cohesive virtual world [19, 21].

1.5.1 Representing Style

Current Procedural Content Generators (PCGs) can create content that embodies a specific style. However, there is a challenge in developing methods that can learn or infer a particular style and generate content accordingly. For example: Human artists are capable of studying other works of art and mimicking their style. Given an example, it is possible to learn that style. Many existing PCGs work to generate content that matches a specific style, genre, and search space. They often need to be custom tailored for that content. This limitation hinders the creation of diverse and stylistically consistent content [21].
1.5.2 General Content Generators

Most PCGs are designed to generate a single type of content for a specific game. This specificity means the opportunity to reuse these algorithms to generate content in other environments is limited. Plug-and-play solutions, where PCGs can generate multiple types of content, are rare [21]. This lack of versatility makes it difficult to use PCGs across different projects or within different contexts. Often each type of content, or, each search space, has a variety of algorithms that are custom tailored to generating content which is feasible for that domain.

1.5.3 Interfaces and Controllability for PCG systems

Many existing PCGs are not easy for a human to control. Many classic PCGs take no parameters at all. Their only input is a random seed, and a level is generated as an output. In many cases, this kind of generation is useless to a user (or a game). Solver-based PCGs offer alternative ways of specifying objectives [21] where a user might declare the desired properties of an object. In declarative modeling, designers state their intent by encoding objectives. An example of a designer’s intent could be to have a clear line of sight between locations in a virtual world [19]. Other research explores the combination of multiple content generation methods into hybrid methods by “nestling” one method within another [22].

The potential of PCG to produce complex artefacts and save development time motivates further research into this area. However, it is not possible to address all limitations of traditional PCG within this thesis. So, we evaluate our method for performance and its ability to satisfy a user’s requirements. We are interested in providing interactive feedback and simple constraints. Style representation, and other challenges in the field, are left as a future work.
1.6 Constraint Satisfaction Problems

In this thesis, we view the construction of a scene as Constraint Satisfaction Problem (CSP), and use a constraint solver to build a solution to those constraints. The placement of an object in a scene must satisfy a set of constraints provided by the user.

A CSP is a type of problem that can be expressed in terms of variables, domains, and constraints. A CSP solution is a set of values that satisfy the set of constraints given to the solver. The variables are the values being solved for, the domains are the range of values allowed for each variable, and the constraints are restrictions on the values the variables can take. For example: In our method, a variable is a location where an object could be, an object type is a value, a domain is the set of types an object at a location could be, and constraints are the required neighbors of an object type.

The original Wave Function Collapse (WFC) [10] was used to generate textures, and, in the context of WFC-style image generation, a variable is associated with each pixel in the output image (see the next section, Section 1.7, for more details on WFC).

In a CSP, variables are associated with domains of possible values, and constraints define the relationships between these variables. The property arc consistency describes the enforcement of consistency between pairs of variables connected by constraints. Arc consistency is important since it allows for reducing the search space of a CSP while only considering a pair of variables. Two variables are said to be arc consistent if each admissible value of the first is consistent with some admissible value of the second.
The concept of arc consistency is based on the notion of the arc, which represents a directed link between two variables connected by a constraint. An arc \((X, Y)\) between variables \(X\) and \(Y\) is said to be arc-consistent if, for every value \(a\) in the domain of \(X\), there is at least one value \(b\) in the domain of \(Y\) such that \((a, b)\) satisfies the binary constraint between \(X\) and \(Y\). So, an arc is considered consistent if it is impossible to find a value for \(X\) that violates a constraint with any value of \(Y\).

To enforce arc consistency, an algorithm would iterate over all the arcs in the CSP and remove values from the domains of variables to make them arc consistent. This is done by checking each value in the domain of the variable associated with the starting point of the arc and removing any value that does not have a compatible value in the domain of the variable associated with the endpoint of the arc.

The process of enforcing arc consistency involves propagating constraints and updating the domains of variables until no further changes can be made. This process progressively narrows the domain of possible assignments for each of the variables.

### 1.7 Wave Function Collapse

Maxim Gumin’s Wave Function Collapse (WFC) algorithm is an example-driven image generation algorithm that takes patterns from a given input and uses them to model outputs with similar local properties [10]. It has been used in the generation of images, 2D textures, and more recently, 3D objects. More generally, WFC can be viewed as a CSP.

As described by Karth and Smith [10], the original WFC algorithm, performs four key tasks: it extracts local patterns from an input image; it processes those patterns to speed up constraint checking; it incrementally generates the output image assigning
values to grid locations in that image; and, finally, it writes the total assignment back
as an image in the same format as the input [10].

Karth and Smith [10] also outline Gumin’s WFC generation process. During the
generation process, decision variables (grid locations in the image), are repeatedly
assigned by picking a value from their domains. Figure 1.1 shows WFC assignments to
an image grid. This example assigns color values to tiles. In constraint solving, solvers
typically maintain information about the remaining possible values for unassigned
variables (the domain) in addition to the current partial assignment. In Gumin’s Wave
Function Collapse, this information is stored in a table called “wave,” which broadly
resembles a probabilistic wave function. The entries in this table are Boolean values
that indicate if the algorithm can still assign a specific value to a location. Initially, all
coefficients in the wave table are set to true, representing unreduced initial domains
for each decision variable. As the assignment and propagation processes repeat, the
domains of variables are gradually reduced, causing coefficients to change from true
to false. Gumin’s algorithm does not implement backtracking but instead performs a
global restart if variable assignments conflict [10].

There are a number of limitations in the WFC algorithm. One is that it can only
solve problems irrespective of the order of the solution [12]. This means that it cannot
ensure a specific ordering of items in the output, and further search is required to
achieve the desired order. The reason for this is that WFC evaluates nodes in a
somewhat random order based on the entropy heuristic where the most constrained
variable is selected, or variable with minimum remaining values [10]. Introducing
a more complex backtracking algorithm is the only way to expand the search space
when an invalid state is reached [10].
Figure 1.1: Wave Function Collapse with partially complete output. Karth and Smith [10].

1.8 Space Colonization Algorithm

Space Colonization Algorithm (SCA) is a method designed to populate a 3D volume with a natural tree-like structure. It has been successfully used to generate geometry for tree models [15], and city road generation [8].

In general, SCA follows three stages [15, 8]. These are also outlined in Figure 1.2:

- (a) At the beginning of generation, the space within an envelope is seeded with a set of attraction points. These points indicate available space for branch growth.

- (b) Then, a tree skeleton is formed by an iterative process. In each iteration, new nodes are added to extend the skeleton in the direction of nearby attraction points.
• (c) Finally, once all attraction points are removed, the tree geometry can be modeled using generalized cylinders.

![Figure 1.2: SCA initial stage of tree skeleton generation example from Runions, Lane, and Prusinkiewicz [15].](image)

A tree is composed of tree nodes. The tree growth is influenced by a set of attraction points that dictate how the tree grows and develops. Attraction points are initially scattered in a desired volume, along with an initial tree node. The tree will grow from the initial tree nodes, towards the attraction points placed in the volume. An attraction point can only influence the closest tree node within a given radius of influence. Once the tree nodes grow to within the kill distance of an attraction node, that attraction node is eliminated. The tree growth and attraction node elimination is repeated until all attraction points are consumed [8, 15].

However, there are several limitations of the Space Colonization Algorithm that prevent it from being used in a general PCG context:

• Complexity: SCA is computationally expensive, which makes it difficult to use for large-scale PCG applications. It requires a large amount of memory and computational resources to simulate the growth of a single tree.

• Limited applicability: SCA is primarily designed for generating organic structures such as trees and plants, which limits its applicability to other types of content. It may not be suitable for generating geometric structures like buildings or vehicles.
• Lack of control: The algorithm is highly dependent on the initial parameters set by the designer, and it can be challenging to predict how the generated content will evolve. As a result, it may be difficult to achieve specific design goals or ensure a consistent level of quality. However, the algorithm is iterative, which allows it to be used interactively.

