Two Methods for Voxel Detail Enhancement

Adam M. Smith
Hacker Dojo
140A South Whisman Road
Mountain View, California 94041
+1 (408) 335-0362
adam.smith@hackerdojo.com

ABSTRACT

In this paper I describe a novel technique called voxel detail enhancement that, inspired by pixel art scaling algorithms for 2D images, produces finely detailed (3D) voxel maps from the coarser maps that would be edited by a player's actions during gameplay. I describe two methods which deterministically generate fine voxel fragments that depend only on the occupancy of a local window of coarse voxels. Enhanced voxel maps can provide attractive visuals for voxel-based games without requiring the player to manipulate the world at a finer scale. Decoupling the geometry used in graphics and physics from the construction and destruction mechanics of the game opens up new gameplay possibilities in the design space occupied by games like *Minecraft* and *Voxelstein 3D*.

Categories and Subject Descriptors

K.8.0 [**Personal Computing**]: General – *Games.*

General Terms

Algorithms, Design.

Keywords

Games, procedural content generation, voxels, pixel art.

1. INTRODUCTION

Voxels (volumetric picture elements) are the 3D generalization of pixels; they represent objects with shape and texture by specifying colors and/or density values for each point on a regular grid. Where triangle meshes are a very popular vector representation for geometry in videogames (with display often accelerated by dedicated hardware designed around the triangle representation), voxels are the raster-graphics equivalent. Recently, general purpose programmable graphics hardware has enabled acceleration for interactive rendering of voxel data [4].

In this paper I introduce a novel technique which I call *voxel detail enhancement* that can be used to generate detailed, high-resolution voxel maps from coarser voxel maps in a visually coherent, computationally deterministic and parallelizable manner. This technique opens up new possibilities in an already-underappreciated region in the space of game designs.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
PCGames 2011, June 28, 2011, Bordeaux, France.
Copyright 2011 ACM 978-1-4503-0872-4/11/06…$10.00.

1.1 Voxel-based Gameplay

Games with player-editable voxel worlds are uncommon, but they allow the player much greater flexibility in creating (or destroying) 3D shapes than a vector representation affords because surface topology is represented implicitly rather than explicitly.

Voxelstein 3D [2] is a voxel-based first-person shooter inspired by *Wolfenstein 3D* [1] in which the environment is modeled with cubes approximately four centimeters on a side. While the detail level of the voxel graphics of *Voxelstein* is similar to that of the pixel graphics in *Wolfenstein*, a one-to-one mapping between player actions and voxel edits implies that would-be trivial operations in another game, such as cutting through thin metal bars, take tens of actions in *Voxelstein* (slowing down an otherwise action-oriented game).

Minecraft [3] is a sandbox survival and construction game that uses a much coarser level of detail in its use of voxels. *Minecraft*'s world is modeled in cubes that are approximately one meter on a side in relation to the player's avatar. While this scale is appropriate for the large-scale geography and player-created architectural works for which the game is known, this design choice bars the player from influencing finer-scale visual details (a workbench object, for example, is represented by a single unit voxel).

1.2 Pixel Art Scaling Algorithms

Towards decoupling display-voxels from editable-voxels (merging *Voxelstein*'s visual details with *Minecraft*'s construction mechanics), I was inspired by the pixel art scaling algorithms used in arcade and console game emulators. These algorithms upsample the carefully-crafted pixel art of classic games for display on modern, high-resolution screens in a manner that attempts preserve the character of the original work. In contrast to standard image scaling algorithms that introduce blur into the scaled output via interpolation, pixel art scaling algorithms preserve flat patches and angled edges in the original work when rendering output pixels.

Edge detection and related logic in these algorithms operates by examining a small window of pixels in the source image to decide which features (lines, edges, corners, gradients, etc.) are present in the original and should be reproduced in finer detail in the output. In particular, the basic Scale2x[1] algorithm used in several emulators produces a 2x2 patch of output pixels for each pixel in the input (conditioned on its eight surrounding pixels). Any power-of-two expansion factor can be achieved using algorithms similar to Scale2x by repeated scaling with the same algorithm.

