Martian Squirrel Cave Dwellers

CP-3

Robert Boerwinkle

Spring 2023

1 Introduction

Squirrels are moving to Mars, and need a place to live. It has to be underground to protect them from radiation. Since the Martian soil is relatively homogeneous, the only other consideration is the cost. Every piece of dirt costs 2 acorns to move and must be conserved. Because the search space is so large, emphasis is not placed on finding the best answer, but finding a good answer in a reasonable time.

The cave consists of a 512x512x512 boolean map of dirt / nondirt. The structure to be built (out of dirt) is a cube of side length 61 with a cube of side length 51 hollowed out (concentric). Only dirt within a cube of side length 71 can be moved (also concentric). These concentric regions are called the central hollow, the walls, and the buffer zone. In addition, there cannot be any floating dirt. It can be assumed that no dirt starts off floating.

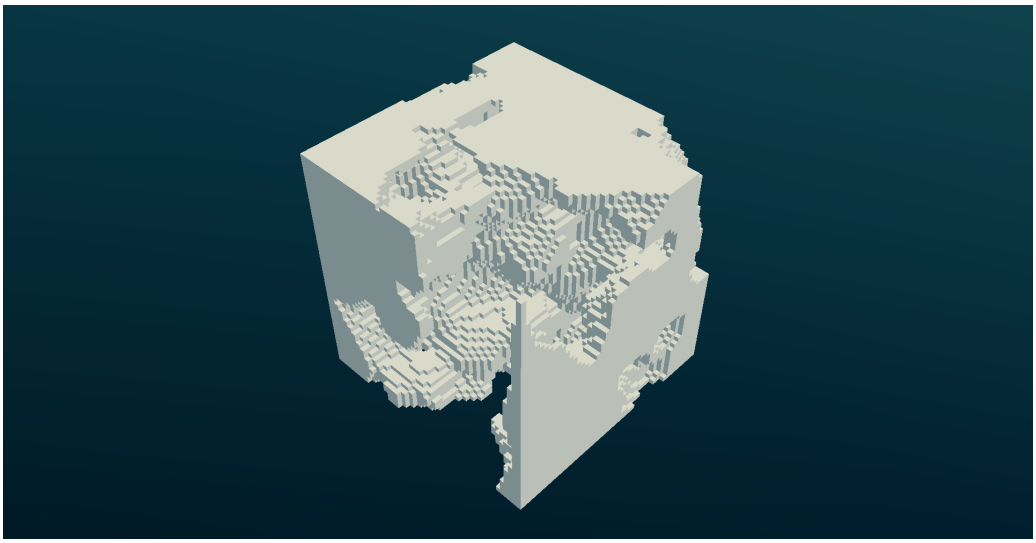

Figure 1: A section of the cave

All programs were implemented in Python with the use of `NumPy` (as `np`). The 3D rendering workflow consisted of Python → OpenSCAD → fstl.

2 Cave Construction; Attempt #1

The first step is an algorithm which can construct a dwelling. The central hollow and walls are simple with a little clever slicing. The bulk of the algorithm was placing the leftover dirt. Because we can be sure of the dirt that composes the walls, leftover dirt can be layered over it in concentric shells. It should be noted that because of the volumes of the buffer, wall, and interior hollow, some positions simply cannot be built in. If there is too much dirt in the central hollow to fit in the buffer zone or not enough in the entire region to build the walls, no dwelling can be built. Here is a snippet of Python code which implements dwelling construction (`SIDE` is the side length of the central hollow, `WALL` is the thickness of the walls, and `BUFF` is the thickness of the buffer zone):

```
X,Y,Z = np.mgrid[:cave.shape[0], :cave.shape[1], :cave.shape[2]]
X = abs(X-x)
Y = abs(Y-y)
Z = abs(Z-z)
sdf = np.maximum(X, Y)
sdf = np.maximum(sdf, Z)
sdf -= (SIZE//2)+WALL+1
for shell in range(BUFF):
        shellView = np.where((sdf == shell) * (cave==0))
        if len(shellView[0]) < unusedDirt:
                unusedDirt -= len(shellView[0])
                cave[shellView] = 1
        else:
                cave[
                        shellView[0][:unusedDirt],
                        shellView[1][:unusedDirt],
```

```
                shellView[2][:unusedDirt]
        ] = 1
        unusedDirt = 0
        break
```

In order to check multiple positions, the cave needs to be copied, edited, differenced with the original cave, and then summed. This took roughly 5 seconds per check. This is inefficient and not feasible for such a large search space. This algorithm is still useful for producing the final result, just not for evaluating cost.

