Irregular Surfaces: a CPSC 5006 Term Project

Darren Janeczek
Laurentian University
128487

dc_janeczek@laurentian.ca

December 20, 2016

Abstract

The problem of generating realistic models of natural irregular surfaces is shared by a variety of applications including scientific modelling, art, animation, and physical simulation. When details are either unknown or completely irrelevant to the task, it is desirable to use a stochastic approach in generating realistic-looking surfaces based on a simple set of parameters. This work develops upon Fournier and Fussell's "Computer Rendering of Stochastic Models" [4], to create an irregular mesh surface from a flat square mesh.

1 Introduction

Fractals are a popular geometric phenomenon which have a wide variety of applications and provide a limitless source of wonder. This work focusses on the application of a procedure inspired by fractals for the purpose of generating irregularities in a surface as a stochastic model.

1.1 Polygon Subdivision

Fournier and Fussell provide an approach to Polygon Subdivision [4], where an initial square can be subdivided into smaller quadrilaterals by inserting points to divide the edges. These new points are displaced by a stochastic amount in a direction along the axis defined by the original polygon's normal vector. The criteria for the stochastic amount is strongly based on the classic fractal "midpoint displacement" methodology, except Fournier and Fussell's choice to use a static normal vector is a simplification which improves performance and ensures consistency of the mesh. The sequence of determining new points is sometimes called the "Diamond Square" algorithm [3], and is illustrated in Figure 1.

2 General Procedure

The procedure is implemented as a Python [2] function called *interpolate*, and makes heavy use of the NumPy [1] numerical processing library, and its high performance n-dimensional array [6]. This interpolate function is detailed in Section 2.1. An inner function called *displace* is called frequently during interpolation, and is detailed in Section 2.2

2.1 Interpolation Loop

The *interpolate* function will take the following parameters:

- $\vec{x}$ and $\vec{y}$, two 3D vectors, which define the initial quadrilateral:

  $$S = (0, \vec{x}, \vec{x}+\vec{y}, \vec{y})$$

  the initial normal unit vector:

  $$\hat{z} = \|\vec{x} \times \vec{y}\|$$

  and scale value:

  $$s = \frac{\|\vec{x}+\vec{y}\|}{2}$$

Figure 1: Demonstration of the "Diamond Square" sequence for two levels of iteration.

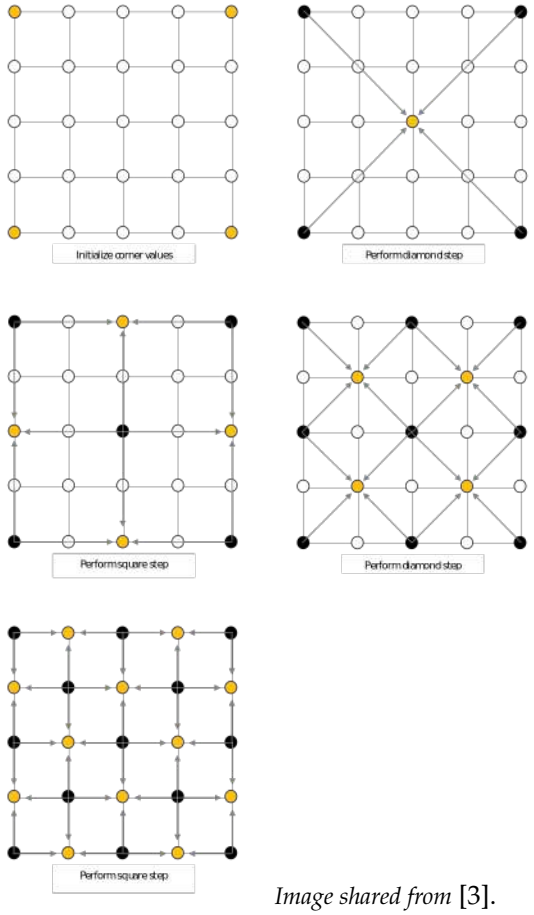


Image shared from [3].

- n, the number of levels of depth to iterate the algorithm (the default is 9); and
- H, the self-similarity parameter which determines the "fractal dimension" (since the 1D examples in [4] allow values between 0 and 1, this 2D application suggests values between 1 and 2);

1. A ratio variable r is calculated from H and a variable c (set to 2 for this study):

$$r = c^{-H}$$

2. A 2×2 matrix of 3D points, P is defined with the four points of S. The positions in the matrix correspond to the positions of the point.

3. An initial σ, standard deviation is calculated:

$$\sigma = sr$$

This σ value will be updated after each iteration as the level of detail increases.

4. The loop begins, and will repeat n times:

   (a) P is a $M \times N$ matrix of points. So, $\hat{P}$, a new $\hat{M} \times \hat{N}$ matrix of points allocated, where $\hat{M} = 2M - 1$ and $\hat{N} = 2N - 1$[1].

   (b) Initialize *original* values of P onto $\hat{P}$ (similar to "Initialize corner values" on Figure 1). $\hat{P}_{\hat{i}\hat{j}} \leftarrow P_{ij}$, where $\hat{i} = 2i - 1$ and $\hat{j} = 2j - 1$, for all i and j on P.

