Cone Step Mapping:

An Iterative Ray-Heightfield Intersection Algorithm

Jonathan “LoneSock” Dummer

Introduction

This paper presents Cone Step Mapping (CSM), a new approach to the Ray-Heightfield Intersection problem. CSM uses preprocessed information about the heightfield to dynamically adjust step sizes along the ray when finding the first intersection. It is an iterative method that is unconditionally stable, without the need to tune algorithm parameters. This method is suitable for real-time rendering and has been implemented using the OpenGL Shading Language (GLSL) on a GeForce 6600.

Cone Step Mapping

The Ray-Heightfield Intersection problem is as follows: given a ray starting at S_0, and traveling along the normalized vector dS, find the first point P where (S_0 + sc*dS) intersects the heightfield. This is done by varying “sc” (for “scalar”). Note that the following drawings are in 2D, but the concept works well with 3D cones because of their symmetry about the vertical (Z) axis.

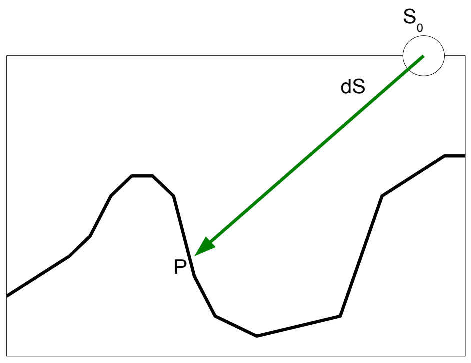


Illustration 1: Ray-Heightfield Intersection

In previous methods, “sc” was incremented by constant steps, or using a simple heuristic to predict step sizes, or found using a binary search, or through some combination of these. These methods (including Relief Mapping, Steep Parallax Mapping, Parallax Occlusion Mapping, and variants) faced the same fundamental problem, namely the trade-off between speed and guaranteed accuracy. For a given heightfield there is a Nyquist sampling frequency that will guarantee that you have found the first intersection, and this had to be tuned for each heightfield or errors could occur. Note that standard iterative techniques such as Newton's Method fail because the heightfield is so often discontinuous in the 1st derivative.

What is needed, then, is a method of predicting the appropriate step size for any given texel, and heading in any given direction. This has been previously solved using “space leaping” or “sphere stepping”, but such methods require 3D textures (voxels, not texels), thus using an order of magnitude more memory then standard 2D textures. Not only are the memory requirements higher for 3D textures, but most (all?) existing game art pipelines are designed for 2D. The 3D versions can do things that 2D can't, however (for example, disconnected blobs or geometry that curves back over itself).

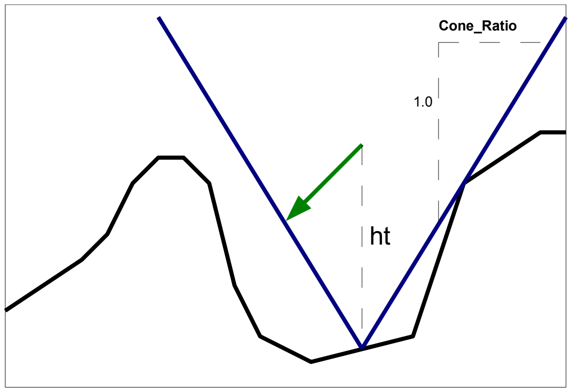


Illustration 2: Cone Bounding Volume

Cone Step Mapping solves this problem by giving each texel its own bounding volume in the shape of - you guessed it - a cone [8^). Thus for each texel, there is a cone sitting on it, point down. The tip of the cone is on the surface of the heightfield at that texel, and the only other parameter needed to fully define the cone is the Cone_Ratio. The volume is, in fact, a bounding volume for open space. So at any point (distance “ht” above the heightfield) the cone of the texel directly beneath describes how large a step can safely be taken (in any direction). Note that the cone volume is conservative, that is, usually only two points (the cone's tip and one other) will actually be touching the cone. All other points are outside the cone.

