VoxWorld: A Destructible & Procedurally Generated 3D Voxel Engine in C++

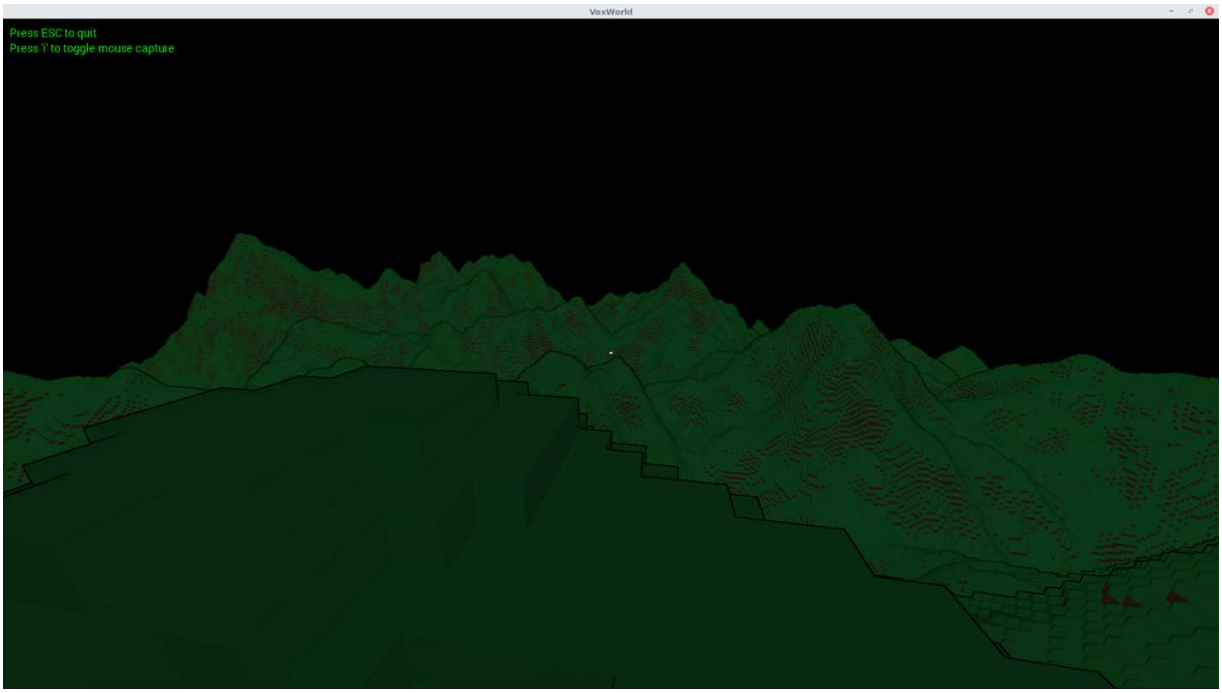
Mitchell Pleune

3/28/2017

Abstract

VoxWorld is a three-dimensional voxel game engine built from the ground up using SDL2 and OpenGL calls. It allowed for research in compressed voxel storage using a custom octree data structure, memory efficient mesh generation, the pros and cons of different rendering pipelines, procedural generation with 2D and 3D noise functions, and post-processing rendering effects. VoxWorld effectively utilizes multiple threads. VoxWorld's code is on Github (https://github.com/Pleune/VoxWorld), and builds nicely on any Linux system. Windows binaries are also available.

Table of Contents



VoxWorld using its 2D perlin generation

Libraries Used

VoxWorld uses two libraries to create windows in a platform agnostic way (SDL2), and wrangle OpenGL function pointers (GLEW).

SDL2 (Simple DirectMedia Layer) is an industry standard library that provides low-level accesses to audio, keyboard, mouse, joystick, and graphics hardware. The library also provides a cross-platform window creation API. SDL2 supports Windows, Linux, Mac OS X, iOS, and Android.

The OpenGL Extension Wrangler Library (GLEW) is a cross-platform open-source C/C++ extension-loading library. GLEW provides efficient run-time mechanisms for determining which OpenGL extensions are supported on the target platform.

