

# Snowballs: An experiment in Winter frivolity

Edward Dale

Computer Graphics II

4005-762

Professor Warren Carithers

<http://www.scompt.com/school/classes/computer-graphics-2/snowballs>

## Project Description

I plan to implement the technique described in [SOH99]. It is “a simulation model of ground surfaces that can be deformed by the impact of rigid body models”. I would like to apply it to a ground cover of snow that has objects rolled around on it. The ultimate goal is to create an animation of a ball being rolled on the ground and creating a path in the snow. Ideally, the ball would grow in size as it rolls and picks up snow, simulating the process of creating a snowman. This project will illustrate numerous aspects of the rendering pipeline. Realism is a definite goal of the project. To this end, I plan to create a RenderMan shader to give the snow a realistic appearance.

## Approach

The approach I took is exactly the one from [SOH99]. They modeled the ground as a height field. A height field is generated using one of a number of methods. An object placed onto the ground displaces material from a point to all points lower than it. These values are determined by making a contour map of the object.

## User’s Guide

There is a very limited number of parameters that can be modified by the user. All of these are in the snow module. How the height field is generated can be changed by changing which `heightField*` method is uncommented. Similarly, how the height field is rendered can be changed by changing which `draw*` method is uncommented at the end of the module. The size and resolution of the height field can be changed, but numbers other than the ones currently in the program will likely not work.

To execute the program, the `viewer.py` script from `cgkit` [`cgk`] must be executed, passing `snow.py` as a parameter. This will show an OpenGL window where the camera moves around the scene a bit. The `render.py` script from `cgkit` can be used to render the scene with a RenderMan renderer.

# Technical Documentation

## Program Description

The program creates rendered images of a field of snow with an object placed in the snow. The ultimate goal is that of realism. The snow should be realistically deformed around the edges of the object. Currently, only a simple cube is allowed as an object.

## Historical Development of Program

Program development started in the middle of Winter quarter 2005 at RIT by Edward Dale. Development continues by Edward Dale and it will likely only ever be developed by Edward Dale.

## Overall System Structure

The system is broken into three modules. The main (snow) module calls the fields module to create the height field and then creates an object to deform the snow. Both the height field and the object are passed to the contour module to deform the height field. The height field is then rendered using methods in the snow module.

## Snow Module

The snow module is the entry point into the program. The first thing it does is call one of the `heightField*` methods in the fields module to generate the height field. Next, a call to `getFaces` is made to generate random faces in the height field. Next an ODE triangle mesh is built that represents the generated triangle mesh. This object will be used in the intersection calculations. Next, the cube is placed generated that intersects with the ground. It can either be rendered or not, depending on whether one is just testing. The real work happens when the `deform` method from the contour module is called. This method will be explained in full below, but it changes the height values stored in the height field. Finally, the snow module sets up a camera and renders the height field. Rendering can either be done as a triangle mesh, boxes, or lines. Boxes is the author's current favorite because of the appearance of the resulting image. However, as the number of boxes scales linearly with the number of values in the height field, it can be very slow to render. When the size of the height field is increased, it is expected that triangle mesh will be the only reasonable rendering method.

## Fields Module

The field module is concerned solely with the generation of initial height fields. A height field is two-dimensional array of height values represented by floating point numbers.

Four methods of generating have been created. The `constantHeightField` method returns a height field where all the values are the same. This is primary useful just for testing. The `noiseHeightField` method creates a height field using Perlin noise. The `randomHeightField` method creates a height field using by storing random numbers at each location. This method isn't too useful except to see how the deformation algorithm responds to it. The `gradientHeightField` method generates a height field by making sure that no two adjacent cells have a height value difference that is greater than a constant. It takes in an initial value and all other values are generated from this seed.

The fields module also contains a `getFaces` method that generates face information for the height field. If one imagines the height field as having a number of squares with height values at each corner, then this method breaks each square into two triangles. The triangles can be oriented one of two ways, so this method randomly chooses which orientation to use. This is to prevent any artifacts from all of the triangles going the same direction.