• Difficulty in generating specific shapes: SCA works by simulating the growth of a tree, and as a result, it may be challenging to generate specific shapes or structures. While it is possible to guide the growth of the tree using attractor nodes and a target shape, it may not always produce the desired outcome.

• Repetitive structures: SCA generates repetitive structures, which is beneficial for fractal structures, but this may not be desirable for all PCG applications.

Overall, Space Colonization has several advantages that make it a good candidate for generating organic structures. But, it is difficult to use in general PCG applications. In the context of this thesis, its most important property is that it can be used to progressively expand a tree like structure into an arbitrary volume.
Chapter 2

RELATED WORK

There have been many investigations into using constraint solving methods for PCG. It works for puzzle, narrative, and level design by appropriate choice of constraints. We focus on techniques that have been applied to scene and level generation. This section discusses some of the related work in scene generation and level generation methods.

2.1 Answer Set Programming

The level-based research is focused on constraint problems called Answer Set Programming (ASP). Shaker, Togelius, and Nelson [16] suggest that in order to implement constraint solving in practical applications, a specific language and solver are required. By utilizing ASP, which is a logic-programming method that offers strong support for constrained generation, a user can declare their requirements in a text based format. Although there are alternative approaches for PCG with constraint solving, ASP stands out due to its well-established programming language, AnsProlog, and reliable tools. ASP enables the specification of both game logic and constraints in a single language, making it a suitable choice for developing PCG systems based on constraints. ASP is used to generate grid maps and is not directly applicable to 3D environments, so we do not explore it in this thesis.
2.2 Graph-Based Wave Function Collapse

Kim et al. [12] propose a graph-based version of the Wave Function Collapse (WFC) algorithm for procedural content generation in 3D worlds. Since the authors utilize a graph data structure, this method can be easily integrated with an existing navigation mesh data structure. The proposed method can handle input data in the form of text descriptions that specify a graph structure and uses Poisson disc sampling to obtain nodes on the navigation mesh. The adjacency relation of the nodes, shown in Figure 2.2, is used to determine the placement of content elements, and the Voronoi diagram is used to discretize the 3D world and construct a 3D vector field for non-navigation-based terrain. The proposed method offers controllability and scalability for generating non-grid shape content with adjacency relations. Kim et al. provide experimental results demonstrating the effectiveness of the proposed Graph-Based Wave Function Collapse (GWFC) algorithm for procedural content generation of scenes divided by a navigation mesh.

Kim et al. [12] demonstrate GWFC with neighbors for arbitrary cells and a regular rectangular grid. Figure 2.1 shows Neighbors of the Sudoku game grid are determined by the game rules. The right two examples show neighborhoods based on adjacency [12].

In the context of 3D content generation, this technique relies on an existing structure already in the game world. It does not introduce any method of generating that graph beyond splitting a 3D space into a set of similarly sized cells. These cells are irregular, and have an irregular number of neighbors, so this approach solves the problem of WFC on an unstructured graph. However, the cell size does not address the problem of differently sized objects when considering the search space of scenes and object placements.
Figure 2.1: Graph Based WFC neighbor examples from Kim et al. Neighbors for arbitrary cells and a regular rectangular grid. This shows examples for a Sudoku game grid and a Voronoi non-grid. Neighbors of the Sudoku game grid are determined by the game rules. The right two examples show neighborhoods based on adjacency [12].

Figure 2.2: Graph Based WFC Adjacency Rules examples and the corresponding output results from Kim et al. [12].

Another noteworthy extension to WFC is Tessera. Newgas [13] propose Tessera, a tool that addresses the physical usability issues of WFC. They provide user friendly tools that make constraint editing more practical. These improvements include a User Interface (UI) to paint tiles for encoding adjacency rules, multi tile modules so that a grid can have tiles that cover multiple cells, and additional global constraints. However, this tool is only applicable to the same regular grid structures used by the original WFC.
2.3 Space Colonization for Procedural Road Generation

Dias Fernandes and Fernandes [8] propose a modified Space Colonization Algorithm for procedurally generating road networks. The algorithm is based on the principles of Space Colonization, where tree-like structures are grown from a set of “seed” points. The seed points are initially placed on the terrain, and a set of control points are generated around them. These control points are used to attract new road segments towards them, forming a path between the seed points. As more segments are added, the control points are removed to continue extending new road segments into new areas.

One of the strengths of this algorithm is its ability to generate a wide range of road networks, from simple straight paths to complex, meandering networks. Also, the algorithm can generate road networks that conform to a terrain.

One of the main limitations is that the generated road networks are not guaranteed to be the most efficient or optimal path between the seed points. This is because the algorithm is based on a heuristic approach, where the road segments are grown towards the control points based on a simple set of rules. Additionally, interconnections between branches in the road network must be added after the first phase of the algorithm. The first phase follows the original SCA algorithm and produces a road network shaped similarly to a tree. The second phase of the proposed algorithm chooses nodes on the generated graph and connects them to neighboring roads based on distance. This relies on snapping nodes to new positions and forming connections that are close to an “ideal angle,” which the authors define as parallel, or perpendicular, based on the situation.
SCA can generate road networks procedurally. Its ability to generate a wide range of road networks that conform to the terrain makes it a valuable tool for scene design. Similarly, we use SCA to populate a terrain with positions for objects, however, we place arbitrary objects on the terrain, not control points for road segments.

2.4 City Generation

Kelly and McCabe propose CityGen, which is a procedural city generation system that primarily focuses on the generation of road networks and buildings that align with a given terrain model.

The system starts by mapping road networks onto the provided terrain model. These road networks are adjusted to better fit the underlying geometry of the terrain [11]. Figure 2.3 shows an example output from the road generation phase of Citygen. This process ensures that the generated roads follow natural contours and align with the landscape, resulting in a more realistic and visually correct city layout.

Figure 2.3: Citygen roads. Citygen initially places roads that conform to the terrain. In this primary pass, the road network is created. This figure from Kelly and McCabe [11] shows an example output from this first pass.
In addition to the road networks, CityGen also generates building footprints that conform to the road networks. The footprints are designed to fit within the contours of the generated roads, ensuring proper connectivity and alignment. Figure 2.4 is an example of these generated lot divisions. This allows for the automatic placement and generation of buildings throughout the city, saving significant manual labor in the content creation process.

![Figure 2.4: Citygen Down-town and Suburban lots. This figure from Kelly and McCabe [11] shows an example output from the lot division phase of their algorithm. It is able to generate different local structures for building lots: both regular grids and suburban like lot divisions.](image)

One aspect of CityGen is its focus on providing developers with hands-on control over the generation process. The system enables real-time interaction and feedback, allowing developers to tweak various parameters and observe the immediate results. This iterative and interactive workflow empowers developers to have fine-grained control over the generated city, ensuring that the final output meets their specific design goals and requirements.

By automating the placement of road networks and buildings, while offering real-time control, CityGen streamlines the process of generating detailed cities within a given terrain. The system reduces the manual effort required from developers, enabling them to focus on higher-level design decisions and creative aspects, ultimately
leading to visually correct city generation. Our method is different from CityGen as we attempt to model scene generation more generally, we do not have explicit road placement and building placements phases since we do not explicitly generate a city. CityGen uses an explicit subdivision of city “blocks” to create “lots” in which buildings can be placed.
In this chapter, we describe the implementation of the Scene Modeling Application that enables us to integrate PCG into the editing process. The focus of this application is to facilitate the extension and modification of the scene structure while a PCG algorithm is operating. This chapter describes the design and features of our Scene Modeling Application.

