3D Nature Environment with Procedurally Generated Terrain and Trees

Liam Ozog
lozog
20515121

CS 488, Fall 2016

Introduction:

Much of the inspiration for this project comes from David Whatley's chapter in GPU Gems 2, "Toward Photorealism in Virtual Botany". The aim was to combine principles of procedural generation with other graphics techniques to create an immersive, organic environment.

Manual:

Build :

Building this project is mostly similar to the previous CS488 assignments. First, run premake in the top level CS488 project directory:

```
premake4 gmake
make
```

There is an additional library, SOIL, used for image loading. Copy the file *libSOIL.a* from the A5/ directory to the lib/ directory.
Next, build the actual project. Navigate to the A5/ folder and compile:

```
cd A5
premake4 gmake
make
```

Program Input :

Usage of the program is as follows:

```
./A5 [ input-param-file ]
```

Specifying the input file name is not required. If it isn't supplied, the program will load the file **inparams1**, which *must* exist.

The input file is used to specify program parameters. I chose not to use Lua because the input is simple enough to not require it. The input file is assumed to be well-formed. Each line contains a parameter name, then a value. Depending on the parameter, the value might be a number or a vector. Lines beginning with a # are ignored. The input system recognizes the following parameters:

- TERRAIN_SIZE
- WATER_HEIGHT
- NUM_OCTAVES
- REDIST
- GROUND_TEXTURE
- TREE_DENSITY
- GRASS_DENSITY
- GRASS_COLOUR
- LEAF_COLOUR
- SKYBOX_NAME
- SUN_DIRECTION
- SUN_COLOUR
- SUN_INTENSITY

- AMBIENT_COLOUR
- CAMERA_POS
- CAMERA_FRONT
- CAMERA_PITCH
- CAMERA_YAW
- CAMERA_SPEED
- LSYSTEM

The LSYSTEM parameter takes an integer n (n >0) that specifies the number of production rules associated with that L-system. The following n lines are then read as production rules.

Controls :

The program employs a standard first-person camera (controlled by the mouse) with the following keyboard shortcuts:

- W - Forward
- A - Left
- S - Backward
- D - Right
- Q - Down
- E - Up
- Z - Speed down
- X - Speed Up
- R - Reset camera
- P - print position information

In addition to the camera controls, the user may also alter some of the program parameters at runtime, and the results are immmediately applied. Here are the relevant keyboard shortcuts:

- 1 - Increase the Sun's X direction
- 2 - Decrease the Sun's X direction
- 3 - Increase the Sun's Y direction
- 4 - Decrease the Sun's Y direction
- 5 - Increase the Sun's Z direction
- 6 - Decrease the Sun's Z direction
- O - Increase the number of octaves used by terrain generation
- I - Decrease the number of octaves used by terrain generation
- G - Raise the water level
- H - Lower the water level
- M - Increase the distribution exponent used by terrain generation
- N - Decrease the distribution exponent used by terrain generation

Finally, the **B** key toggles display of the shadow map, and the **ESC** key quits the program.

Implementation:

The biggest objectives of this project are procedural generation of terrain with Perlin noise and procedural generation of trees with L-systems.

Procedural Generation of terrain using Perlin noise

Perlin noise was first developed in the 1980s by Ken Perlin for the movie TRON. Since then, it has seen widepsread use in many application, such as bump mapping, texture generation, and, in this project, terrain generation.

The terrain is a 2-dimensional grid of vertices, each with an x, y, and z coordinate. The y coordinate corresponds to height, and the noise function is used to assign y coordinates to all the vertices. By assigning appropriate values, the program can generate "realistic"-looking terrain (for some definition of realistic).

Perlin noise is a type of gradient noise. In a nutshell, you assign pseudo-random gradients at regularly-spaced points (the *noise lattice*) then interpolate a smooth function between those points. This can be done in any number of dimensions, but it made sense to use 2D for my purposes. That means that each gradient will be a 2D vector. In the case of Perlin noise, the gradient are uniformly spaced on the unit circle. To find the noise value at a point P, the gradients of the four closest points on the noise lattice are lineraly interpolated. This yields the noise value at P.