   (c) Perform the "diamond step", where the *middle* nodes are now set using the same $\hat{P}_{\hat{i}\hat{j}}$ values that were set above. Effectively, $\hat{P}_{ij}$, where $i = (1, 3, 5, \ldots, \hat{M})$ and $j = (1, 3, 5, \ldots, \hat{N})$[2] will be set to the result of a call to the *displace* function

[1] Both matrices are square, but it should be reasonable to adapt this algorithm to operate on non-square matrices.

[2] Values 3, 5, etc. not necessarily in sequence for early iterations, but the point is to show that the sequence increments by 2 starting at 1, and ending at the specified value. This pattern of illustrating sequences will be repeated movine forward.

using two pairs of points: first, the upper-left neighbour and lower-right neighbour, and second, the lower-left neighbour and the upper-right neighbour.

(d) The matrix $\hat{P}$ will be buffered on both sides to contain additional values to simplify the following "square step". This involves adding a fake point on the other side of each boundary point initialized in step (b). This point will continue the slope started from the recenty created *middle* point to its neighbouring boundary *original* point.

(e) Perform the "square step" by setting all unset values in $\hat{P}$ (not including the buffer around the matrix) to the result of *displace* using two pairs of points: first, the upper neighbour and the lower neighbour, and second, the left neighbour and the right neighbour (these neighbours *may* include points in the matrix buffer).

(f) Perform the "mean correction" step to repair the inner *original* points which are "sticking out". See Section 2.3 for more detail.

(g) Update the variables for the next iteration of detail:

$$\sigma \leftarrow \sigma r$$

$$P \leftarrow \hat{P}$$

(h) Continue to the next iteration until n iterations have concluded[3].

2.2 Dual Midpoint Displacement

Instead of restricting the displacement to a single normal vector as done by Fournier and Fussell [4], this implementation seeks to allow more variety in interpolation displacement. Each interpolated point will be the mean of two midpoint displacement procedures, based on two pairs of opposing points, where the interpolated point is initially considered between both pairs before it is displaced.

Considering one pair of points at a time, the two opposing points are P_1 and P_2. A displacement direction unit vector $\hat{d}$ is calculated as the normal of the line from P_1 to P_2 parallel to $\hat{z}$. A sample from the normal distribution is randomly drawn, and multiplied by the current iteration standard deviation σ, and this $\hat{d}$ is multiplied by this randomly determined displacement magnitude to produce $\vec{d}$, the midpoint displacement of a point in the middle between P_1 and P_2.

Both displacements are considered together – a mean is taken, and added to the mean of *all* points. That is the result of this dual midpoint displacement.

2.3 Mean Correction

It was quickly discovered that the above methodology (without this correction) resulted in very interesting looking artifacts: very sharp peaks and pits. Sometimes smaller peaks and pits dotted the surface and the result was very unnatural. To investigate these artifacts, the use of the normal distribution was disabled, and instead the standard deviation was used to determine the displacement magnitude. By taking away the stochastic nature from the surface, a very fractal-looking pattern emerged.

To resolve this fractal pattern, which appeared to have an impact where the *original* points from P were copied onto $\hat{P}$ in the "Initialize Step", an additional process was added where all copies of the original points (except the 4 original corners defined as the first S quadrilateral) where set to new values after all points were interpolated. These would be set to the mean of its neighbours, using a pattern similar the the "Square Step", but with no displacement. The border "originals" would only be set to the mean of neighbours along the border, and corner points would remain the same.

[3] Notice that only one diamond and square step occur per iteration.

Illustrations of the interesting fractal artifacts are shown in Figure 4 under Section 3.

3 Results

From the matrix of points P, a triangular mesh is created, splitting each quadrilateral into two triangles. Each mesh is constructed and displayed using the Visualization Toolkit [5] (VTK) version 7.1.

The figures below illustrate examples of meshes obtained using different h parameters for $h = 1.8$, $h = 1.6$, $h = 1.4$, and $h = 1.2$. After resolving the issue with the fractal artifacts, the resulting meshes look like reasonable, realistic irresgular surfaces, ranging from tissue paper to hills to mountains.

4 Conclusion

This implementation successfully implemented a variant of Fournier and Fussell's methodology, ignoring some simplification, and some complexities. It is a reasonable start toward generating fractal surfaces, although some immediate improvements could be explored:

- Each displacement can use a single random sample if the cross product of the crossing lines from the pairs of points is used instead of taking the mean of two displacements (Originally, the displace function could optionally work on a single pair – but now it can be assumed that there are always two).
- More variation could be included in the displacement direction to make more interesting surfaces.

Future goals to work toward include establishing more known starting points and adding the ability to interpolate unknowns between and around knowns while still respecting knowns.

Figure 2: Fractal peak and pit artifacts which appeared before the 'initial points' were averaged by interpolated neighbours (left column: $h = 1.8$, right column $h = 1.2$; first row: randomized, second row: non-randomized; bottom: non-randomized $h = 1.2$ from above).