At a high level, the scene modeling application consists of a few separate systems:

- **Renderer**: The renderer is responsible for managing resources required by the OpenGL API. This includes texture, buffer, shader, and vertex data. The renderer also abstracts away the details of shaders for each graphics technique. Rather than interacting directly with the OpenGL API, a developer creates drawable objects and modifies their properties to change the visual appearance.

- **Resource Management**: Resource management is responsible for finding and loading image, material, and model data from disk. Any resource can be requested from multiple parts of the application. The Resource Manager is responsible for keeping references to those loaded resources and sharing them as required.

- **Scene Graph**: The scene graph is responsible for managing parent-child relationships among nodes. Since this rendering engine was designed to work in 3D, the focus of the scene subsystem is managing the hierarchical relationships, updating transforms, and notifying other parts of the application when these data change.
- User Interface (UI): imgui (an immediate mode Graphical User interface for C++ with minimal dependencies https://github.com/ocornut/imgui) is used to build an interface for editing both the parameters for SC-GWFC and the scene.

### 3.1 Deferred Rendering

Deferred rendering is a rendering technique commonly used in computer graphics to efficiently render complex scenes with multiple light sources. It involves splitting the rendering process into two passes to reduce unnecessary lighting calculations.

![Geometry-Buffer example for Deferred Rendering](image)

**Figure 3.1: Geometry-Buffer example for Deferred Rendering.** These images show the contents of the G-Buffer after the Geometry Pass. This is only the material information, other position information is necessary to complete the lighting calculation.

In traditional forward rendering pipelines, the lighting calculations are performed per-pixel during the rendering of geometry. That is, every surface in the camera’s view results in a full lighting calculation, even for surfaces occluded by others. Forward
rendering is also computationally expensive when there are many lights in the scene, or there are complex shaders, as the lighting calculations need to be evaluated for each pixel for each light, including those invisible in the final image.

Deferred rendering addresses these performance issue by decoupling the geometry and lighting calculations into two separate rendering steps: (1) the geometry pass, and (2) the lighting pass.

In the geometry pass, the scene is rendered into multiple buffers called the G-buffer. The G-buffer typically consists of several textures, including the position, normal, albedo, and additional material properties of each pixel in the scene [20]. Figure 3.1 shows examples of each of these buffers. These buffers store information about the geometry of the scene without performing any lighting calculations. Consequently, geometry pass shaders are only responsible for transforming vertices and copying material properties.

In the lighting pass, lighting calculations are only performed on the pixels that are visible in the G-buffer. This effectively acts as a visibility check since each pixel in the G-buffer can only take on a single value. For each light source, the lighting pass retrieves the necessary data from the G-buffer and calculates the lighting contribution for each pixel. This significantly reduces the computational cost compared to per-pixel lighting calculations [20]. The final output, Figure 3.2, is the combined output of the lighting calculations for all lights in the scene.

Deferred shading has the following advantages over traditional forward shading:

- Multiple light sources: Deferred rendering can handle scenes with numerous light sources efficiently, as the lighting calculations are performed only on visible pixels. Decoupling geometry from shading results in a complexity for shading
Figure 3.2: Final output of Deferred Rendering.

of $O(Objects + Lights)$ [20], versus $O(Objects \cdot Lights)$, of traditional forward rendering.

- Less wasted shading: Deferred rendering results in much less wasted shading than in forward rendering [20].

- Flexibility: Deferred rendering allows for various lighting models and complex shading effects since all the required information is available in the G-buffer.

- Post-processing: The G-buffer can be used for post-processing effects such as depth of field, motion blur, or screen-space ambient occlusion, as it contains all the necessary data for these effects. Thaler and Wien [20] outlines how these different data can be used for each type of effect.

However, deferred rendering also has some limitations:

- Memory requirements: The G-buffer can consume a significant amount of memory, especially in scenes with high-resolution textures and additional material properties.

- Transparency and anti-aliasing: Handling transparent objects and anti-aliasing can be more challenging in deferred rendering, as they require additional techniques and may not fit well into the deferred pipeline. Transparency is not
supported within a deferred renderer as the first hit with a surface is the only information recorded [20].

In this project deferred rendering is used along with a technique like that of Shishkovtsov [17] where the material-id is stored in the G-Buffer during the geometry pass. Multiple materials and many material properties are supported with a material-id stored in the G-Buffer. During the lighting pass, the shader behavior changes dynamically. This material-id is used to index a shader storage buffer which contains all material properties. Unlike color, many material properties are fixed for an entire object. These fixed properties do not need to be stored on a per pixel basis like color and normals, which are texture mapped properties. If, for example, specular maps were to be used, there would need to be another channel in the G-buffer for specular data.

3.2 Managing OpenGL Resource Handles

In OpenGL, resources like textures, shaders, and buffers need to be explicitly created and destroyed. The functions that allocate resources in OpenGL return object handles as integers: with unique values for each object. Tying these integer resource handles to object handles in C++ only requires creating a wrapper object with a constructor and destructor that call the corresponding `glCreate` and functions for each resource type. Tying OpenGL resource lifetimes with object lifetimes in C++ is advantageous as it allows for simpler resource management in the C++ code. The wrapper object can have their lifetimes managed with smart pointers like `std::shared_ptr<>`.
3.3 Scene Graph

A game engine scene graph is a hierarchical data structure that represents the objects and their relationships within a scene. It provides a way to organize and manage the objects in a game world, enabling efficient rendering, collision detection, and other game-related operations. This project uses a scene graph to maintain scene information. It also acts to generate events on state changes created by the user.

At the top of the scene graph is the root node, which serves as the parent for all other nodes in the hierarchy. Each node in the scene graph represents an object in the game world, these objects can be things such as: characters, terrain, props, lights, and cameras. These nodes can have child nodes, forming a tree-like structure.

In this project, the scene graph is used to manage the following data:

- Transformation Hierarchy: Each node in the scene graph has transformation information associated with it such as position, rotation, and scale. These transformations are applied hierarchically; child nodes inherit the transformations of their parent nodes. This allows for efficient hierarchical transformations of objects in the game world. Since the nodes notify when this data changes, children nodes can update their cached transformation information, and only update it when required. Transformation relationships are parent-child relationships, allowing for hierarchical structuring of objects. This hierarchy builds relationships between objects an can be used to create a nested structure for organizing game entities. Usually, related nodes appear as children under another node. This organization is used to differentiate spawned objects from things like the camera, ground plane, character, and terrain in this project.
• Resource Management: The scene graph also acts to manage resource usage in the scene modeling application. As an example, a visual node represents a visible object in the scene. A visual node must own the mesh data used to display it. When a visual node is created, it will create that mesh data resource, and register it with the renderer. When the user assigns a model to that visual node, the node will then modify its mesh data. Then, when that node is removed from the scene, it will clean up the mesh data. This management hides the specifics of the renderer from the scene graph.

• Scene Management: Nodes allow for insertion, removal, and modification of their children nodes. These are scene changes, such as spawning or despawning objects, activating or deactivating objects based on gameplay conditions, or creating and managing object instances during gameplay. Scene management expands on resource management since all resource lifetimes are tied to node lifetime.

By utilizing a game engine scene graph, it becomes simpler as a developer to organize and manipulate objects within a game scene. It provides a structured representation of the world, along with scene and resource management. The scene graph is a foundational component in many game engines, so implementing one for this project made experimenting with SC-GWFC easier.

