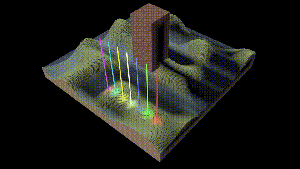
**[Voxel lighting](https://0fps.net/2018/02/21/voxel-lighting/)**

Posted on [February 21, 2018](https://0fps.net/2018/02/21/voxel-lighting/)by [mikolalysenko](https://0fps.net/author/mikolalysenko/" \o "View all posts by mikolalysenko)

Been a long time since I’ve written anything here.  Lately I’ve been working on a new multiplayer WebGL voxel engine for [codemao](https://codemao.cn/) (an educational platform to teach kids programming via games).  It’s not live yet, but a bunch of neat ideas have come out of this project.  In this post I want to talk about the lighting model and share some of the tricks which were used to speed up processing.



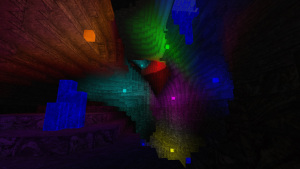
Conceptually the basic lighting system is similar to the one used in Minecraft and Seed of Andromeda with a few small tweaks.  For background information, I would recommend the following sources:

* [Ambient occlusion for Minecraft-like worlds](https://0fps.net/2013/07/03/ambient-occlusion-for-minecraft-like-worlds/)
* [Lighting – Minecraft Wiki](https://minecraft.gamepedia.com/Light)
* [Fast flood fill lighting in a voxel game – Part 1](https://www.seedofandromeda.com/blogs/29-fast-flood-fill-lighting-in-a-blocky-voxel-game-pt-1)
* [Fast flood fill lighting in a voxel game – Part 2](https://www.seedofandromeda.com/blogs/30-fast-flood-fill-lighting-in-a-blocky-voxel-game-pt-2)

This post describes some extensions and optimizations to the basic flood fill lighting model proposed by Ben Arnold.  In particular, we show how to adapt ambient occlusion to track a moving sun, and how to improve flood fill performance using word-level parallelism bit tricks.

Recap of flood fill lighting

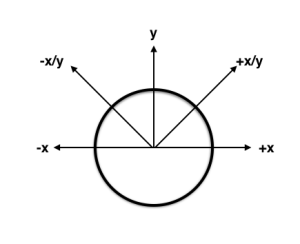
Flood fill lighting is an approximation of ambient occlusion where lighting values are propagated using a breadth-first search along the 6-faces of each cube.  To the best of my knowledge, Minecraft is the first game to use technique to approximate global illumination, but Ben Arnold is the first to write about it in detail.  Minecraft tracks two separate lighting channels.  One for global light values based on time of day, and one for block light levels derived from objects like torches.  This allows for the color of the day light to change dynamically without requiring large terrain updates.  Ben Arnold improves on this basic pattern by storing a separate light level for the red/green/blue allowing for colored light sources:



* **Minecraft**: 1 byte =  4 bits for torch light + 4 bits for sky light
* **Seed of andromeda**: 2 bytes = 4 bits red + 4 bits green + 4 bits blue + 4 bits sky
* **This post**: 4 bytes = 4 bits red + 4 bits green + 4 bits blue + 5 \* 4 bits sky

Multidirectional sun light

The first improvement we propose in this model is modifying the sky light to support multiple directions.  Instead of sampling only sunlight propagation along the y-axis, we also sample along the +/-x axes and the 45-degree diagonal x-y lines and compute ambient contributions for these angle independently:



This requires storing 4 extra light coefficients per-voxel, and if we use 4-bits per coefficient, then this is a total of 16 extra bits.  Combined with the previous 16-bits, this means we need 32-bits of extra lighting data per voxel.

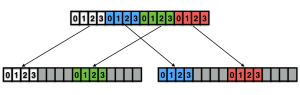
To track the lighting values for the other axes we do essentially the same thing for each extra component.  In the case of the +/-x axes, it should be clear enough how this works.  For the diagonal axes we can trace through a sheared volume.  To index into the diagonal textures we take either the sum/ difference of the x and y components and to get the distance along the ray we can just use the y-value.

Word level parallelism

When propagating light field, we need to often perform component-wise operations on the channels of each light field, which we can pack into a single machine word.  Here’s a picture of how this looks assuming 4-bits per channel for the simpler case of a 32-bit value:



We could do operations on each channel using bit masking/shifting and looping, however there is a better way: [word level parallelism.](https://en.wikipedia.org/wiki/Bit-level_parallelism) We’ll use a general pattern of splitting the coefficients in half and masking out the even/odd components separately so we have some extra space to work.  This can be done by bit-wise &’ing with the mask 0xf0f0f0f:



Less than

The first operation we’ll consider is a pair-wise less than operation.  We want to compare two sets of lighting values and determine which components are < the other.  In pseudo-JS we might implement this operation in the following naive way:

function lightLessThan (a, b) {

let r = 0;

for (let i = 0; i < 8; ++i) {

if ((a & (0xf << i)) < (b & (0xf << i))) {

r |= 0xf << i;

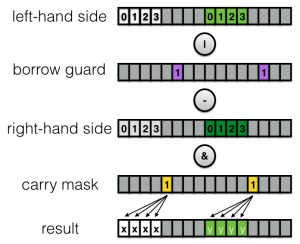
}

}

return r;

}

We can avoid this unnecessary looping using word-level parallelism.  The basic idea is to subtract each component and use the carry flag to check if the difference of each component is negative.  To prevent underflow we can bitwise-or in a guard so that the carry bits are localized to each component.  Here’s a diagram showing how this works:



Putting it all together we get the following psuedo-code:

const COMPONENT\_MASK = 0xf0f0f0f

const BORROW\_GUARD = 0x20202020

const CARRY\_MASK = 0x10101010

function wlpHalfLT (a, b) {

const d = (((a & COMPONENT\_MASK) | BORROW\_GUARD) - (b & COMPONENT\_MASK)) & CARRY\_MASK;

return (d >>> 1) | (d >>> 2) | (d >>> 3) | (d >>> 4);

}

function wlpLT (a:number, b:number) {

return wlpHalfLT(a, b) | (wlpHalfLT(a >> 4, b >> 4) << 4);

}

Maximum

Building on this we can find component-wise maximum of two light vectors (necessary when we are propagating light values).  This key idea is to use the in place bit-swap trick from the identity:

Combined with the above, we can write component-wise max as:

function wlpMax (a, b) {

return a ^ ((a ^ b) & wlpLT(a, b));

}

Decrement-and-saturate

Finally, in flood fill lighting the light level of each voxel decreases by 1 as we propagate.  We can implement a component-wise-decrement and saturate again using the same idea:

function wlpDecHalf (x) {

// compute component-wise decrement

const d = ((x & 0xf0f0f0f) | 0x20202020) - 0x1010101;

// check for underflow

const b = d & 0x10101010;

// saturate underflowed values

return (d + (b >> 4)) & 0x0f0f0f0f;

}

// decrement then saturate each 4 bit component of x

function wlpDec (x) {

return wlpDecHalf(x) | (wlpDecHalf(x >> 4) << 4);

}

Conclusion

Thanks to [codemao](https://codemao.cn/) for supporting this project.