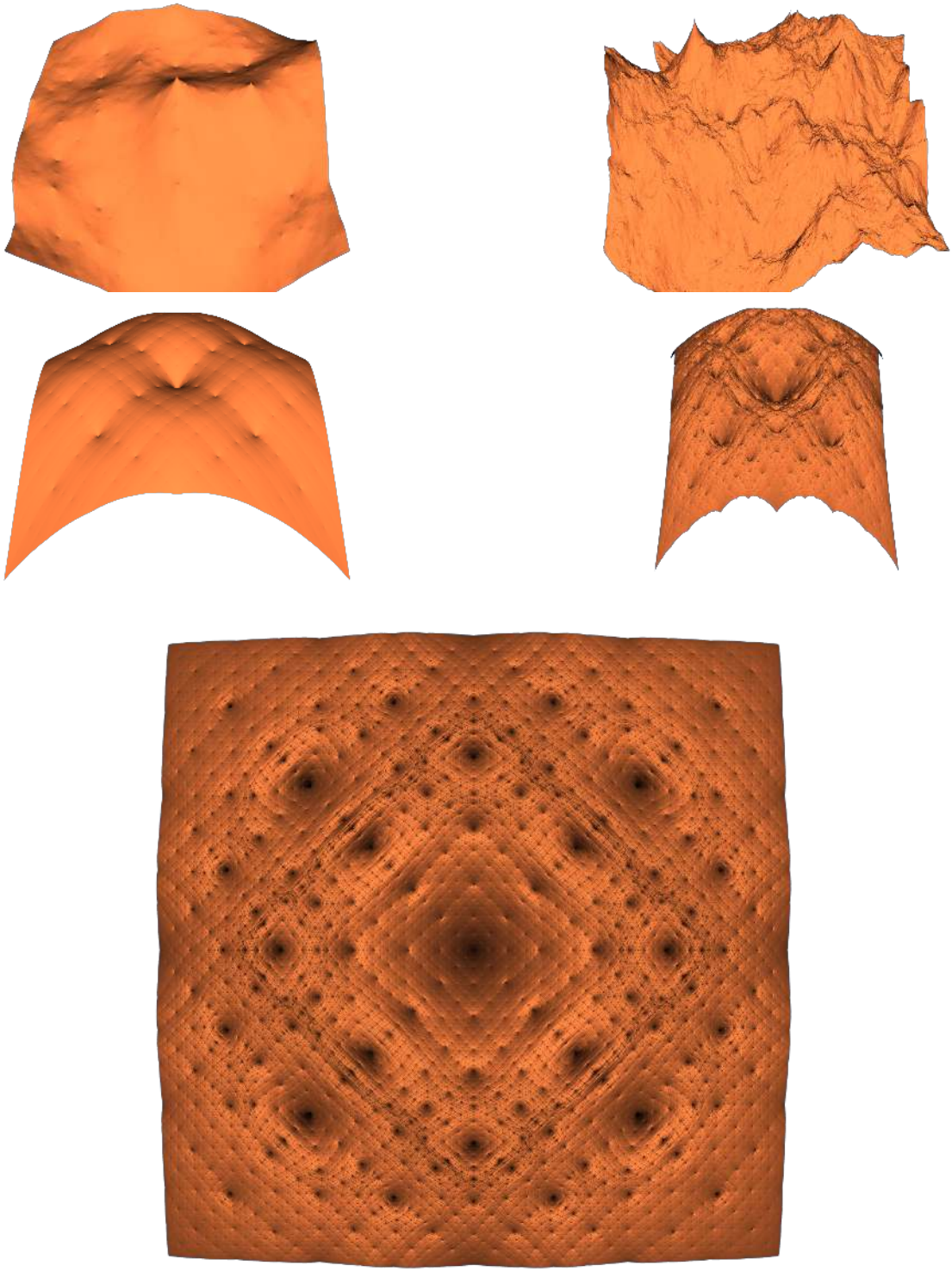


Figure 3: Example of fractal routine results for ten levels with $h = 1.8$

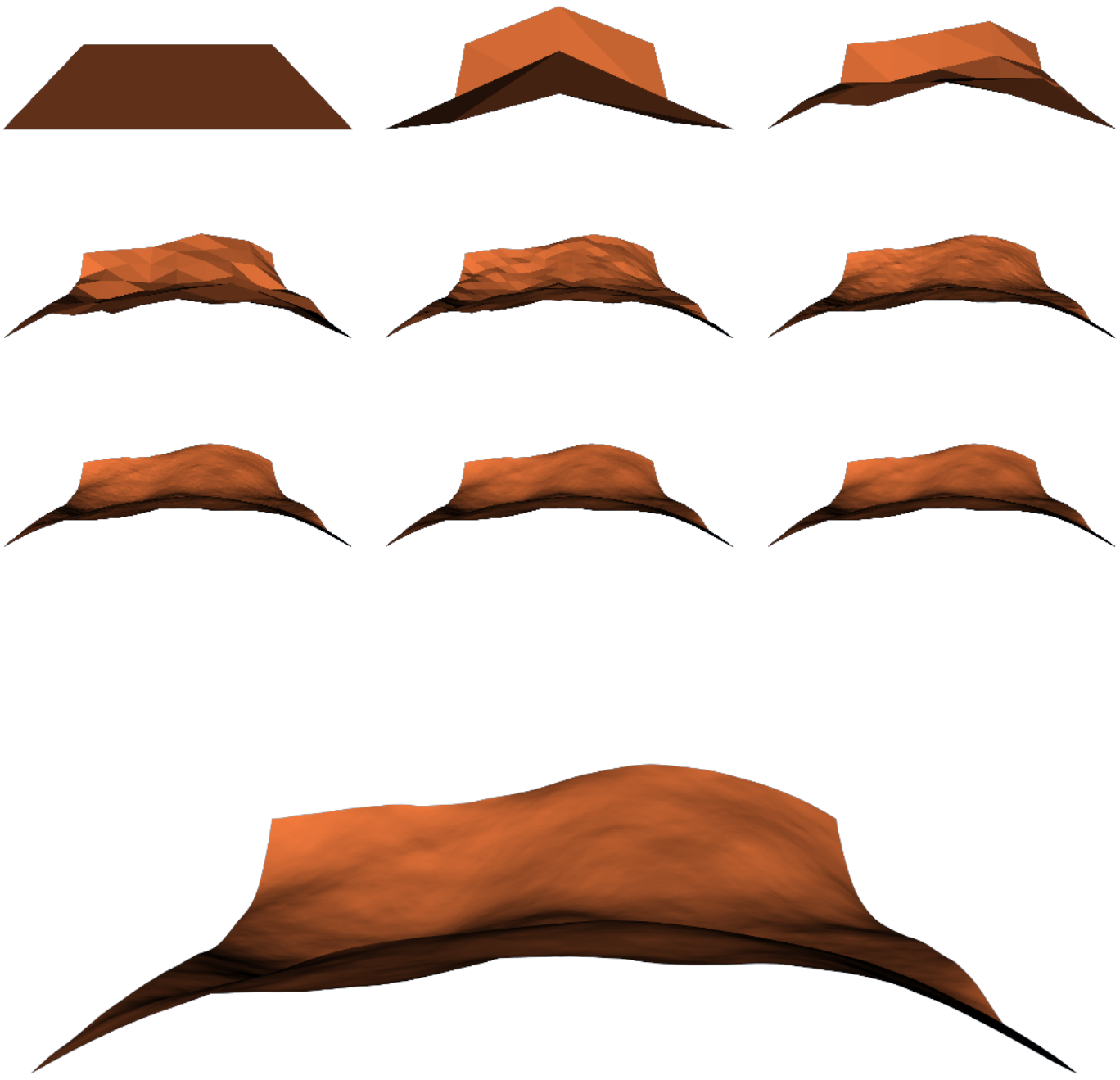


Figure 4: Example of fractal routine results for ten levels with $h = 1.6$

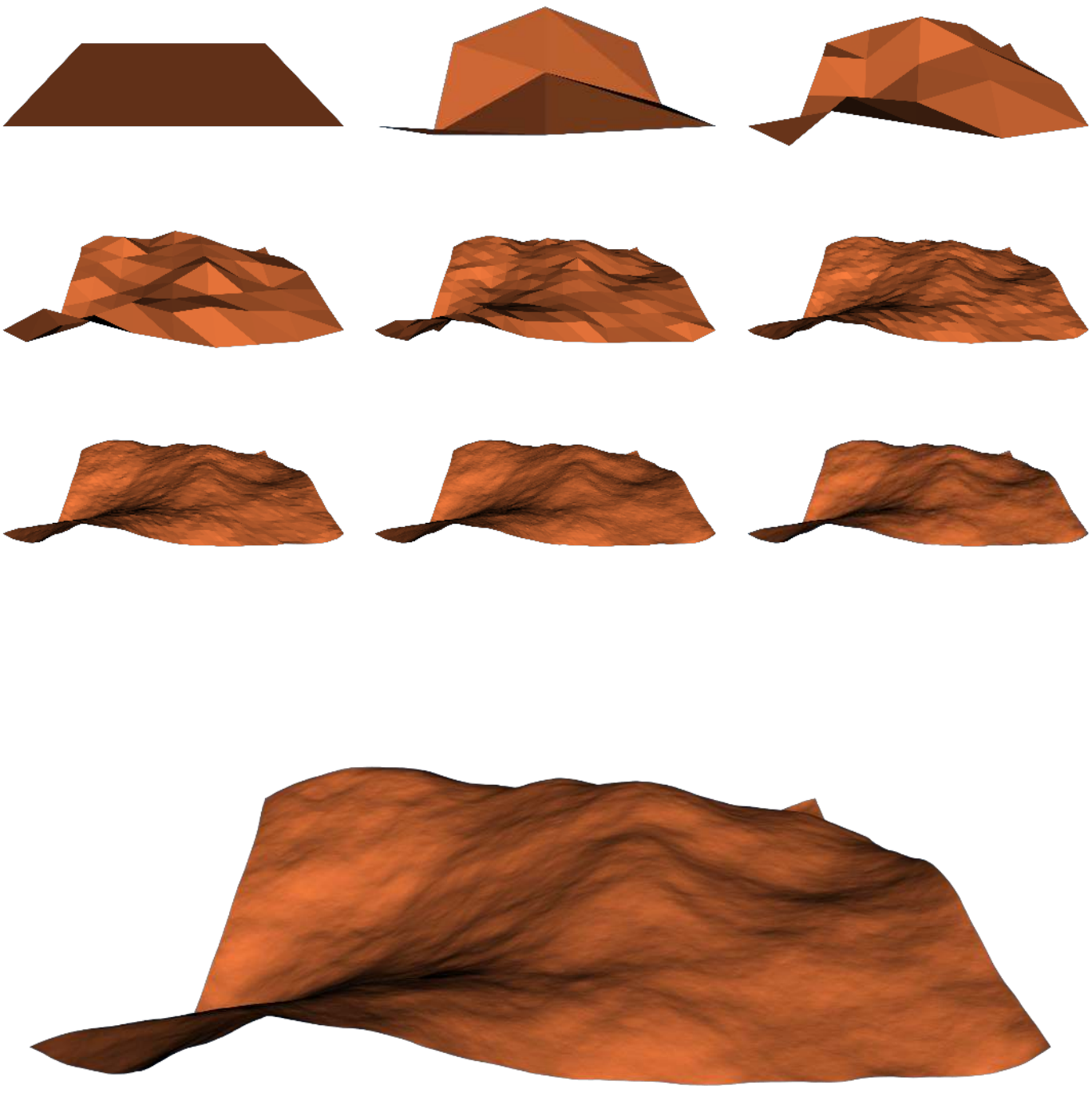


