A5: Final Project Documentation

Title: The Weeping Forest
Name: Philippe Demontigny
Student ID: 20557658
User ID: pdemonti

April 17, 2015

1 Overview

The Weeping Forest is a horror-survival game that combines elements of "Slender: The Eight Pages" with the weeping angels from Dr. Who. The game was created using the THREE.js API for WebGL, and runs in the browser. The goal is to turn on all of the lanterns scattered throughout the forest while avoiding the weeping angel. This angel, like the aliens from Dr. Who, takes the form of a statue, that only moves when the player is not looking in its direction. If it gets too close to the player without them noticing, the game ends, and the player must start over. Only by turning on all five lanterns can the angel be stopped. But be careful, as the more lights you activate, the faster the angel becomes...

2 Playing the Game

Game: phdemontigny.github.io/TheWeepingForest
Source: www.github.com/phdemontigny/TheWeepingForest

2.1 Supported Browsers

Currently The Weeping Forest will only run on one of the following browsers:

- Google
- Mozilla Firefox

The game has only be tested on Chrome v41.0.2272.118 and Firefox v36.0.4.

2.2 Controls

Use WASD or the arrow keys to move and the mouse to look around. When close to a lantern, use the mouse buttons turn it on and off.

Note: This game requires that your browser supports full-screen mode and pointer lock. Please ensure that this functionality is not blocked in your settings.

3 Implementation

3.1 THREE.js Overview

This game has been created primarily using the THREE.js JavaScript API, which greatly simplifies the process of creating 3D web games. This API is built off of WebGL, and its main functionality revolves around the following JavaScript objects:

1. A WebGL *renderer* is an object that communicates with the WebGL renderer. The render takes as input a scene object and a camera object.
2. A *camera* object stores all information necessary for placing the camera in world space and updating the view/perspective matrices.
3. A *scene* object is a display list implementation in WebGL. Each element added to the scene is a *mesh* object that takes two parameters: a geometry, and a material.
4. A *geometry* is an object that stores the vertices and normals for a given mesh. This geometry can either be a primitive (such as a box, sphere, or cylinder) or values loaded from an external source (such as an .obj file).

5. A *material* is an abstract shader program that controls the lighting of a mesh. There are many varieties, including PhongMaterial, LambertMaterial, and BasicMaterial.

To create a THREE.js scene, first a renderer object must be created and added to the HTML document. Then, once each mesh object has been created and added to the scene, this renderer can be called upon to draw the scene in the specified WebGL context. In the next section, we describe the various types of objects that are found in The Weeping Forest scene.

3.2 Scene Objects

- Skybox

  The skybox is created using the algorithm from [2]. It is modeled as a single, 1x1x1 cube that is centered at the camera, and drawn during a first render pass that uses a separate skybox_scene in which depth writing has been turned off. Texture mapping is used on the backside of each cube face to create a seamless night sky.

- Ground

  The ground is modeled as a single plane object. Texture mapping with repeat wrapping is used to add grass to the scene from a seamless texture.

- Flashlight

  The flashlight is modeled using a THREE.js Spotlight object. Details on the functionality of this object can be found in [3]. The spotlight takes an origin and a direction and creates a light source from the origin that expands outwards with a given angle of dispersion. To achieve the effect of a limited light source, the angle was set to the relatively low value of $\pi/9$. To make the flashlight always point in the same direction as the camera, during each render pass it is moved 1 unit in the opposite direction of the camera, and then made to look at the camera object. Since the light source is always extends away from the camera in this fashion, shadow mapping was turned off, as shadows shouldn't be visible to the player anyway.

- Lanterns

  The lanterns were modeled using a combination of 6 objects: four cylinders and two light sources. Two of the cylinders are used to create the pole and the top of the lantern, and these cylinders are given a metallic texture mapping. The other two cylinders are used to make the glass, but only one is rendered at a time. The first is made visible when the lantern is off, and uses a standard material lit using Phong Shading. The second is rendered when the lantern is on, and no shading calculations are performed on it (it's color is always set as the exact texture color). This gives the impression that the light is bright inside the glass when compared to the much darker background. Both glass cylinders are transparent, with the first having an alpha value of 0.2, and the second having an alpha value of 0.6. The same glass texture is applied to both objects.

  Two different light sources are used to add illumination to the scene when the light is turned on. The first is a spotlight that is placed at the top of the glass cylinder and directed straight downwards. Shadow mapping is applied to this light, and only the pole is set to cast a shadow. This creates the small circular shadow at the base of the lantern. However, this light source only creates a dim circular light on the ground. To create more ambient light, a single point light source with a limited projection radius is added as well.

- Trees

  Trees are modeled in one of two ways depending on how close to the tree the camera is. If the tree is far away, then it is rendered as a screen-aligned billboard [1]. These billboards are 2D plane objects that have a transparent texture of a tree mapped onto them. They are then rotated about the y-axis before each render pass to face the camera. This process creates the illusion that the trees are 3D even when they are not. This effect is particularly strong in The Weeping Forest because trees that are far away are not illuminated.

