Procedural Generation of City Level

Date: 10/06/2024

Backstage Pass Gaming Institute

By: CH Goutham

Abstract

In many games that require city levels, such as tower defense games, the need for diverse cities is paramount. However, the time and resources required to create new level environments can be significant constraints. This thesis aims to address this issue through procedural generation, specifically utilizing the Wave Function Collapse (WFC) algorithm. By leveraging WFC, the project aims to develop a method for automatically generating varied and detailed city levels efficiently. This approach not only reduces the resource burden on developers and artists but also enhances the gameplay experience with dynamically generated urban environments. The study explores the implementation of this algorithm in Unity, discusses its effectiveness, and highlights potential areas for further improvement and application.

Table of Contents

Introduction

Contents

Introduction

What Is Procedural Generation?

- Procedural generation is a method of creating content algorithmically, allowing for the automatic generation of game elements such as levels, landscapes, and cities based on predefined rules and parameters. This technique offers infinite and diverse gameplay experiences.

Background Information

- Video games are increasingly adopting procedural generation techniques to create vast and diverse game worlds efficiently. One area of interest is the procedural generation of city levels, which has the potential to significantly enhance gameplay experiences by providing players with dynamically generated urban environments to explore. The procedural generation of city levels presents unique challenges and opportunities, as developers seek to create realistic, varied, and immersive urban landscapes.

Problem and Objective

- Creating environments for city levels is time-consuming and resource-intensive for both the art and development teams. To alleviate this, procedural generation can be used not only at runtime for players but also to create levels in the editor. The objective of this thesis is to create a tool for Unity that uses game objects and scriptable objects as structures for levels, along with parameters such as size and objectives, to generate detailed city levels procedurally.

Already Existing Techniques

L-System

L-System is a procedural method for generating complex structures by iteratively applying a set of rules to a starting structure. It has been adapted to generate road networks, building layouts, and other urban features, allowing for the creation of realistic and organic city environments.

How L-System Works

L-System operates through a set of rules and symbols that define how to generate a complex structure from a simple initial state. It involves the following key components:

- **Axiom**:

  The initial state or starting structure.

  Represents the simplest form from which the generation process begins.

  Ex: x

- **Production Rules**:

  Rules that define how symbols in the system are replaced or expanded.

  These rules are applied iteratively to generate increasingly complex structures.

  Ex : x-> xy

- **Iteration:**

  The process of repeatedly applying the production rules to the axiom.

Each iteration adds more detail and complexity to the structure.

Ex : **Rules**

x-> xy

y->yx

Iteration	**Result**
0	**x**
1	**xy**
2	**xyyx**
3	**xyyxyxxy**
4	**xyyxyxxyyxxyxyy**

- **Alphabet:**

A set of symbols that can be used to represent various elements of the structure.

In the context of city generation, symbols could represent roads, intersections, buildings, and other urban features.

Application in City Generation

Road Networks:

Building Layouts:

Urban Features:

Advantages of L-System

- ***Scalability***:

Can generate structures of any size, making it suitable for large-scale city generation.

- **Flexibility**:

  The production rules can be easily modified to create different styles and types of cities.

- **Organic Growth:**
  Mimics natural growth processes, resulting in more realistic and organic urban environments.

References and Further Reading

Books and Papers:

"The Algorithmic Beauty of Plants" by Przemyslaw Prusinkiewicz and Aristid Lindenmayer provides an in-depth exploration of L-System and its applications.

Online Resources:

Wikipedia Page Link : provides a comprehensive overview and links to further reading.

Research Papers:

"Procedural Modeling of Cities" by Pascal Müller., discusses various procedural generation techniques, including L-Systems, for urban modeling.

- By leveraging the principles of L-System, we can create dynamic and realistic city environments that enhance the gameplay experience and reduce the time and resources required for manual city design.

Noise Functions

Noise functions, such as Perlin noise, are used to generate natural-looking terrain and city layouts. By generating random variations in height, density, and other parameters, noise functions can create realistic and varied urban landscapes that mimic the randomness and complexity of real-world cities.

Perlin Noise:

- Perlin noise, developed by Ken Perlin in 1983, is a type of gradient noise used to generate natural-looking textures and patterns. It's particularly well-suited for creating terrain, landscapes, and city layouts due to its ability to produce smooth, continuous variations.

How Perlin Noise Works:

- Perlin noise operates by combining multiple gradient noise functions at different frequencies and amplitudes. It generates a grid of pseudo-random values that smoothly transition between adjacent points, creating a coherent and organic appearance.

