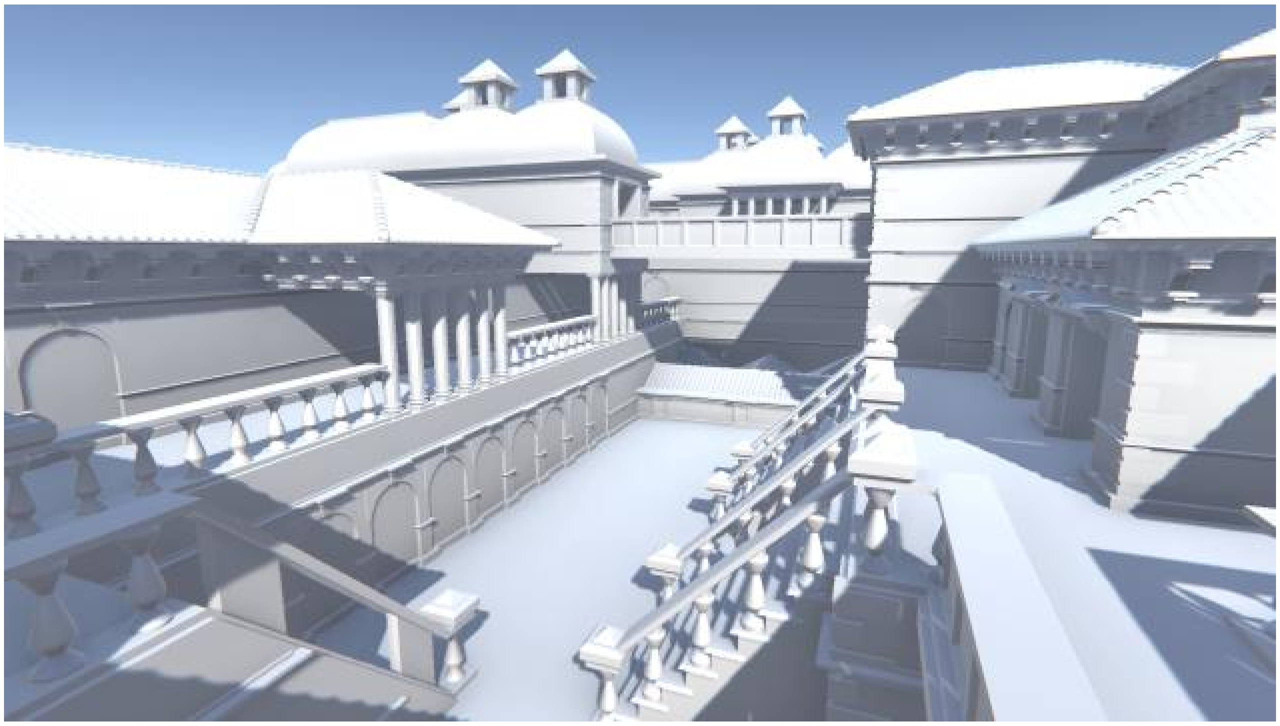
Wave Function Collapse

Name: Jasmine C, Maddox P
Class: Game Design II
Date: 10/18/21

What Is Wave Function Collapse

Wave Function Collapse is an algorithm that fills all tiles with all possible solutions and slowly collapses them into one possibility by using adjacency rules. In general terms, Wave Function Collapse can take a set of tiles with rules defined by the programmer, and then procedurally generate game worlds. One of the most famous examples of this game design is the infinite city found on Marian's blog. The algorithm fills all grid tiles with all possible solutions (superpositions).