[1] http://scale2x.sourceforge.net/index.html

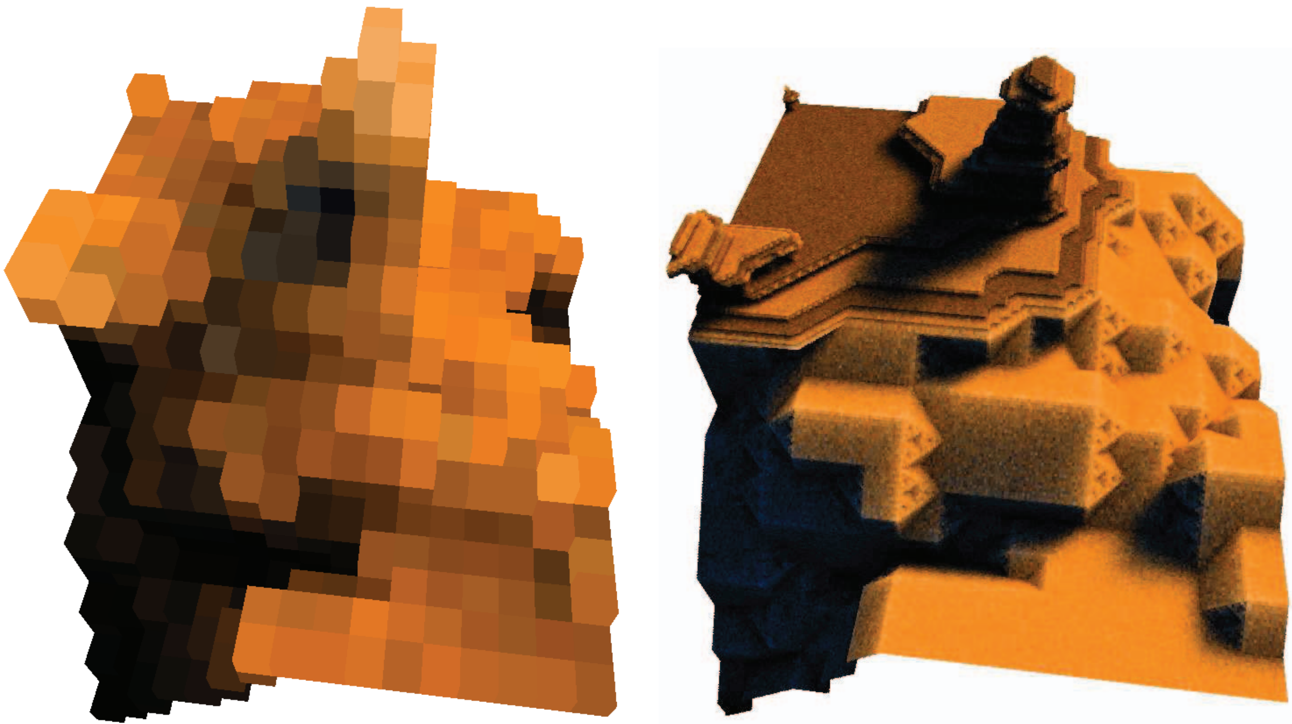

Figure 1. Enhancing a 16x16x16 input voxel map (left) to produce a detailed, 256x256x256 output fragment map (right) using the *recursive subdivision* operator. Below a certain altitude, a single threshold value was used (1), and above, two other values were vertically alternated (3 and 4) to yield a variety of terrain types. The input map was initialized with 3D Perlin noise, and both maps were lit with the same lighting algorithm based on wandering particles (see Section 4.4 for lighting details).

Another family of pixel art scaling algorithms, the hq*nx*[2] family, works by determining which determining which of a source pixels neighbors are similar to it, and then applying a hand-crafted lookup table to decide the appropriate action to take in the output image. One instantiation, hq4x, examines 4x4 patches centered on each pixel of the input to produce 4x4 output. This algorithm is able to exactly reproduce lines of several distinct slopes and present them, anti-aliased, in the output. Further, it is able to smooth dithered gradients from the source images where pixel colors are similar, yielding an appropriate mixture of sharp and soft details in the output.

The inherently two-dimensional logic of these algorithms does not directly generalize to voxel maps (primarily because the shape and size of adjacency neighborhoods changes when adding another dimension). However, it is possible to translate their basic intent to 3D, yielding a kind of voxel art scaling.

1.3 Voxel Art Scaling Algorithms

Currently, there exist no algorithms that do for voxels what pixel art scaling algorithms do for pixels. If such algorithms existed, they could be used to programmatically rescale the art in games employing voxel sprites (such as the in-development arcade shooter *Voxatron*[3] by Lexaloffle Games), saving artist effort and enabling novel visual effects. Further, voxel art scaling algorithms should allow detail-preserving rotation of voxel sprites akin to the use of pixel art scaling in the RotSprite[4] utility for 2D sprites.