## Contour Module

The main entrypoint into the contour module is the `deform` method. It is called with a triangle mesh representing the height field and an ODE [ode] object. First of all, the two objects are intersected. At each point in the height field, a ray is shot up and intersected with the object. The difference between the height field value and the height of the intersection is interpreted as the amount of ground material that needs to be displaced. A contour map is then built to determine how to distribute the displaced material. Any point in the plane of intersections that didn't intersect with the object is initialized with a value of zero. Each point in the object's bounding box is then examined. If there is a lower number in any adjacent point, that point takes on that number incremented by one. This process is run until no changes are made in an iteration. After completion, a post-processing step is done to create a maps that maps from a location in space to a list of all the points that material should be distributed to. If desired, the material that needs to be distributed is then distributed to the lower cells. One may not want to distribute ground material in order to get a better look at the crater created by the object. This also runs iteratively until no changes are made to the ground material. There are also a number of utility methods in the contour module that allow the height field and contour information to be dumped out to the terminal.

## Data Structures

The main data structure is the height field itself. It is a two-dimensional array that is stored in a flat form in a list data structure. The  $(0, 0)$  location is stored at location 0, the  $(0, 5)$  location is stored at location 5, and the  $(5, 0)$  location is stored at location  $5 \times rows$ . Each location stores a float value that represents the height of the field at that point. The field is created in the methods in the fields module and is accessed everywhere in the

program. A number of downsides to this implementation have come to light recently. First of all, because the height field is stored as a list, the indices must be integers, which means that float point locations aren't fully supported right now. There is a parameter called *resolution* that lets one increase the number of height values stored per grid step. This only works for generating the field right now because of the index problem just mentioned. The proper fix for this is to create a height field class that can encapsulate the list operations. With this done, the height field generation methods from the fields module could be made static factory methods in the height field class.

The contour values are similarly stored as a two-dimensional array in a list. The contour relates to the object being placed in the snow and needs to be slightly larger than the bounding box of the object to account for displaced ground material. It needs to have the same resolution as the height field it's being placed in.

## Maintenance Aids

There are unit tests built-in to the contour module. This was the only way to actually write the code as it complex enough that I couldn't put my head around it immediately. The tests can be run by executing the contour module directly. The rest of the code doesn't do any complicated algorithms, so there are no built-in tests.

## Building the System

Because the program is written in Python, there is no building step. The libraries used by the system do, however, need to be built. The Open Dynamics Engine [ode] is a library to do rigid body dynamics that is used to do object collisions. The program actually uses pyODE [pyo], a python wrapper of the ODE library. Both of these can be installed relatively painlessly using instructions available from the respective websites.

The program uses cgkit [cgk] to do all of its drawing and rendering. To say installing cgkit is a challenge is an understatement. Following the instructions from the website is a good first start. The one change to make is to use the CVS version of cgkit instead of the latest release.

## Results

Rendered results can be seen at <http://www.scompt.com/photos/school/cg2-project/>.

## Future Enhancements

The biggest enhancement that needs to be made is to create a height field class, as mentioned above. This should facilitate the creation of higher resolution height fields and

images. The next change that needs to be made is to clean up and improve the contour code. It will need to be extended to allow for three-dimensional contours. Right now, it can only objects such as cubes where the bottom side is flat. The rest of the steps from the original paper need to be implemented as well.

## References

- [cgk] The python computer graphics kit. <http://cgkit.sourceforge.net/>.
- [ode] Open dynamics engine. <http://www.ode.org/>.
- [pyo] Pyode. <http://pyode.sourceforge.net/>.
- [SOH99] Robert Sumner, James F. O'Brien, and Jessica K. Hodgins. Animating sand, mud, and snow. *Computer Graphics Forum*, 18(1):17–26, March 1999. ISSN 1067-7055.