The key components of Perlin noise generation include:

Gradient Vectors:

- At each grid point, Perlin noise assigns a random gradient vector. These vectors define the direction and magnitude of change between neighboring points.

Dot Product:

- To calculate the noise value at a given point, Perlin noise takes the dot product between the gradient vectors and the distance vectors from the grid points to the query point.

Interpolation:

- Perlin noise interpolates between the dot products to produce smooth transitions. Various interpolation methods, such as linear, cubic, or smoothstep, can be used to achieve different visual effects.

Octaves and Persistence:

- Perlin noise can be generated at multiple frequencies, or octaves, by scaling the grid and adjusting the amplitude of each octave. The persistence parameter controls how quickly the amplitude diminishes with each successive octave, affecting the overall roughness of the generated noise.

Applications in City Generation:

- *Perlin noise is incredibly versatile and can be used in various ways to generate city layouts*

Terrain Elevation:

Density and Distribution:

Natural Variation:

Advantages of Perlin Noise:

Smoothness:

Control and Flexibility:

Performance:

References and Further Reading:

"Texturing & Modeling: A Procedural Approach" by David S. Ebert et al. provides a comprehensive overview of Perlin noise and its applications in computer graphics.

"Real-Time Rendering" by Tomas Akenine Möller et al. covers various techniques for procedural content generation, including Perlin noise, in real-time rendering applications.

Wave Function Collapse

The Wave Function Collapse algorithm operates by taking an input pattern, such as a small section of a city layout, and expanding it to generate a larger, more complex pattern that satisfies certain constraints. In the context of city generation, the input pattern could be a small section of a road network or building layout, and the algorithm would then expand this pattern to create an entire city layout.

How Wave Function Collapse Works:

- The basic principle of WFC is to iteratively collapse possibilities based on local constraints until a complete pattern emerges.

Here's a simplified overview of how the algorithm operates:

Input Pattern:

1. The algorithm starts with a small input pattern, typically represented as a grid of tiles or cells. Each cell contains a set of possible states or values.

Local Constraints:

2. The algorithm considers the neighboring cells of each cell in the input pattern and enforces local constraints. These constraints determine which states are compatible with each other based on their adjacency and relationships.

Collapse:

3. Starting from the input pattern, the algorithm iteratively collapses possibilities by propagating constraints and eliminating incompatible states. At each iteration, the algorithm selects a cell with multiple possible states and collapses it to a single state based on the constraints.

Backtracking:

4. If the algorithm encounters a contradiction where no valid state can be selected for a cell, it backtracks to a previous state and explores alternative possibilities. Backtracking allows the algorithm to explore multiple potential solutions and avoid getting stuck in dead ends.

Completion:

5. The algorithm continues collapsing possibilities and propagating constraints until either a complete pattern is generated or it reaches a predefined stopping condition.

Applications in City Generation:

- In the context of city generation, the Wave Function Collapse algorithm can be applied to various aspects of urban planning

Road Networks:

Building Layouts:

Urban Features:

Advantages of Wave Function Collapse:

Flexibility:

Local Coherence:

Scalability:

References and Further Reading:

"GitLink" by Maxim Gumin, the original author of the algorithm,

"Procedural Content Generation for Unity Game Development" by Ryan Watkins covers practical implementations of procedural generation techniques, including Wave Function Collapse, in Unity game development.

These are a few existing algorithms that can be used.

Our Methodology

- For this project we will be using mostly Wave Function Collapse method

Methods and Techniques

Grid:

- The grid system divides the map into uniform cells, providing a structured framework for organizing the city layout.

Cells:

- Within this grid, each cell serves as a container for storing essential data related to its position, contents, and other relevant attributes. This includes information such as the type of tile placed within the cell, its position within the grid, and any additional metadata associated with it.

Scriptable Objects (SO):

- Scriptable Objects play a vital role in the procedural generation process by providing a versatile tool for creating and managing tiles. They offer a modular approach where all tiles share the same base scripts, allowing for efficient management and customization of tile properties. This enables developers to define various types of tiles with distinct characteristics while maintaining consistency and reusability across the entire tileset.

Implementation

WFC Algorithm Implementation

Grid

- The foundation of our procedural generation system is the Grid class. This class allows users to define parameters for various aspects of the grid, such as its size and spacing between cells.
- The Grid class divides the level into a grid structure, with each grid unit represented by an instance of the Cell class. These cells serve as the building blocks for constructing the city layout.