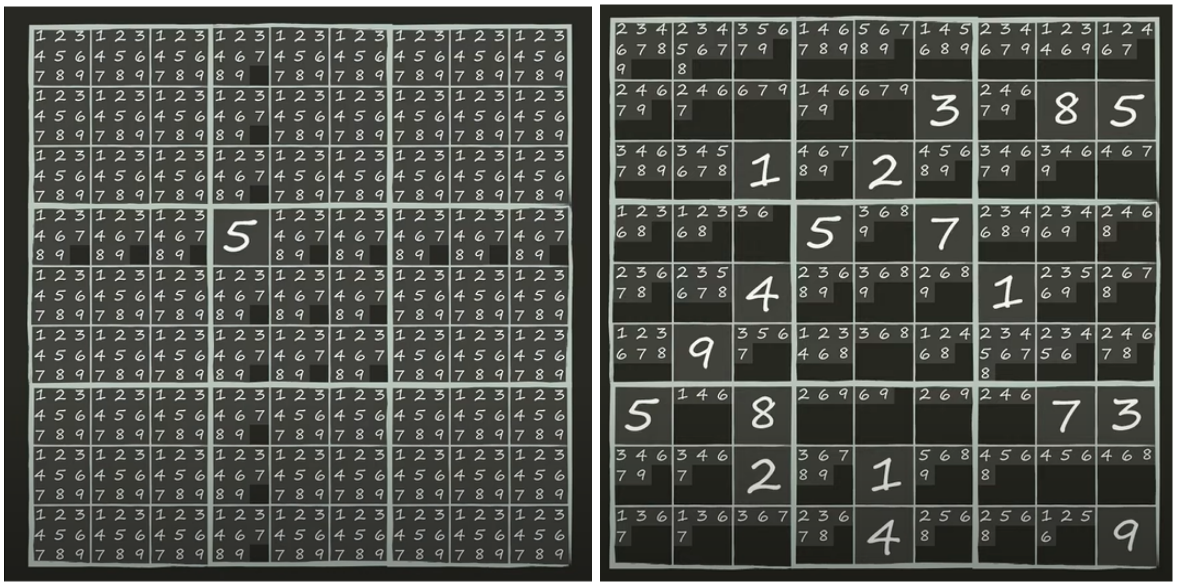
How can Wave Function Collapse be Used

Wave Function collapse is most commonly used in games to randomly generate a game world. This can be in the form of an infinite 3D world or set 2D rooms/dungeons. The 3D application of Wave Function Collapse is a bit more complex than the 2D application, but it is also far more versatile. In general, handcrafted levels are "better" than algorithmically generated ones, but if the main focus of the game isn't on the levels or world then Wave Function collapse can still be extremely useful.

How Does the Wave Function Collapse Algorithm Work

Important Definitions	
Superposition	When a grid piece is filled with all tile possibilities at once. When a number is filled in, superpositions are collapsed, removing any possibility that would not fit with the possible grid tiles
Propagate	To remove any tile options in a grid would not be a possible solution. (Passing knowledge)
Entropy	The number of possible solutions. Low Entropy would be only a few possible solutions while high entropy would be many possible solutions.

To visualize how the waveform collapse algorithm works, I will use the game sudoku as an example (for the purposes of length I will assume you know the rules to sudoku). Each cell in a sudoku chart has 9 possible solutions (i.e. the numbers 1-9). To algorithmically find the solution, we first need to fill each cell with its superpositions. Then, we need to fill in the given values, crossing off any of the same values in that given row/column. Now that we have input all the given data, we need to look for the cell with the least entropy and collapse it to a single possibility. We want to prioritize low entropy cells since it minimizes the chance of making a bad choice.


[Visual of sudoku Wave Form Collapse]

Since we have collapsed another cell to a single possibility, we need to cross off this number in the column and rows of this cell, leading to a greater amount of low entropy cells. We continue this pattern of choosing low entropy cells and collapsing adjacent cells until the sudoku puzzle is complete.

Algorithm Steps

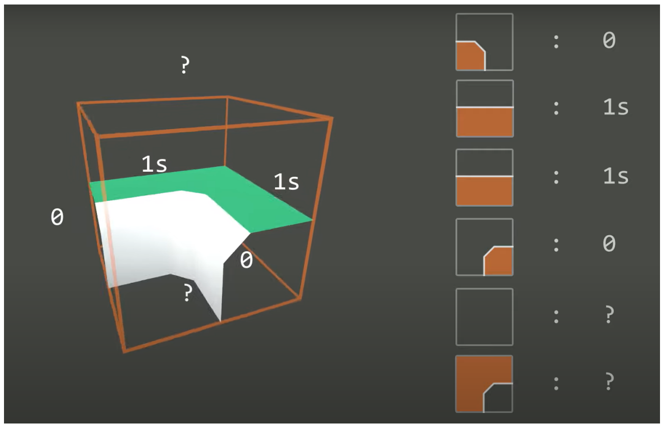
1. Populate all grid cells with all possible solutions (superposition)
2. Fill in all given information (if applicable)
3. Propagate all applicable cells from the given information
4. Choose the cell with the lowest entropy and collapse it to a single solution
5. Repeat steps 3 and 4 until all cells are filled in

Applying the Wave Function Collapse Algorithm to Game Creation

The Wave Function Collapse Algorithm in game development is very similar to the Wave Function Collapse algorithm used to solve a sudoku puzzle, except the programmer must define their own sets of rules for what tiles can go together.