Compiling (Linux)

1. Install the required software through your distro's package manager:
    a. Debian Based:

```
# apt-get install build-essential git libsdl2-dev libsdl2-ttf-dev libglew-dev
```

    b. Arch:

```
# pacman –S base-devel git sdl2 sdl2_ttf glew
```

2. Clone the git repo

```
$ git clone https://github.com/Pleune/VoxWorld
```

3. Make and run the project

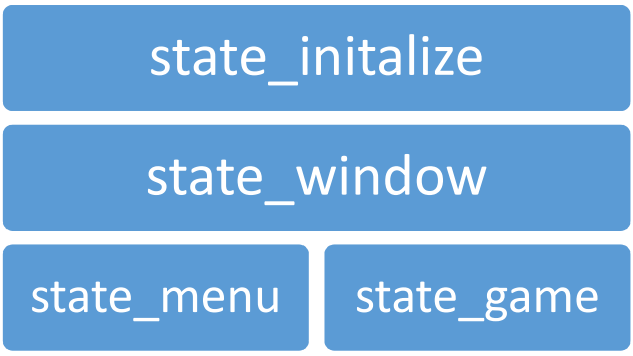
```
$ cd VoxWorld
$ make run
```

State Stack (gameengine.cpp)

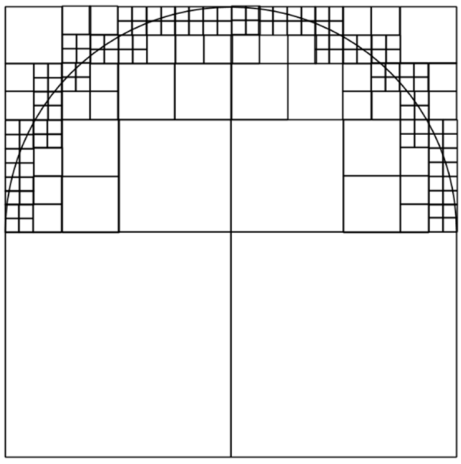
VoxWorld uses a stack based state machine to keep track of the macro structure of the project. The state engine repeatedly calls the `run()` and `input()` functions in the active state (the state on the top of the stack). Each state has three options to control the flow of the program:

- `pop()`, to pop itself off the stack and resume the state under it
- `push()`, to pause itself, and to initialize a new state on top of itself
- `change()`, to pop itself off the stack then immediately push a new state onto the stack.

The state engine is created on program execution, and initializes itself with state_initalize on the stack. The program terminates once all states have popped off the stack. State_initalize and state_window automatically pop themselves off the stack if they are resumed.



Voxel Storage (voxeltree.hpp)



Voxel data is stored in a 3D array like container very similar to an octree. Octrees work by representing a three-dimensional space by a node, then dividing node's space into eight octant-nodes (each side is cut in half) recursively, n-times. Any nodes whose child nodes are all the same are then simplified into a single node. A 2D representation of a curved surface is show to the right.

Octrees proved to be an extremely efficient storage structure, as many huge areas of the world are either entirely land or air, and are the compressed into a small number of large nodes.

The VoxelTree data type goes a step beyond the standard octree implementation by allowing different numbers of divisions per node ($2^3, 2^4, 2^5, etc.$). This is done through templates to allow for far better compiler optimization.

This method of compression takes what would be tens to hundreds of GiB of voxel data and compresses it down to ~100MiB.

Mesh Generation (chunk.cpp)

The mesh generation did not focus on reducing the number of triangles, because the generally accepted solution to this already exists and is well documented: greedy meshing. Instead, efficient usage of memory per vertex is valued. The world mesh is separated into 32x32x32 voxel chunks that are individually generated and rendered.

This is done by creating a master array (also known as an element array) containing all possible vertices (all integer positions within a chunk, and all possible voxel types). The size of this array is $chunkWidth^3 * numVoxelTypes$. A chunk mesh is then simply a list of vertices (every three represents a single triangle). Each vertex is represented by a two byte short that indexes into the master array. This saves considerable amounts of memory, because without the master array six shorts (3 positional and 3 color) represented each vertex.

In the case of a 10,000 chunk world (13 chunk render radius), the 6 short vertex representation used 5 GiB of graphics memory. The element array representation brought this down to 1.1 GiB. There is a slight overhead of ~.3GiB for the element array itself.