  When the tree is close to the camera, the full object mesh is rendered. Only a single mesh is used, but each instantiated tree is given a random uniform scale and a random rotation to distinguish them. This mesh is very large, and uses about 4000 vertices, so using the billboards at a distance helps to reduce the cost of rendering the full scene. A tree bark texture is applied to the mesh to give it a realistic appearance.

3.3 Game Loop

A typical game runs as follows:

1. First, a string representation of the map is generated. This string contains the following symbols, each of which correspond to a single scene object.

'T' -	denotes a tree object.
'L' -	denotes a lantern object.
'1' -	denotes the starting position of the angel.
'S' -	denotes the starting position of the player.
'.' -	denotes an empty region

2. The map string is parsed to create the scene objects and add them to the scene. For each object, a JavaScript object is also created which stores the following information:

   - The (x,z) world position of the scene object.
   - The rotation and scale of the scene object.
   - References to the scene objects (meshes, billboards, cylinders, etc.).

   Storing these JavaScript objects allows us to access and update the scene objects at any point during the game loop.

3. Finally, the main loop begins, in which methods are repeatedly called to process player and angel movement, sound effects, and object rendering.

4 Code Map

All contributed code can be found in one of the following three files:

- index.html (loads the game in the browser and imports required libraries)
- style.css (handles simple canvas formatting)
- main.js (contains all JavaScript code for running the game)

The code has been divided into four sections, each of which are outlined in the subsectons below. Note that the titles used in this manual correspond to the commented regions in the "main.js" file.

4.1 Setting Up The Scene

This section loads all of the geometry, materials, and lights needed for instantiating the scene objects. Functions for establishing pointer lock and fullscreen mode can also be found here.

4.2 Utility Functions

This section contains various small functions for performing distance calculations and handling player input. Mouse control functions can be found here, as well as code for testing whether or not a given object is visible on the screen.

4.3 Game Logic

The main functions for running the game can be found here. In addition to the various functions mentioned in the Game Loop section, code for handling collisions and instantiating objects can also be found here.

4.4 Menus and Animation

The final section contains functions for creating and drawing the title screen and running the introductory text animation. Also included in this section is code for running the "game over" animation.

5 Special Algorithms

5.1 Static Collision Detection

Static collision detection has been implemented for all scene objects and the camera. To do this, each type of object (angel, lantern, and tree) was given a radius value, which defined an implicit cylinder about each object. When a player moves the camera, the new location is calculated, and then tested against each object in the scene. if the distance from the camera to any object is less than its radius, the move is not completed. Otherwise, the move proceeds as normal.

5.2 Testing if an Object is on the Screen

This test was required both the check whether the angel was visible to the player, and to test whether the lantern was visible when the player was in range to turn it on. To accomplish this, the view and projection matrices were used to create a THREE.js frustum object, as explained in [3]. Then, a collision test is performed between the frustum and the object in question. If they collide, then it must be the case that the object is at least partially visible on the screen. Otherwise, the object is not visible to the player.

5.3 Television Static Effect

To create the gradual TV static effect that plays when the player is caught by the angel, the canvas element of an HTML page was employed. The canvas is an HTML context that is typically used for displaying graphics. However rendering to the canvas is very slow, and thus not suited for heavy graphics usage. Instead, THREE.js makes use of a WebGL context, which has speeds similar to that of OpenGL. However, the canvas is useful for simple 2D drawing, which makes it ideal for creating post-processing effects.

The canvas in this game is situated above the WebGL context, and thus any images drawn to the canvas will appear above the rendered THREE.js scene. When the angel gets too close to the player, the following algorithm is applied to create a static effect:

1. First, a timer is set to 10 seconds, and a probably value p is chosen based on the elapsed time of the timer.
2. Next, for each pixel on the screen, with probably p a random grayscale color is assigned to that pixel. If no color is chosen, the pixel is set to be the same color as the THREE.js rendered pixel.
3. This process is repeated for 10 seconds, with p increasing linearly from 0 to 1. After 10 seconds pass, the page is refreshed and the game restarts.

6 Acknowledgments

I would like the acknowledge the following internet sources and API libraries, without which this project would not have been possible.

1. THREE.js (threejs.org)
2. Snappy Tree and proctree.js (snappytree.com)
3. Buzz.js (buzz.jaysalvat.com)
4. Freesound (freesound.org)

References

[1] Tomas Akenine-Möller, Eric Haines, and Natty Hoffman. *Real-Time Rendering 3rd Edition.* A. K. Peters, Ltd., Natick, MA, USA, 2008.

[2] J. Dirksen. *Three.js Essentials.* Community experience distilled. Packt Publishing, 2014.

[3] J. Dirksen. *Three.js Cookbook.* Packt Publishing, 2015.