Adjacency Rules

The first step in creating a Wave Function Collapse algorithm is to create a set of tiles with specified adjacency rules. Adjacency Rules are the defining characteristics of each edge of the tiles that either allow it to fit with another tile or prevent it from fitting with another tile. The most common way to implement Adjacency Rules is a socket system, where each edge of a tile is labeled with numbers/letters that identify a connection type. For example, two edges both labeled with 1s can fit together but an edge labeled 1s and 0 cannot fit together.



[Cube sides and their labeled sockets]

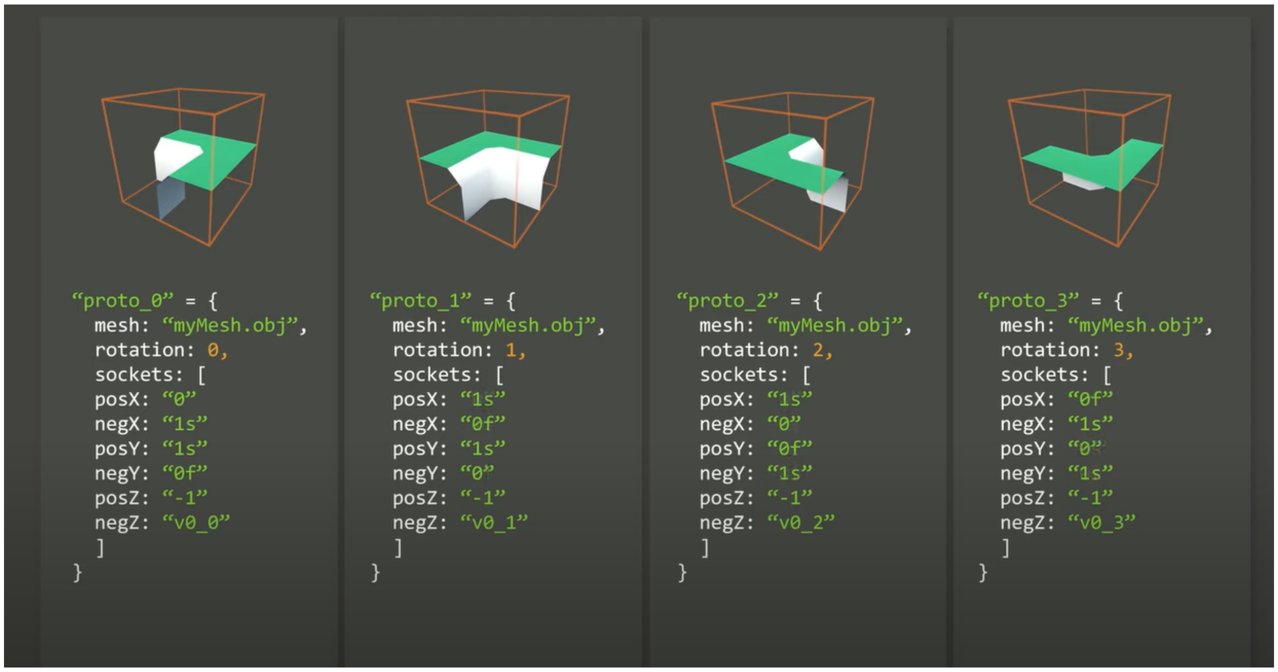
Wave Function Collapse Steps (pseudocode)

1. Create a list of 6 lists that will hold all valid neighbors (one for each cube face)
    a. Whenever the algorithm collapses a cell, remove any cell types from neighboring cells that aren't in the list of valid neighbors.
2. Label each module (cube face) with a socket identifier
3. Loop over every module in the set and store the position of each vertex that sits along each edge of the 6 boundaries **[this is all scripted in Blender, do not manually do this]**
4. Store and label each of these (above). This creates a dictionary of module names and profiles.
5. There is special markup needed for symmetrical and asymmetrical modules:
    a. Tag symmetrical with an s next to its socket label (ex 1s)
    b. Since symmetrical tiles will always fit with itself, we can add that as a rule for matching cube faces
    c. Asymmetrical tiles, on the other hand, will fit with a mirrored version of themselves. This means we can store an asymmetrical module as two different sockets and mark one with F (for flipped)
6. There is also special markup needed for Top and Bottom modules:
    a. For each top/bottom module store 4 versions of each socket and label them with a rotation index

b. There is an extra check needed for these modules as well; vertical sockets will only be considered valid if they have the same socket index and rotation index

Creating Prototypes

Prototypes are the metadata for modules. They contain information about which mesh to use, what the mesh rotation is, and each cube faces' list of 6 valid neighbors.



```
"proto_0" = {
  mesh: "myMesh.obj",
  rotation: 0,
  sockets: [
  posX: "0"
  negX: "1s"
  posY: "1s"
  negY: "0f"
  posZ: "-1"
  negZ: "v0_0"
  ]
}
```