Properties:

```
public static Grid_ instance;

public int gridSizeX ;
public int gridSizeZ ;
public int spacing ;
public Cell[,] cells ;
```

Functions:

```
//initializes singleton and calls GenerateGrid()
//and initializes cells array size
Unity Message | 0 references
private void Awake()...

//Generates Grid of size gridSizeX,gridSizeY
//with spacing in between each point
1 reference
private void GenerateGrid()...

//returns cell at index
7 references
public Cell GetCell(int x, int z)...
//Overload of aboveFn
4 references
public Cell GetCell(Vector2Int CellIndex)...
```

Key Features of the Grid Class:

Generation of Grid:

Getting Cell Ref:

Cell Class:

- The Cell class represents individual units within the grid. Each cell contains properties and methods to manage its position, size, and occupancy status.

Properties:

```
//positons to place walls on edge of each cell
public Vector3 Xpos;
public Vector3 Xneg;
public Vector3 Zpos;
public Vector3 Zneg;

//center of cell
5 references
public Vector3 cellCenter {  get; private set; }

//size of cell(spacing between them )
1 reference
public int cellSize { get; private set; }

//if any tile already occupies the cell
5 references
public bool isOccupied { get; set; }

//what tile is palced if any
public Tile TilePlaced;

//the instance of prefab placed in case
//we need to remove/change it
public GameObject PrefabPlaced;
```

Functions:

```
//constructor to initilize all the values
1 reference
public Cell(Vector3 position,int cellSize)...
```

Map Generation:

Class GenerateMap : inheriting MonoBehaviour

Properties:

sizeOfBase : size of map by default (20,20)

BaseAssets : a custom class to store assets/prefabs that might be needed

base_Bounds: for calculating the starting and ending of map if needed

```
//Size of map to generate
public Vector2Int sizeOfBase = new Vector2Int(20, 20);

// class to store assets (more organised)
[SerializeField] private BaseAssets m_BaseAssets;

//just to store the start and end corners ntg much ignore for
[SerializeField] private Vector2 base_Bounds;
```

Functions:

GenerateBase()

- A function that can be called through Inspector and calls other necessary functions

```
//context menu to access this fn through Inspector
[ContextMenu("GenerateMap")]
//super function to call other functions required for generation
1 reference
public void GenerateBase()
{
    //index of starting cell from grid
    //leaving (0,0) for now due to null ref excep
    //in later code due to no cells on -ve x axis
    Vector2Int cellIndex = new Vector2Int(1, 1);

    //using Coroutine just to better visulise the process

    //builds the walls around map as in bounds
    StartCoroutine(BuildBaseWall(cellIndex));
    //fills the insize of walls using WFC Algo
    StartCoroutine( GenerateInsideOfBase(new Vector2Int(cellIndex.x, cellIndex.y)));
}
```

BuildBaseWall(vector2Int cellIndex);

- Takes index of beginning cell and uses 2 for loops to create a wall around the perimeter of map

```
//use Void as return type this is just for better
//visulisation of generation process

//builds a border by going through 2 for loops
1 reference
public IEnumerator BuildBaseWall(Vector2Int cellIndex)...

//to place walls
4 references
private void Placewall(GameObject Obj, Vector3 pos, Cell cell)...


// for placing tile in a cell
0 references
private void PlaceTileIntoCell(Tile tile, Vector2Int cellIndex)...
```

GenerateInsideOfBase(Vector2Int cellIndex):

- Starts from cellIndex and uses 2 for loops to go through each cell placing suitable tile in each
- Before placing a tile in cell checks for adjacent tiles using **CheckAdjCells().**(more about it later)

```
//use Void as return type this is just for better
//visulisation of generation process

//Uses WFC Algo
1 reference
private IEnumerator GenerateInsideOfBase(Vector2Int cellIndex)
{
    //2 forloops to go through grid
    for (int i = 0; i < sizeOfBase.x; i++)
    {
        for (int j = 0; j < sizeOfBase.y; j++)
        {
            //ignore this
            yield return new WaitForSeconds(.01f);

            //get current cell in grid using cell index
            Cell cell = Grid_.instance.GetCell(cellIndex.x + i, cellIndex.y + j);
            if (cell.isOccupied) continue;

            //select tile to place in cell
            Tile selectedTile = CheckAdjCells(new Vector2Int(cellIndex.x + i, cellIndex.y + j));

            //instantiating the selected tile
            Instantiate(selectedTile.Prefab, cell.cellCenter, selectedTile.Prefab.transform.rotation);
            cell.TilePlaced = selectedTile;

        }
    }

}
```

CheckAdjCells(Vector2Int cellIndex):

