Walking Through a Forest Simulator 2017

The goal of this project is to generate a forest for the user to move through. The objects in the forest will be loaded into the program through a lua file that also specifies how dense they should be placed and the variances for how each instance of an object should be rotated and scaled. They objects don't necessarily have to be trees, for example a secondary scene of geometric shapes or a scene containing different models of people could be interesting to explore too.

The purpose here is to explore procedural generation of objects in a scene. In previous projects I disliked creating scenes in a lua file and constantly having to reload the program to check if I had created what wanted. For this project I wanted to make a program where the user is free to explore a scene, but I didn't want to have to actually make the scene. I think procedural generation is a good compromise.

Goals

1. **Object Generation**

   The program should take the inputs from a script and generate a scene to be rendered based on randomly varying the placement, rotation, and scale of the objects in the script. There should be a reasonable maximum radius of generation that puts the user reasonably within the centre of a "forest" while not generating too much to destroy performance. This objective is for generating the objects defined in the script, and objective 3 is for generating variations of the objects.

   The script should contain a list of objects and their variance specifications. The script should return a single list of objects. Each object will contain a density, name, .obj file, model transform, and rotation and scale variances.

   We define a function $gr.make\_list()$ that creates a list of objects. The list object will include a method $list : add\_object(object)$ that adds an object to the list.

   For objects, we define a $gr.make\_object(name, obj, density)$ function that creates an object with a specified name, .obj file, and density. Obj is the name of the object file. Density is a number that specifies how close together on average each instance of the object should be.

   We define these additional functions for creating objects:

   - $object : translate(\{x, y, z\})$ - translates the model transform
   - $object : rotate(axis, amount)$ - rotates the model transform
   - $object : scale(\{x, y, z\})$ - scales the model transform

   The program should generate multiple instances of every object in the script and display them in a scene on top of a basic ground plane that is common to every script as a "ground".

2. **Skybox**

   There should be an interesting skybox generated by the program. If there is a sun in the sky, the directions of the light and the sun should match.

3. **Variation Generation**

   The objects in the forest should be varied randomly according to these two extra functions on objects:

   - $object : set\_scale\_variance(\{x, y, z\})$ - sets the amount of variance in the scale. The provided object and model transform should be the smallest possible, and when this full transform is applied the new object will be the largest possible in x, y, and z directions.
   - $object : set\_rot\_variance(\{x1, y1, z1\}, \{x2, y2, z2\})$ - x, y, and z should be numbers in rads indicating the minimum and maximum rotation allowed on each axis-aligned axis of rotation.

   The objects should be rotated randomly in the provided axes. It should be noted that in a forest of trees, the y axis might be the only desired axis of rotation to keep the trees upright. The objects should also be scaled to have a final scale transform between {1, 1, 1} and the set scale variance amount.

4. **Camera Movement**

   There should be movement controls based on those in the CryEngine Map Editor. I have found these to be very powerful for moving in any direction on demand. The directions of movement are all based upon the direction the camera is currently facing. They are defined as follows:

   - WASD keys - Move the camera forward, backward, left, or right
   - Holding the right mouse button - Rotate the camera
   - Hodling the centre mouse button - Pan the camera up, down, left, or right
   - Scrolling the mouse - Move the camera forward or backward
   - Holding shift while performing one of the others - Move or twice as fast. This does not apply to rotations.

   Combinations of multiple actions at a time is left up to implementation to choose something that feels good to use.

5. **Texture Mapping**

   The program should load in linked textures from .obj files and correctly display them to screen. If an object does not have a linked texture then a simple white texture with reasonable lighting values should be applied instead.

6. **Phong Shading**

   A Phong shading model should be applied in the shader using vertex normals and normal interpolation across faces. This interpolation should take place on the GPU with the final shading applied to each fragment.

   The only light in the scene should be a directional "sun" placed at infinite distance from the scene. This sun will be common to every scene.

7. **Shadow Mapping**

   Shadow mapping should be used to create shadows below objects in the scene. The general strategy will have two phases.