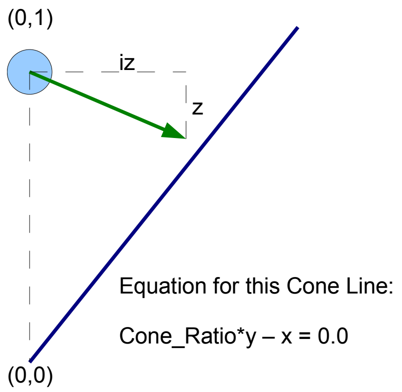


Illustration 3: Solving the Cone

Given the cone ratio, and the height above the heightfield, compute the next step size (“dsc”). Taking the 3D problem into 2D, break the dS vector into two components: “z” is the vertical component, and “iz” is the horizontal component (note that because dS is normalized, iz = sqrt (1.0-z*z), by definition). Find a value “Step_Ratio”, such that (0,1)+Step_Ratio*(iz,-z) is on the cone line. As you can see from illustration 3, the height is normalized to 1.0, so the step “dsc” must be post-multiplied by height. Here's the math:

Cone Line Equation:

$$\text{Cone\_Ratio}*y - x = 0$$

Position after Step_Ratio:

$$x = 0 + \text{Step\_Ratio}*iz$$

$$y = 1 - \text{Step\_Ratio}*z$$

Plugging that in:

$$\text{Cone\_Ratio}*(1 - \text{Step\_Ratio}*z) - \text{Step\_Ratio}*iz = 0$$

Solve for Step_Ratio:

$$\text{Step\_Ratio} = \text{Cone\_Ratio} / (iz - z*\text{Cone\_Ratio})$$

Rearrange (slightly faster):

$$\text{Step\_Ratio} = 1 / (iz/\text{Cone\_Ratio} - z)$$

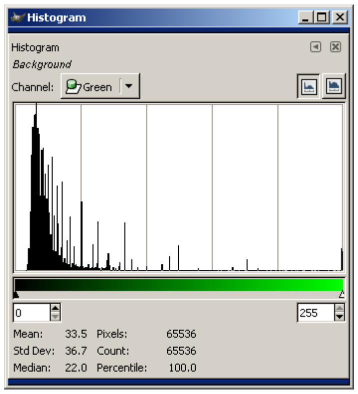
Scale by the original height:

$$dsc = ht * \text{Step\_Ratio}$$

And to take a step:

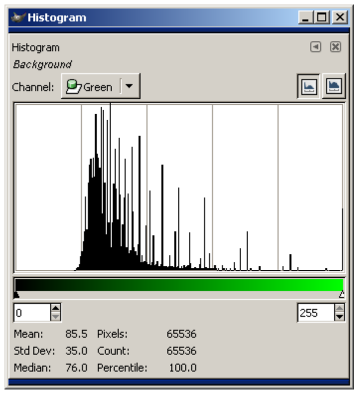
$$sc \mathrel{+}= dsc$$

That's it for the math. Please note that Cone_Ratio is always in the range (0,1), partly because that is so extremely convenient as a texture lookup, and partly because hardly any pixels ever had a Cone_Ratio of > 1. Also, because of the shape of the cone, any rays pointing straight down will arrive in a single iteration, while nearly horizontal rays have to go through many more iterations. Another note is that by looking at the histogram of the Cone_Ratios, it is obvious that most of the values were clustered at the low end. So to increase the spread (and thus accuracy, since the texture typically only has 8 bits per channel) the square root of Cone_Ratio is stored in the green channel. The added accuracy gained by using the sqrt makes up for the minor cost of the extra multiply in the shader.



```
Mean:     33.5  Pixels:      65536
Std Dev:  36.7  Count:       65536
Median:   22.0  Percentile:  100.0
```

Illustration 4: Green = Cone_Ratio



```
Mean:     85.5  Pixels:      65536
Std Dev:  35.0  Count:       65536
Median:   76.0  Percentile:  100.0
```

Illustration 5: Green = sqrt (Cone_Ratio)

This algorithm was implemented three different ways: the _exact version has to use branching (PS3.0 card required). The _loop version was given a uniform variable “conesteps”, and it would execute that number of iterations. The _fixed version had the loop unrolled to do a fixed number of iterations, but there is no branching so it was extremely fast.

The _exact version is the slowest of the CSM variants. It is approximately the same speed as Relief Mapping, but more accurate. And it does not need to be tuned per heightfield (though it can be sped up somewhat at the expense of guaranteed accuracy). At each iteration “dsc” is computed as shown previously, but an additional step of 1.0/texture_size is taken. This speeds up the process and simultaneously allows the ray to actually pierce the heightfield (if you only ever take steps defined by the cone, you can approach the surface asymptotically, but never get there, give or take machine error).

The _loop version still uses dynamic branching, so it is slower than _fixed, but faster than _exact (depending on the number of conesteps, obviously). This version is nice for its ability to adapt to any desired Level Of Detail. Smaller/further objects can be processed with fewer iterations. This allows the engine to determine how much accuracy (and therefore time) should be given to each object.

The _fixed version is the fastest of all, with no dynamic branching. With 10 unrolled iterations this version performs significantly faster than either of the other two versions, but of course lacks the ability to be tuned after compilation. It can still look anywhere from very good to merely OK, depending on the heightfield (e.g. a fur heightfield looks bad at 10 iterations). The example below shows the difference between the three versions of CSM (look for darkening on silhouettes).