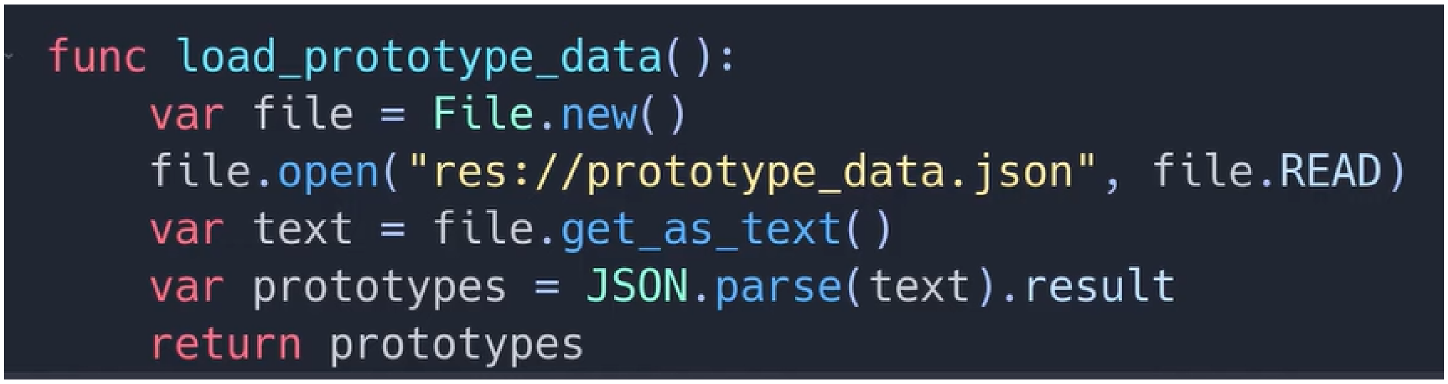
```
"proto_1" = {
  mesh: "myMesh.obj",
  rotation: 1,
  sockets: [
  posX: "1s"
  negX: "0f"
  posY: "1s"
  negY: "0"
  posZ: "-1"
  negZ: "v0_1"
  ]
}
```

```
"proto_2" = {
  mesh: "myMesh.obj",
  rotation: 2,
  sockets: [
  posX: "1s"
  negX: "0"
  posY: "0f"
  negY: "1s"
  posZ: "-1"
  negZ: "v0_2"
  ]
}
```

```
"proto_3" = {
  mesh: "myMesh.obj",
  rotation: 3,
  sockets: [
  posX: "0f"
  negX: "1s"
  posY: "0"
  negY: "1s"
  posZ: "-1"
  negZ: "v0_3"
  ]
}
```

[An image of 4 completed prototypes all referencing the same mesh, just rotated]