Instead of addressing the full complexity of voxel art scaling, this paper focuses on simply expanding voxels maps in a way that preserves the character of the input while elaborating with plausible details on the finer scale. Voxel detail enhancement refers to this simpler problem.

1.4 Vocabulary

In the description of my techniques to follow, I will refer to the inputs to each algorithm as the "input map" composed of coarse "voxels" over which I imagine a player having direct control (as in *Minecraft*). The "output map" is composed of fine "fragments" which the player can see and interact with (via walking/driving/climbing) but not destructively manipulate. I assume the input map encodes the density at each voxel as a single bit (either solid or empty) and that every solid voxel is made of a diffuse white material for the purposes of shading.

2. RECURSIVE SUBDIVISION

My first method of voxel detail enhancement, recursive subdivision, is most similar to the Scale2x pixel algorithm. Recursive subdivision works by producing a 2x2x2 patch of fragments for each input voxel. To decide whether an output fragment is turned on (solid vs. empty) I examine the occupancy of the source voxel and its three immediately neighboring voxels in the direction of the fragment in question. Concretely, when generating the front-top-right fragment of a voxel, I examine the source voxel's front,

[2] http://www.hiend3d.com/hq3x.html

[3] http://www.lexaloffle.com/voxatron.php

[4] http://info.sonicretro.org/RotSprite

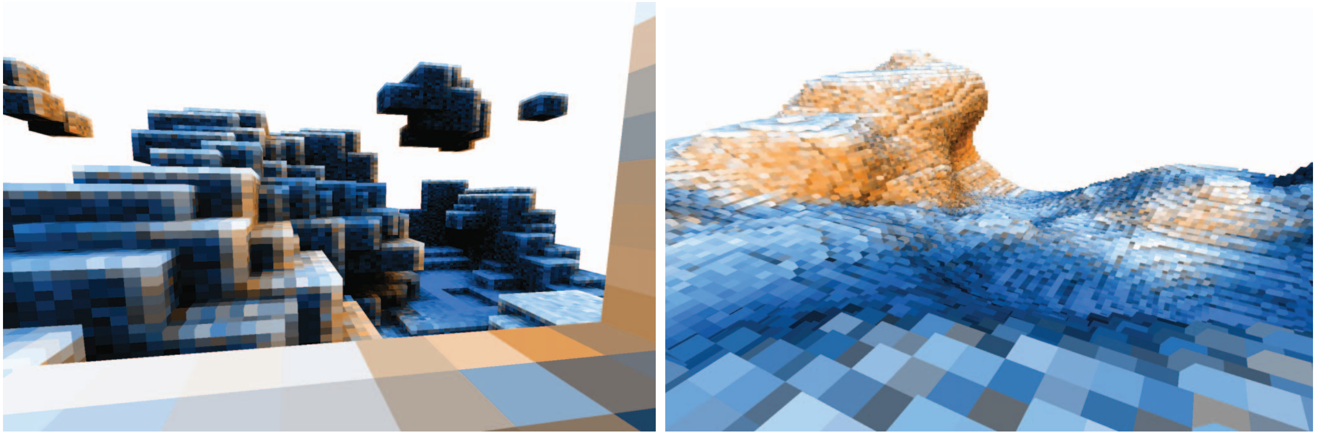

Figure 2. First-person views of two 128x128x128 fragment maps generated by enhancing separate, very coarse 8x8x8 input maps (not shown) via the *direct resampling* operator with a spread of 1.0 voxels and threshold of 25%. The image at left used a crude 4-sample pattern to yield a jittered, cubist reinterpretation of the input map while the image at right used 512 probe samples to achieve a softer feel. Input maps were again initialized with 3D Perlin noise.

top, and right neighbors. By comparing the count of solid voxels in this neighborhood to a constant (described later), I set the output fragment's occupancy bit.

In computing the eight output fragments, only the source voxel and its six face-adjacent neighboring voxels need be considered. The fragments of one voxel do not depend on the fragments of another. This implies that voxel detail enhancement is an embarrassingly parallelizable process.