Figure 5: Example of fractal routine results for ten levels with $h = 1.4$

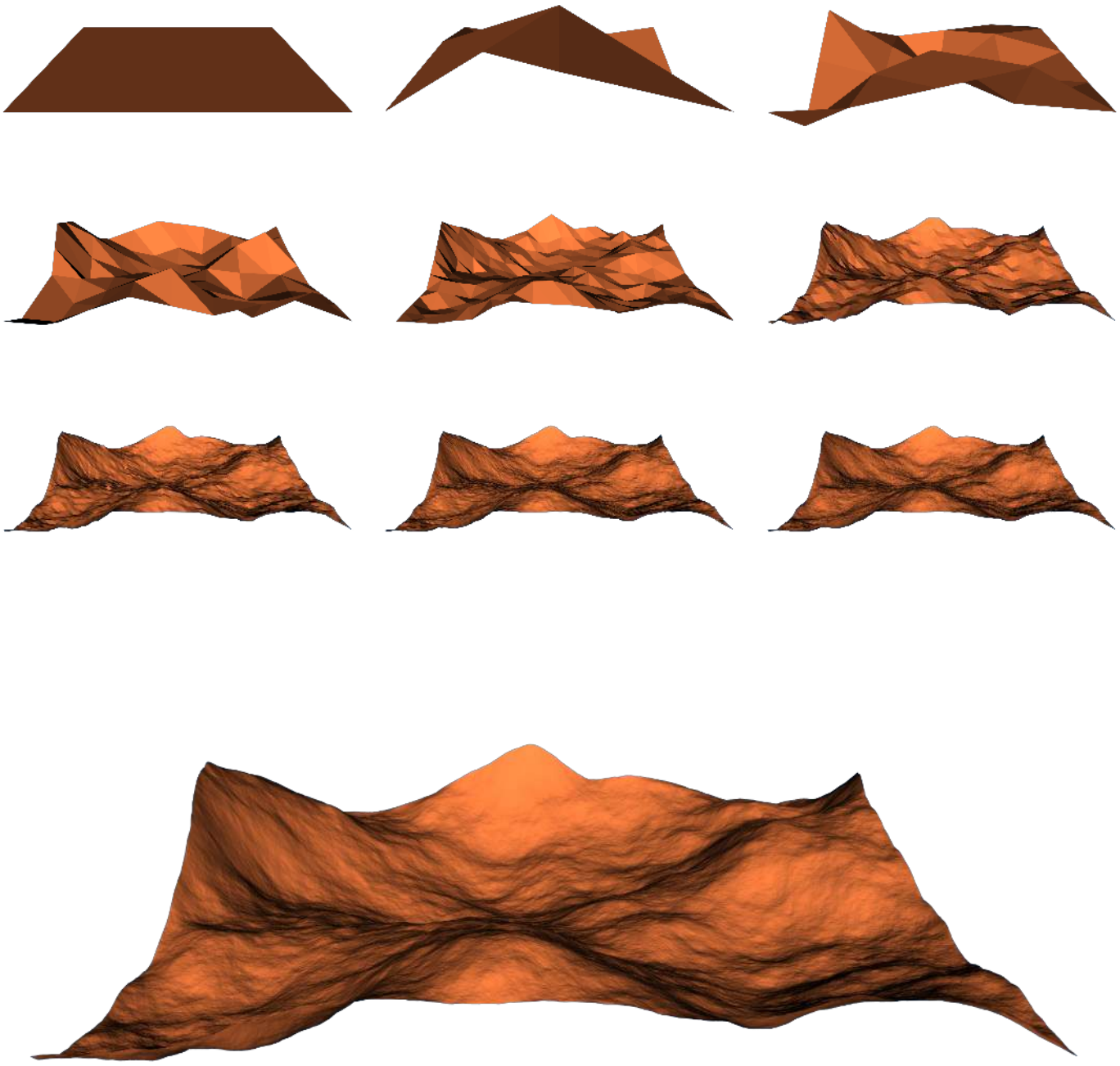
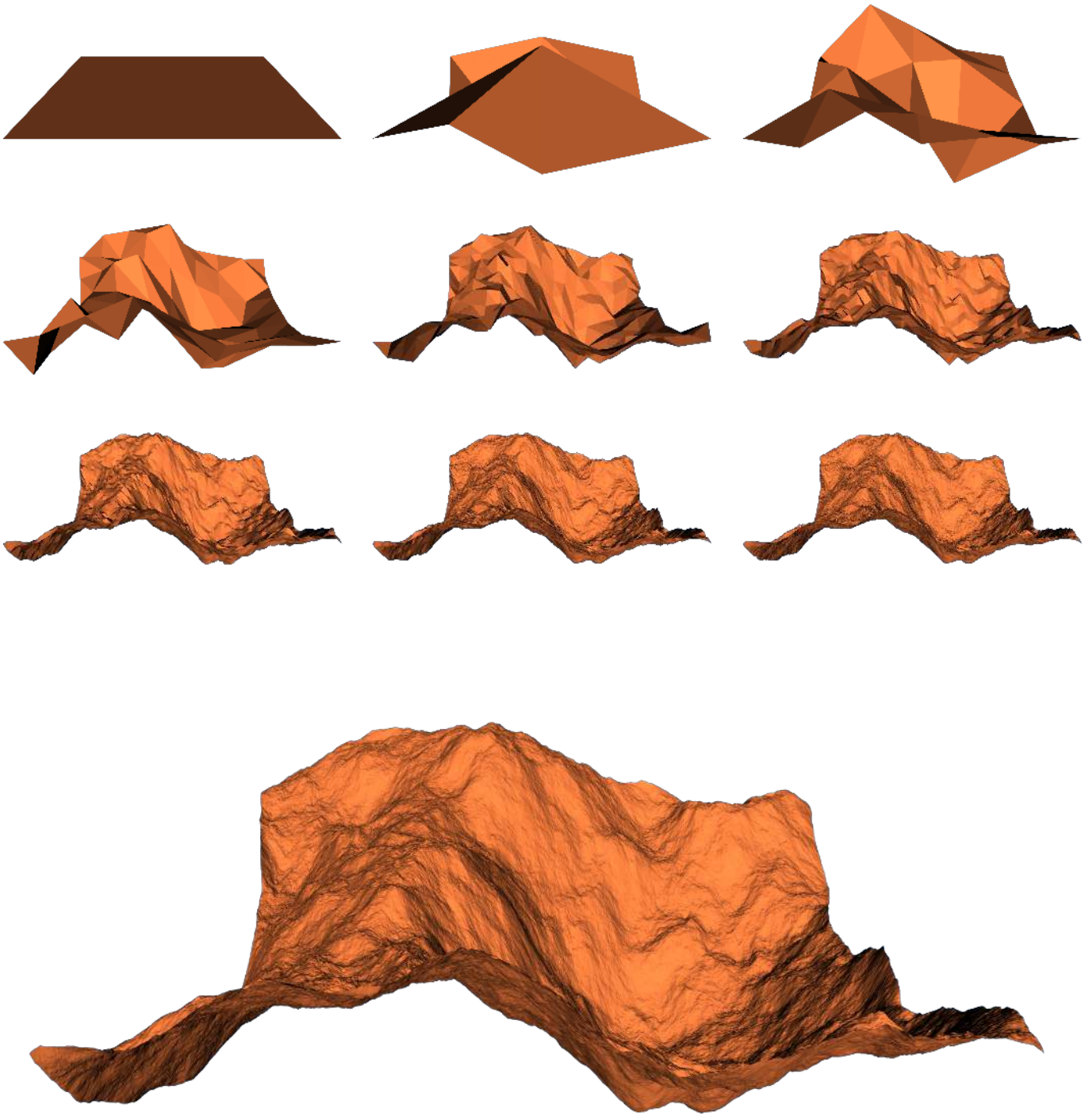


Figure 6: Example of fractal routine results for ten levels with $h = 1.2$



A Setting up VTK/Python environment

Download and install the *Standalone Python Interface (Installer)* for VTK (the version used is 7.1.0) from the website:

`http://www.vtk.org/download/`

VTK comes supplied with a version of Python as well as the NumPy numerical libraries. The interpreter's executable is `vtkpython` or `vtkpython` in windows.

On Mac or Linux systems, the available script file can be executed as follows.

```
$ ./vtkpython CPSC5006.py
```

On Windows:

```
> vtkpython CPSC5006.py
```

In both cases, this is assuming that the command-line environment is currently in the VTK's *bin* directory, and that the script file is also located in the same directory. If this is not the case, specific paths will need to be used.