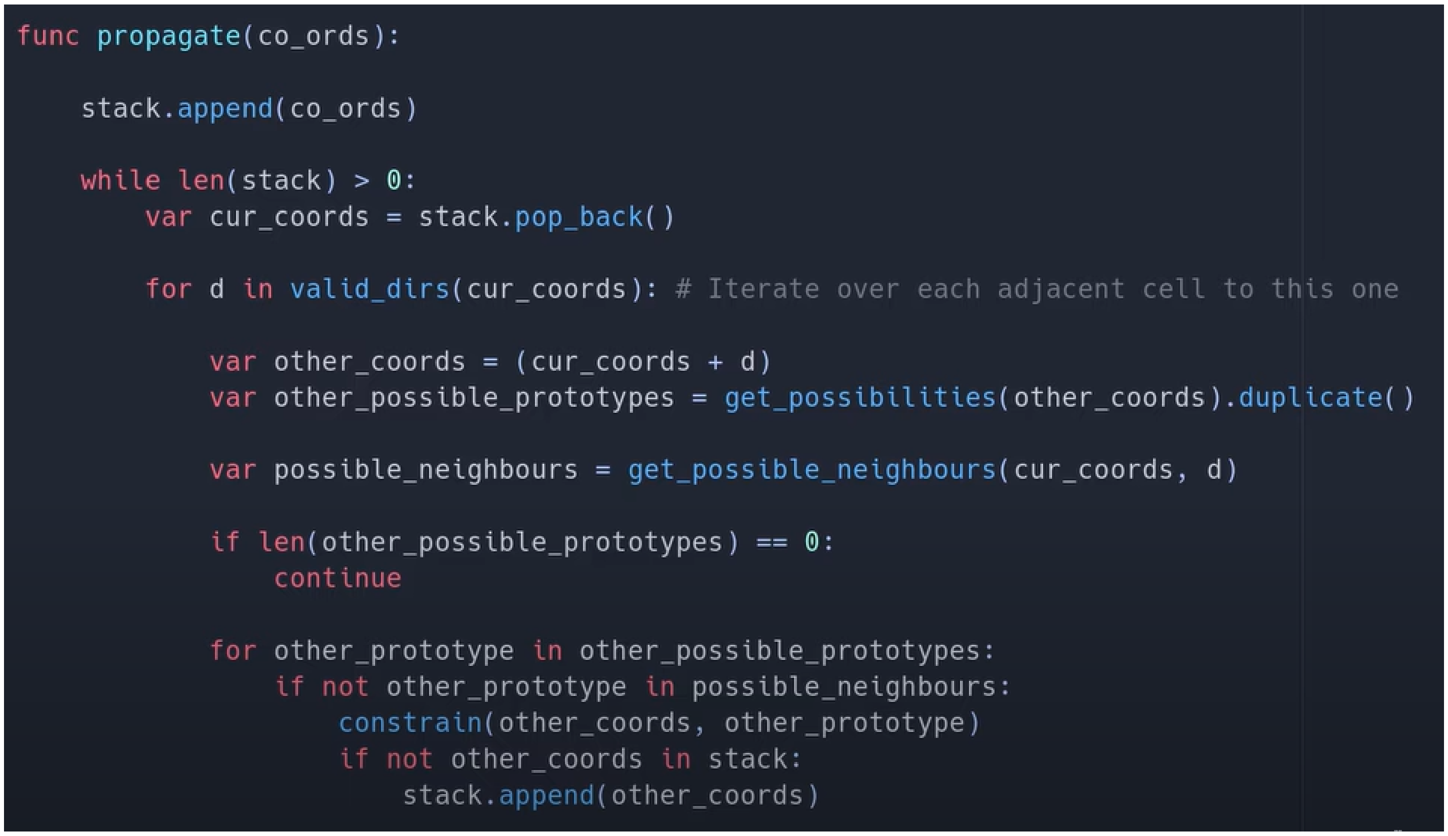
Since each mesh has a rotation number, the programmer doesn't have to export four different meshes (one for each rotation), and instead, each prototype references the same mesh and a rotation value. The prototype data is written to a JSON file, which can be loaded as a dictionary in Unity.



```
func load_prototype_data():
    var file = File.new()
    file.open("res://prototype_data.json", file.READ)
    var text = file.get_as_text()
    var prototypes = JSON.parse(text).result
    return prototypes
```

[Python code that imports the prototypes and loads them]

Coding the Wave Function Collapse algorithm



```
func propagate(co_ords):

    stack.append(co_ords)

    while len(stack) > 0:
        var cur_coords = stack.pop_back()

        for d in valid_dirs(cur_coords): # Iterate over each adjacent cell to this one

            var other_coords = (cur_coords + d)
            var other_possible_prototypes = get_possibilities(other_coords).duplicate()

            var possible_neighbours = get_possible_neighbours(cur_coords, d)

            if len(other_possible_prototypes) == 0:
                continue

            for other_prototype in other_possible_prototypes:
                if not other_prototype in possible_neighbours:
                    constrain(other_coords, other_prototype)
                    if not other_coords in stack:
                        stack.append(other_coords)
```

[Python code of the main loop for Wave Function Collapse]

The function propagate is what is called to run the Wave Function Collapse algorithm, after instantiating all types of tiles.

Steps of the propagate function

1. Add all possible coordinates to the “stack” list
2. Loops through all stack objects in the list, and checks for adjacency rules
3. Cross out any prototypes that fail the adjacency rules
4. Check if there are still possible solutions in the list of all possible solutions
5. If yes, then repeat, if no then place the solution

References

1. Martin Donald “Superpositions, Sudoku, the Wave Function Collapse algorithm.” https://www.youtube.com/watch?v=2SuvO4Gi7uY
2. Itch.io Wave Function Collapse Examples
    a. https://bolddunkley.itch.io/wfc-mixed
    b. https://bolddunkley.itch.io/wave-function-collapse
3. Osker Stalvberg Wave Function Collapse Example http://oskarstalberg.com/game/wave/wave.html
4. Robert Heaton “The Wavefunction Collapse Algorithm explained very clearly” https://robertheaton.com/2018/12/17/wavefunction-collapse-algorithm/
5. Marian “Infinite procedurally generated city with the Wave Function Collapse algorithm” https://marian42.de/article/wfc/
6. Github for Wave Function Collapse algorithm https://github.com/mxgmn/WaveFunctionCollapse