The implementation of Perlin noise has a few more nuances. The gradient of a point (x,y) on the lattice is chosen by using a hashing function on (x,y) to obtain an index in a permutation array. The permutation array contains two copies each of all the numbers from 0-255 (it could be higher, but most sources use 256), shuffled. The particular arrangement of the permutation array can be thought of as the "seed" for Perlin noise.

In the case of terrain generation, an extra step is taken to make the terrain more realistic. The resulting noise value at P is multiplied by two factors, *amplitude* and *frequency*. This process is then repeated, with *amplitude* and *frequency* halved and doubled each time, respectively. Each step is sometimes called an *octave*. This is repeated a certain number of times (usually 5 or 7), and the result is realistic-looking terrain. The lower octaves shape the low-level details of the terrain (global hills and valleys), whereas the higher octaves yield smaller, local features of the land.

You may find the relevant code in the following places:

- perlinnoise.cpp, perlinnoise.hpp
  The Perlinnoise class encapsulates the noise algorithm.
- terrain.cpp: init() function, lines 64-101
  The Terrain::init() function uses the Perlinnoise::noise() function to generate heights for the terrain.

Procedural Generation of trees with Lindenmayer systems

Lindenmayer Systems (L-systems) are a type of formal grammar that are commonly used to procedurally generate vegetation, because they lend themselves nicely to the self-similar structure of plants. They are distinguished from other grammars in that they use parallel rewriting (this helps with the self-similarity aspect).

3D models of trees can be generated from an L-system (which consists of a set of production rules) by interpreting the characters in the generated strings as instructions for a turtle in *turtle graphics*. The turtle, in this case, begins at a point in 3D space, and each character instructs the turtle to either draw a branch or change its direction. Branching is achieved through the use of bracketing; a [instruction tells the

turtle, "save your current position and orientation", and] means "return to your last saved position and orientation".

There are several classes of L-systems. The one used by this project is *bracketed* (explained above) and *deterministic*, meaning each non-terminating character is the LHS of exactly *one* production rules.

I have added variation to my trees by randomizing the divergence angle of each branch (within a specified range) and by varying the colours of leaves. The effect is that trees generated using an L-system will look similar, but not exactly the same (like tree species). More variation could be added by using *stochastic* L-systems, which are non-deterministic: a non-terminating symbol can be the LHS of several production rules, each with a certain probability of being chosen.

You may find the relevant code in the following places:

- lsystem.cpp, lsystem.hpp
  The LSystem class encapsulates the rewriting of production rules. The Rule struct encapsulates a single production rule.
- ltree.cpp. ltree.hpp
  The LTree class encapsulates a single tree generated from an LSystem.
- branchnode.cpp, branchnode.hpp
  The BranchNode class encapsulates the geometry of a single branch.
- A5.cpp: initFoliage() function, lines 716-736
  This section of the initFoliage function is where an LTree is generated from a randomly chosen LSystem.

Procedural Generation of trees with Lindenmayer systems

3D Nature Environment with Procedurally Generated Terrain and Trees

Name: Liam Ozog
User ID: lozog
Student ID: 20515121

Objectives

___ **1:** UI: Implement a first-person camera with associated controls to allow navigation of the scene, including movement in 3 axes, speed adjustment, and camera rotation.

___ **2:** Modelling: Add a skybox to the scene using cube mapping.

___ **3:** Implement reflections for water using OpenGL's stencil buffer.

___ **4:** Generate a pseudo-random terrain heightmap with Perlin noise.

___ **5:** Add grass to the scene using billboards to create the illusion of many blades of grass.

___ **6:** Add texture to the ground and foliage using texture mapping.

___ **7:** Use L-systems to procedurally generate trees.

___ **8:** Implement shadows using a depth map stored in an OpenGL frame buffer.

___ **9:** Implement bloom using framebuffers and Gaussian blur.

___ **10:** Objective ten. Use a "screen-door" effect to simulate alpha transparency for grass.