"""

Edward Dale  
2006-2-27  
Computer Graphics 2  
Project

Copyright (c) 2006, Edward Dale  
All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- \* Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- \* Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- \* Neither the name of Edward Dale nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

"""

```
from ode import *
from cgkit.all import *
from math import ceil
```

```
def isIn(bb, point):
```

"""

Checks if a point is inside a bounding box. This will have to get much more complicated for complex geometries.

"""

```
    itsin = point[0] >= bb[0] and point[0] < bb[1] and point[1] >= bb[2] and point[1] < b
```

]

```
    return itsin
```

```
def buildContour(geom, z=0):
```

"""

Builds a contour map of a geometry at the specified depth. Only works for simple geometries (Cubes) right now.

```
>>> odebox=GeomBox(lengths=(4,4,4))
```

```
>>> odebox.setPosition((2,2,0))
```

```
>>> cont=buildContour(odebox, 0)
```

```
>>> cont[(2,2,0)]
```

2

```
>>> cont[(1,1,0)]
```

2

```
>>> cont[(0,0,0)]
```

1

```
>>> cont[(-1,-1,0)]
```

```

0
>>> cont[(3,3,0)]
1
>>> cont[(4,4,0)]
0

>>> odebox=GeomBox(lengths=(5,5,5))
>>> odebox.setPosition((10,10,0))
>>> cont=buildContour(odebox, 0)
>>> cont[(10,10,0)]
3
>>> cont[(9,9,0)]
2
>>> cont[(8,8,0)]
1
>>> cont[(7,7,0)]
0
>>> cont[(11,11,0)]
2
>>> cont[(12,12,0)]
1
>>> cont[(13,13,0)]
0
"""
contour={}
bb=geom.getAABB()

# Intialize the contour map to all zeroes, can possible be
# done using a default value of zero in the next loop.
minx,maxx=int(bb[0])-1,int(ceil(bb[1]))+1
miny,maxy=int(bb[2])-1,int(ceil(bb[3]))+1

# Build the contour map
while 1:
    changes=0
    for x in range(minx,maxx):
        for y in range(miny,maxy):
            if isIn(bb, (x,y,z)):
                a=min(contour.get((x-1,y,z),0),contour.get((x+1,y,z),0),contour.get(
-1,0),0),contour.get((x,y+1,0),0))
                cont=contour.get((x,y,z),0)
                if a==cont:
                    changes += 1
                    contour[(x,y,z)] = cont+1
            else:
                contour[(x,y,z)] = 0
        if not changes: break

    return contour

def printContour(contour,width):
    """
    Dumps out the contour from the previous method.

    >>> cont={(4, -1, 0): 0, (-1, 2, 0): 0, (3, -1, 0): 0, (1, 0, 0): 1, (4, 4, 0): 0, (3
, 0): 0, (-1, 3, 0): 0, (1, 3, 0): 1, (2, 3, 0): 1, (3, 3, 0): 1, (2, 2, 0): 2, (4, 3, 0)
, (0, 2, 0): 1, (1, 4, 0): 0, (4, 0, 0): 0, (-1, 0, 0): 0, (1, 2, 0): 2, (2, 0, 0): 1, (3
, 0): 1, (2, 4, 0): 0, (0, 3, 0): 1, (3, 1, 0): 1, (4, 1, 0): 0, (0, 0, 0): 1, (-1, 4, 0)
, (0, 4, 0): 0, (-1, 1, 0): 0, (1, 1, 0): 2, (2, 1, 0): 2, (2, -1, 0): 0, (1, -1, 0): 0,
, -1, 0): 0, (3, 0, 0): 1, (4, 2, 0): 0, (0, 1, 0): 1, (0, -1, 0): 0}
    >>> print printContour(cont,6)
    <BLANKLINE>
    0 0 0 0 0 0
    0 1 1 1 1 0
    0 1 2 2 1 0

