Procedural Asset Generation for Space-Themed Graphical Applications

Ben Yuan, California Institute of Technology
CS174 Seminar Day
Overview

● Tried to create something game-like (for CS145).
● This didn't work too well.
● However, we got some procedural asset generation done.
● We'll talk about that today.
Lots of computer games are set in space
A typical game will have hundreds to thousands of art assets
  ● with a space-based game, this number could be potentially boundless!!
Requiring artists to create every asset can be quite labor-expensive
Procedural generation aims to automate the process of asset creation
  ○ create something unique and plausible
Case in point: an asteroid field
○ Every asteroid is 'supposed' to be unique
○ Using one mesh for every asteroid object is unnaturally repetitive
○ Don't want artists to be stuck creating 100s of asteroids by hand

Case in point: planet textures in a boundless universe
○ Hand-sculpting terrain is labor-intensive - especially when planets aren't the main focus
○ Every planet needs to look unique
Case study: Spore

- **Large** world
  - several tens of thousands of stars
  - most stars have multiple planets
- Every corner of the galaxy is explorable
- Impossible for traditional content creation!
What is 'procedural generation'?

- Given some small seed data, algorithmically create an asset.
  - input: number(s), e.g. systematic ID
  - output: name, texture, mesh, terrain, entire game world, ...

- Can be used to aid the creative process...
  - quickly prototype an asset, then have artists iterate

- ...or to generate finished assets autonomously
  - artists set parameters; results are otherwise determined entirely by algorithm
What problems are involved?

● Need a way to turn a number into a finished asset
  ○ Ideally, a small change in input data should result in a large change in output
  ○ A small amount of input data needs to be expanded into an arbitrarily large amount of intermediate / output data
  ○ Identical input data should result in identical output

● Most procedural algorithms rely on high-quality PRNGs (pseudo-random number generators)
  ○ predictable output given a seed
  ○ desirable statistical properties
What problems are involved?

● Perfectly random numbers are not always desirable
  ○ for instance, "static" is not usually suitable for generating terrain

● Would like a notion of "smooth" / "coherent" noise
A powerful tool: Perlin noise

- Perlin noise: a coherent, smoothly varying noise primitive
  - initially developed by Ken Perlin for TRON (1982)
  - now ubiquitous
- The default primitive produces "blurry" noise
- "Blurry" noise can be resized and additively applied
  - this creates a "fractal" appearance
Perlin noise: basic ideas

- Divide up space into integer grid points
- Assign a 'pseudorandom' gradient unit vector to each grid point
  - these assignments should be consistent
  - one way to do this: pseudorandom coordinate-based lookup into table of evenly distributed gradients
- Given a sampling point:
  - look at 'surrounding' grid points
  - for each: compute dot product of relative position with assigned gradient vector
  - take weighted component-wise sum based on component distance (weighted based on 'S curve')
    - e.g. $6t^5 - 15t^4 + 10t^3$
Perlin noise: worked example

∃ whiteboard?
Perlin noise: iteration

- Take a weighted sum of multiple noise functions
  - e.g. $0.5 \times \text{noise}(x) + 0.25 \times \text{noise}(2x) + 0.125 \times \text{noise}(4x) + ...$
  - number of octaves: number of terms in sum
  - lacunarity: factor by which frequency changes per octave
  - persistence: factor by which amplitude changes per octave

- More octaves => more detail
- Result is 'fractal noise'
Using Perlin noise: asteroids

- It turns out that 3D Perlin noise can be leveraged to generate decent asteroid meshes
- Add 3D Perlin noise to a distance metric
- Take an isosurface using the "marching cubes" algorithm
  - Divide up sample space into cubical grid cells
  - Sample grid cell corners
  - Use corner values (above or below target value?) to determine which edges to connect with triangles
  - Use linear interpolation to find edge intersection with isosurface (where triangle vertices are placed)
● Grid resolution: 8 in live demo; 32 for last screenshot
● OpenMP-enabled CPU marching-cubes implementation
  ○ SLOW for high grid resolution!
● Occasional artifacts where isosurface intersects edge of grid
● Lighting is done by FFP; textures not dynamically computed
  ○ per-pixel texturing, lighting would require sampling noise field somehow
  ○ probably doable, but not done for this segment
Using Perlin noise: planets

- The focus of this project is not on planetary landings - everything always gets seen from orbit
- Coloring based on generated heightmap seems sufficient
- How to texture-map a planet?
  - standard UV mapping puts too much resolution on poles
  - alternative: cube mapping (usually used for reflection calculations)
    - small amounts of distortion near edges / corners
    - much easier to texture than icosphere
Using Perlin noise: planets

- **How to heightmap?**
- **For every pixel on cubemap texture:**
  - Normalize pixel coordinates (to sphere surface)
  - Sample 3D Perlin noise
  - Record value as pixel intensity ("height")
- **Inverse operation (cubemap querying) is harder to express mathematically...**
- **but is natively supported by GPU!**
  - GPU can project any vector onto cubemap, and return a color value
Implementation notes: planet textures

- First iteration: CPU heightmap, CPU texture
  - texture resolution-limited = grainy, pixelated
  - SLOW initial data generation
  - fixed-function pipeline = only basic lighting
Implementation notes: planet textures

- Second iteration: CPU heightmap, GPU fragment shader
  - still slow initial generation - but faster than CPU texturing
  - pixel-exact coloring with heightmap interpolation = smoother-looking even with low-res height data
Implementation notes: planet textures

- Second iteration: CPU heightmap, GPU fragment shader
  - can do pixel-perfect lighting with varying 'shininess'
  - can do more advanced techniques
    - e.g. bump mapping
Implementation notes: planet textures

- Third iteration: everything in fragment shader
  - reimplemented libnoise algorithms in shader code
    - older GPUs don't support bitwise operations - but newer ones do
  - **every** pixel is recomputed every frame
    - permits "infinite" complexity
    - high GPU load!
  - buggy >_>
    - PRNG doesn't work well with large seed values!?
Further work

- **General improvements**
  - working GPU noise (that doesn't break)

- **Asteroid generation improvements**
  - Dynamic texturing
  - GPU mesh generation

- **Planet generation improvements**
  - Changing noise properties
    - combining multiple noise sources with different operations can lead to different results
      - addition, multiplication, selection based on value, ...
  - GPU heightmap generation
  - Atmospheric shader effects
    - modeling atmospheric scattering (for more realistic temperate planets, etc)