

Haunted House Ride – Final

Dennis Reimer

1. Control & Installation

Camera cannot be controlled. It moves automatically. The application can be exited with the “Escape” key. After the fade-out in the end the application closes automatically.

One can toggle between full screen and windowed mode with the “F” key.

In the submissions “Data” folder is the “settings.txt” file, where one can set **brightness, contrast, saturation, intensity of the main light and if vsync is on (1) or off (0)**.

When the source code is not compiling because of missing NuGet libraries, the packages have to be restored in the NuGet Manager.

2. Description

This demo is a short ride on a wagon through a spooky scenery. The ‘user’ is locked to a cart and automatically looks at point of interests. The scene can be pretty dark. So the brightness and light power values can be adjusted in the settings as explained in chapter 1.

Following effects are implemented. A description of the implementation can be found later:

- Basic Phong lighting model
- Vignette Postprocessing Effect
- Color Settings Postprocessing Effect
- Random Noise Postprocessing Effect
- Particle System for Smoke and Fire
- Fog
- ShadowMapping for Directional Light

3. Used libraries

The current demo uses the current OpenTK framework for OpenGL programming in C#.

(<https://www.nuget.org/packages/OpenTK/>)

The code is based on and extends my own Framework which I wrote some time ago.

(<https://github.com/Densen90/BasicEngine>)

For sound I use an implementation from Daniel Scherzer’s ‘Zenseless’ Framework, which uses the NAudio sound library.

(<https://github.com/danielscherzer/Zenseless>)

4. Implementation

a. Classes

SceneManager:

The SceneManager is the main class for holding all elements in the scene. It contains the ShaderManager (which itself holds all relevant shaders) and a list of all objects which should be rendered. It also has the ParticleSystems, the Lights and the Postprocessing object. In the corresponding methods, it initializes, and calls the render and update method of all objects.

ShaderManager:

The ShaderManager holds all relevant shaders of the program. A shader consists of a vertex and fragment shader code. It handles, that every shader only exists once during the runtime and can be shared by multiple 3D Objects.

MeshManager:

A singleton which holds all unique instances of a mesh object. The Model3D class (see later section) can either set its mesh via the MeshManager or load a new mesh from a file if it is not already loaded. So we do not need to load the same mesh multiple times.

Camera:

The camera delivers the view- and projection-matrix for the mesh to give it to the shaders. Depending on its location in the scene and its FOV, Aspect and Near- and Far-Plane it calculates the matrices every time the user makes an input.

For the projection matrix the method `Matrix4.CreatePerspectiveFieldOfView(..)` and for the view matrix the method `Matrix4.LookAt(..)` is used.

For automatic movement it can hold a Transform which it should follow and a list of targets with timestamps to look at. It counts the time and selects the next target to look at depending on the timecode. This is done smooth with Lerp and the matrices are recalculated every frame.

PointLight:

The PointLight class holds information about the location, intensity and color of a point light in the scene. The corresponding uniform ids are stored in every mesh for the corresponding point light and set for the uniforms inside the shader. It can also flicker by adjusting the intensity every frame.

Model3D:

The representation of a 3D object inside the scene. It holds a mesh, which is responsible for its rendering, as well as a transform, which holds information about the 3D orientation of the model inside the scene.

Some classes which have special implementation of some Model logic derive from this class (JumpScareModel.cs, RotatingSkeleton.cs, Wagon.cs)

Mesh:

The main class which holds all information of a model file/3D object. When created, it either loads it mesh from the MeshManager, or loads an .obj file from disk with the help of a MeshLoader.

Currently, the application can only load and apply one texture from the corresponding mtl file.

The MeshLoader was created with the help of online tutorials:

<http://neokabuto.blogspot.com/2015/04/opentk-tutorial-7-simple-objects-from.html> &

<http://www.opengl-tutorial.org/beginners-tutorials/tutorial-7-model-loading/>

After that, all uniform ids and buffers for all data, which needs to be sent to the shader are created.

When rendering, the model-view-projection matrix, depending on the given transform, is created with the values from the camera. Then it is told, which shader to use, all uniforms (lights, camera pos, matrices and textures) and the attribute buffers for vertices, normal and texture-coordinates (all read from the .obj) are set. Since an obj can consist of triangles and quads, these are drawn depending on their existence.

vertex2.glsl:

The default vertex shader. It calculates the world position of a vertex for the fragment shader. This is calculated with the model matrix. When the model is scaled, the vertex normal is the transpose of the inverse of the upper-left corner of the model matrix. Then the uv is set correctly. The v component is negated, because of different mapping of OpenGL and DirectX

(<https://stackoverflow.com/questions/44021587/opengl-texture-distortion-on-3d-model>)

fragment2.glsl:

The default fragment shader for the models. A structure is created for a PointLight. For every point light given, the lighting calculations for this fragment are done. These consist of a phong lighting model. Afterwards the fog is applied. The implementation of these effects can be found in chapter 4b.

