Procedural terrain generation

ICS 162
Lecture #5
Announcements

• HWK #2 out later today, due Thursday (10/25)
  – Learn about scripting in Maya using MEL
  – Experiment with procedural generation
Procedural generation

Generate interesting geometry, texture, animation, etc. automatically (programmatically)

Why?

- Manpower: Generate content for very large virtual worlds without requiring an artist to create and place every single blade of grass, pebble, etc.

- Compression: Code to generate content is often much smaller than the content generated
Example: Crysis 2

Procedural animation of vegetation blowing in the wind.
Example: Spore

Gameplay based on “evolution” from single cell organisms up to interplanetary civilization

- Makes heavy use of procedural generation (planet geometry, character animations, behavior, etc.)

- Allows the import of content from other users games
Procedural generation

Some examples we will discuss

- Fractals and terrain generation
- L-systems and Grammars
- Texture synthesis

later: physics and simulation
Today: Virtual Terrain Generation
Terrain generation history

• The defense industry created virtual terrains as early as the 1970’s
  – Their purpose were ballistics and training simulations

• TINs (Triangulated Irregular Networks) use in GIS
  – TIN : vector based representation of the physical land surface or sea bottom
    (Randolf Franklin, 1973)

• In the 1980’s procedural techniques are developed and they are used to
  generate artificial terrains
  – Perlin noise functions

• Late 1980’s: fractals and polygonal subdivision techniques become more
  widespread in artificial terrain generation *(we will talk about this today)*

• Present: High-quality generators synthesize whole virtual planets with
  diverse, interesting terrain features.
Military simulation software SIMDIS

Triangulated Irregular Network (TIN)

Emil Multifractal terrain by Kenton Musgrave
A virtual terrain generated using Terragen 2 from Planetside software

Image created by Hannes Janetzko.

http://www.planetside.co.uk/gallery/f/tg2/
Hannes-Another+jungle+flyover.jpg.html
Terrain generation

Treat height as a function $\text{height} = f(x,y)$

– Given $f$ we can generate a mesh (and a colormap)

– Could use real data (e.g. given by artist or a real topo map) but our goal is to synthesize a value for $f$ at each location
Adding color and texture

Color based on height
Adding color and texture

Color based on height and surface normal
First attempt

Use “random” values for height field $f(x,y)$

hmmm...lacking structure

SrO clusters randomly distributed on smooth TiO2 surface observed on fractured (pristine) Nb-doped SrTiO3 surface.

--Argonne National Labs
Fault line algorithm

- Created to approximate real world terrain features such as escarpments, mesas, and seaside cliffs

First step in faulting process

Terrain generated after 400 iterations

One way of generating fault lines in a height field grid

- randomly pick two grid points \( p_1 \), \( p_2 \)
- calculate the line between them
- Go through all the points in the height field and add or subtract an offset value depending on what side of the line they are located
- Before the next fault is drawn, reduce the range of the offset by some amount
Need to smooth!

- Compute local average in order to smooth values in height map
- Referred to as “low-pass filtering”
- Can control smoothness based on size of averaging window
Now with smoothing!
some variations to the fault line algorithm

- Cosine
- Sine
Fractals

• Many natural objects have a fractal structure. They appear similar at different levels of magnification (self-similar scaling)
Self-similarity

Similarity at different locations: 
(periodic texture)

Similarity at different scales: 
(fractal)
Generating Self-similarity

• Use recursion...
Koch “Snowflake”
Fractal Terrains

Idea: use fractal functions to generate height-map, color, texture
Midpoint displacement (1D version)

- Type of polygon subdivision algorithm,
- “Simulate” tectonic uplift of mountain ranges (Fournier & Fussell (1980))
- Input parameters: roughness constant \( r \)

**Step 0**

Displace the midpoint of the line by some random value between (-\( d \), \( d \))

**Step 1**

Reduce the range of your random function by \( d = d^* \times 2^{-r} \)

**Step 2**

Repeat displace the midpoint of all the line segments
Iterate until you get the required detail

How does $r$ affect the outcome?

If $r = 1$
your $d$ will half each iteration

If $r > 1$
$d$ increases faster
generates smooth terrain

If $r < 1$
$d$ increases slowly
generates chaotic terrain
Generalize to 2D
Diamond- Square Subdivision

- The 2D version of the midpoint displacement algorithm
- Also called the cloud fractal, plasma fractal or random midpoint displacement
• the algorithm starts with a 2 x 2 grid
  – The heights at the corners can be set to either zero, a random value or some predefined value
The first step involves calculating the midpoint of the grid based on its corners and then adding the maximum displacement for the current iteration. This is called the Diamond step, because if you render this terrain you will see four diamond shapes.

E = (A+B+C+D)/4 + Rand(d)

rand(d) can generate random values between -d and +d.
Next is the Square step

Calculate the midpoints of the edges between the corners

\[ G = \frac{(A+B+E)}{3} + \text{rand}(d) \]
\[ H = \frac{(B+D+E)}{3} + \text{rand}(d) \]
\[ I = \frac{(D+C+E)}{3} + \text{rand}(d) \]
\[ F = \frac{(A+C+E)}{3} + \text{rand}(d) \]

Now reduce d is reduced by

\[ d = d \times 2^{-r} \]

where \( r \) is the roughness constant
• Start the second iteration.... perform the diamond step

\[
\begin{align*}
J &= (A+G+F+E)/4 + \text{rand}(d) \\
K &= (G+B+E+H)/4 + \text{rand}(d) \\
L &= (F+E+D+I)/4 + \text{rand}(d) \\
M &= (E+H+I+C)/4 + \text{rand}(d)
\end{align*}
\]