```

```

    0 1 2 2 1 0
    0 1 1 1 1 0
    0 0 0 0 0 0
    """
    out=""
    k=contour.keys()
    k.sort()
    count=0
    for i in k:
        if count%width == 0:
            out += '\n'
            count=0
        count +=1
        out += ' ' + str(contour[i])
    return out

def contourToDict(contIn):
    """
    Converts a contour created using buildContour to a dictionary that maps
    from a point in space to all of the points that material should be
    distributed to.
    >>> cont={(4, -1, 0): 0, (-1, 2, 0): 0, (3, -1, 0): 0, (1, 0, 0): 1, (4, 4, 0): 0, (3
, 0): 0, (-1, 3, 0): 0, (1, 3, 0): 1, (2, 3, 0): 1, (3, 3, 0): 1, (2, 2, 0): 2, (4, 3, 0)
, (0, 2, 0): 1, (1, 4, 0): 0, (4, 0, 0): 0, (-1, 0, 0): 0, (1, 2, 0): 2, (2, 0, 0): 1, (3
, 0): 1, (2, 4, 0): 0, (0, 3, 0): 1, (3, 1, 0): 1, (4, 1, 0): 0, (0, 0, 0): 1, (-1, 4, 0)
, (0, 4, 0): 0, (-1, 1, 0): 0, (1, 1, 0): 2, (2, 1, 0): 2, (2, -1, 0): 0, (1, -1, 0): 0,
, -1, 0): 0, (3, 0, 0): 1, (4, 2, 0): 0, (0, 1, 0): 1, (0, -1, 0): 0}
    >>> myDict=contourToDict(cont)
    >>> # TODO: test
    """
    # Build a dictionary of points to points of lower contour
    distribution={}
    big=1000
    for k,v in contIn.iteritems():
        lowers=[]
        if contIn.get((k[0]-1, k[1], k[2]), big) < v:
            lowers.append((k[0]-1, k[1], k[2]))
        if contIn.get((k[0]+1, k[1], k[2]), big) < v:
            lowers.append((k[0]+1, k[1], k[2]))
        if contIn.get((k[0], k[1]-1, k[2]), big) < v:
            lowers.append((k[0], k[1]-1, k[2]))
        if contIn.get((k[0], k[1]+1, k[2]), big) < v:
            lowers.append((k[0], k[1]+1, k[2]))
        distribution[k] = lowers
    return distribution

def myCollide(mesh, odegeom):
    """
    Collides the odegeom object with the mesh using Rays fired from each point
    on the ground up at the box. Returns the contacts that occurred and the
    contour that was created.

    >>> verts=[vec3(0, 0, 10), vec3(0, 1, 10), vec3(0, 2, 10), vec3(0, 3, 10), vec3(0, 4
), vec3(1, 0, 10), vec3(1, 1, 10), vec3(1, 2, 10), vec3(1, 3, 10), vec3(1, 4, 10), vec3(2
, 10), vec3(2, 1, 10), vec3(2, 2, 10), vec3(2, 3, 10), vec3(2, 4, 10), vec3(3, 0, 10), ve
3, 1, 10), vec3(3, 2, 10), vec3(3, 3, 10), vec3(3, 4, 10), vec3(4, 0, 10), vec3(4, 1, 10)
ec3(4, 2, 10), vec3(4, 3, 10), vec3(4, 4, 10)]
    >>> faces=[(5, 1, 0), (5, 6, 1), (6, 2, 1), (6, 7, 2), (7, 3, 2), (7, 8, 3), (8, 4, 3
(8, 9, 4), (10, 6, 5), (10, 11, 6), (11, 7, 6), (11, 12, 7), (13, 8, 7), (12, 13, 7), (14
, 8), (13, 14, 8), (15, 11, 10), (15, 16, 11), (16, 12, 11), (16, 17, 12), (17, 13, 12),
, 18, 13), (19, 14, 13), (18, 19, 13), (21, 16, 15), (20, 21, 15), (22, 17, 16), (21, 22
), (22, 18, 17), (22, 23, 18), (23, 19, 18), (23, 24, 19)]
    >>> a=TriMeshData()
    >>> a.build(verts=verts, faces=faces)
    >>> mesh=GeomTriMesh(data=a)