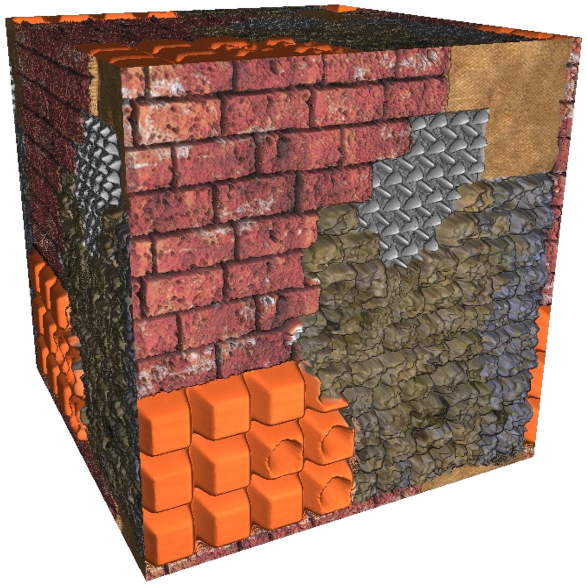


Illustration 6: _fixed: 10 steps

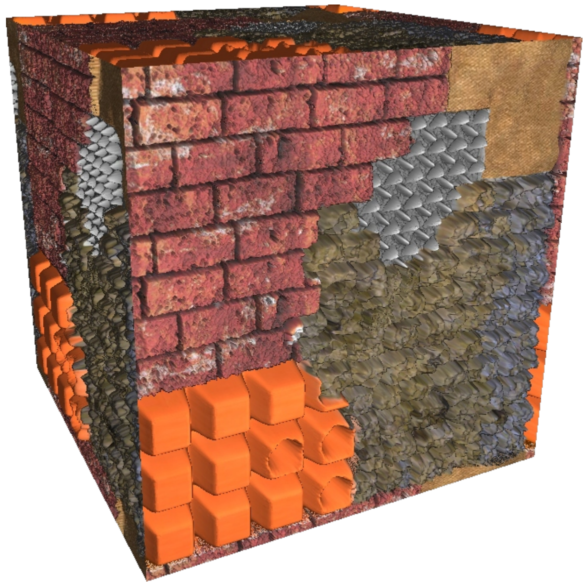


Illustration 7: _loop: 20 steps

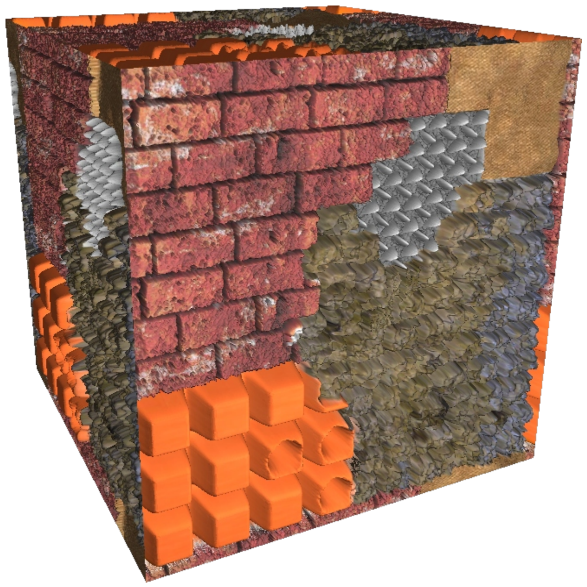


Illustration 8: _exact: variable steps

See Appendix A for the source code to the GLSL fragment shader.

The Preprocessor

The preprocessor's main job is to find the Cone_Ratio for each texel. For each texel it searches outward in expanding squares until it find the steepest possible cone. The preprocessor also computes the slope of the heightfield at each point. It uses the central-difference method for all interior pixels, forward- or backward-difference for the edges as appropriate. The derivative is stored as a number (0,1), which actually maps to (-Texture_Size/2, Texture_Size/2) inside the shader, recovering slope information. This method is actually more versatile then storing the normal vectors explicitly as you can scale the depth of the texture and it will automatically adjust the normals (note that the surface normal = normalize (-df/dx*depth, -df/dy*depth, 1.0)). The output of the preprocessor looks like this:

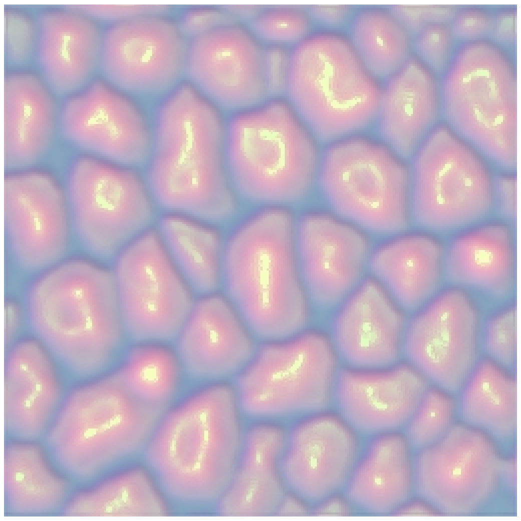


Illustration 9: RGBA Cone Step Map

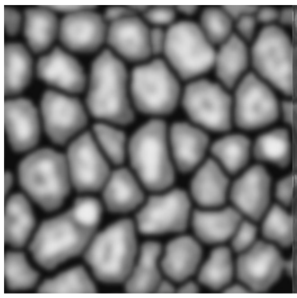


Illustration 10: Red = Height

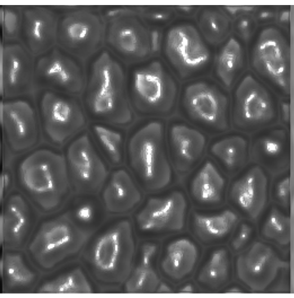


Illustration 11: Green = sqrt(Cone_Ratio)

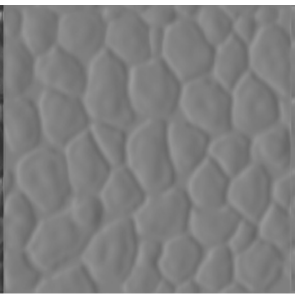


Illustration 12: Blue = dF/dx

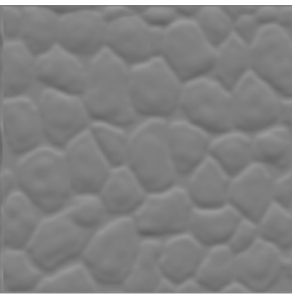


Illustration 13: Alpha = dF/dy

Some tricks for optimizing the processing of the heightfield include computing min_ratio_squared, and only performing one sqrt per pixel. Also, the expanding squares are used because the search can short circuit when the current Cone_Ratio cannot be outdone by any subsequent square (this is a function of the current min_ratio_squared and the value 1.0-height (the overhead or ceiling)). This sped up the preprocessor tremendously. See Appendix B for the C++ source code for the preprocessor.

Future Work

In my opinion, the most outstanding improvement to be made to the Cone Step Mapping method has to do with taking the local slope of the heightfield into account. That information is already available for the purpose of normal vector calculations, so there is no computation cost (it is sampled at the same time the height and cone information is). Imagine a point on a steep surface...generally the “uphill” side will determine the minimum (bounding) step ratio, while in the downhill direction will typically have a larger step ratio. Taking this asymmetry into account could significantly speed up the method while keeping it accurate.

Another simple improvement would be to move from a circular cone to an elliptical one. Thus X and Y information could be encoded separately. There are two detractors for this: you need one more value per texel, so either you include another texture fetch or remove access to the slope information. Also, the shader would be a bit more complex (intersecting an ellipse instead of a circle).

Adding self shadowing to the method would be quite simple. The algorithm would be essentially identical to how it is done in relief mapping or steep parallax mapping: start another ray at the intersection of the light-ray and the polygon. If this shadow ray does not reach the original intersection point then it is in shadow.

Finally, it would be nice to modify the silhouette of the CSM polygons, much like the Curved Relief Mapping of Policarpo and Oliveira. A direct port of their method would not work, as the step cone would be affected by (arbitrary) modification to the heightfield. If there was a simple way to modify the cone values (accurately) as the underlying curvature of the polygon changes, then it would work. More research needs to be done in this area.

Conclusion

The _exact version of Cone Step Mapping performs as well (or as poorly [8^) as standard relief mapping, but the result is more accurate, and there is no need to hand tune algorithm parameters (such as “number of steps”). For these reasons it is an ideal candidate for applications requiring precise surface rendering, or for games whose target hardware is a few years in the future.

The _fixed version of Cone Step Mapping performs much faster than the _exact version, but with reduced quality. The actual quality difference is dependent on many factors, such as the type of heightfield, the (perceived) depth of the heightfield, and the number of iterations. It is still slower than parallax mapping, but looks better.

The Cone Step Mapping method is in its early stages, however it is already a fast and robust method. There are still improvements that can be made. Programmers wishing more speed at the cost of stability can tweak the algorithm for better performance.

