Performance optimizations when traveling thru world

1. Identify the cause

* Its not the chunk rendering...
* Its NOT mesh generaiton or mesh sending!

# It’s the memory usage

* There are stair bumps in memory that align perfectly to when the player is traveling thru the world
* The reason why running thru the dev terrain is faster is because the memory hill is much smoother

What could be causing too much memory usage?

* The first culprit is from Terrain.generateChunkInner()
  + This is only an issue with complex terrain, default terrain is ok.
* There could be some memory usage from sunlight generation?
  + I couldn’t tell a difference. Don’t test for this until mesh gen is resolved first
* **The second culprit is DEFINITELY chunk mesh generation**

|  |  |
| --- | --- |
| Without mesh gen (default terrain) | With mesh gen (default terrain) |
|  |  |

Specifically, this is the code producing all the spikes:

|  |
| --- |
| meshesHaveAllSides = chunk.neghbors.allFacingNeghborsLoaded;  *//We should guarantee that the buffers get sent to the mesh, because we determine //if a mesh is empty by the size of the verteces* opaqueBuffer.reset(); transBuffer.reset();  greedyMesher.compute(opaqueBuffer, transBuffer, stack, 1,false); naiveMesher.compute(opaqueBuffer, transBuffer, chunk.position,false); opaqueBuffer.makeVertexSet(); *//Buffer will automatically not make verteces if it is empty* transBuffer.makeVertexSet(); |

### Diving deeper (“CODE A” is NOT the problem)

We disabled the below code, we will call this code CODE A:

|  |
| --- |
| IntBuffer vert = MemoryUtil.*memAllocInt*(VertexSet.*VECTOR\_ELEMENTS*); vert.put(0, firstInt); vert.put(1, secondInt); vert.put(2, thridInt); verts1.add(vert); |
| opaqueBuffer.makeVertexSet(); *//Buffer will automatically not make verteces if it is empty* transBuffer.makeVertexSet(); |

* Creating temporary verticies to add to the gpu
* And making the vertex set by creating a new intBuffer

And there is no visible difference in memory consumption (if there is a problem with this, I will only address it AFTER I have COMPLETELY solved the MAIN bottleneck)

* If this is causing problems, I don’t think I will be able to do anything about this, other than using something like an SSBO

|  |
| --- |
| Without adding temp vertex + index buffer |
|  |
| With temp vertex + index buffer |
|  |

# The meshers are the memory culprit!!!

The core memory culprit is greedy and naïve meshers:

* I tested the code snippet with and without the following code, and it is true that the culprit is indeed the meshers, and nothing else.
  + I tested it while leaving “Code A” enabled
* The vertices themselves have nothing to do with it. I already determined that “CODE A” is innocent from memory usage

*greedyMesher.compute(opaqueBuffer, transBuffer, stack, 1, false);  
naiveMesher.compute(opaqueBuffer, transBuffer, chunk.position, false);*

## The Naïve mesher plays a role as well

|  |
| --- |
| When we remove greedy mesher, and just have naïve mesher generating everything:   * “CODE A” is disabled * Some of the memory is being spent with the naïve mesher, but what?   A graph with a line drawn on it  Description automatically generated |

Could hashmap.get be the memory bottleneck for the naïve mesher?

We could replace it with an array where the element index is the block ID

2. Fix it

We have identified that the culprit is mesh gen

\* Lets strip the chunk generation down as much as possible, to eliminate any auxilary problems  
 \* Get rid of sunlight generation  
 \* We will keep the regular terrain for 2 reasons  
 \* The default terrian is fairly ok  
 \* The complex geometry means more vertex memory