By utilizing a scene graph in this project, both user and generator triggered changes can be handled in the same way. It provides a structured representation of the world, enabling scene and resource management. The scene graph is a foundational component in many game engines, so implementing one for this project made it simple to connect the generator state with the rendered scene state.
Figure 3.3: Scene Editor Window interface. The Scene Editor Window provides an interface to the properties of nodes in the scene while the application is running. The UI allows for selection and editing of transform data. When a node is selected, a gizmo is drawn over it with handles for transform editing. Developers can write custom UI functions draw additional information to the Node Properties tab.
METHOD: SPACE COLONIZATION AND GRAPH-BASED WAVE FUNCTION COLLAPSE

This thesis focuses on the “Levels and Scenes” search space, so, we are attempting to place objects in outdoor scenes by calculating their location and orientation in 3D space. The goals of this project were build an algorithm capable of generating scenes in an unstructured 3D environment and then use it to generate procedural placements for 3D objects in a 3D environment. This chapter describes the implementation of our proposed Space Colonization and Graph-Based Wave Function Collapse (SC-GWFC) algorithm.

Space Colonization Algorithm (SCA) is integrated with Graph-Based Wave Function Collapse (GWFC) by running the two algorithms in independent but coupled stages. SCA is used to expand the existing graph by placing new possible object locations in the scene, and GWFC is used to determine if the placement of those possible locations are valid and the object type for those new locations. SC-GWFC operates in two stages: (1) SCA propagation, described in Sections 4.2 - 4.5; and (2) GWFC evaluation, described in Sections 4.6 - 4.9.

In the first stage, SCA expands the existing GWFC graph by placing new unsolved nodes within the scene. The SCA algorithm is known for simulating the growth of natural structures, such as trees and branching patterns [8, 15], and, here is used to find potential placements for unsolved nodes. SCA growth determines the placement and connectivity of new nodes based on proximity and repulsion “forces” from existing nodes. New nodes are placed following propagation patterns specified by the user.
This allows different types of objects to have different relative orientations with their respective neighbors.

In the second stage, GWFC can optionally evaluate the expanded graph. When it operates on the expanded graph, it validates and collapses the object domains for the nodes. GWFC tries to satisfy the requirements supplied by the user by finding valid object types for neighboring nodes. It checks the constraints supplied by the user to ensure that the generated object layouts align with the design restrictions.

SCA expands the scene by adding new nodes, while GWFC validates and assigns object types to those nodes. This iterative process can be repeated, allowing for further modifications and additions to the scene until the desired result is achieved. Since both algorithms can be run independently, the user can choose to expand the graph multiple times with SCA without solving the object types using GWFC.

4.1 SC-GWFC Terminology

For reference:

- The *type* of an object describes its visual appearance and its propagation pattern. Object types are created by the user and can represent any collection of assets. These can be things like trees, houses, and rocks. The types exist as an organisational tool and are what the pattern solved for by GWFC represents.

- A *pattern* is used in a similar way to the original WFC algorithm [10]. Patterns are adjacency rules that specify what type of node needs to be near the current node for it to be valid. Figure 4.4 shows an example of these rules. A pattern represents only a single type, but a single type may have multiple patterns representing it.
• A propagation pattern is associated with every type. The propagation pattern supplies information to SC-GWFC about how objects like trees and buildings appear in relation to their neighboring objects. They are used only during the SCA phase of SC-GWFC. Figure 4.1 shows an example propagation pattern.

• A propagation node is the node SCA is using to expand the existing GWFC graph by placing new unsolved nodes within the scene. It is the node SCA expands the scene from in any single step. Figure 4.1 shows the propagation node and the resulting SCA propagation.

• A node is an object in the scene graph and directly represents an object that is visible to the user. Nodes can either be solved or unsolved. Solved nodes have a known type and only a single value in their domain $D$. Unsolved nodes still have uncertainty of their type, and have a domain larger than one.

4.2 Space Colonization Algorithm (SCA)

In the first stage of SC-GWFC, SCA expands the existing GWFC graph by placing new unsolved nodes. We use SCA-like growth to progressively find potential placements for unsolved nodes. SCA growth determines the placement and connectivity of new nodes based on proximity and repulsion from existing nodes.

As outlined in Section 1.8, SCA generally follows three stages: At the beginning of generation, the space within an envelope is seeded with a set of attraction points [15, 8]. These points indicate the availability of space for branch growth. Then, a tree skeleton is formed by an iterative process. In each iteration, new nodes extend the skeleton in the direction of nearby attraction points. Finally, the tree geometry can be modeled using generalized cylinders [15]. However, we do not follow this same
process. Rather, we rely on repulsion alone to push new nodes away from existing ones.

![Figure 4.1: Illustration of a single step of SCA graph expansion. Left, a node is selected as the propagation node (green). Right, the selected node is expanded according to a placement pattern in its domain. Scene graph events on node insertion update the adjacency information (dashed lines). Then, the newly spawned nodes (green) will used in the next iteration of SCA propagation.](image)

The *propagating node* is the node SCA expands the scene from in any single step. Figure 4.1 shows the propagation node and the resulting SCA propagation. The figure shows a case where a *branching factor* setting of three is used. In our experiments, we used a fixed branching factor of 20, meaning that one iteration of SCA results in 20 new nodes being added to the scene.

### 4.3 SCA Propagation Patterns

During the SCA propagation step, each domain value in the propagating node results in new nodes being placed in the scene. The placement of new nodes is guided by local patterns that represent the orientation of objects in proximity. These *propagation patterns* go beyond the adjacency requirements satisfied by GWFC. Propagation patterns supply information to SC-GWFC about how objects like trees and buildings appear in relation to their neighboring objects. For example, trees tend to exhibit random orientations relative to other trees, whereas urban objects such as buildings
have a more ordered arrangement. To account for these orientation differences, the SC-GWFC algorithm has additional properties that are specified by the user to guide object placement.

These propagation patterns capture the spatial relationships between objects, and allow for variation in both orientation and location. This information lets SCA place new nodes in locations that, ideally, conform to patterns observed in real-world scenarios. The position and orientation of a new object depends on those of the previously placed object during the SCA propagation process.

SCA propagation patterns are stored as lists of Oriented Bounding Boxes (OBB). Each object type has a single SCA propagation pattern associated with it. Each propagation pattern may have multiple OBBs. The OBBs in the propagation pattern model a 3D normal distribution where new nodes are allowed to be placed with respect to the propagating node.

Figure 4.2 shows the editing window for propagation patterns. In this example, there is only a single OBB. Any new node SCA places using this one may end up anywhere in this OBB, but the location will be biased toward the center due to the use of a normal distribution on each axis. The orientation requirements, appearing after the object extent parameter, lock and unlock each axis. A locked axis means that the orientation of any newly spawned node must match that of the previous. A free axis has a random orientation, and a stepped axis has a discrete number of possible orientations. In this implementation, we fix the stepped setting to four random orientations, but this can easily be added as a user configurable parameter.
Figure 4.2: The Object Propagation Pattern editor allows for visually editing the Oriented Bounding Boxes (OBB) of the placement patterns for any object type. Users can add OBBs that specify the distribution of positions objects can take with respect to a propagating node.

4.4 SCA Repulsion

In the SCA propagation phase, we place new nodes with repulsion rather than attraction like that used in Runions, Lane, and Prusinkiewicz [15]. Since we do not have an upper limit on the volume that can be grown into by SCA, the use of attraction nodes was not something we investigated, but may be interesting future work.