B Code

All code is written in Python [2].

```
''' COSC5006.py
@author: Darren Janeczek
'''

from vtk import *
import numpy
from numpy import sin, cos, pi, arcsin, arccos
from numpy.linalg import norm

def interpolate(x_vector, y_vector, h, c=2.0, scale_factor=1.0,
                target_divisions=9,
                save_images=None, show_steps=False):

    zero = numpy.zeros(3, dtype=float)
    x = numpy.zeros(3, dtype=float)
    y = numpy.zeros(3, dtype=float)

    x[:] = x_vector
    y[:] = y_vector
    z = numpy.cross(x_vector, y_vector)
    z_unit = z / numpy.linalg.norm(z)

    del x_vector
    del y_vector

    P = numpy.zeros((2, 2, 3), dtype=float)
    P[0,0] = zero
    P[1,0] = x
    P[0,1] = y
    P[1,1] = x + y

    scale = scale_factor * (numpy.linalg.norm(x) +
                            numpy.linalg.norm(y)) / 2
    ratio = c ** (-h)

    def displace(pt_pairs, std):
        pts = []
        displacements = []

        for pt1, pt2 in pt_pairs:
            pts.append(pt1)
            pts.append(pt2)

            M, N, _ = pt1.shape

            delta = pt2 - pt1
            delta_mag = numpy.linalg.norm(
                delta, axis=2)[:,:,numpy.newaxis]
            delta_unit = delta / delta_mag

            gauss = numpy.random.randn(M, N, 1) * std

            displacement = numpy.zeros((M, N, 3), dtype=float)
            dp = numpy.tensordot(z_unit, delta_unit, axes=([0],[2]))
            theta = arcsin(dp)

            displacement[:,:] = (
                cos(theta)[:,:,numpy.newaxis] * z_unit +
                delta_unit * sin(theta)[:,:,numpy.newaxis])
            displacement[:,:,:] *= gauss

            displacements.append(displacement)

        midpoint = numpy.mean(pts, axis=0)
        return midpoint + numpy.mean(displacements, axis=0)

    divisions = 0

    std = scale * ratio

    while target_divisions >= divisions:
        M, N, _ = P.shape
        if show_steps:
            if save_images:
                screenshot = "../output/%s-%02d.png" % (
                                save_images, divisions)
                view_polydata(mesh_from_point_matrix(P),
                              screenshot_file=screenshot)
            else:
                view_polydata(mesh_from_point_matrix(P))

        if target_divisions == divisions:
            break

        divisions += 1

        QM = M * 2 - 1
        QN = N * 2 - 1

        Q = numpy.zeros((QM + 2, QN + 2, 3), dtype=float)
        # COPY ORIGINALS ###################################
        Q[:QM:2,:QN:2] = P[:,:]

        # DIAMOND STEP #####################################

        # middle nodes
        UL = P[:M-1,:N-1]
        LL = P[:M-1,1:]
        UR = P[1:,:N-1]
        LR = P[1:,1:]
        Q[1:QM:2, 1:QN:2] = displace([(LR, UL), (LL, UR)], std)

        # SET UP BUFFER ####################################

        # Continue the slope to the left of Q (i=-1)
        U = P[0, :N-1]
        D = P[0, 1:]
        R = Q[1, 1:QN:2]
        mean_UD = numpy.mean((U, D), axis=0)
        Q[-1, 1:QN:2] = mean_UD + (mean_UD - R)

        # Continue the slope to the right of Q (i=QM)
        U = P[M-1, :N-1]
        D = P[M-1, 1:]
        L = Q[QM-2, 1:QN:2]
        mean_UD = numpy.mean((U, D), axis=0)
        Q[QM, 1:QN:2] = mean_UD + (mean_UD - L)

        # Continue the slope to the up of Q (j=-1)
        L = P[:M-1, 0]
        R = P[1:, 0]
        D = Q[1:QM:2, 1]
        mean_LR = numpy.mean((L, R), axis=0)
        Q[1:QM:2, -1] = mean_LR + (mean_LR - D)

        # Continue the slope to the down of Q (j=QN)
        L = P[:M-1, N-1]
        R = P[1:, N-1]
        U = Q[1:QM:2, QN-2]
        mean_LR = numpy.mean((L, R), axis=0)
        Q[1:QM:2, QN] = mean_LR + (mean_LR - U)

        # Square Step ######################################

        # Even rows
        L = P[0:M-1, :]
        R = P[1:M, :]
```

```
        U = Q[1:QM:2, xrange(-1,QN-1,2)]
        D = Q[1:QM:2, 1:QN+1:2]
        Q[1:QM:2, 0:QN:2] = displace([(U, D), (L, R)], std)

        # Odd rows
        L = Q[xrange(-1,QM-1,2), 1:QN:2]
        R = Q[xrange(1,QM+1,2), 1:QN:2]
        U = P[:, 0:N-1]
        D = P[:, 1:N]
        Q[0:QM:2, 1:QN:2] = displace([(U, D), (L, R)], std)

        # CORRECT ORIGINAL POINTS ##############################

        # Smooth original points to match
        Q[2:QM-1:2,2:QN-1:2] = numpy.mean(
            (
                Q[2-1:QM-1-1:2,2:QN-2:2],
                Q[2+1:QM-1+1:2,2:QN-2:2],
                Q[2:QM-1:2,2-1:QN-2-1:2],
                Q[2:QM-1:2,2+1:QN-2+1:2],
            ), axis=0
        )

        #LEFT edge originals
