Hexblock algorithm

HackSurrey 2019

Justin Chadwell

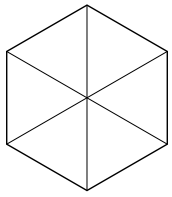
3D printing data

The idea behind the hexblock algorithm is to able to 3D print arbitrary data onto a surface, with two major requirements: to be able to perfectly reconstruct the data, and to have it look visually appealing.

When we started designing the look of the printed piece, we decided to use hexagons as our main design focus, simply because hexagons are easy to generate, easy to tessalate, and most importantly, look quite pretty.

The hexagon

The hexagon itself is where all the data is stored. To store data within the hexagon, we decided to divide it into 6 equilateral triangles. The information and data is then stored within the heights of the individual triangles.



Now, while the data could be directly encoded in the heights, i.e. a value is represented as a function of the height, this approach is too prone to error. Slight 3D printer inaccuracies, layers of dust or simply just normal wear-and-tear could cause the heights to not represent the value they were originally meant to.

Therefore, we chose to encode data, not as a function of the height of a single triangle, but as a function of the collection of heights of the six triangles arranged into a hexagon.

Permutation

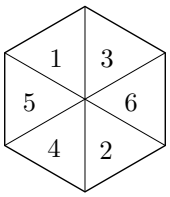
Each hexagon can represent a number between 0 (inclusive) and 720 (exclusive). Through our `permute` function (see Appendix A for function definitions), we can create a unique permutation of numbers for each number within that range. These permutations can then be translated into heights.

For example, take the number 435. We create a permutation of the set $\{1, 2, 3, 4, 5, 6\}$ based on it, using our `permute` function, which yields the following array:

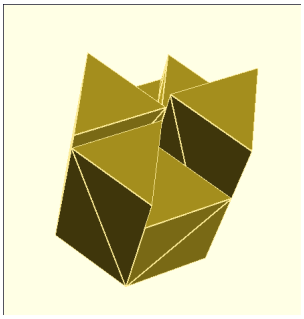
$$\text{permute}([1, 2, 3, 4, 5, 6], 435) = [4, 5, 1, 3, 6, 2]$$

Encoding

Based on this permutation, we can now assign each of the items in our permutation to one of the triangles, starting with the triangle in the bottom left.



Then, we can make the tallest triangle the triangle with the highest value, the second-tallest triangle the triangle with the second-highest value, etc., and finally the shortest triangle the lowest value.



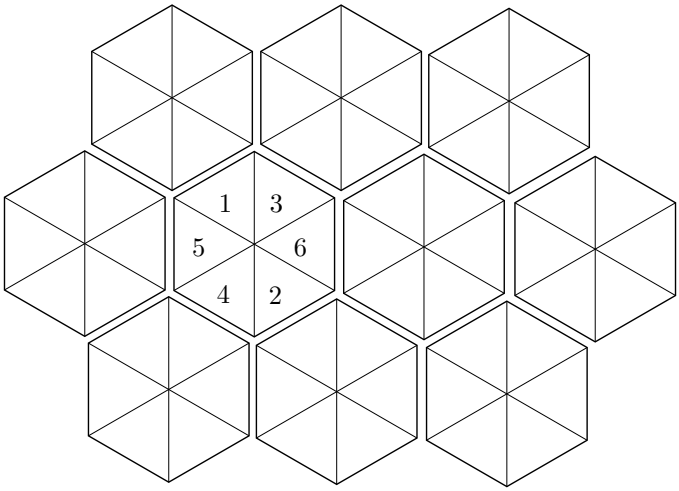
We now have a way of translating any number from 0 to 720 into a hexagon. We can use this as a component within our larger representation.

Block

To represent larger data structures, we need to find a way to compose our hexagons together into a more complex structure. While there are probably quite a few ways to do this, we decided to use a base-encoding scheme. To do this, we first convert our data into an integer, then convert the integer into a base-720 number (see `to_base`) and then use the hexagons as digits in that number.

Once we have our base-720 number, we can then create a series of hexes representing each digit. For visual effect, we arrange them into rows and columns, although they could conceivably arranged in any configuration. To read the data back in, we read from left to right and then top to bottom.

In the case of our previous example, with the number 435, it would appear like this somewhere in the middle of the array of hexes:



Obviously, it is difficult to make out which side is right way up without a visual marker, so we create a small triangle in the bottom-left corner to indicate the orientation of the piece.

Here is an example of the final result:

A note on reconstruction

Data reconstruction is not likely to be easy, but it is definitely possible.

To reconstruct the original data, the decoder must first calculate the value of each individual hex by recording the heights of the triangles and then calculating their order. The decoder must then feed the order into the `depermute` function which will then return the number in the range 0-720. After we have completed this operation for each hexagon, we can use those as the digits in the base-720 number and reconstruct the original integer. This can then trivially be re-encoded back into bytes.

Appendix A: functions

Below are listed the various functions we used in our implementation of the hexblock algorithm, which was in python3.

These are definitely not the only possible implementations, however, they are what we originally used, and we believe that they are not too difficult to understand.

Permutations

```
def permute(ordered, number):
    '''
    Converts an ordered set of values into a unique permutation based on the
    value of a number.
    '''

    li = ordered[:]
    permuted = []

    seq = to_factoradic(number, len(li))
    for i in seq:
        permuted.append(li.pop(i))

    return permuted

def depermute(permuted, ordered):
    '''
    Converts a permuted array of values into a number based on the original set
    of ordered values.
    '''

    li = ordered[:]
    seq = []

    for i in permuted:
        index = li.index(i)
        seq.append(index)
        li.pop(index)

    return from_factoradic(seq)
```

Base conversion

```
def to_base(number, base):
    '''
    Converts an integer into an arbitrary base. The result is encoded as a list
    of big-endian digits.
    '''

    top = 1
    while base ** top <= number:
        top += 1

    parts = []
    for i in range(top - 1, -1, -1):
```

```
        value = base ** i
        parts.append(number // value)
        number %= value

    return parts

def from_base(parts, base):
    '''
    Converts a number encoded as a list of big-endian digits encoded in an
    arbitrary base into an integer.
    '''
    number = 0

    place = 1
    for value in reversed(parts):
        number += value * place
        place *= base

    return number
```

Helper functions

The functions above use a couple of utility methods which we have listed below:

```
def to_factoradic(number, places):
    '''
    Convert an integer into a factoradic base (see https://en.wikipedia.org/wiki/Factoradic)
    '''

    data = []

    assert number < math.factorial(places)

    for i in range(places - 1, -1, -1):
        result = number // math.factorial(i)
        data.append(result)
        if result > 0:
            number %= math.factorial(i)

    return data

def from_factoradic(data):
    '''
    Convert a number in a factoradic base into an integer.
    '''

    number = 0
    for place, value in enumerate(reversed(data)):
        number += value * math.factorial(place)
    return number
```