Appendix A: Fragment Shader Source

```
uniform vec4 ambient;
uniform vec4 diffuse;
uniform float depth;
uniform float texsize;
uniform int conesteps;

varying vec2 texCoord;
varying vec3 eyeSpaceVert;
varying vec3 eyeSpaceTangent;
varying vec3 eyeSpaceBinormal;
varying vec3 eyeSpaceNormal;
varying vec3 eyeSpaceLight;

uniform sampler2D stepmap;
uniform sampler2D texmap;

float intersect_cone_fixed(in vec2 dp,in vec3 ds);
float intersect_cone_loop(in vec2 dp,in vec3 ds);
float intersect_cone_exact(in vec2 dp,in vec3 ds);

void main(void)
{
    vec4 t,c;
    vec3 p,v,l,s;
    vec2 uv;
    float d,a;

    // ray intersect in view direction
    p  = eyeSpaceVert;
    v  = normalize(p);
    a  = dot(eyeSpaceNormal,-v)/depth;
    s = normalize(vec3(dot(v,eyeSpaceTangent),dot(v,eyeSpaceBinormal),a));

    // pick _one_ of the following variations
    //d = intersect_cone_fixed(texCoord,s);
    //d = intersect_cone_loop(texCoord,s);
    d = intersect_cone_exact(texCoord,s);

    // get rm and color texture points
    uv=texCoord+s.xy*d;
    c=texture2D(texmap,uv);

    // expand normal from normal map in local polygon space
    // blue = df/dx
    // alpha = df/dy
    // note: I _need_ the texture size to scale the normal properly!
    t=texture2D(stepmap,uv);
    t.xy=t.ba-0.5;
    t.x = -t.x * depth * texsize;
    t.y = -t.y * depth * texsize;
    t.z = 1.0;
    t.w = 0.0;
    t.xyz=normalize(t.x*eyeSpaceTangent+t.y*eyeSpaceBinormal+t.z*eyeSpaceNormal);

    // compute light direction
    p += v*d*a;
    l=normalize(p-eyeSpaceLight.xyz);

    gl_FragColor = vec4(
                       ambient.xyz*c.xyz+
                       c.xyz*diffuse.xyz*max(0.0,dot(-l,t.xyz)),
                       1.0);
}

// slowest, but best quality
float intersect_cone_exact(in vec2 dp, in vec3 ds)
{
    // minimum feature size parameter
    float w = 1.0 / texsize;
    // the "not Z" component of the direction vector
    // (requires that the vector ds was normalized!)
    float iz = sqrt(1.0-ds.z*ds.z);
    // my starting location (is at z=1,
    // and moving down so I don't have
    // to invert height maps)
    // texture lookup
    vec4 t;
    // scaling distance along vector ds
    float sc=0.0;

    // the ds.z component is positive!
    // (headed the wrong way, since
    // I'm using heightmaps)

    // find the starting location and height
    t=texture2D(stepmap,dp);
    while (1.0-ds.z*sc > t.r)
    {
        // right, I need to take one step.
        // I use the current height above the texture,
        // and the information about the cone-ratio
        // to size a single step.  So it is fast and
        // precise!  (like a coneified version of
        // "space leaping", but adapted from voxels)
        sc += w + (1.0-ds.z*sc-t.r) / (ds.z + iz/(t.g*t.g));

        // find the new location and height
        t=texture2D(stepmap,dp+ds.xy*sc);
    }
    // back off one step
    sc -= w;

    // return the vector length needed to hit the height-map
    return (sc);
}

// the super fast version
// (change number of iterations at run time)
float intersect_cone_fixed(in vec2 dp, in vec3 ds)
{
    // the "not Z" component of the direction vector
    // (requires that the vector ds was normalized!)