        Q[0,2:QN-1:2] = numpy.mean(
            (
                Q[0,2-1:QN-2-1:2],
                Q[0,2+1:QN-2+1:2],
            ), axis=0
        )

        #RIGHT edge originals
        Q[QM-1,2:QN-1:2] = numpy.mean(
            (
                Q[QM-1,2-1:QN-2-1:2],
                Q[QM-1,2+1:QN-2+1:2],
            ), axis=0
        )

        #UP edge originals
        Q[2:QM-1:2,0] = numpy.mean(
            (
                Q[2-1:QM-1-1:2,0],
                Q[2+1:QM-1+1:2,0],
            ), axis=0
        )

        #DOWN edge originals
        Q[2:QM-1:2,QN-1] = numpy.mean(
            (
                Q[2-1:QM-1-1:2,QN-1],
                Q[2+1:QM-1+1:2,QN-1],
            ), axis=0
        )

        std *= ratio
        P = Q[:QM,:QN]

    return P

def ensure_directory_exists(screenshot_file):
    import os
    directory = os.path.dirname(screenshot_file)
    if not os.path.exists(directory):
        os.makedirs(directory)

def mesh_from_point_matrix(P):
    ''' Utility '''
    M, N, _ = P.shape

    cells = vtkCellArray()
    points = vtkPoints()
    polydata = vtkPolyData()

    polydata.SetPolys(cells)
    polydata.SetPoints(points)

    for i in xrange(M):
        for j in xrange(N):
            points.InsertNextPoint(P[i,j])


    for i in xrange(M-1):
        for j in xrange(N-1):
            cells.InsertNextCell(3)
            cells.InsertCellPoint(i * N + j)
            cells.InsertCellPoint(i * N + j + 1)
            cells.InsertCellPoint((i+1) * N + j)
            cells.InsertNextCell(3)
            cells.InsertCellPoint((i+1) * N + j)
            cells.InsertCellPoint((i+1) * N + j + 1)
            cells.InsertCellPoint(i * N + j + 1)
    return polydata

def view_polydata(mesh, screenshot_file=None):
    ''' Utility '''

    mapper = vtkPolyDataMapper()

    mapper.SetInputData(mesh)
    mapper.SetScalarRange(0, 0)

    actor = vtkActor()
    actor.SetMapper(mapper)
    actor.GetProperty().ShadingOn()
    actor.GetProperty().SetColor(.6, .3, .15)
    actor.GetProperty().SetSpecular(1.0)

    mapper.ScalarVisibilityOff()

    renderer = vtkRenderer()
    renderer.SetTwoSidedLighting(1)

    renderWindow = vtkRenderWindow()
    renderWindow.AddRenderer(renderer)
    renderWindowInteractor = vtkRenderWindowInteractor()
    renderWindowInteractor.SetInteractorStyle(
        vtkInteractorStyleTrackballCamera())
    renderWindowInteractor.SetRenderWindow(renderWindow)
    renderWindow.SetSize(1920, 1080)

    renderer.AddActor(actor)
    renderer.SetBackground(.2, .3, .4)

    camera = vtkCamera()
    camera.SetPosition(0.5, -1.0, 0.5)
    center = mesh.GetCenter()
    print center
    camera.SetFocalPoint(0.5, 0.5, 0.0)

    renderer.SetActiveCamera(camera)
    renderer.SetUseDepthPeeling(0)
    camera.SetViewUp(0, 0, 1.0)

    renderWindow.Render()


    if screenshot_file:
        windowToImageFilter = vtkWindowToImageFilter()

        magnification = 1

        windowToImageFilter.SetInput(renderWindow)
        windowToImageFilter.SetMagnification(magnification)
        windowToImageFilter.SetInputBufferTypeToRGBA()
        windowToImageFilter.ReadFrontBufferOn()
        windowToImageFilter.SetShouldRerender(1)
        windowToImageFilter.Update()

        writer = vtkPNGWriter()
        writer.SetFileName(screenshot_file)
        writer.SetInputConnection(windowToImageFilter.GetOutputPort())
        ensure_directory_exists(screenshot_file)
        writer.Write()

    else:
        renderWindowInteractor.Start()

if __name__ == '__main__':

    save_images = True
    show_steps = True

    stamp = "report"

    for h in [1.8, 1.6, 1.4, 1.2]:
        interpolate([1,0,0], [0,1,0], h,
                    save_images=stamp+str(h).replace('.','p'),
                    show_steps=show_steps)
```

References

[1] Numpy. http://www.numpy.org/. Accessed: 2016-12-20.

[2] Python 2.7.13 documentation. https://docs.python.org/2/. Accessed: 2016-12-20.

[3] Christopher Ewin. https://commons.wikimedia.org/wiki/File%3ADiamond_Square.svg, CC BY-SA 4.0 (http://creativecommons.org/licenses/by-sa/4.0), via Wikimedia Commons. Accessed: 2016-12-20.

[4] Alain Fournier, Don Fussell, and Loren Carpenter. Computer rendering of stochastic models. *Commun. ACM*, 25(6):371–384, June 1982.

[5] Will Schroeder, Ken Martin, and Bill Lorensen. *The Visualization Toolkit–An Object-Oriented Approach To 3D Graphics*. Kitware, Inc., fourth edition, 2006.

[6] S. van der Walt, S. C. Colbert, and G. Varoquaux. The numpy array: A structure for efficient numerical computation. *Computing in Science Engineering*, 13(2):22–30, March 2011.