- Has a list of all posible tiles that can be placed
- Goes through all the 4 side (you can have 6 if you wish just need to worry about 2 more sides /j)
- Checks and removes non compatable Tiles With help of RemoveNonCompatableTiles()
- Selects random tile form all compatable tiles

```
//checks adjcent side to find matching tiles
//that fit in there cells while connecting
//to other tiles well
1 reference
private Tile CheckAdjCells(Vector2Int cellIdex)
{
    Tile selectedTile = null;

    //list of all posible tiles
    List<Tile> PossibleTiles = m_BaseAssets.tiles;

    //checking all 4 sides

    Cell cell;
    //check xpos cell
    cell = Grid_.instance.GetCell(new Vector2Int(cellIdex.x + 1, cellIdex.y));
    if (cell.TilePlaced != null)
    {
        //checking the connecting edge to find compatable
        //tile with matching edge
        TypeOfEdges xposEdge = cell.TilePlaced.xneg;
        //removing non compatable tiles
        PossibleTiles = RemoveNonCompatableTiles(xposEdge, new Vector2(1, 0), PossibleTiles);
    }
    //check xneg cell
    cell = Grid_.instance.GetCell(new Vector2Int(cellIdex.x - 1, cellIdex.y));
    if (cell.TilePlaced != null)...


    //check zpos cell
    cell = Grid_.instance.GetCell(new Vector2Int(cellIdex.x, cellIdex.y + 1));
    if (cell.TilePlaced != null)...

    //check zneg cell
    cell = Grid_.instance.GetCell(new Vector2Int(cellIdex.x, cellIdex.y - 1));
    if (cell.TilePlaced != null)...

    //selecting 1 tile out of remaining possible tiles after elemination
    //this is the collapse in WaveFn Collapse lol
    selectedTile = PossibleTiles[UnityEngine.Random.Range(0, PossibleTiles.Count - 1)];

    return selectedTile;
}
```

RemoveNonCompatableTiles (TypeOfEdges typeOfEdge, Vector2 side, List<Tile> tiles):

- Has an empty list named result of compatible tiles
- Has a switch case for 4 sides each to check for compatible sides and add them to result list
- Side is indicated by Vector3Int (1,0) = xpositive , (-1,0) = xnegative , (0,1) = ypositive,(0,-1) = ynegative
- Returns list of compatible tiles

```
//removes non compatable tiles
4 references
public List<Tile> RemoveNonCompatableTiles(TypeOfEdges typeOfEdge, Vector2 side, List<Tile> tiles)
{
    //new list that contains compatable tiles
    List<Tile> result = new List<Tile>();


    //switch case for 4 sides
    switch (side)
    {
        case Vector2 v when v == new Vector2(1, 0):

            foreach (Tile tile in tiles)
            {
                //checking if edge match tyoe of edge of the connecting cell
                if (tile.xpos == typeOfEdge)...
            }
            return result;

            break;

        case Vector2 v when v == new Vector2(-1, 0):

            foreach (Tile tile in tiles)...
            return result;
            break;

        case Vector2 v when v == new Vector2(0, 1):

            foreach (Tile tile in tiles)...
            return result;
            break;

        case Vector2 v when v == new Vector2(0, -1):

            foreach (Tile tile in tiles)...
            return result;
            break;

        default:...
    }
}
```

Class BaseAssets:

- Used to store assets/prefab
- For better maintenance

```
[Serializable]
1 reference
public class BaseAssets
{
    public GameObject VerticalWall;
    public GameObject HorizontalWall;

    public List<Tile> tiles;


}
```

Tile: Scriptable Object:

- Scriptable Object to store data related to tile such as GameObject that needs to be Instantiated, what type of tile it is and what is on its edges

```
[CreateAssetMenu(fileName ="Tile",menuName ="Tile Creation",order = 0)]
Unity Script | 15 references
public class Tile :ScriptableObject
{
    //prefab to spawn
    public GameObject Prefab;

    //to identify type such as road,grass,building,etc
    public TileType TileType;

    //size of tile such as how many cells it occupies
    public Vector2Int Size;

    public TypeOfEdges xpos;
    public TypeOfEdges xneg;
    public TypeOfEdges ypos;
    public TypeOfEdges yneg;
}

1 reference
public enum TileType
{
    StraightPath,Xroads,turn,Tjunction,grass,

}
9 references
public enum TypeOfEdges{
    grass,road,Building,river,footpath
}
```

Software and tools used.