As with Scale2x, higher levels of refinement can be achieved by applying the refinement operator several times (using the fragment output of one level as the voxel input to another), assuming a power-of-two scaling factor is desired. Figure 1 shows the result of applying the recursive subdivision technique four times to achieve a 1:16 linear expansion (4,096 potential fragments for each voxel). In this case, the input map was created by thresholding a 3D Perlin noise function similar to the one used in *Minecraft*'s world generator.

This technique has a single tunable constant: the activation threshold. In the lower altitudes of Figure 1, a value of one was used (i.e. a fragment is turned on if any voxel in its neighborhood is turned on) yielding the emergence of Sierpinski triangle patterns on angled facets. In the higher altitudes, the value alternated between three and four (depending on the least significant bit of the fragment's altitude), yielding delicate strata with rounded corners.

I have considered, but not implemented, using a different activation threshold (or spatially-varying threshold pattern) based on the in-game material type of the input voxel. This should result in a variety of player-visible materials without any major changes to the algorithm.

3. DIRECT RESAMPLING

The self-similar patterns produced by the recursive subdivision enhancement operator are visually interesting, but they would not be appropriate for the visual themes of every game that might use voxel detail enhancement. To address this, and overcome the power-of-two limitation of recursive subdivision, I experimented with another technique: a direct resampling operator.

Direct resampling is more inspired by image interpolation than pixel art scaling. Where the four-voxel neighborhood in recursive subdivision is specifically designed for 2x scaling, a more flexible (and chaotic) neighborhood is used in direct resampling. The core of both algorithms, however, is the same: if probing neighboring voxels yields more than a certain number of solid voxels, the output fragment will be turned on.

Before direct resampling begins, a random sampling pattern is generated once and stored. This pattern consists of a set of offsets to be applied to the current fragment's location when probing for neighbors. The spread of these samples controls the detail enhancement operator's ability to smooth corners and diagonal lines, and the count of these samples controls the smoothness of the resulting resampled surface.

Figure 2 shows two concrete examples of the direct resampling operator in action. In the two worlds depicted, sampling patterns consisting of 4 and 512 probe samples were created using offsets in the range -1.0 to 1.0 (in voxel-units, this is a relatively conservative spread). During fragment generation, these offsets are added to the fragment's center position and the resulting number (converted to an integer) is used to look up the occupancy of a voxel in the input map. If more than 25% of these probes reach a solid voxel, the fragment is made solid in the output. This process was carried out for every fragment on a 16x16x16 grid within each voxel yielding a similar expansion ratio to that shown in Figure 1 (however, any positive integer could have been used instead of 16 in this place).

In essence, the direct resampling operator is carrying out a very crude form of density estimation followed by a threshold operation. Using a larger number samples and a 50% activation threshold would reproduce box-filtered anti-aliasing of the input map. Per our motivation, such perfect resampling is not actually desirable (and could be achieved in a much easier way). Thus, getting attractive results from the direct resampling method requires careful attention to the count, spread, and threshold for probe samples. Achieving a desired texture requires balancing the natural grit from sampling noise with blur that erodes the coherence between the output fragments and the player's intentions as expressed in the input map. In manual experimentation with these tweakable parameters, I have been able to achieve effects ranging from a cubist reinterpretation of the input, to rolling hills that capture the input in the form of broad curves (styles shown in Figure 2).

4. RELATED CONCERNS

4.1 Visibility and Computational Complexity

Voxel detail enhancement quickly produces large amounts of data. However, for terrain-like voxel maps, most regions of the input map are either completely solid or completely empty (transparent). Knowing that only fragments on the surface between solid an empty (between land and air) need to be displayed to the player or used in game physics, it is possible to quickly determine if a voxel should be completely skipped during enhancement (by checking if any of its immediate neighbors differ in occupancy), thus saving both computation and storage.

In voxel worlds dominated by terrains (as opposed to a volumetric froth), the visible fragments roughly form a 2D surface (albeit heavily distorted). Storage and computation for the fragments on this manifold should scale quadratically with the side-length expansion factor despite the cubic growth of the number of potential fragments in the worst case.

4.2 Neighbor Dependence and Caching

When applying voxel detail enhancement in an interactive setting, it is important to keep track of which nearby voxels can possibly affect the fragment generation for a given voxel. When the player interactively toggles a voxel occupancy bit in the coarse input map, a small cluster of voxels must be re-enhanced.