    float iz = sqrt(1.0-ds.z*ds.z);
    // my starting location (is at z=1,
    // and moving down so I don't have
    // to invert height maps)
    // texture lookup (and initialized to starting location)
    vec4 t;
    // scaling distance along vector ds
    float sc;

    // the ds.z component is positive!
    // (headed the wrong way, since
    // I'm using heightmaps)

    // find the initial location and height
    t=texture2D(stepmap,dp);
    // right, I need to take one step.
    // I use the current height above the texture,
    // and the information about the cone-ratio
    // to size a single step.  So it is fast and
    // precise!  (like a coneified version of
    // "space leaping", but adapted from voxels)
    sc = (1.0-ds.z*sc-t.r) / (ds.z + iz/(t.g*t.g));

    // and repeat a few (4x) times
    t=texture2D(stepmap,dp+ds.xy*sc);
    sc += (1.0-ds.z*sc-t.r) / (ds.z + iz/(t.g*t.g));
    t=texture2D(stepmap,dp+ds.xy*sc);
    sc += (1.0-ds.z*sc-t.r) / (ds.z + iz/(t.g*t.g));
    t=texture2D(stepmap,dp+ds.xy*sc);
    sc += (1.0-ds.z*sc-t.r) / (ds.z + iz/(t.g*t.g));
    t=texture2D(stepmap,dp+ds.xy*sc);
    sc += (1.0-ds.z*sc-t.r) / (ds.z + iz/(t.g*t.g));
    // and another five!
    t=texture2D(stepmap,dp+ds.xy*sc);
    sc += (1.0-ds.z*sc-t.r) / (ds.z + iz/(t.g*t.g));
    t=texture2D(stepmap,dp+ds.xy*sc);
    sc += (1.0-ds.z*sc-t.r) / (ds.z + iz/(t.g*t.g));
    t=texture2D(stepmap,dp+ds.xy*sc);
    sc += (1.0-ds.z*sc-t.r) / (ds.z + iz/(t.g*t.g));
    t=texture2D(stepmap,dp+ds.xy*sc);
    sc += (1.0-ds.z*sc-t.r) / (ds.z + iz/(t.g*t.g));
    t=texture2D(stepmap,dp+ds.xy*sc);
    sc += (1.0-ds.z*sc-t.r) / (ds.z + iz/(t.g*t.g));

    // return the vector length needed to hit the height-map
    return (sc);
}

// pretty fast version
// (and you can do LOD by changing "conesteps" based on size/distance, etc.)
float intersect_cone_loop(in vec2 dp, in vec3 ds)
{
    // the "not Z" component of the direction vector
    // (requires that the vector ds was normalized!)
    float iz = sqrt(1.0-ds.z*ds.z);
    // my starting location (is at z=1,
    // and moving down so I don't have
    // to invert height maps)
    // texture lookup (and initialized to starting location)
    vec4 t;
    // scaling distance along vector ds
    float sc=0.0;

    //t=texture2D(stepmap,dp);
    //return (max(0.0,-(t.b-0.5)*ds.x-(t.a-0.5)*ds.y));

    // the ds.z component is positive!
    // (headed the wrong way, since
    // I'm using heightmaps)

    // adaptive (same speed as it averages the same # steps)
    //for (int i = int(float(conesteps)*(0.5+iz)); i > 0; --i)
    // fixed
    for (int i = conesteps; i > 0; --i)
    {
        // find the new location and height
        t=texture2D(stepmap,dp+ds.xy*sc);
        // right, I need to take one step.
        // I use the current height above the texture,
        // and the information about the cone-ratio
        // to size a single step.  So it is fast and
        // precise!  (like a coneified version of
        // "space leaping", but adapted from voxels)
        sc += (1.0-ds.z*sc-t.r) / (ds.z + iz/(t.g*t.g));
    }

    // return the vector length needed to hit the height-map
    return (sc);
}
```

