intro3dassign3

Jan Kjærgaard

August 2015

1 Introduction

In this article, I will discuss my implementation of a minecraft-like interactive world.

2 Code overview

2.1 Rendering and updating the world

Voxel worlds are fun to interact with, as any part of the world may be manipualted. However, voxels can be a pretty heavy structure to work with, as there is often a lot of data to process. Voxels by themselves are not easy to visualize on modern hardware, and to visualize them we often have to resort of polygonization of the voxel data. This adds another layer of overhead that is hard to work with.
So, to counter this I created a fairly robust system for updating part of the scene in a efficient manner.
My approach is split up into two parts:

- Vertex Memory Management
- Scene Management

If we start by vertex memory management:
I decided to built on top of the fairly primitive buffer system in webgl, and implement a sort of dynamic memory. This leads itself well to the nature of voxel terrains, as large chunks of the terrain is continuously updated, either by changing, gettig unloaded or loaded again. The dynamic memory is spread over however many vertex buffers the application needs, and from the allocations point of view, it can always allocate/free continious chunks of memory.
The top object in this system is the vertex memory manager (defined in vertex-memory.js), this object can do the follow: allocate page, free page and update. Whenever an application requests a continuous block, the vertex memory manager will find a page for the application. After use, or when the geometry changes the application will deallocate the page again. The operations mark

pages as dirty, and when all operations have been finalized the manager will update the webgl buffers with new memory.
The second object is the vertex buffers themselves, these objects are allocated on demand whenever there are no more pages left to allocate to the application. Each vertex buffer governs a large continuous block of memory and is split into multiple pages. The pages themselves just act as a stack or array like object. They will automatically link another page to themselves when they run out of space, these new pages are allocated by the manager, and may exist on an entirely different vertex buffer.
The system is designed in such a way, that applications allocate and use pages however they need, and then just discard the page after use. The manager cleans the page, and will serve it again the next time memory is needed.

This fits into the scene management pretty nicely, as many other implementations of voxel world, my world is split up into chunks that are further divided into geometryChunks. Each chunk contains the raw voxel data, and each geometry chunk renders some part of the chunk (files defined in chunk.js). Whenever a chunk is marked as dirty, the chunk will locate any dirty geometryChunk and regenerate it. Each geometryChunk manage their own vertex memory through the use of pages explanined above. In this way, any changes to the world affect a most a handful pages at the time. Meaning, the mesh can be partially updated in an efficient manner.

This approach to rendering the world also means that actually rendering the world is fairly easy. First, update dirty/vertexbuffers/pages, then simply walk through all the allocated vertexbuffers, bind them, iterate over pages; rendering continious 'alive' pages.

2.2 Block geometry generation

I implemented a relatively flexible system for generating block geometry. First thing to note is that one should only generate the sides that are actually visible, so two blocks sharing a side should not waste resources rendering the hidden side. In my system I simply encode a integer with the nth bit set meaning the side should be visible. Then a function for rendering each side is called, this function has access to all state/information needed to render said block. Each block type possible to render in the world is defined in block-definition.js and uses block-geometry.js to generate blocks. I use the same code for generating small minirature versions of the blocks that drop when blocks are removed from the world.

2.3 Rest

The rest of the code is mostly the glue or system using these subsystems to generate the code.

3 Which buffers do I have and why?

For the terrain I use some large vertex buffers.
The moon and sun are rendered using a more traditional mesh/material approach.
I have a single framebuffer that I use to query picking information. I query block position and normal info.

4 Which drawcall do I have and why?

I have a single large drawcall to the terrain as explained a few sections ago.
Other than that I have a drawcall to render the wireframe, moon/sun and minirature blocks.
I split these into seperate drawcalls as there really was not reason to be smart about a few small drawcalls.

5 Which parameters are transfering between js/glsl

5.1 Uniforms

I transfer transformations such as perspective/modelview, but also sun/moon lightdirection, ambient lighting and the camera position. The camera position is possibly a bit redundant as I am just using it for calculating the view vector for the specular reflection vector. But I found it practical to have when calculating the distance attenuation.

6 Attributes

- aPosition, aNormal, aUV for rendering
- aID when picking blocks

7 Optimality of my solution

I actually expected the memory system to perform better. It runs very well, but my test computer is a mid 2015 macbook, and I can fly around the world with only minor hitches when generating new chunks. Obivously there are probably a lot of ways to optimize it. Looking at the performance text I put in the upper left corner I am generally updating only a few pages at the time which is what I wanted with this overly complicated system.
I think an even better solution would do the following:

- Merge faces of similar blocks, such that less geometry is generated.

- Find better way to pack vertex attributes (it does not make sense to send a three normals that can a most have three values)
- Allow different sizes of pages and chunks
- Implement some basic culling to guide which geometry chunks to generate, at the very least implement frustrum culling.
- Some subsystems are way to messy and responsibility could probably be seperated better
- Do not implement picking using rendertextures.

So I believe it is possible to find an even better solution for managing minecraft like worlds.