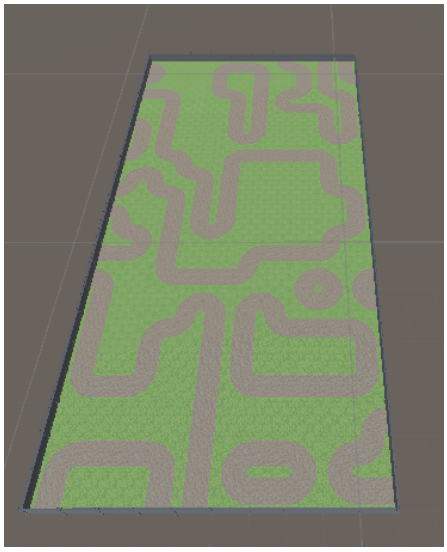
- **Unity**
- **Visual Studio 2022**

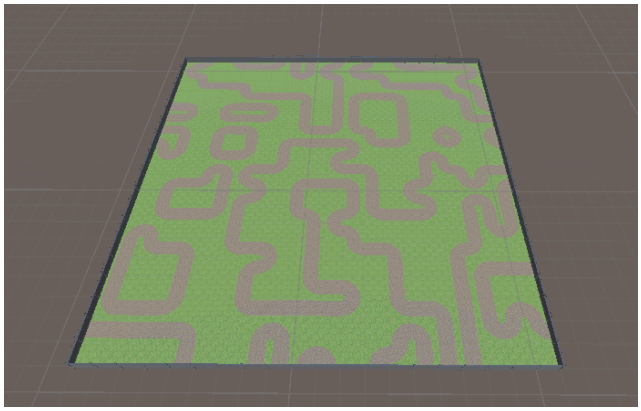
Challenges faced during implementation and how they were overcome.

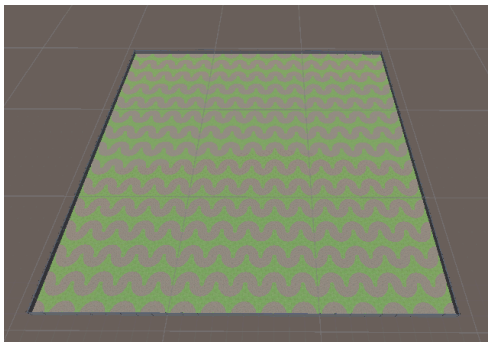
- Many Null Reference exceptions due to cells and tiles
- Null reference exceptions when placing tiles on the edge of grind
  we could have an if statement that stops the error by returning if cell==null ,etc but I decided to just either increase the size of grid by 1 or decrease the size of map that was meant to be build by 1
- Having multiple rotated copies of same piece instead of manually doing it ended up using a script that rotates the piece for you
- Didn't have enough time to implement back tracking
- Did try to have back tracking for 1 step behind attest but got some other issues so it's on the side for now

Results:

(imgs)







(Video): scan this for yt video

Analyzing the results:

From my experience with this thesis I could make a lot more improvement in this project just as:

1. Adding Backtracking functionality such that we don't form repeated unnatural patterns
2. Adding more parameters such as what type of city , density of houses-roads-trees
3. Adding support for trees/props such as garbage cans ,street lights , other such props
4. Can make it easy to use and implement for users who just want to use this for there projects using editor tool
5. These will be a QR to feedback form at end please let me know your findings too

Interpretation of Results

- does this project implement WFC yes kind of. I feel that we could do a lot more with this and the result need a lot more work done on it for it to be a proper use case tool

implications of the findings.

- WFC is pretty simple and yet very effective way to procedurally generate map/levels it want too complex to understand (I mean we still have backtracking left) but even tho this is a useful Algo/method for procedurally generating world/level and I do wanna expand my knowledge on this subject

Conclusion:

In conclusion, the results reveals several areas for improvement and highlights. The potential of the Wave Function Collapse (WFC) method in procedural generation. While the project implements WFC to some extent, there is room for enhancements to make it more robust and user-friendly. The identified improvements include adding backtracking functionality to prevent the formation of repeated unnatural patterns, incorporating additional parameters for specifying city types and densities of various elements, and extending support for additional props such as trees and street furniture. Moreover, simplifying the implementation process and providing user-friendly editor tools can make the project more accessible and usable for a wider audience.

In summary, while there is still work to be done to optimize and expand the project, the insights gained from this study underline the significance of WFC in procedural content generation. By continuing to refine the methodology, we can unlock new possibilities for creating dynamic and immersive game worlds.

References

"GitLink" by Maxim Gumin, the original author of the algorithm,

"Procedural Content Generation for Unity Game Development" by Ryan Watkins covers practical implementations of procedural generation techniques, including Wave Function Collapse, in Unity game development.

https://youtu.be/Jsc3BQaJndQ : Procedural Generation of a City in Unity