Procedural Generation (chunkgen*.cpp)

World generation is handled through an abstract chunk generation class (chunkgen.hpp). There are a couple test generators (chunkgen_simple.hpp). The rest of the generators use either 2D or 3D perlin noise to create infinite procedural terrain.

Perlin noise generates a grid of random values (-1 to 1), and then interpolates smoothly between them. An arbitrary point can be calculated without any contextual information, making it perfect for procedural generation.

In the 2D case, four frequencies of perlin noise are used: 10 blocks, 33 blocks, 125 blocks, and 1000 blocks between points on the grid. The output from each noise function is weighted and summed together to produce a single pixel in a height map.

```
void ChunkGeneratorPerlin2D::gen_map(Heightmap &map)
{
    long3_t cpos = map.pos;
    for(int x=0; x<map.width; x++)
    for(int z=0; z<map.width; z++)
    {
        map.set(x, z,
                Noise::Perlin2D::noise(1, .001,
                          x + cpos.x*c_width, z + cpos.z*c_width)*200 +
                Noise::Perlin2D::noise(2, .008,
                          x + cpos.x*c_width, z + cpos.z*c_width)*90 +
                Noise::Perlin2D::noise(3, .03,
                          x + cpos.x*c_width, z + cpos.z*c_width)*20 +
                Noise::Perlin2D::noise(4, .1,
                          x + cpos.x*c_width, z + cpos.z*c_width)*4
                );
    }
}
```

In the 3D case, a similar approach is used. However, the sum of the Perlin3D functions represents a density. If this density is positive, a block is created, otherwise the block is left as air. Each density value is then adjusted by the world height, so that lower voxels are almost always solid, and more and more voxels become air higher up.

```
//For each voxel
        float density =
            Noise::Perlin3D::noise(1, 0.01, X, Y*2, Z);

        density -= Y/60.0; //Y represents the world height-coordinate
                           //of the voxel
```

Rendering Pipeline (world.cpp)

All world data (voxel data and mesh data) is separated into 32x32x32 voxel chunks. All of these chunks are constructed into a std::map, using its position as a key. Only a single thread accesses this data directly: `World::client_tick_func()`.

This thread continually sweeps the map, and generates a series of vectors containing pointers into the map data. For example, a vector called chunks_for_render is generated containing all of the chunks that have been fully generated and meshed. This is given to the render thread to display:

```
chunks_for_render_m.lock();
if(chunks_for_render)
{
    for(std::vector<Chunk *>::iterator it =
    chunks_for_render->begin(); it != chunks_for_render->end(); it++)
            (*it)->render(camera.pos);
}
chunks_for_render_m.unlock();
```

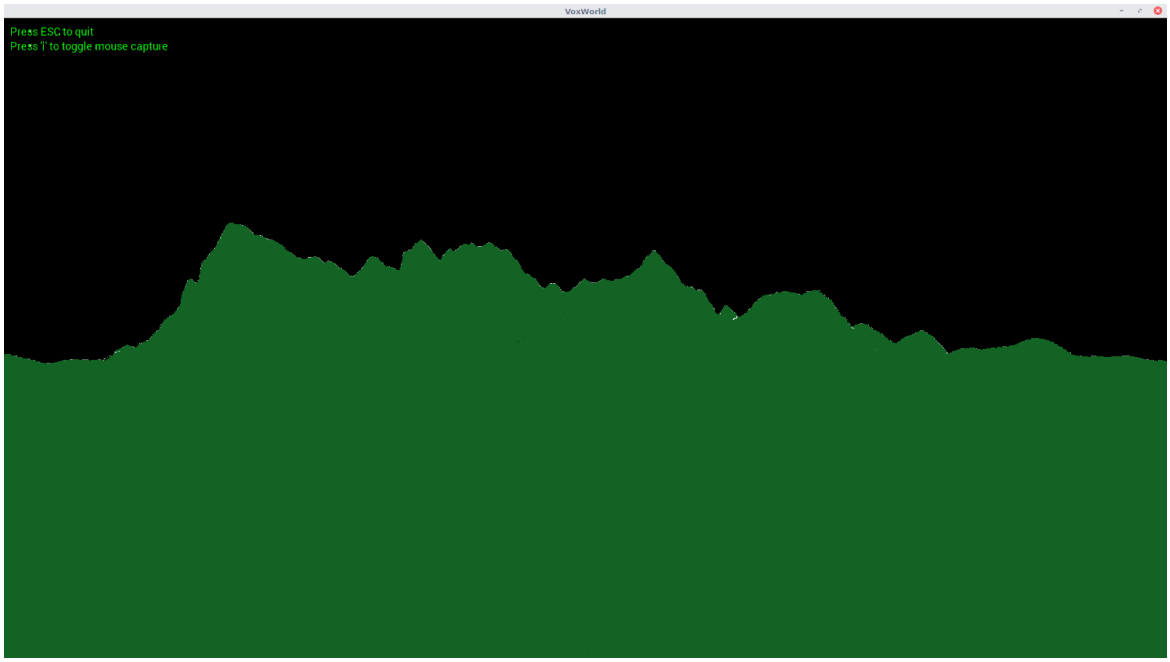
A mutex is used alongside each list, so that the client_tick thread may change the list without disrupting the other threads.