When a new node is placed, a repulsion parameter provided by the user is used to determine the distance at which the new node will be pushed away from existing nodes. To calculate the repulsion vector acting on a new node, the algorithm considers the cumulative repulsive “forces” exerted by all existing nodes in the scene. Each existing node contributes to the repulsion force on any new node. The calculation of this repulsion force is the negation of gravitational attraction.
The repulsion force between two nodes is based on the “mass”, distance between them, and their bounding sphere radius. The repulsion force is inversely proportional to the square of the distance between the nodes. A sphere \((c, r)\) has a center \(c\) and a radius \(r\). The repulsion vector exerted by node \(N_1\) with sphere \((c_1, r_1)\) on node \(N_0\) with sphere \((c_0, r_0)\) with \(d\) repulsion factor is calculated as follows:

\[
s = c_0 - c_1 \quad (4.1)
\]

\[
\text{repulsion}(N_0, N_1) = d \frac{s}{||s||(s \cdot s)r_0r_1} \quad (4.2)
\]

**Figure 4.3:** Labelled diagram for node bounding sphere properties.

By summing the repulsive forces from all neighboring nodes, the SCA propagation stage determines the overall repulsion vector acting on the new node. This repulsion mechanism pushes the new node spawn locations away from existing ones, resulting in node spacing affected by the desired repulsion parameter, and existing nodes.
SCA Node Neighborhood Radius  Each node in the scene has two associated spheres; a bounding sphere, and a neighborhood sphere. The bounding sphere determines when two objects overlap and one of them needs to be removed from the scene. The bounding sphere radius for a node is calculated using the weighted average diagonal size of the Axis Aligned Bounding Boxes for all the types in its domain. The weight is the weight \( w \) assigned to a pattern representing this object type. We call this the \( weightedDiagonal \) of a domain \( D \). The neighborhood bounding sphere determines what nodes can be considered in its neighborhood. When two neighborhood spheres intersect, the two nodes correspond to those spheres are marked as adjacent in the GWFC sparse graph.

4.5  SCA New Node Domains

As outlined in Section 4.3, the SCA propagation step involves placement of new nodes based on propagation patterns for each domain value. This addresses the position of new nodes, however, it does not address the assignment of an initial domain to those nodes. One option is to spawn new nodes with a \textit{full domain}, where their domain contains every possible pattern. This would closely match the original WFC [10], where the grid is initialized with a complete domain for every pixel. However, we found that this resulted in poor object selection and a uniform object layout (see Figure 5.11). To address this issue, we introduce a method of spawning nodes with a \textit{dependent domain}, where the domain of the new node depends on the domain of the propagating node SCA used to place it.

When a new node is placed with a dependent domain, we determine its domain by considering the type of the node SCA used to generate its placement. The propagating node is the node SCA uses to place the new node. A dependent domain contains all
of the domain values required by a single pattern in the propagating node. We choose
the domain values this way as it puts nodes that are expected to be adjacent in the
same neighborhood.

The user provided data is: The set of types $T$, and the set of patterns $S$. With this
information we define the multiset of requirements over $T$ as $R$. The set of patterns
then has the structure $S = \{(t, w, R_S) \mid t \in T, w \in \mathbb{R}_{>0}, R_S \subset R\}$.

Given the current propagating node $N_{prop}$, its node domain $D_{prop} = \{x_i \mid x_i \in S\}$,
the probability that a pattern $x_i$ will be selected during node collapse of $N_{prop}$ is:

$$P_{prop}(x_i) = \frac{w_i}{\sum_{j=1}^{|D_{prop}|} w_j} \quad (4.3)$$

Where $w_i$ and $w_j$ are the weights for pattern $x_i$ and the $j$th pattern $x_j$ in the domain
$D_{prop}$.

Which results in a probability that the propagating node will have a type of $t \in T$.

When propagating for pattern $x_i$, the probability that a new node will be spawned
with a dependent domain of the propagating node is $P(x_i)$. When the propagating
node is highly certain of its type (fewer types in its domain), there is a higher chance
that the spawned neighborhood will contain the required types for that high certainty
type.

Algorithm 1 Algorithm for finding a new node domain based on pattern $x \in D_{prop}$
for node $N_{prop}$

```
procedure: newDomain($N_{prop}$, $x$)
    if binomialTrial($P_{prop}(x)$) then
        $D_{new} \leftarrow$ required types $R_S$ of $x$
    else
        $D_{new} \leftarrow T$
    end if
    return $D_{new}$
```

37
Where \( \text{binomialTrial} \) is a function returns true with probability \( P_{prop}(x) \).

With the repulsion force \( f \) on the selected propagation node, the process for the SCA propagation step on \( D_{prop} \) is outlined in Algorithm 2.

**Algorithm 2** Algorithm for SCA propagating of \( n \) nodes around node \( N_{prop} \)

**procedure:** SCAPropagation\((N_{prop}, n)\)

\[
f \leftarrow 0
\]

for each node \( N \) in scene do

\[
f \leftarrow f + \text{repulsion}(N_{prop}, N)
\]

end for

for \( x_i \in D_{prop} \) do

\[
s \leftarrow P_{prop}(x_i)
\]

\[
D_{new} \leftarrow \text{newDomain}(N_{prop}, x_i)
\]

\[
r \leftarrow \text{weightedDiagonal}(D_{new})
\]

\[
k \leftarrow 0
\]

for \( k < n \cdot s \) do

\[
k \leftarrow k + 1
\]

\[
AABB \leftarrow \text{Axis Aligned Bounding Box for object } t \text{ of pattern } x_i
\]

\[
pos \leftarrow (X, Y, Z) \sim \mathcal{N}(\bar{c}_{AABB}, \frac{d_{AABB}}{3})
\]

if \( \text{pos} \cdot f > 0 \) then

\[
\text{SpawnNode}(D_{new}, \text{pos} + f, r)
\]

end if

end for

end for
Where \textit{weightedDiagonal} is the weighted average diagonal of all object types in domain $D_{\text{prop}}$ (see the end of Section 4.4 for more details on node radius), $c_{\text{AABB}}$ is the center of bounding box $\text{AABB}$, $\vec{d}_{\text{AABB}}$ is the diagonal vector between minimum and maximum points of bounding box $\text{AABB}$, and \textbf{pos} is a random location sampled from a Gaussian distribution $\mathcal{N}(\mu, \sigma^2)$. 
4.6 Graph-Based Wave Function Collapse (GWFC)

Figure 4.4: Example of a pattern that a user can supply to the Graph-Based WFC (GWFC) algorithm. Here, for there to be an object of type 13, there must be four neighbors with type 14.

Similar to Kim et al. [12], our implementation of GWFC is responsible for finding an assignment of a type to a node that satisfies pattern adjacency requirements. However, we follow a different process for propagating constraints and evaluating the validity of a value in a node’s domain. Rather than considering the pattern requirements as restricting possible neighbors for a node with a given type, we use them as a requirement for a node with the given type to exist. So, we assume that any type can be next to any other type. This is a less restrictive requirement than that used by Kim et al. [12], and WFC [10] in general, but, our graphs are more strongly connected than those generated by Voronoi cell adjacency or Delaunay triangulation, and, consequently, our nodes have many more neighbors. It was too restrictive to force a node to only have neighbors required by its corresponding pattern. We leave this type of constraint as future work. See Section 4.9 for details about checking pattern validity with this decision.

To find an assignment of a type to a node that satisfies the pattern adjacency requirements, GWFC performs two main operations:

1. it collapses the node, forcing the domain to have a single type.
2. propagates that new domain information to all neighbors.

**Node Collapse** When an unsolved node is visited by the solver, it first needs to be collapsed. When a node is collapsed, its domain is reduced to a single type [10, 12]. The probability that a type will be selected in this step is based on the weight of that pattern. Since updates are propagated during the previous iterations of the algorithm, any node domain should already contain a set of valid values for its neighborhood.

**Constraint Propagation** After a node’s domain has been collapsed, the node is considered solved. However, the domains of the nodes in that node’s neighborhood are no longer valid, so the information must propagate from the collapsed node to all of its neighbors. This propagate step iterates through all the neighboring nodes and reevaluates the validity of each type in their domain. This propagation follows a similar idea to that of the original WFC, described in Karth and Smith [10], where constraint information is used to update neighbors until no more changes are made to any node domains. Figure 4.5 shows how domain updates might be propagated to neighbors in the graph.