3 Convoluted Convolutions; Attempt #2

A convolution is a mathematical operation particularly suited to this application. The definition of a convolution is complex, unimportant, and generally beyond the scope of this paper. What is important is that `SciPy` has a function for doing them. A carefully constructed kernel could possibly evaluate the cost of construction. Unfortunately, the memory usage for the algorithm is simply too high for arrays of this size. Some possible solutions include subsampling the original cave and kernel or splitting the cave up into portions and then re-stitching after the convolutions. Neither were implemented, and at this point option #3 was found to be viable.

4 A Number of NumPy Processes; Attempt #3

This approach is effectively a custom implementation of a convolution-like operation. It consists of a number of sums of slices from the original cave. During this operation no data is copied or changed, making it significantly faster. Each check takes about 0.001s. This method ignored positions which would require pulling dirt in from the buffer zone. After checking to make sure there is enough dirt in the central hollow and wall regions to build the dwelling and that there isn't too much dirt to fit into the buffers, the cost is 2 * (total dirt in the central hollow). This method had a search time of about 0.001s, allowing the space to be searched in about 24 hours. The cost was saved out. In this case, the cost map was used to choose the top 4 spots.

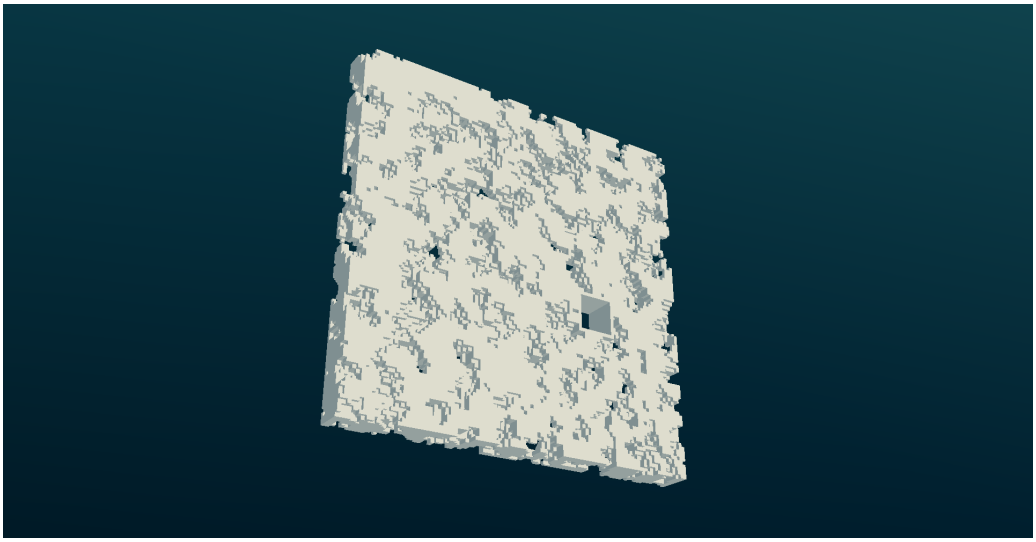

Figure 2: A slice of of the cave containing the lowest cost spot (1:5 subsampling)

