# Greedy meshing demo

Saturday, October 14, 2023

9:04 AM

## [GreedyMesh/src/mygame/Main.java at master · roboleary/GreedyMesh · GitHub](https://github.com/roboleary/GreedyMesh/blob/master/src/mygame/Main.java)

Below is the code written in UI4J. It includes an entire runnable demo.

package com.xbuilders.engine.mesh;

import com.xbuilders.window.Window;

import java.io.IOException;

import java.util.logging.Level;

import java.util.logging.Logger;

import org.joml.Vector3f;

import processing.core.PShape;

import processing.core.UIFrame;

import processing.event.KeyEvent;

import processing.event.MouseEvent;

/\*\*

\* This is a Java greedy meshing implementation based on the javascript

\* implementation written by Mikola Lysenko and described in this blog post:

\*

\* <http://0fps.wordpress.com/2012/06/30/meshing-in-a-minecraft-game/>

\*

\* The principal changes are:

\*

\* - Porting to Java - Modification in order to compare \*voxel faces\*, rather

\* than voxels themselves - Modification to provide for comparison based on

\* multiple attributes simultaneously

\*

\* This class is ready to be used on the JMonkey platform - but the algorithm

\* should be usable in any case.

\*

\* @author Rob O'Leary

\*/

public class GreedyMeshing extends UIFrame {

/\*

\* In this test each voxel has a size of one world unit - in reality a voxel engine

\* might have larger voxels - and there's a multiplication of the vertex coordinates

\* below to account for this.

\*/

private static final int VOXEL\_SIZE = 10;

/\*

\* These are the chunk dimensions - it may not be the case in every voxel engine that

\* the data is rendered in chunks - but this demo assumes so. Anyway the chunk size is

\* just used to populate the sample data array. Also, in reality the chunk size will likely

\* be larger - for example, in my voxel engine chunks are 16x16x16 - but the small size

\* here allows for a simple demostration.

\*/

private static final int CHUNK\_WIDTH = 3;

private static final int CHUNK\_HEIGHT = 3;

/\*

\* This is a 3D array of sample data - I'm using voxel faces here because I'm returning

\* the same data for each face in this example - but calls to the getVoxelFace function below

\* will return variations on voxel data per face in a real engine. For example, in my system

\* each voxel has a type, temperature, humidity, etc - which are constant across all faces, and

\* then attributes like sunlight, artificial light which face per face or even per vertex.

\*/

private final VoxelFace[][][] voxels = new VoxelFace[CHUNK\_WIDTH][CHUNK\_HEIGHT][CHUNK\_WIDTH];

/\*

\* These are just constants to keep track of which face we're dealing with - their actual

\* values are unimportantly - only that they're constant.

\*/

private static final int SOUTH = 0;

private static final int NORTH = 1;

private static final int EAST = 2;

private static final int WEST = 3;

private static final int TOP = 4;

private static final int BOTTOM = 5;

@Override

public void settings() {

size(400, 400, P3D);

}

@Override

public void draw() {

/\*

\* And now that the sample data is prepared, we launch the greedy meshing.

\*/

noFill();

background(204);

camera(70.0f, 35.0f, 120.0f,

0.0f, 0.0f, 0.0f,

0.0f, 1.0f, 0.0f);

stroke(1);

rotateY((float) mouseX / 100);

rotateZ((float) mouseY / 100);

beginShape(TRIANGLES);

greedy();

endShape();

// noLoop();

}

@Override

public void mouseEvent(MouseEvent me) {

}

@Override

public void keyTyped(KeyEvent ke) {

}

@Override

public void keyPressed(KeyEvent ke) {

}

@Override

public void keyReleased(KeyEvent ke) {

}

@Override

public void windowFocusGained() {

}

@Override

public void windowFocusLost() {

}

@Override

public void windowResized() {

}

@Override

public void windowCloseEvent() {

}

/\*\*

\* This class is used to encapsulate all information about a single voxel

\* face. Any number of attributes can be included - and the equals function

\* will be called in order to compare faces. This is important because it

\* allows different faces of the same voxel to be merged based on varying

\* attributes.

\*

\* Each face can contain vertex data - for example, int[] sunlight, in order

\* to compare vertex attributes.

\*

\* Since it's optimal to combine greedy meshing with face culling, I have

\* included a "transparent" attribute here and the mesher skips transparent

\* voxel faces. The getVoxelData function below - or whatever it's

\* equivalent might be when this algorithm is used in a real engine - could

\* set the transparent attribute on faces based on whether they should be

\* visible or not.

\*/

class VoxelFace {

public boolean transparent;

public int type;

public int side;

public boolean equals(final VoxelFace face) {

return face.transparent == this.transparent && face.type == this.type;

}

}

/\*\*

\* This is just the main function used to start the demo on JMonkey.

\*

\* @param args

\*/

public static void main(String[] args) {

final GreedyMeshing app = new GreedyMeshing();

}

public GreedyMeshing() {

super();

startWindow();

}

/\*\*

\* This is an initialization function used here to set up the sample voxel

\* data and launch the greedy meshing.

