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.

You may view screenshots of the final project with various different input parameters in the **screenshots/final/** directory. You may also view screenshots of the program in the process of being developed in the **screenshots/progress/** directory.

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. Five sample input files, **inparams1**, **inparams2**, **inparams3**, **inparams4**, and **inparams3** have been provided. Each produces a unique scene.

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. However, I will talk about each of my objectives in order.

First-Person Camera

The camera is a fairly standard first-person camera. The controls section has more info on how to use it.

You may find the relevant code in the following places:

- A5.cpp: resetCamera() function, lines 327-336
  This function sets the default settings for the camera
- A5.cpp: mouseMoveEvent() function, lines 1115-1142
  This function updates the camera vectors (front, up, and right) which are used to control direction of motion, as well as the camera's pitch and yaw. The pitch is limited to a range of [-89.0f, 89.0f] to restrict the range of motion (and also to keep the sin and cos functions behaving nicely).

Skybox

The camera is a cubemap texture, which is specifically supported by OpenGL.

You may find the relevant code in the following places:

- A5.cpp: loadSkybox() function, lines 424-473
  This function opens the skybox texture files and loads them into OpenGL's GL_TEXTURE_CUBE_MAP.
- A5.cpp: drawSkybox() function, lines 859-872
  This function uses the skybox shaders to actually draw the skybox. It is necessary to disregard the translation component of the view matrix (line 865).
- Assets/skyboxVertexShader.vs, Assets/skyboxFragmentShader.vs

Water reflections

My original plan was to use OpenGL's stencil buffer to implement reflections of geometry (trees, terrain) on the water, but that proved to be problematic. Instead, I reflected the skybox on the water by passing the skybox texture to the water's fragment shader and calculating the reflection.

You may find the relevant code in the following places:

- Assets/waterFragmentShader.vs, lines 72-78
  Calculate the reflection
- A5.cpp: drawWater() function, lines 922-925
  Here, I pass the skybox texture to the water's fragment shader.

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.

Billboards

Billboards are 2D sprites that exist in a 3D environment and always face the camera. It’s much cheaper to draw clumps of grass with these than to model individual blades of grass. I also used billboards for the tree leaves. One billboard of each type is created, then instanced many times by drawing it many times in different poisitions.

You may find the relevant code in the following places:

- billboard.cpp, billboard.hpp
  The Billboard class encapsulates billboards.
- A5.cpp: drawBillboards() function, lines 979-1023
  Here, many instances of the billboards (one for grass, one for leaves) are drawn. The variation of leaf colour is also applied here.
- Assets/billboardVertexShader.vs, Assets/billboardFragmentShader.fs

Texture mapping

Textures are applied to the terrain, water, trees, grass billboards, leaf billboards, and skybox. In the case of the terrain, the terrain is blended depending on slope: flat ground is given a “ground” texture, and steeper ground is given a “cliff” texture. The ground texture image is passed in as a parameter, which can help create different environments (e.g. grass, snow, sand). The two textures are blended manually using a weighted sum calculated from the normal vector at each point of terrain.

You may find the relevant code in the following places:

- A5.cpp: loadTexture function, lines 370-394
  Opens a texture file and loads it into GL_TEXTURE_2D.
- Assets/FragmentShader.fs, lines 72-83
  Here, the ground and cliff textures are blended according to a weighted sum of the slope of the ground at each point. Each slope is compared (dot product) with the up direction (0,1,0). Some values have been scaled according to what I've found to look best.

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 fcrom a randomly chosen LSystem.

Shadows

Shadows are created by first rendering the scene from the direction of the light and storing the resulting depth values in an OpenGL Framebuffer object, creating the shadow map. The shadow map is then passed to the fragment shader, which checks each point to see if it's in a shadow. If it is, no diffuse or specular light is applied to that point (ambient light only).

You may find the relevant code in the following places:

- A5.cpp: initShadowMap function, lines 648-673
  Initializes a framebuffer object to be used as the shadow map, which will be written to a 2D texture that can be read by the shaders.
- A5.cpp: drawShadowMap function, lines 834-857
  Renders scene from light's point of view, then writes results to the framebuffer object. To render the scene from the light's POV, an orthographic projection matrix (which is calculated from the terrain size to make sure it covers everything) is created, as well as a new view matrix. Terrain and trees cast shadows, but water and billboards do not.
- Assets/shadowVertexShader.vs, Assets/shadowFragmentShader.fs
- Assets/FragmentShader.fs, lines 29-36, 68-69
  The fragment shader only adds diffuse & specular light to a vertex if it isn't in a shadow. A bias of 0.005 is used to eliminate shadow acne with minimal peter panning (when the shadow is offset from the object).

Bloom

I did not implement bloom.

Screen-door effect for billboards

Alpha-based transparency requires sorting, which is slow. Instead, the screen-door (also known as dissolve) effect is a cheaper approximation. Each billboard texture also has an associated screen-door texture. The screen-door texture matches the billboard's shape, but instead of colour, it has greyscale noise. The noise is interpreted by the shader as an alpha value (e.g. white is fully transparent, black is fully opaque). The shader then tests the value of the noise at each pixel; if it does not exceed a certain threshold, the pixel is discarded, and the billboard will be transparent at that point. The effect of this is that as billboards are further away, fewer and fewer pixels pass the threshold, and billboards "fade" away with distance.

You may find the relevant code in the following places:

- A5.cpp: loadTextureAlpha function, lines 397-421
  Identical to the loadTexture() function, but also loads the alpha channel.
- Assets/billboardFragmentShader.fs, lines 12-20
  Threshold tests for each fragment. screen.a checks against the outline of the texture, a cheap alpha test. Then, screen.r then checks against the value of the (greyscale) noise.

Acknowledgements

The function RenderQuad(), which can be found in A5.cpp on lines 806-832, was taken from learnopengl.com's tutorial on shadow mapping. It renders a single quad that fills the screen with a texture, and I used it to debug the shadow map. It isn't really used by the program itself, but is still used if you toggle the shadowmap debug mode.

The textures used in this program were all retrieved from opengameart.org. They were all made by various artists. Individual credits for each texture may be found in the readme file.

The FASTFLOOR function in perlinnoise.hpp is taken from Stefan Gustavson's paper on Simplex Noise (see first Perlin noise acknowledgement below)

Calculating the normal vector at a point on the terrain grid is done using a method provided by user Gigi on this Stackoverflow question: http://stackoverflow.com/questions/13983189/opengl-how-to-calculate-normals-in-a-terrain-height-grid

My understanding of Perlin noise is largely from the following sources:

- `http://staffwww.itn.liu.se/~stegu/simplexnoise/simplexnoise.pdf`
- `http://flafla2.github.io/2014/08/09/perlinnoise.html`
- `http://www.scratchapixel.com/lessons/procedural-generation-virtual-worlds/procedural-patterns-nois`
- `http://www.redblobgames.com/maps/terrain-from-noise`

I used the first two chapters of the book *The Algorithmic Beauty of Plants* and from chapter 5 of *Procedural Content Generation in Games* to learn about L-systems:

- `http://algorithmicbotany.org/papers/cabop`
- `http://pcgbook.com/wp-content/uploads/chapter05.pdf`

The websites `learnopengl.com` and `opengl-tutorial.org` were invaluable resources for learning about other various OpenGL features.

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