```

```

>>> odebox=GeomBox(lengths=(3,3,3))
>>> odebox.setPosition((2,2,10))
>>> contacts,contour=myCollide(mesh,odebox)
>>> len(contacts)
9
>>> for i in contacts.values():
...     if i != 8.5: print False
"""
cont = buildContour(odegeom, 0)
contour=contourToDict(cont)

bb=odegeom.getAABB()

ray=GeomRay()
ray.set((0,0,0), (0,0,1))
ray.setLength(100)

# The bounds
minx,maxx=int(bb[0])-1,int(ceil(bb[1]))+1
miny,maxy=int(bb[2])-1,int(ceil(bb[3]))+1

contacts={}
for i in range(minx,maxx):
    for j in range(miny,maxy):
        # Shoot the ray from 0 up to the level of the ground
        ray.setLength(100)
        ray.set((i,j,0), (0,0,1))
        c1=collide(mesh,ray)

        if c1:
            # If the ray hits the ground, then shoot it again
            # with the length equal to the height of the ground
            # at this point. If it hits the object, then it is
            # intersecting with the ground and needs to displace material.
            ray.setLength(c1[0].getContactGeomParams()[2])
            c2=collide(odegeom,ray)
            if c2:
                # Record where the intersection occurs.
                con = c2[0].getContactGeomParams()
                contacts[(con[0][0],con[0][1])] = con[2]
return contacts,contour

def deform(mesh, odegeom, verts,x,y, distribute=True):
    """
    Collides the ground and the object and distributes material
    to the neighboring, lower locations.

    >>> verts=[vec3(0, 0, 10), vec3(0, 1, 10), vec3(0, 2, 10), vec3(0, 3, 10), vec3(0, 4,
), vec3(1, 0, 10), vec3(1, 1, 10), vec3(1, 2, 10), vec3(1, 3, 10), vec3(1, 4, 10), vec3(2
, 10), vec3(2, 1, 10), vec3(2, 2, 10), vec3(2, 3, 10), vec3(2, 4, 10), vec3(3, 0, 10), ve
3, 1, 10), vec3(3, 2, 10), vec3(3, 3, 10), vec3(3, 4, 10), vec3(4, 0, 10), vec3(4, 1, 10)
ec3(4, 2, 10), vec3(4, 3, 10), vec3(4, 4, 10)]
    >>> faces=[(5, 1, 0), (5, 6, 1), (6, 2, 1), (6, 7, 2), (7, 3, 2), (7, 8, 3), (8, 4, 3
), (8, 9, 4), (10, 6, 5), (10, 11, 6), (11, 7, 6), (11, 12, 7), (13, 8, 7), (12, 13, 7), (14
, 8), (13, 14, 8), (15, 11, 10), (15, 16, 11), (16, 12, 11), (16, 17, 12), (17, 13, 12),
, 18, 13), (19, 14, 13), (18, 19, 13), (21, 16, 15), (20, 21, 15), (22, 17, 16), (21, 22
), (22, 18, 17), (22, 23, 18), (23, 19, 18), (23, 24, 19)]
    >>> a=TriMeshData()
    >>> a.build(verts=verts, faces=faces)
    >>> mesh=GeomTriMesh(data=a)
    >>> odebox=GeomBox(lengths=(3,3,3))
    >>> odebox.setPosition((2,2,10))
    >>> dumpHeights(verts,5,-1,5,-1,5)
10.0 10.0 10.0 10.0 10.0 10.0
10.0 10.0 10.0 10.0 10.0 10.0

