Xbuilders 3 notes

# Steps to matching all xbuilders 2 features:

**Assuming it takes 40 hours to complete this, it will take me 2months if I work on it 1hr a day or 1 month if I work on it 2hr a day.**

1. Make, light updating, block set/remove events, block local change event, and block movement events like liquid, fire, or grass.
2. Make Sunlight updating capabilities
3. Create Torch lighting
   1. For channels, use the light byte for complete torch/sun lighting, and if you must, use an int>int hashmap to hold 8 intensity channels, 1,2,3,4,5,6,7,8 (8 should be the shortest distance we can get)
4. Fix saving/loading bug and add light to chunk saving
   1. Pack multiple chunks into a region file? It shouldn’t be too hard (with 32^3 chunks, we save/load 2x more voxels per file than xb2, but with 64^3 regions, we will save 16x more)
      1. Each chunk can still save its data individually, but it must:
         1. mark where it is in the file.
         2. Only allow said file to be written to, one at a time
5. Add events of all blocks
6. add torch and lamp block types
7. Fix bugs with block type normals
8. place multiple blocks simultaneously.
   1. copy/paste.
   2. Decide how sun and events should be handled in bulk
9. Water and grass event handling
10. Add all entities.
    1. Convert doors, slabs into blocks.
    2. Import all animals and vehicles
11. Holograms
12. Clouds
13. TNT billboard animation (hologram)

# Sunlight generation

## Lightning in Xbuilders 3

* We need to wait for all neighbors to be loaded before sunlight generation (skipping neighbors that won’t be generated on the top and bottom of the world )
  + We could also wait for all neighbors before loading chunk due to the creation being the first step.
* We need to implement the sunlight algorithm just like in Xb2.
* We need to test the mesh generation with light to see if we need to multithread the mesh generation algorithm first
  + We don’t need to do that.

# Chunk generation performance on the rendering thread:

TLDR: When flying fast through the world, and with a large render distance, the game becomes choppy. This is closely associated with the memory usage going up and down over and over again rapidly. **After experimentation, I discovered that the memory bottleneck was the creation of int buffers to be sent to the GPU after the chunk mesh was created.**

* **the garbage collector is playing a big role in performance issues.**
  + I think the GC is the cause of the choppiness that occurs when driving thru big worlds.
  + This is definitely true, when the GC is set to 8GB, there is choppiness that occurs when the GC cleans out, this especially happens when GC is cleaned manually.

|  |
| --- |
| What could be the cause of the memory stacking up?  * Its not the greedy mesher. The mesher only uses about 11kb per compute   + Deduce why basic terrain uses significantly less memory than complex (default) terrain * the buffer set is the memory bottleneck   + The arraylist creation of vertices is so minor that It is insignificant.   + **The real bottleneck is the constant creation of intBuffers, to be sent to the GPU** |
| How to optimize int buffers that must be constantly created and we don’t know what size they are beforehand? When you're dealing with unknown or dynamic sizes for data that needs to be sent to the GPU, you can still optimize memory usage with the following strategies:   * Over-Allocation: Allocate a buffer that is larger than what you typically need, and then only fill up as much as required. This can reduce the need for re-allocation if the size varies within a known range. * Dynamic Resizing: Start with a buffer of a certain size, and if you find that it's not large enough, double the size. This follows a similar approach to what ArrayList does in Java when it needs to grow. * Buffer Sub-Data: Allocate a large buffer up front and then use glBufferSubData (in OpenGL) for updating parts of the buffer with new mesh data as it becomes available. This way, you can avoid reallocating the entire buffer when new data comes in. * Memory-Mapped Buffers: In some cases, memory-mapped buffers can be used to efficiently handle large or dynamic data sets.   + A memory mapped buffer is just a shared piece of memory between the CPU and GPU * Object Pooling: Implement an object pool that dynamically adjusts the size of the buffers based on the demands of the application. Unused buffers can be kept around for a while in case they are needed again soon. * Buffer Orphaning: This is a technique where you essentially tell the GPU to disregard the old buffer and you allocate a new one. It can be done using glBufferData with a null data pointer before updating the buffer with new data. This can sometimes be faster than updating the buffer in place and can avoid stalling the graphics pipeline. * Streaming VBOs: Use streaming vertex buffer objects (VBOs) for dynamic data where the buffer contents are updated frequently. |
| Using ArrayList like intBuffer that can be reused within our bufferSet We can have 1 index buffer per bufferSet   * We will have to figure out how to avoid crashes from doing this FIRST * We will make an index buffer of size 1000 * If we need more room, double the size * Don’t worry about shrinking the buffer yet * We will use an index to specify where the buffer ends, and leave the rest alone * We can use subData to only write portions of the data that have changed   + Experement with buffer subData     - I think subData requires a bufferData call first to set the actual list.     - See if we can write more than the original buffer size       * If we can’t do this, we will have to use regular glBufferData if the new intBuffer is larger |