Remember this $d$ is smaller than the one in the first iteration
Perform the square step:

\[
O = \frac{(A+G+J)}{3} + \text{rand}(d)
\]

\[
N = \frac{(A+F+J)}{3} + \text{rand}(d)
\]

\[
P = \frac{(J+G+K+E)}{4} + \text{rand}(d)
\]

\[
Q = \frac{(J+E+L+F)}{4} + \text{rand}(d)
\]

and so on.... continue subdividing until you reach the desired level of detail
“Diamond Square” Subdivision
Diamond- Square

- The 2D version of the midpoint displacement algorithm

- The diamond-square alg. works best if it is run on square grids of width $2^n$
  - This ensures that the rectangle size will have an integer value at each iteration

- Still lacking structure....
Still too homogenous

- Real terrains are not actually fractals... they have more structure
- A better choice would be to use Multifractals
  - These are fractals whose dimension/roughness varies with location

Multifractal terrain by Kenton Musgrave.
Texturing and Modeling: a Procedural Approach 3rd edition, pg 490
Multi-fractals

Varying the roughness of the fractal

Multi-fractal terrain patch

- Difficult to work with...
  - Fractal terrain varies from highly heterogeneous to flat
  - Output usually needs to be rescaled to combine different roughness

Instead of varying roughness, use some other tricks to generate variety
"Calderas" (reflect around a specified max value)

**FIGURE 4.19.6** The caldera line.

**FIGURE 4.19.7** Invert the height field values above the caldera line.
Ridged multi-fractals

Use $1 - \text{abs}(f(x,y))$ to get sharp “ridges”

Ridged multifractal terrains: taken from Texturing and Modeling: A Procedural Approach pg 518 (left) pg 480 (right)
This produces some nice ridges....

Ridged multi-fractal
Add a height-based texture
More Controls

• To create plateaus, add a $\text{min}$ function to flatten out high areas.
More Controls

• To create a plain, add a $max$ function to flatten out low areas.
Combine multiple height fields

- To create valleys, create a lower amplitude, smooth terrain. Then take the \textit{max} of the two
Ridges

• Quantize a terrain to create ridges
  – Take the *min* of original and quantized function
  – In general can define a *transfer function* that maps $f(x,y) \rightarrow g(f(x,y))$
Particle Deposition

- Simulates volcanic mountain ranges and island systems
- drop random particles in a blank grid
- Determine if the particle’s neighboring cells are of a lower height
  - If this is the case increment the height of the lowest cell
  - keep checking its surrounding cells for a set number of steps or until it is the lowest height among its surrounding cells
  - If not increment the height of the current cell

Generated after 5 series of 1000 iterations
Erosion
Limitation of heightmaps

They cannot generate overhangs or caves

Fix: Generate a 3d offset vector instead of a height?

Other possibilities...

Voxel Terrain


Mandelbulb
Next time... L-systems and Textures
MEL Scripting Intro

• MEL is Maya’s built in scripting language
  – Allows programmatic access to underlying functions in Maya
  – Almost all of Maya’s own user interface is built on top of MEL
  – Allows you to build custom tools that interact with Maya
    • High demand in animation and game industry for technical artists
Accessing MEL

• Click on “script editor” icon in lower right-hand corner
  – Command history shows MEL commands that have been executed. Try creating a sphere and note the command that pops up in the history window. *This is very useful!*
  – Enter a command in the MEL window. Highlight text and hit Ctrl+Enter to execute it.
  – Create scripts, save them, add them to custom “shelf”
MEL commands

Example:

//create a sphere or radius 12
polySphere -r 12;

//move it distance 10 along x-axis
move -r 10 0 0
Online help:

cmd | polySphere

Go to: Synopsis, Return value, Related, Flags, MEL examples.

Synopsis

```

polySphere is undoable, queryable, and editable.
The sphere command creates a new polygonal sphere.

Return value

```
string? Object name and node name.
```

In query mode, return type is based on queried flag.

Related

polyCone, polyCube, polyCubes, polyCylinder, polyPlane, polyTorus

Flags

```
axis, constructionHistory, createUVs, name, object, radius, subdivX, subdivY, subdivZ, texture
```

<table>
<thead>
<tr>
<th>Long name (short name)</th>
<th>Argument types</th>
<th>Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>axis (a)</td>
<td>linear</td>
<td></td>
</tr>
<tr>
<td>-axis (-a)</td>
<td>linear</td>
<td></td>
</tr>
<tr>
<td>radius (r)</td>
<td>linear</td>
<td></td>
</tr>
<tr>
<td>-radius (-r)</td>
<td>linear</td>
<td></td>
</tr>
<tr>
<td>subdivX (sX)</td>
<td>int</td>
<td></td>
</tr>
<tr>
<td>-subdivX (-sX)</td>
<td>int</td>
<td></td>
</tr>
<tr>
<td>subdivY (sY)</td>
<td>int</td>
<td></td>
</tr>
<tr>
<td>-subdivY (-sY)</td>
<td>int</td>
<td></td>
</tr>
<tr>
<td>subdivZ (sZ)</td>
<td>int</td>
<td></td>
</tr>
<tr>
<td>-subdivZ (-sZ)</td>
<td>int</td>
<td></td>
</tr>
</tbody>
</table>

This flag specifies the primitive axis used to build the sphere.
Q: When queried, this flag returns a `float`.

This flag specifies the radius of the sphere.
C: Default is 0.5.
Q: When queried, this flag returns a `float`.

This specifies the number of subdivisions in the X direction for the sphere.
C: Default is 20.
Q: When queried, this flag returns an `int`.