Appendix B: Preprocessor Source

```
// Theirs
#include <iostream>
#include <fstream>
#include <cstdlib>
#include <cmath>
#include <vector>

// in-between
#include <corona.h>

using namespace std;

/*
 basically, 99% of all pixels will fall in under 2.0
  (most of the time, on the heightmaps I've tested)
 the question:
  Is reduced resolution worth missing
  the speedup of the slow ones?
*/
const float max_ratio = 1.0;

/*
    And for the cone version, how tolerant am I?
    (and should it be a ratio, tolerance*r^2, or flat?)
*/
const float cone_tol2 = 4.0 / (255.0 * 255.0);

// Do I want the textures to be computed as tileable?
bool x_tileable = true;
bool y_tileable = true;

int main(int argc, char *argv[])
{
    char OutName[1024];
    int FileCounter;
    long TheSize, ScanWidth;
    long width, height;
    float iwidth, iheight;
    long chans;
    corona::Image *image, *outimage;
    unsigned char *Data;
    long tin;
    int wProgress;
    float really_max = 1.0;

    cout << "********** Height Map Processor **********" << endl << endl;

    // Did I get a file name?
    if (argc < 2)
    {
        // Oops, no file to convert
        cout << "usage: HeightProc input_file" << endl << endl;
        system ("pause");
        return (0);
    }

    for (FileCounter = 1; FileCounter < argc; ++FileCounter)
    {
        cout << "Converting file #" << FileCounter << endl;
        // OK, open the image file
        image = corona::OpenImage (argv[FileCounter], corona::PF_R8G8B8A8);
        if (!image)
        {
            cout << "I could not open " << argv[FileCounter] << endl << endl;
            system ("pause");
            return (0);
        }

        cout << "Loading " << argv[FileCounter] << ":" << endl;

        width = image->getWidth ();
```

```
height = image->getHeight ();
chans = 4; // forced for now (by corona)
ScanWidth = chans * width;
TheSize = ScanWidth * height;
Data = (unsigned char *)(image->getPixels());

// invert this (used to convert depth-map to height-map)
if (false)
{
    for (int px = 0; px < width*height; ++px)
        Data[px*chans] = 255 - Data[px*chans];
}

iheight = 1.0 / height;
iwidth = 1.0 / width;

wProgress = width / 50;

cout << endl << "The image " << argv[FileCounter] << " has been loaded." << endl;
cout << " size: " << width << " x " << height << " by " << 32 << " bits" << endl;
cout << endl << "Computing:" << endl;

//  warning message if the texture is not square
if (width != height)
{
    cout << endl << "Warning: The image is not square!  Results not guaranteed!" << endl;
    system ("pause");
}

// Redo the best, and save it
// (only writing formats supported: PNG, TGA)
//  ((and PNG is compressed and has alpha!!))
strcpy (OutName, argv[FileCounter]);
strcat (OutName, "-step.png");
tin = clock ();

//  pre-processing: compute derivatives
cout << "Calculating derivatives [";
for (int y = 0; y < height; ++y)
{
    //  progress report: works great...if it's square!
    if (y % wProgress == 0)
        cout << ".";
    for (int x = 0; x < width; ++x)
    {
        int der;

        //  Blue is the slope in x
        if (x == 0)
            der = (Data[y*ScanWidth + chans*(x+1)] - Data[y*ScanWidth + chans*(x)]) / 2;
        else if (x == width - 1)
            der = (Data[y*ScanWidth + chans*(x)] - Data[y*ScanWidth + chans*(x-1)]) / 2;
        else
            der = Data[y*ScanWidth + chans*(x+1)] - Data[y*ScanWidth + chans*(x-1)];
        Data[y*ScanWidth + chans*x + 2] = 127 + der / 2;

        //  Alpha is the slope in y
        if (y == 0)
            der = (Data[(y+1)*ScanWidth + chans*x] - Data[(y)*ScanWidth + chans*x]) / 2;
        else if (y == height - 1)
            der = (Data[(y)*ScanWidth + chans*x] - Data[(y-1)*ScanWidth + chans*x]) / 2;
        else
            der = (Data[(y+1)*ScanWidth + chans*x] - Data[(y-1)*ScanWidth + chans*x]);
        //  And the sign of Y will be reversed in OpenGL
        Data[y*ScanWidth + chans*x + 3] = 127 - der / 2;
    }
}
cout << "]" << endl;

//  OK, do the processing
for (int y = 0; y < height; ++y)
{
    cout << "img " << (argc - FileCounter) << ": line " << (height - y) << " [";
    for (int x = 0; x < width; ++x)
```

```
{
    float min_ratio2, actual_ratio;
    int x1, x2, y1, y2;
    float ht, dhdx, dhdy, r2, h2;

    if ((x % wProgress) == 0)
        cout << ".";

    //  set up some initial values
    // (note I'm using ratio squared throughout,
    // and taking sqrt at the end...faster)
    min_ratio2 = max_ratio * max_ratio;

    //  information about this center point
    ht = Data[y*ScanWidth + chans*x] / 255.0;
    dhdx = +(Data[y*ScanWidth + chans*x + 2] / 255.0 - 0.5) * width;
    dhdy = -(Data[y*ScanWidth + chans*x + 3] / 255.0 - 0.5) * height;

    // scan in outwardly expanding blocks
    // (so I can stop if I reach my minimum ratio)
    for (int rad = 1;
            (rad*rad <= 1.1*1.1*(1.0-ht)*(1.0-ht)*min_ratio2*width*height)
            && (rad <= 1.1*(1.0-ht)*width) && (rad <= 1.1*(1.0-ht)*height);
            ++rad)
    {
        // ok, for each of these lines...

