Optimised Morton Encoding

Lookup Tables

Consider a given set of indices describing the anchor of an octree node (x, y, z). The Morton encoding can be statically encoded in a lookup table.

For example, (x, y, z) = (4, 55, 132) which in binary is (100, 110111, 10000100). As we have to interleave these bits, we can add zero bits and perform a bitwise `or' operation to find the final Morton key:

$x_{\text{shift}} \mid y_{\text{shift}} \mid z_{\text{shift}} =$

01000000
| 000100100000100010010
| 100000000000000100000000

If x, y, z ∈ [0, 255], these shifts can be stored in small statically stored lookup tables, and we can encode a Morton key for larger indices by considering their bits byte by byte.

This is a divide and conquer strategy, that has been shown to be faster than on-the-fly implementation using for-loops and bit-shifts[1].

1. https://www.forceflow.be/2013/10/07/morton-encodingdecoding-through-bit-interleaving-implementations/

Shared Memory Parallelism With Rayon

This lookup strategy can be performed in parallel over each anchor being encoded, which is easy to to using the Rayon crate for shared memory parallelism.

Rust defines `iterators', which are a functional programming abstraction to apply a single transformation to a set of data.

For example, consider the calculation of the squares of a vector of numbers, which are then summed. This can be expressed as:

```
let vec: Vec<i32> = vec![0,1,2,3,4,5];

let res = vec.iter()
   .map(|&i| i*i)
   .sum();
```

Rayon's main abstraction is to extend this to a parallel setting with *parallel iterators*:

```
let res = vec.par_iter() // <- only change
   .map(|&i| i*i)
   .sum();
```

Rayon's parallel iterators are an abstraction built on top of its work-stealing based parallel backend. Rayon fully incorporates Rust's multithreading safety features, and therefore guarantees data-race freedom.