\*/

@Override

public void setup() {

VoxelFace face;

for (int i = 0; i < CHUNK\_WIDTH; i++) {

for (int j = 0; j < CHUNK\_HEIGHT; j++) {

for (int k = 0; k < CHUNK\_HEIGHT; k++) {

if (i > CHUNK\_WIDTH / 2 && i < CHUNK\_WIDTH \* 0.75

&& j > CHUNK\_HEIGHT / 2 && j < CHUNK\_HEIGHT \* 0.75

&& k > CHUNK\_HEIGHT / 2 && k < CHUNK\_HEIGHT \* 0.75) {

/\*

\* We add a set of voxels of type 1 at the top-right of the chunk.

\*

\*/

face = new VoxelFace();

face.type = 1;

/\*

\* To see an example of face culling being used in combination with

\* greedy meshing, you could set the trasparent attribute to true.

\*/

face.transparent = true;

} else if (i == 0) {

/\*

\* We add a set of voxels of type 2 on the left of the chunk.

\*/

face = new VoxelFace();

face.type = 2;

} else {

/\*

\* And the rest are set to type 3.

\*/

face = new VoxelFace();

face.type = 3;

}

voxels[i][j][k] = face;

}

}

}

}

/\*\*

\*

\*/

void greedy() {

/\*

\* These are just working variables for the algorithm - almost all taken

\* directly from Mikola Lysenko's javascript implementation.

\*/

int i, j, k, l, w, h, u, v, n, side = 0;

final int[] x = new int[]{0, 0, 0};

final int[] q = new int[]{0, 0, 0};

final int[] du = new int[]{0, 0, 0};

final int[] dv = new int[]{0, 0, 0};

/\*

\* We create a mask - this will contain the groups of matching voxel faces

\* as we proceed through the chunk in 6 directions - once for each face.

\*/

final VoxelFace[] mask = new VoxelFace[CHUNK\_WIDTH \* CHUNK\_HEIGHT];

/\*

\* These are just working variables to hold two faces during comparison.

\*/

VoxelFace voxelFace, voxelFace1;

/\*\*

\* We start with the lesser-spotted boolean for-loop (also known as the

\* old flippy floppy).

\*

\* The variable backFace will be TRUE on the first iteration and FALSE

\* on the second - this allows us to track which direction the indices

\* should run during creation of the quad.

\*

\* This loop runs twice, and the inner loop 3 times - totally 6

\* iterations - one for each voxel face.

\*/

for (boolean backFace = true, b = false; b != backFace; backFace = backFace && b, b = !b) {

/\*

\* We sweep over the 3 dimensions - most of what follows is well described by Mikola Lysenko

\* in his post - and is ported from his Javascript implementation. Where this implementation

\* diverges, I've added commentary.

\*/

for (int d = 0; d < 3; d++) {

u = (d + 1) % 3;

v = (d + 2) % 3;

x[0] = 0;

x[1] = 0;

x[2] = 0;

q[0] = 0;

q[1] = 0;

q[2] = 0;

q[d] = 1;

/\*

\* Here we're keeping track of the side that we're meshing.

\*/

if (d == 0) {

side = backFace ? WEST : EAST;

} else if (d == 1) {

side = backFace ? BOTTOM : TOP;

} else if (d == 2) {

side = backFace ? SOUTH : NORTH;

}

/\*

\* We move through the dimension from front to back

\*/

for (x[d] = -1; x[d] < CHUNK\_WIDTH;) {

/\*

\* -------------------------------------------------------------------

\* We compute the mask

\* -------------------------------------------------------------------

\*/

n = 0;

for (x[v] = 0; x[v] < CHUNK\_HEIGHT; x[v]++) {

for (x[u] = 0; x[u] < CHUNK\_WIDTH; x[u]++) {

/\*

\* Here we retrieve two voxel faces for comparison.

\*/

voxelFace = (x[d] >= 0) ? getVoxelFace(x[0], x[1], x[2], side) : null;

voxelFace1 = (x[d] < CHUNK\_WIDTH - 1) ? getVoxelFace(x[0] + q[0], x[1] + q[1], x[2] + q[2], side) : null;

/\*

\* Note that we're using the equals function in the voxel face class here, which lets the faces

\* be compared based on any number of attributes.

\*

\* Also, we choose the face to add to the mask depending on whether we're moving through on a backface or not.

\*/

mask[n++] = ((voxelFace != null && voxelFace1 != null && voxelFace.equals(voxelFace1)))

? null

: backFace ? voxelFace1 : voxelFace;

}

}

x[d]++;

/\*

\* Now we generate the mesh for the mask

\*/

n = 0;

for (j = 0; j < CHUNK\_HEIGHT; j++) {

for (i = 0; i < CHUNK\_WIDTH;) {

if (mask[n] != null) {

/\*

\* We compute the width

\*/

for (w = 1; i + w < CHUNK\_WIDTH && mask[n + w] != null && mask[n + w].equals(mask[n]); w++) {

}

/\*

\* Then we compute height

\*/

boolean done = false;

for (h = 1; j + h < CHUNK\_HEIGHT; h++) {

for (k = 0; k < w; k++) {

if (mask[n + k + h \* CHUNK\_WIDTH] == null || !mask[n + k + h \* CHUNK\_WIDTH].equals(mask[n])) {

done = true;

break;

}

}

if (done) {

break;

}

}

/\*

\* Here we check the "transparent" attribute in the VoxelFace class to ensure that we don't mesh

\* any culled faces.