```

```
10.0 10.0 10.0 10.0 10.0 10.0
10.0 10.0 10.0 10.0 10.0 10.0
10.0 10.0 10.0 10.0 10.0 10.0
10.0 10.0 10.0 10.0 10.0 10.0
>>> deform(mesh,odebox,verts,5,5)
>>> dumpHeights(verts,5,-1,5,-1,5)
10.0 10.0 10.0 10.0 10.0 10.0
10.0 10.0 10.0 10.0 10.0 10.0
10.0 10.0 8.5 8.5 8.5 10.0
10.0 10.0 8.5 8.5 8.5 10.0
10.0 10.0 8.5 8.5 8.5 10.0
10.0 10.0 10.0 10.0 10.0 10.0
"""
contacts,contour = myCollide(mesh,odegeom)

while 1:
    changes=0

    for cont in contacts.iteritems():
        contactdepth=cont[1]
        vertidx=int(cont[0][0]*y+cont[0][1])
        #print verts[vertidx]
        vertdepth=verts[vertidx].z
        if contactdepth < vertdepth:
            diff = vertdepth-contactdepth
            verts[vertidx].z = contactdepth

            if distribute:
                lowers = contour[(cont[0][0], cont[0][1], 0)]
                #print lowers
                for lower in lowers:
                    changes += 1
                    verts[lower[0]*y+lower[1]].z += diff/len(lowers)
    if not changes: break

def dumpHeights(verts,width,minx,maxx,miny,maxy):
    """
    Prints out a textual representation of the height field.
    """
    for i in range(minx,maxx):
        for j in range(miny,maxy):
            print verts[i*width+j][2],
        print

def _test():
    """
    Run the unit tests for each method using the doctest module.
    """
    import doctest
    doctest.testmod()

# If the module's being run directly, then run the unit tests.
if __name__ == "__main__":
    _test()
```

"""

Edward Dale  
2006-2-27  
Computer Graphics 2  
Project

Copyright (c) 2006, Edward Dale  
All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- \* Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- \* Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- \* Neither the name of Edward Dale nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

"""

```
from cgkit.all import *
from cgkit import noise
from random import random, uniform, seed, choice
```

```
def getFaces(num_x,num_y):
```

"""

A height field is rectilinear, but the faces are only going to be triangular. This method chooses 3 vertices out of each 4 to make a face and the other set to make another triangular face. This is done randomly to eliminate artifacts.

"""

"""

```
faces=[]
for y in range(num_y-1):
    for x in range(num_x-1):
        one = (num_x*y)+x
        two = (num_x*y)+x+1
        three = (num_x*(y+1))+x
        four = (num_x*(y+1))+x+1
        faces += choice( ( [( three,two,one ),( three,four,two )],
                          [( four,two,one ), ( three,four,one )] ) )
return faces
```

```
def constantHeightField( num_x, num_y, height=10.0 ):
```

"""

A height field where all the values are the same.

"""

"""

```
verts=[]
for x in range(num_x):
    for y in range(num_y):
```

```
        verts.append(vec3(x,y,height))
    return verts

def gradientHeightField( num_x, num_y, initial=5, scale=1.0, depthOffset=10, gradient=0.3):
    """
    A height field where values from adjacent columns are no greater than a certain gradient
    value.
    """
    verts=[]
    lastCol=None
    for x in range(num_x):
        thisCol=[]
        lastCell=None
        for y in range(num_y):
            if y==0 and x==0:
                lastCell = vec3(x*scale,y*scale,initial)
            elif x==0:
                lastCell = vec3(x*scale,y*scale, lastCell.z + uniform(-gradient,gradient))
            elif y==0:
                lastCell = vec3(x*scale,y*scale, lastCol[y].z + uniform(gradient,gradient))
            else: # everywhere else
                avgZ = (lastCol[y].z + lastCell.z)/2
                lastCell = vec3(x*scale,y*scale, avgZ+ uniform(gradient,gradient))
            thisCol.append(lastCell)
            verts.append(lastCell+vec3(0,0,depthOffset))
        lastCol=thisCol
    return verts

def noiseHeightField(num_x, num_y, resolution=1.0, height=10, scale=5):
    """
    A height field where all values come from Perlin noise.
    """
    verts=[]
    blah = resolution * 10
    for x in range(int(num_x*resolution)):
        for y in range(int(num_y*resolution)):
            vert=vec3(x/resolution,y/resolution,
                    noise.noise(x/blah,y/blah, 0)*scale+height)
            verts.append( vert )
    return verts

def randomHeightField( num_x, num_y, scale=1.0 ):
    """
    A height field where values are completely random.
    """
    verts=[]
    for x in range(num_x):
        for y in range(num_y):
            verts.append( vec3(x,y, random()*scale) )

    return verts
```

```
"""
```

```
Edward Dale  
2006-2-27  
Computer Graphics 2  
Project
```

```
Copyright (c) 2006, Edward Dale  
All rights reserved.
```

```
Redistribution and use in source and binary forms, with or without modification,  
are permitted provided that the following conditions are met:
```