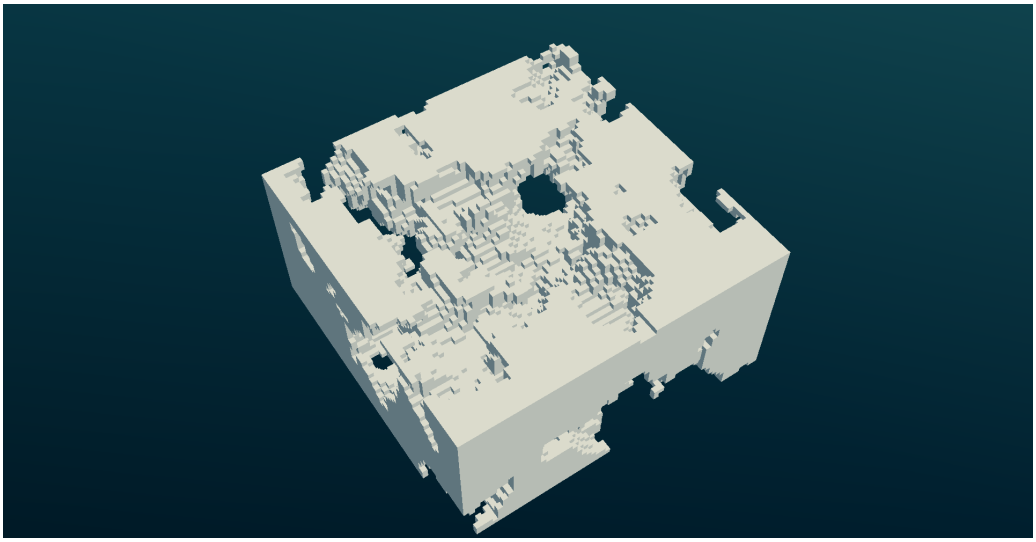

Figure 3: The lowest cost spot, before any work has been done (1:2 subsampling)

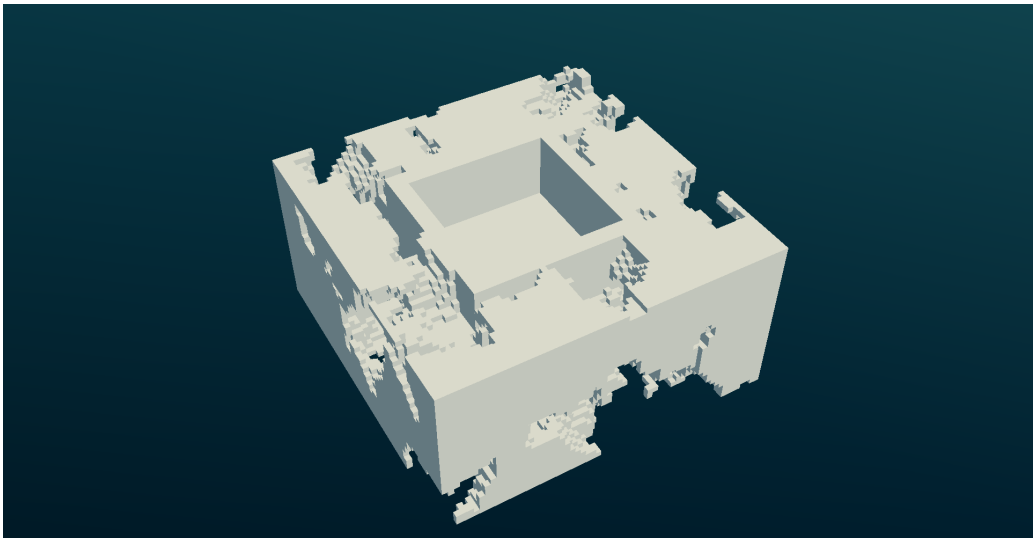

Figure 4: The lowest cost spot with the dwelling constructed (1:2 subsampling)

5 Conclusion

Although the buffer zone does not have to be entirely included in the cave, the algorithm ignored that possibility. As such there are some dwellings at the edge of the cave which were not checked. There could also be some spots in which it would be cheaper to pull dirt from the buffer. At that point, the algorithm would have to take into account the no floating dirt clause. While the solutions found might not be the best solutions, they are certainly adequate. Had the search space been bigger, or the dwelling cost been more difficult to calculate, more optimizations would have been needed. As it stands, the solution of 'throw more computer at it' worked pretty well. Even without multiprocessing, a single personal computer (HP Elitebook 840 G5) was able to finish the search in under 24 hours.