\*/

if (!mask[n].transparent) {

/\*

\* Add quad

\*/

x[u] = i;

x[v] = j;

du[0] = 0;

du[1] = 0;

du[2] = 0;

du[u] = w;

dv[0] = 0;

dv[1] = 0;

dv[2] = 0;

dv[v] = h;

/\*

\* And here we call the quad function in order to render a merged quad in the scene.

\*

\* We pass mask[n] to the function, which is an instance of the VoxelFace class containing

\* all the attributes of the face - which allows for variables to be passed to shaders - for

\* example lighting values used to create ambient occlusion.

\*/

quad(new Vector3f(x[0], x[1], x[2]),

new Vector3f(x[0] + du[0], x[1] + du[1], x[2] + du[2]),

new Vector3f(x[0] + du[0] + dv[0], x[1] + du[1] + dv[1], x[2] + du[2] + dv[2]),

new Vector3f(x[0] + dv[0], x[1] + dv[1], x[2] + dv[2]),

w,

h,

mask[n],

backFace);

}

/\*

\* We zero out the mask

\*/

for (l = 0; l < h; ++l) {

for (k = 0; k < w; ++k) {

mask[n + k + l \* CHUNK\_WIDTH] = null;

}

}

/\*

\* And then finally increment the counters and continue

\*/

i += w;

n += w;

} else {

i++;

n++;

}

}

}

}

}

}

}

/\*\*

\* This function returns an instance of VoxelFace containing the attributes

\* for one side of a voxel. In this simple demo we just return a value from

\* the sample data array. However, in an actual voxel engine, this function

\* would check if the voxel face should be culled, and set per-face and

\* per-vertex values as well as voxel values in the returned instance.

\*

\* @param x

\* @param y

\* @param z

\* @param face

\* @return

\*/

VoxelFace getVoxelFace(final int x, final int y, final int z, final int side) {

VoxelFace voxelFace = voxels[x][y][z];

voxelFace.side = side;

return voxelFace;

}

/\*\*

\* This function renders a single quad in the scene. This quad may represent

\* many adjacent voxel faces - so in order to create the illusion of many

\* faces, you might consider using a tiling function in your voxel shader.

\* For this reason I've included the quad width and height as parameters.

\*

\* For example, if your texture coordinates for a single voxel face were 0 -

\* 1 on a given axis, they should now be 0 - width or 0 - height. Then you

\* can calculate the correct texture coordinate in your fragement shader

\* using coord.xy = fract(coord.xy).

\*

\*

\* @param bottomLeft

\* @param topLeft

\* @param topRight

\* @param bottomRight

\* @param width

\* @param height

\* @param voxel

\* @param backFace

\*/

void quad(final Vector3f bottomLeft,

final Vector3f topLeft,

final Vector3f topRight,

final Vector3f bottomRight,

final int width,

final int height,

final VoxelFace voxel,

final boolean backFace) {

final Vector3f[] vertices = new Vector3f[4];

vertices[2] = topLeft.mul(VOXEL\_SIZE);

vertices[3] = topRight.mul(VOXEL\_SIZE);

vertices[0] = bottomLeft.mul(VOXEL\_SIZE);

vertices[1] = bottomRight.mul(VOXEL\_SIZE);

final int[] indexes = backFace ? new int[]{2, 0, 1, 1, 3, 2} : new int[]{2, 3, 1, 1, 0, 2};

// final float[] colorArray = new float[4 \* 4];

//

// for (int i = 0; i < colorArray.length; i += 4) {

//

// /\*

// \* Here I set different colors for quads depending on the "type" attribute, just

// \* so that the different groups of voxels can be clearly seen.

// \*

// \*/

// if (voxel.type == 1) {

//

// colorArray[i] = 1.0f;

// colorArray[i + 1] = 0.0f;

// colorArray[i + 2] = 0.0f;

// colorArray[i + 3] = 1.0f;

//

// } else if (voxel.type == 2) {

//

// colorArray[i] = 0.0f;

// colorArray[i + 1] = 1.0f;

// colorArray[i + 2] = 0.0f;

// colorArray[i + 3] = 1.0f;

//

// } else {

//

// colorArray[i] = 0.0f;

// colorArray[i + 1] = 0.0f;

// colorArray[i + 2] = 1.0f;

// colorArray[i + 3] = 1.0f;

// }

// }

switch (voxel.type) {

case 1 ->

fill(255, 0, 0);

case 2 ->

fill(0, 0, 255);

case 3 ->

fill(0, 255, 0);

default ->

fill(255);

}

for (int i : indexes) {

vertex(vertices[i].x, vertices[i].y, vertices[i].z);

}

}

}