### 4.7 GWFC Node Visitation Order

GWFC should assign types to nodes in an order that results in the least number of invalid assignments. Original WFC, described by Karth and Smith, makes assignments in order of entropy. Lower entropy nodes have a smaller domain of available values [10]. We experiment with two different visitation orders in Chapter 5. The first is in order of discovery: nodes are collapsed in the order they are discovered. Nodes are discovered as part of the visitation process. The second order we use is the same as the original WFC algorithm [10], where we use a priority queue to pick nodes with
the lowest entropy. See Section 5 for examples showing the differences between these two evaluation orders.

In either case, the process for expanding the set of solved nodes remains the same. First, a node in the boundary set is selected for evaluation (see Figure 4.6), then GWFC will push all its undiscovered neighbors to the boundary queue. Figure 4.7 shows the state of the boundary node set after this stop. In the last step, the node is collapsed and its new domain information propagated to neighbors as described in Section 4.6.

4.8 GWFC Required Neighbors Specification

Users need to give the SC-GWFC algorithm information on what types of objects need to be near other types of objects. The Pattern and Object Database editor interface allows users to manage and customize assets and adjacency requirements for object types.

The Object section of the editor allows users to edit the asset files associated with each object type. Users specify and modify the visual representation and properties of objects. A single object type can have multiple asset files associated with it, allowing for variation in appearance for a type of object. Users can add, remove, or modify these object types.

In the Pattern section, users can specify the adjacency requirements for each object type. Patterns define the constraints governing how objects can be placed and arranged in the generated content. Users can define which objects are required to be in the neighborhood.
The interface enables users to add, remove, or modify both object types and patterns. This flexibility allows for changes to the content generation parameters based on the behavior of the generator. Users can experiment with different asset files, adjust adjacency requirements, or introduce new object types as needed.

By providing a user-interface for managing objects and patterns, the editor streamlines the customization process and allows for direct control over the PCG parameters. This flexibility and control over assets and adjacency requirements integrates this tool more tightly with the scene editing experience.

4.9 GWFC Adjacency Requirement Satisfaction

Since users give the SC-GWFC algorithm information on what types of objects need to be near other types of objects, we have to check the validity of each object type in a node’s domain with respect to its entire neighborhood.

During the propagation stage, a node neighboring another updated node needs its domain to be reevaluated. For a type in the domain to be valid, it must be possible to satisfy all of its requirements with types in the domains of its neighbors. The node must be arc consistent with all neighboring nodes.

Kim et al. [12] rely on rules for what types can appear next to other types. They use a Compatible function to determine if a tile may be assigned a type based on checking if one type may appear next to another. As described in Section 4.6, it was too restrictive to only allow neighbors required by the type pattern.

We determine if a type in a node’s domain is valid by checking if there exists a matching of neighborhood nodes to type requirements in the pattern for that type.
This lets patterns model any adjacency requirement: a type may require multiple nodes of any other type and any type may be required by any pattern.

The problem of determining the validity of a pattern in the domain then becomes a matching problem. Each requirement must be matched with a single other node, and each node can only satisfy a single requirement in the pattern. Figure 4.10 shows and example matching problem for a set of requirements and a node neighborhood. To determine the validity of a type in a node’s domain, the algorithm checks if there exists an assignment of neighborhood nodes that satisfies the type requirements specified in the pattern. This flexible approach allows patterns to model various types of adjacency requirements, including the need for multiple nodes of a specific type.

Each requirement in the pattern needs to be matched with a single neighboring node, and each node can only satisfy a single requirement in the pattern. Figure 4.11 illustrates an example matching problem, where requirements are matched with neighborhood nodes that fulfill those requirements.

If a valid matching can be found, then there can exist a valid configuration of assignments to the neighborhood nodes that would result in the current pattern being valid. So, assigning the valid type to the node being evaluated could maintain arc consistency. The match solution ensures that the chosen pattern and its corresponding node assignments adhere to the specified requirements.

4.10 Sparse Graph

To track adjacency between nodes in the scene, a sparse graph was implemented in the C++ programming language. This graph data structure effectively handles the dynamic nature of the scene, updating itself whenever new nodes are added or
removed. It maintains a lightweight representation of the scene’s structure, focusing only on the relevant connections between nodes for GWFC, and no other unnecessary information.

When a new node is added to the scene, the sparse graph is updated accordingly. This ensures that the adjacency information remains up to date and accurately reflects the relationships between nodes. Similarly, when a user triggers events that lead to the movement or removal of nodes, the sparse graph is updated to reflect these changes.

An advantage of this approach is its integration with the scene graph’s event system. As changes to child nodes generate events within the scene graph, the sparse graph uses these events to update the adjacency information. By subscribing to the relevant events, the sparse graph can respond in real-time to node additions or removals, ensuring that the adjacency tracking remains synchronized with the changing scene.
Figure 4.5: Propagation stage of GWFC solver step. All neighboring nodes will have their domains reevaluated for validity. Green node is the node being evaluated. $n_0$ is collapsed, and this information is propagated to all neighboring nodes $n_1, n_3, n_4$. In this example, only $n_1$ has a domain change after it is reevaluated. In the second step, the domain change of $n_1$ is once again propagated to all neighbors. Since $n_0$ is already solved, the allow revisiting setting determines if its domain is again reevaluated.
Figure 4.6: Initial Boundary example. All nodes in the boundary and solved node areas of this diagram are considered as “discovered”. Red nodes have not yet been evaluated. Green node is the node being evaluated for a single GWFC solver step.

Figure 4.7: Expansion of boundary for a single GWFC solver step. Green node is the node being evaluated.
Figure 4.8: Pattern and Object Database editor. The object section allows users to edit the asset files for each object type. A single object type can have more than one asset associated with it. The patterns section allows users to specify the adjacency requirements for each object type. Both Objects and Patterns can be added and removed through this interface.

Figure 4.9: Adjacency requirements example. Similar to Kim et al. [12], adjacency requirements can be represented as plain text (a). They are used to evaluate the domains on the generated graph. (b) shows the node and neighborhood. The application draws adjacency information when a SC-GWFC node is selected in the Scene Editor Window.
Figure 4.10: Example matching problem of requirements and neighborhood nodes that satisfy those requirements. The right side shows domain values $x_0, x_1, x_2$ in each neighborhood node $n_0, n_1, n_2, n_3$.

Figure 4.11: Example matching solution that satisfies pattern requirements.
Chapter 5

EVALUATION

To evaluate the effectiveness of the SC-GWFC algorithm, we experimented with a number of different solver configurations. It was difficult to determine how SCA will interact with GWFC in an environment where both the adjacency graph and the scene geometry can change. So, we performed these experiments to find a configuration of SC-GWFC settings that best fit the original design.

Experiments 1-6 demonstrate the effect of different behaviors on the same object rules. The object rules used for these experiments are a model for a simple forest.

The following algorithm behaviors were changed in these experiments:

- **New Domain**: Domain of the nodes spawned by the Space Colonization propagation step. Could either be full domain or dependent domain. With full domain, every new node is placed with the entire set of patterns as its domain. In dependent domain, new nodes may have their domain restricted based on the domain of the node SCA is growing the graph from.

- **Order**: Visitation Order. Was an entropy heuristic used in SC-GWFC? This was either a minimum entropy priority queue, or stack with nodes inserted in order of discovery.

- **Revisit**: Can the SC-GWFC solver reevaluate the validity of a node after it was already solved? yes or no. This can remove nodes from the scene that are found without their required neighbors.
• **Neighborhood:** Size of the neighborhood radius. Can be: **fixed**, where the radius stays constant after node is spawned in scene; or **adjusted**, where the neighborhood radius depends on the size of the nodes domain.

• **Validity:** Which implementation of Adjacency Requirement Satisfaction is used: **approximate**, or **correct**.