- \* Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- \* Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- \* Neither the name of Edward Dale nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

```
THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND  
CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES,  
INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF  
MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE  
DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS  
BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR  
CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT  
OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR  
BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF  
LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING  
NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS  
SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.  
"""
```

```
from cgkit.all import *  
from random import random, uniform, seed  
from ode import *  
from math import pi  
  
from fields import *  
from contour import *  
  
def drawTriMesh(verts, faces):  
    """  
    Draws the height field as a triangle mesh.  
    """  
    print "Rendering a %d triangle mesh." % len( faces )  
    TriMesh("Trimesh", True, False, verts, faces )  
  
def drawBoxMesh(verts):  
    """  
    Draws the height field as a bunch of boxes.  
    """  
    print "Rendering %d boxes." % len(verts)  
    for vert in verts:  
        Box(lx=1,ly=1,lz=vert[2], pos=vec3(vert[0],vert[1],0))  
  
def drawLineMesh(verts):  
    """  
    Doesn't really work as desired because the size parameter doesn't  
    do seem to anything.  
    """
```

```
print "Rendering %d lines." % len(verts)
for vert in verts:
    drawLine(pos1=vec3(vert[0],vert[1],0), pos2=vert, size=10)

def setupCamera(x,y):
    """
    Add the camera and light to the scene.
    """
    cameraPos=Expression("vec3(x+a*sin(t/2),y+a*cos(t/2),z)",
                          a=5, x=1.2*x, y=1.2*y, z=0.4*(x+y) )

    cam=TargetCamera(target=(40,40,10))
    light=GLPointLight(diffuse=(1,1,1))

    cameraPos.output_slot.connect( cam.pos_slot)
    cameraPos.output_slot.connect( light.pos_slot)

def placeBox( showBox ):
    """
    Places the red box in the ground and return it.  If the parameter is
    True, then the box is drawn in the scene too.
    """
    mat=GLMaterial(diffuse = vec3(1,0,0),
                   specular = vec3(1,1,1),
                   shininess = 30)
    if showBox: Box(lx=5,ly=5,lz=5, pos=(40,40,10), material=mat)
    odebox=GeomBox(lengths=(5,5,5))
    odebox.setPosition((40,40,10))
    return odebox

resolution=1.0
seed(5)
x,y=(50,50)
#verts=gradientHeightField( x, y )
#verts=constantHeightField(x,y)
#verts=randomHeightField(x,y, 10)
verts=noiseHeightField(x,y, resolution)
faces=getFaces(int(resolution)*x,int(resolution)*y)

# Create the triangle mesh representing the ground
a=TriMeshData()
a.build(verts=verts, faces=faces)
mesh=GeomTriMesh(data=a)

odebox = placeBox( True )
deform(mesh,odebox,verts,x,y, True)

setupCamera(x, y)
drawBoxMesh(verts)
#drawLineMesh(verts)
#drawTriMesh( verts, faces )
```