Client_tick tries to keep all chunks within a distance from the camera in the map, and any chunks outside this distance are deleted to free up memory.

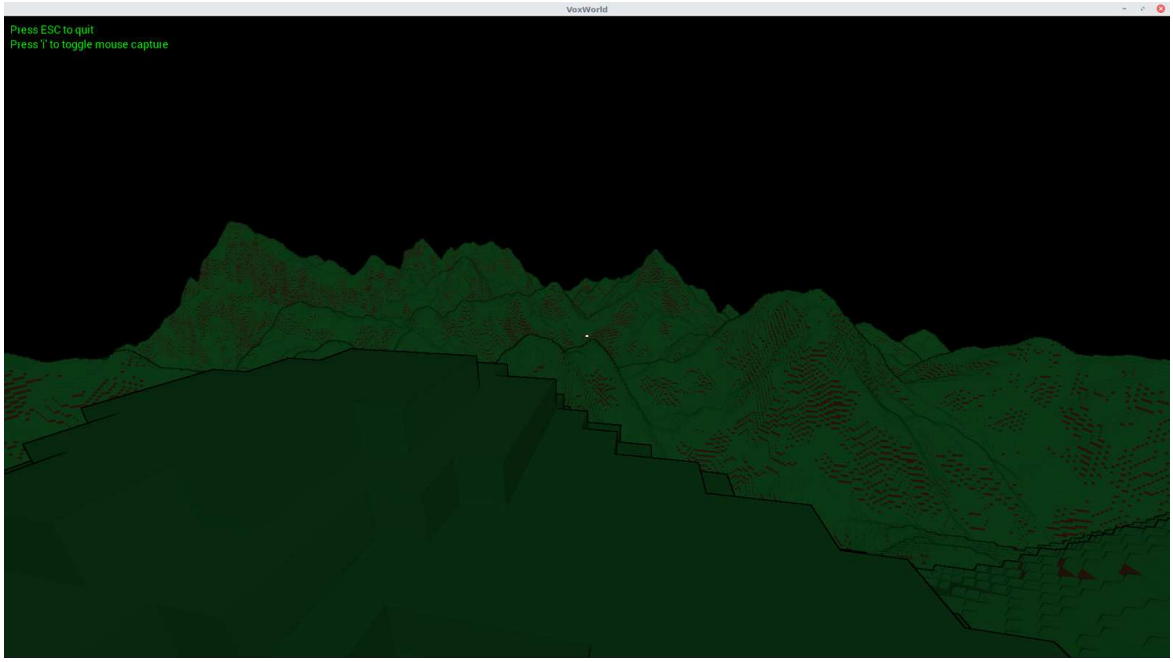
Post Processing (bin/shaders/pfs)

VoxWorld renders its world in a two-step process. First, all of the voxels are rendered as one flat color without any shading. Second, the rendered scene is shaded based on depth-map of the world.

Before:



After:



(Note: the brown dirt voxels in this picture are not in the before picture)

As can be seen, the post-processing adds all of the detail to the rendering. Without it, neither block edges nor mountains are distinguishable.

The depth map of the rendering contains the distances of each pixel from the camera. The post-processing shader effectively takes a two dimensional derivative of the depth map. For every pixel, the larger the magnitude of the derivative map, the more the unshaded render is darkened. This can be seen by the dark outlines between the foreground and the background. The sudden change in distance from the camera creates a bold black line, where every pixel is completely darkened.