Each experiment has two screenshots from the scene modeling application. The first is the raw placements generated by SC-GWFC with the given solver settings, and the second, is the set of nodes remaining after reevaluating discovered nodes for validity. The reevaluation is done to visually show what proportion of node placements were valid according to the user constraints.

Experiment results begin on next page.
5.1 Experiment 1

Figure 5.1: Experiment 1 propagation result. After SC-GWFC propagation and solving all discover-able nodes without reevaluating all solved nodes.

Figure 5.2: Experiment 1 validity result. After reevaluating discovered nodes for validity.

Table 5.1: Experiment 1 Solver Settings

<table>
<thead>
<tr>
<th></th>
<th>SC-GWFC Solver Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>Domain</td>
<td>full domain</td>
</tr>
<tr>
<td>Order</td>
<td>priority queue</td>
</tr>
<tr>
<td>Revisit</td>
<td>no</td>
</tr>
<tr>
<td>Neighborhood</td>
<td>adjusted</td>
</tr>
<tr>
<td>Validity</td>
<td>correct</td>
</tr>
</tbody>
</table>

Experiment 1 SC-GWFC Solver Settings.
5.2 Experiment 2

Figure 5.3: Experiment 2 propagation result. After SC-GWFC propagation and solving all discover-able nodes without reevaluating all solved nodes.

Figure 5.4: Experiment 2 validity result. After reevaluating discovered nodes for validity.

Table 5.2: Experiment 2 Solver Settings

<table>
<thead>
<tr>
<th>Setting</th>
<th>SC-GWFC Solver Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>Domain</td>
<td>full domain</td>
</tr>
<tr>
<td>Order</td>
<td>queue</td>
</tr>
<tr>
<td>Revisit</td>
<td>no</td>
</tr>
<tr>
<td>Neighborhood</td>
<td>adjusted</td>
</tr>
<tr>
<td>Validity</td>
<td>approximate</td>
</tr>
</tbody>
</table>

Experiment 2 SC-GWFC Solver Settings.
5.3 Experiment 3

Figure 5.5: Experiment 3 propagation result. After SC-GWFC propagation and solving all discoverable nodes without reevaluating all solved nodes.

Figure 5.6: Experiment 3 validity result. After reevaluating discovered nodes for validity.

Table 5.3: Experiment 3 Solver Settings

<table>
<thead>
<tr>
<th></th>
<th>SC-GWFC Solver Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>Domain</td>
<td>dependent domain</td>
</tr>
<tr>
<td>Order</td>
<td>queue</td>
</tr>
<tr>
<td>Revisit</td>
<td>no</td>
</tr>
<tr>
<td>Neighborhood</td>
<td>fixed</td>
</tr>
<tr>
<td>Validity</td>
<td>correct</td>
</tr>
</tbody>
</table>

Experiment 3 SC-GWFC Solver Settings.
5.4 Experiment 4

Same as Experiment 3, but allowing the solver to reevaluate solved nodes.

![Image: Experiment 4 propagation result](image1.png)

**Figure 5.7: Experiment 4 propagation result.** After SC-GWFC propagation and solving all discover-able nodes without reevaluating all solved nodes.

![Image: Experiment 4 validity result](image2.png)

**Figure 5.8: Experiment 4 validity result.** After reevaluating discovered nodes for validity.

![Table: Experiment 4 Solver Settings](table.png)

**Table 5.4: Experiment 4 Solver Settings**

<table>
<thead>
<tr>
<th></th>
<th>SC-GWFC Solver Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>Domain</td>
<td>dependent</td>
</tr>
<tr>
<td>Order</td>
<td>queue</td>
</tr>
<tr>
<td>Revisit</td>
<td>yes</td>
</tr>
<tr>
<td>Neighborhood</td>
<td>fixed</td>
</tr>
<tr>
<td>Validity</td>
<td>correct</td>
</tr>
</tbody>
</table>

Experiment 4 SC-GWFC Solver Settings.
5.5 Experiment 5

Same as Experiment 4, but with approximate validity.

**Figure 5.9: Experiment 5 propagation result.** After SC-GWFC propagation and solving all discover-able nodes without reevaluating all solved nodes.

**Figure 5.10: Experiment 5 validity result.** After reevaluating discovered nodes for validity.

**Table 5.5: Experiment 5 Solver Settings**

<table>
<thead>
<tr>
<th></th>
<th>SC-GWFC Solver Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>Domain</td>
<td>dependent</td>
</tr>
<tr>
<td>Order</td>
<td>queue</td>
</tr>
<tr>
<td>Revisit</td>
<td>yes</td>
</tr>
<tr>
<td>Neighborhood</td>
<td>fixed</td>
</tr>
<tr>
<td>Validity</td>
<td>approximate</td>
</tr>
</tbody>
</table>

Experiment 5 SC-GWFC Solver Settings.
5.6 Experiment 6

Same as Experiment 5, but allowing the neighborhood to resize.

**Figure 5.11: Experiment 6 propagation result.** After SC-GWFC propagation and solving all discoverable nodes without reevaluating all solved nodes.

**Figure 5.12: Experiment 6 validity result.** After reevaluating discovered nodes for validity, there are no valid object placements.

<table>
<thead>
<tr>
<th>Table 5.6: Experiment 6 Solver Settings</th>
</tr>
</thead>
<tbody>
<tr>
<td>Domain</td>
</tr>
<tr>
<td>Order</td>
</tr>
<tr>
<td>Revisit</td>
</tr>
<tr>
<td>Neighborhood</td>
</tr>
<tr>
<td>Validity</td>
</tr>
</tbody>
</table>

Experiment 6 SC-GWFC Solver Settings.
We found that Experiments 3, 4, 5 had the best results in terms of the design intent behind the patterns supplied to SC-GWFC. Experiments 2, 6 likely failed due to the adjusted neighborhood setting, which allowed node neighborhoods to change as the solver was attempting to assign types to them. This likely resulted in valid patterns in the domain quickly becoming invalid due to that changed neighborhood. Based on these results, we find that Experiment 4 had the best results in due to the output being interesting, and the larger number of valid node placements as shown in Figure 5.8.

These experiments were done to find a configuration of heuristic settings that best fit the original project objective. Which meant we needed to combine the constraint solving capabilities of GWFC with the arbitrary 3D volumes used by SCA. The following table has the settings we determined to work the best through the experiments.

Table 5.7: Final Settings

<table>
<thead>
<tr>
<th>SC-GWFC Solver Setting</th>
<th>Domain</th>
<th>Order</th>
<th>Revisit</th>
<th>Neighborhood</th>
<th>Validity</th>
</tr>
</thead>
<tbody>
<tr>
<td>dependent</td>
<td>queue</td>
<td>yes</td>
<td>fixed</td>
<td>either approximate or correct</td>
<td></td>
</tr>
</tbody>
</table>

Final settings for SC-GWFC

5.7 Evaluation Order Effect on SC-GWFC Solver

SC-GWFC should assign types to nodes in an order that results in the least number of invalid assignments. Since lower entropy nodes have a smaller domain of available values [10], evaluating nodes in order of entropy should result in the most certain nodes being solved first. We experiment with two different visitation orders. The first is in order of discovery: nodes are collapsed in the order they are discovered. Nodes are
discovered as part of the visitation process. The solver pushes all neighbors not in the discovered set to a queue. The second order we use is the same as the original WFC algorithm [10], where a priority queue is used to pick nodes with the lowest entropy.

Figures 5.13a through 5.13d demonstrate the effect of evaluating nodes in discovery order. This evaluation order results in the SC-GWFC solver evaluating the scene in a boundary that expands away from the initial node. A distinct boundary between solved and unsolved nodes is maintained.