        // West
        x1 = x - rad;
        while (x_tileable && (x1 < 0))
            x1 += width;
        if (x1 >= 0)
        {
            float delx = -rad*iwidth;
            // y limits
            // (+- 1 because I'll cover the corners in the X-run)
            y1 = y - rad + 1;
            if (y1 < 0)
                y1 = 0;
            y2 = y + rad - 1;
            if (y2 >= height)
                y2 = height - 1;

            // and check the line
            for (int dy = y1; dy <= y2; ++dy)
            {
                float dely = (dy-y)*iheight;
                r2 = delx*delx + dely*dely;
                h2 = Data[dy*ScanWidth + chans*x1] / 255.0 - ht;
                if ((h2 > 0.0) && (h2*h2 * min_ratio2> r2))
                {
                    //  this is the new (lowest) value
                    min_ratio2 = r2 / (h2 * h2);
                }
            }
        }

        // East
        x2 = x + rad;
        while (x_tileable && (x2 >= width))
            x2 -= width;
        if (x2 < width)
        {
            float delx = rad*iwidth;
            // y limits
            // (+- 1 because I'll cover the corners in the X-run)
            y1 = y - rad + 1;
            if (y1 < 0)
                y1 = 0;
            y2 = y + rad - 1;
            if (y2 >= height)
                y2 = height - 1;

            // and check the line
    for (int dy = y1; dy <= y2; ++dy)
    {
        float dely = (dy-y)*iheight;
        r2 = delx*delx + dely*dely;
        h2 = Data[dy*ScanWidth + chans*x2] / 255.0 - ht;
        if ((h2 > 0.0) && (h2*h2 * min_ratio2> r2))
        {
            //  this is the new (lowest) value
            min_ratio2 = r2 / (h2 * h2);
        }
    }
}

// North
y1 = y - rad;
while (y_tileable && (y1 < 0))
    y1 += height;
if (y1 >= 0)
{
    float dely = -rad*iheight;
    // y limits
    // (+- 1 because I'll cover the corners in the X-run)
    x1 = x - rad;
    if (x1 < 0)
        x1 = 0;
    x2 = x + rad;
    if (x2 >= width)
        x2 = width - 1;

    // and check the line
    for (int dx = x1; dx <= x2; ++dx)
    {
        float delx = (dx-x)*iwidth;
        r2 = delx*delx + dely*dely;
        h2 = Data[y1*ScanWidth + chans*dx] / 255.0 - ht;
        if ((h2 > 0.0) && (h2*h2 * min_ratio2> r2))
        {
            //  this is the new (lowest) value
            min_ratio2 = r2 / (h2 * h2);
        }
    }
}

// South
y2 = y + rad;
while (y_tileable && (y2 >= height))
    y2 -= height;
if (y2 < height)
{
    float dely = rad*iheight;
    // y limits
    // (+- 1 because I'll cover the corners in the X-run)
    x1 = x - rad;
    if (x1 < 0)
        x1 = 0;
    x2 = x + rad;
    if (x2 >= width)
        x2 = width - 1;

    // and check the line
    for (int dx = x1; dx <= x2; ++dx)
    {
        float delx = (dx-x)*iwidth;
        r2 = delx*delx + dely*dely;
        h2 = Data[y2*ScanWidth + chans*dx] / 255.0 - ht;
        if ((h2 > 0.0) && (h2*h2 * min_ratio2> r2))
        {
            //  this is the new (lowest) value
            min_ratio2 = r2 / (h2 * h2);
        }
    }
}

// done with the expanding loop
```

```
                }

                /********** CONE VERSION **********/
                //  actually I have the ratio squared.  Sqrt it
                actual_ratio = sqrt (min_ratio2);
                //  here I need to scale to 1.0
                actual_ratio /= max_ratio;
                //  most of the data is on the low end...sqrting again spreads it better
                //  (plus multiply is a cheap operation in shaders!)
                actual_ratio = sqrt (actual_ratio);

                //  Red stays height
                //  Blue remains the slope in x
                //  Alpha remains the slope in y

                //  but Green becomes Step-Cone-Ratio
                Data[y*ScanWidth + chans*x + 1] = static_cast<unsigned char>(255.0 * actual_ratio +
0.5);
                // but make sure it is > 0.0, since I divide by it in the shader
                if (Data[y*ScanWidth + chans*x + 1] < 1)
                    Data[y*ScanWidth + chans*x + 1] = 1;
            }

            cout << "]" << endl;
        }

        // end my timing after the computation phase
        tin = clock() - tin;
        cout << "Processed in " << tin * 0.001 << " seconds" << endl << endl;

        outimage = corona::CreateImage(width, height, corona::PF_R8G8B8A8, Data);
        if (outimage != NULL)
        {
            tin = clock();
            cout << "Saved: " << OutName << " " <<
            corona::SaveImage (OutName, corona::FF_PNG, outimage) << endl;
            tin = clock() - tin;
            cout << "(That took " << tin * 0.001 << " seconds)" << endl;
        }
        else
            cout << "Couldn't create the new image" << endl;

        // Report
        cout << endl << endl;

    } // and finish all file names passed in

    /*
    cout << "really_max = " << really_max << endl;
    system ("pause");
    //*/

    // And end it
    cout << endl << "Done processing the image(s)." << endl;
    //system("PAUSE");
    return (0);
}
```