For recursive subdivision, only the six immediately adjacent voxels can possibly affect the fragments generated for a central voxel. If desired, a lookup table can map the occupancy state of the relevant voxels (one of 128 possible configurations) to a pre-computed fragment set of arbitrary detail. For the direct resampling method, depending on the sampling radius, a much larger number of voxels can potentially contribute to the fragments of a central voxel. In this case a lazily-populated cache would store details for only those configurations which actually occur during gameplay (a quickly decreasing fraction of all possible configurations actually occurring, as a function of sampling radius, due to the visibility-culling opportunities in terrain-like input maps).

4.3 Physics

Depending on the gameplay experience desired, either the input voxels or the output fragments can be used to drive the physics simulation in game (i.e. collision detection and resolution for feet, wheels, and claws). Where voxels fall short of triangles, with respect to game physics, is in their poor representation of surface orientation. Triangles naturally possess a normal vector which can point in any direction, but voxels, however much enhanced, still only have surfaces that point in the six axis-aligned directions. Computing surface normals for voxel data would require a local estimation procedure with which I have not yet experimented.

4.4 Lighting

In the examples I show above, lighting was accomplished using a new method unique to the voxel representation. Instead of tracing straight-line rays to a light source and nearby surfaces, I release a number of particles (that act more like snowflakes falling-in-reverse than photons of light) which are allowed to randomly wander in small steps through the fragment-level volume representation. One set of particles representing psuedo-sunlight illumination wanders towards the direction of sunlight while another set representing pseudo-skylight illumination wanders uniformly in all directions. If a particle successfully wanders for a set number of steps without encountering any solid fragments, it contributes a small amount of illumination to the source fragment, colored by its type.

In my experiments with voxel detail enhancement thus far, this *snowflake tracing* mode of lighting has been the dominant source of computational effort. In contrast to traditional method of surface shading (such as the default Gouraud shading often directly supported by hardware), this method emphasizes the overall shape of a voxel map in favor of its surface normals (which point in uninformative directions). Thus, in my experiments, voxels are drawn as small cubes with flat shading, where there emissive color is determined by the pre-computed illumination.

To achieve long shadows from towers and near absolute darkness in caves, I allow the particles used in lighting to wander across a radius of several voxels. This means that, when a single voxel's occupancy bit is toggled, a much larger window of voxels must be reconsidered for enhancement than was strictly required for basic fragment generation (turning on one voxel could potentially cast a soft shadow on a fragment several voxels away). So far, I have accepted this larger dependence window in exchange for more attractive looking results. In a complete game where machine cycles dedicated to pre-computing fragment lighting should be minimized, a mixture of caching and partially voxel-level (as opposed to completely fragment-level) lighting could dramatically reduce the computational effort required with only modest losses to visual quality.

5. FUTURE WORK

In future work, I intend to investigate the inclusion color, a primary element of pixel art scaling algorithms that I omitted in my experiments. In voxel-based games, voxel color is a natural way to encode material type (dirt vs. stone vs. wood, and so on). Even in Scale2x, pixels are checked to see if they have the same (or similar) color, not just whether they are both turned on as in my recursive subdivision operator. Bringing a notion of color into voxel detail enhancement would allow dirt voxels to smoothly blend into other dirt voxels while having a crisp edge where dirt abuts stone.

Additionally, I intend to experiment with integrating normal-vector generation with the existing geometric detail enhancement processes so that better surface orientation information is present in support of shading and collision calculations.

6. CONCLUSION

I have proposed two methods for *voxel detail enhancement*, a way to add fine geometric details to voxel-based representations. Using either of the techniques I describe, future game designs may include both display of and physical interaction with (via vehicle physics, perhaps) fine geometric details which coherently and deterministically correspond to a coarser voxel map that is edited by players on a natural scale.

7. ACKNOWLEDGEMENTS

This work was conceived at *TIGJam3*, prototyped at *SuperHappyDevHouse41*, and further explored in the ongoing *South Bay Game Jam* series, all events hosted at the Hacker Dojo community center for hackers and thinkers.

8. REFERENCES

[1] Id Software. 1992. *Wolfenstein 3D* (videogame).

[2] Krieg, Hans. 2008. *Voxelstein 3D* (videogame).

[3] Persson, Markus. 2009. *Minecraft* (videogame).

[4] Römisch, Kristof. 2009. *Sparse voxel octree ray tracing on the GPU* (Master's Thesis). Retrieved from: http://www.daimi.au.dk/~aquak/MasterThesisKristofRoemisch.pdf