![Images showing the SC-GWFC solver iterations in Experiment 6. First node was on the left size under the axis giszmo.](image)

**Figure 5.13:** SC-GWFC solver iterations in Experiment 6. First node was on the left size under the axis giszmo.

Figures 5.13a through 5.13d demonstrate the effect of evaluating nodes in entropy priority order. This evaluation order results in the SC-GWFC solver evaluating the scene in an incoherent boundary that initially jumps around the scene before closing around more central nodes.
Figure 5.14: SC-GWFC solver iterations in Experiment 1. First node was on the left size under the axis gizmo.
Since the scene modeling application introduced in Chapter 3 is intended to be used interactively, it is important to keep the iteration times for SC-GWFC algorithm low. This would both prevent interruptions to the user and application responsive to user inputs. This requirement stems from the design goal of providing users with a uninterrupted workflow and immediate feedback.

We split the performance evaluation of SC-GWFC into two parts. The first evaluates the performance of sphere intersection in SCA. This phase of the algorithm calculates repulsive forces and generates the adjacency information required by GWFC (see Section 4.6). Then we evaluate the performance of the GWFC algorithm to show its run-time characteristics.

6.1 SCA Performance

Sphere intersection is required to find adjacency information in the scene and calculate repulsive forces applied to new nodes. Our implementation does not have any form of spatial partitioning to accelerate this process. Figure 6.1 plots the processing time for checking sphere intersections while the scene modeling application is running. Note that these timings also captured the time taken to update adjacency information in the sparse graph.
Figure 6.1: Performance timings for sphere intersections and adjacency updates during session in the scene modeling application. This includes adding edges in the sparse graph, so there is a lot of variability in the measured times. However, this adjacency update phase is roughly linear as expected.

6.2 GWFC Performance

Evaluations were done by varying a few different SC-GWFC settings:

- the number of requirements the pattern being evaluated has
- the number of neighboring nodes
- and the size of the domain of those neighboring nodes.

Pattern validity checks occur for each type in a node’s domain while GWFC is propagating constraint information. The times in this evaluation reflect extreme situations during specific performance tests. During normal operation, the neighborhood size is
much lower than the size used in our evaluations. Usually the neighborhood size is in the range \([0, 250]\), see Figure 6.4.

**Figure 6.2:** Performance timings for valid check on a single object type. Varying neighbor node domain sizes. Jump discontinuity in time just past a Domain Size of 4000 is likely due to a reallocation in `std::vector<>`

Figures 6.2 and 6.3 show the different time complexities of the pattern validity check operation used by GWFC when evaluating a node. There are three variables that dictate the complexity of the check validity function: (1) requirements size, which is the number of required types by the pattern being evaluated; (2) domain size, the size of the neighboring nodes’ domains; and (3) neighborhood size, which is the number of nodes that can satisfy requirements for this pattern.

The scene modeling application performs all operations on a single thread, so the processing time for SC-GWFC updates should be kept low. Usually, 3D applications render an image every 16.67 milliseconds, or at 60 frames per second\(^1\). However, this

\(^1\)the rate at which images are displayed to the user. Often measured in frames per second (fps).
high frame rate is not required for this application, as we only require a smooth user experience, so our target is 24 fps. This gives 41.6 milliseconds to perform all scene updates, UI updates, and submit all rendering commands to the GPU. If we run one iteration of GWFC per frame of the application, it prevents large interruptions to the responsiveness of the application. Figure 6.4 shows that the peak processing time for a single GWFC iteration was around 70 milliseconds. While this is large enough for the user to notice, it does not create a disruption to the editing experience. The same cannot be said for the SCA phase of the algorithm. As the scene size grows, the latency induced by adjacency updates on node insertion follows $O(N)$. When the SC propagation step runs it may insert a user configurable number new nodes in the scene. In Figure 6.1, 20 new nodes are added in each iteration. When there are more than 5000 nodes in the scene, the update step can take up to six milliseconds for a single node. In the worst case, adding 20 new nodes would result in 120 milliseconds
of delay on top of all the other scene insertion operations not included in the timings of Figure 6.1. This limits the number of objects that may be in the scene to around 5000 before the modeling application becomes impractically slow. Similarly, we found that the SC propagation step caused much longer, and more noticeable, frame rate reductions in the scene modeling application as the number of objects in the scene approached 5000.

![GWFC Single Iteration Timings during Application use](image.png)

**Figure 6.4:** Timings for GWFC iterations during modeling session.
This thesis presents an interactive Scene Modeling Application that focuses on controllable procedural content generation (PCG) of outdoor 3D scenes. The modeling application allows for iterative modifications to an existing scene while incorporating constraint-driven PCG techniques. To showcase interactive editing and demonstrate our Scene Modeling Application, we implement our proposed method and use this application to perform experiments. Our proposed algorithm, Space Colonization Graph-Based Wave Function Collapse (SC-GWFC), leverages the constraint-based capabilities of WFC and the ability of SCA to populate arbitrary 3D volumes. We successfully demonstrate that objects of variable scale can be used with SC-GWFC. Our scene modeling and rendering application facilitates the experimentation and visualization of constraints for the SC-GWFC scene generator. Users can specify adjacency requirements for objects, and SC-GWFC will attempt place objects in patterns that adhere to these rules. We demonstrate the ability to place up to 5000 objects using the SC-GWFC technique. Additionally, we experiment with different settings of the SC-GWFC algorithm to find the most configuration that results in the largest amount of valid object placements.

7.1 Limitations

Implementing a usable User Interface (UI) turned out to be a much more time consuming part of the project than was expected. Early on, there was little information about all the functionality that would need to be added, so the UI evolved alongside
the SC-GWFC algorithm. This resulted in the Database Editing interface being completely disconnected from the SC-GWFC interface. We believe it would have been better for more of the UI to be contained within a single window in the application.

Due to a limited amount of time for writing our implementation, we were not able to implement a scene saving and loading system. In the future we would like to be able to generate scenes which can be exported and reloaded at a later time. If exporting to other game engines were supported, it would make the tool more useful as an external editing and scene generation system.

7.2 Future Work

Our graphs are more strongly connected than those generated by Voronoi cell adjacency like in Kim et al. [12]. Consequently, our nodes have many more neighbors. It was too restrictive to force a node to only have neighbors required by its corresponding pattern. We would like to experiment with SCA in a setting where a Voronoi cells are used to further explore the work done by Kim et al. [12]. Using this type of spatial subdivision would allow us to rely on the same adjacency rules as the original WFC [10]. Given more time we would have liked to implement this and compare Voronoi cells with our radius based adjacency.

Since the SCA propagation rules and the pattern adjacency requirements are a simple data format, it should be possible to learn rules from an input scene. Style Representation could be implemented in a way similar to the original WFC algorithm [10], using clustering to identify common patterns.

A few miscellaneous features we would have also liked to add to the Scene Modeling Application:
• Better UI, immediate feedback on user actions is an important part of PCG systems [2]. More focus on the UI would have given the opportunity for a tighter integration between our Scene Modeling Application and our SC-GWFC algorithm.

• Better shadows. Our shadow rendering is simple, and causes some artifacts in the Scene Modeling Application. Variance Shadow Maps (VSM) [9] would be a viable alternative to produce some better visual results.

• Scene saving and loading. Right now, there is no ability to save a generated scene. This is a very important feature, and unfortunately we did not have the time to implement all the serialization functions required to save the scene.

• Better placement on terrain and terrain sensitivity during propagation. Terrain placement is based on the height of the Bounding Box of the loaded model. We could consider the size of the object as well when determining placement on the terrain. This would prevent models from floating away from the terrain. We could also consider elevation during the SCA propagation phase. Additional forces could easily be added to push new nodes away from steep areas, or steepness could be part of the parameters used in the SCA propagation patterns.

• Path constraints used by Answer Set Programming [13] [16] could be used to create roads.
BIBLIOGRAPHY