   The first phase is an orthogonal rendering into just a depth buffer from the perspective of the sun. We only need to do this once because the sun will always be the only light in the scene, and it is orthogonal because the sun is at effectively infinite distance from the scene.

   The second phase happens in the fragment shader during lighting. We calculate a vector from the fragment in the direction of the sun and then sample the depth buffer from before. The idea will be to create a shadow if the object is not the object at the depth saved to the shadow depth buffer.

8. **Ambient Occlusion**

   The basic algorithm includes first taking samples from close neighbors and taking their difference in depth into account. We combine the samples attenuated distance to get a value of how much the particular fragment should be occluded.

   The references also include good practices and strategies to avoid artifacts and self occlusion.

9. **Crepuscular (God) Rays**

   This algorithm includes taking samples at each pixel, and then calculating how much light in the media would have been scattered by the sun into the eye instead, and whether the media would scatter any light.

   We sum many samples to give the final approximation of scattered light into the eye. We implement this as a postprocessing effect, where we render occlusion to an extra FBO, then take samples at all the locations that are not occluded.

10. **Fast Approximate AntiAliasing (FXAA)**

    This approach to antialiasing uses a postprocessing filter to blur aliasing edges. We begin by converting the RGB values into an estimate of luminance. We then check local contrast to exit early on non-edges. If we have found an area of high contrast we choose the greatest difference as the edge we are using. We then search for the end of the edge in opposite directions. We shift the position of our sampler, and then resample the given input, followed by blurring the scene depending on the amount of aliasing we have detected.

References

5. **Texture Mapping**

   A basic tutorial on texture mapping: `http://www.opengl-tutorial.org/beginners-tutorials/tutorial-5-a-textured-cube/`

7. **Shadow Mapping**

   A basic tutorial on shadow mapping: `http://www.opengl-tutorial.org/intermediate-tutorials/tutorial-16-shadow-mapping/`

8. **Ambient Occlusion**

   A description of an alternative method of Ambient Occlusion `http://www.derschmale.com/2013/12/20/an-alternative-implementation-for-hbao-2/`

   A description of some good values for the generally included tuning values in an AO algorithm: `http://docs.nvidia.com/gameworks/content/gameworkslibrary/visualfx/hbao/product.html`

   A description of issues with current algorithms and strategies to deal with common artifacts: `http://http.developer.nvidia.com/GPUGems3/gpugems3_ch12.html`

   A simpler explanation and tutorial on ambient occlusion: `http://www.gamedev.net/page/resources/_/technical/graphics-programming-and-theory/a-simple-and-practical-approach-to-ssao-r2753`

9. **Crepuscular (God) Rays**

   A low level and detailed explanation of how to calculate this type of lighting: `http://http.developer.nvidia.com/GPUGems3/gpugems3_ch13.html`

   A higher level explanation with more examples of how lighting should change properly: `http://fabiensanglard.net/lightScattering/`

10. **Fast Approximate AntiAliasing (FXAA)**

    Description of algorithm from conference: `http://iryoku.com/aacourse/downloads/09-FXAA-3.11-in-15-Slides.pdf`

    Further description of a more developed version of FXAA: `http://developer.download.nvidia.com/assets/gamedev/files/sdk/11/FXAA_WhitePaper.pdf`

Objectives

1. The program should take input from the script and produce multiples of every object specified in the script to generate a ”forest” of objects.

2. The program should generate an interesting non-trivial skybox.

3. The program should vary objects size and rotation based on the provided limitations in the script
4. The program should allow the user to move the camera using the movement controls outlined above
5. The program should use the textures specified inside object files, or place a default white texture when there is no linked texture.
6. A Phong shading model should be implemented with one directional light imitating a sun.
7. The program should produce shadows using a shadow mapping technique
8. The program should enhance shadows using the ambient occlusion techniques outlined above
9. The program should show crepuscular rays imitating the media refracting some but not all sunlight that passes through it.
10. The program should implement the Fast Approximate AntiAliasing (FXAA) algorithm.