PostprocessingEffect:

This class handles the postprocessing effects. When initialized, a framebuffer object is created. Also a Texture is created, in which we will render our scene and a render buffer is created as a depth buffer, because we are rendering 3D objects.

Before rendering our scene in SceneManager, the framebuffer is binded. After the scene is rendered, the framebuffer is unbinded and the rendered texture is used to have a postprocessing effect, which is written inside a glsl file. The vertex shader for this is always the same: **quad_vert.glsl** creates a rectangle in the size of the screen. **screen.glsl** implements all effects of the demo. The description can be found in chapter 4b.

For implementation, the tutorial <http://www.opengl-tutorial.org/intermediate-tutorials/tutorial-14-render-to-texture/> was used to look up the principles.

Particle System:

The particle system uses instancing to have many particles rendered without too much performance issues. All particles share the same four vertices.

A vertex buffer object is created for the vertex data, the position and the color of the particle.

An Array with the maximum number of particles holds all current particles. If a particle has a life

smaller than 0, it is currently not used.

In the update method, unused particles from this list are found and initialized to be used in the next frame. Here every particle gets a lifetime, how long it is visible, a speed with which it flies and a wind factor which influences the direction it flies. Also, its initialization color and size are set.

In the Render method (every frame) all particles are simulated to update their position. For every particle the Distance to the camera is calculated and the list is ordered by it. So, they are rendered in the correct order.

After this, all uniforms and attributes are set (vertex and world position, color, size...) and given to the shader. Then, specific for instanced rendering, the "rate at which generic vertex attributes advance during instanced rendering" (<https://www.khronos.org/registry/OpenGL-Refpages/gl4/html/glVertexAttribDivisor.xhtml>). So, every particle reuses the same vertex position, but has one position and one color per quad. Then all Particles are drawn (DrawArraysInstanced) and the AttribDivisors are set to their default values again, so other vertex-array-objects which are rendered later are not influenced by this.

For the implementation, the following tutorials were used:

<https://learnopengl.com/Advanced-OpenGL/Instancing>

<http://www.opengl-tutorial.org/intermediate-tutorials/billboards-particles/billboards/>

<http://www.opengl-tutorial.org/intermediate-tutorials/billboards-particles/particles-instancing/>

Shadow Mapping:

This class creates a depth texture of the rendered scene (similarly to Postprocessing).

For this every object in the scene (except the terrain and the tracks, because I don't want them to cast shadows to prevent unneeded shadows) is rendered with the depth shader (see chapter 4b) from the view of the main light of the scene. This implementation only uses one direction to create the depth map. To have moving shadows later, depending on the position of the main light, the light direction for the depth map is always set to the view direction of the main camera with an orthographic view. The size of the shadow map texture is set to 2048x2048 to have nicer shadows later. Also to avoid wrong located shadows (peter panning), the depth rendering happens with Front-Face-Culling.

After that, the scene is rendered again from the main cameras view with the standard shader. There the created depth map is given to this shader, to calculate shadows.

For the implementation, the tutorials from my lectures and additionally the following tutorials were used:

<https://learnopengl.com/Advanced-Lighting/Shadows/Shadow-Mapping>

<http://www.opengl-tutorial.org/intermediate-tutorials/tutorial-16-shadow-mapping/>

b. Effects / Shaders

Basic Phong lighting model:

Implemented in: fragment2.glsl

Source: Lectures; <https://learnopengl.com/Lighting/Basic-Lighting>

Consists of the combination of three lightings: ambient, diffuse and specular.

In line 39 is the ambient calculation is done.

Line 42-44 has the diffuse calculation depending on the light. It is calculated if the normal of the vertex is facing the light and how far the vertex is away from the light source. This way the lighting gets weaker the farther away the vertex is from the light.

Line 47-50 is the specular calculation. This depends on the view direction of the camera und creates a "reflective" effect. The higher the exponent of the pow method in line 50, the smaller and more focused the specular reflection is.

At last all lightings are combined and multiplied with the color of the vertex (texture) and the color of the light.

Vignette Postprocessing Effect:

Implemented in: screen.glsl

Source: Lectures;

Creates a vignette around the rendered texture (see 4a: PostprocessingEffect). The method is in line 24-28 and has the current uv, as well as a value when the vignette starts and ends to fade.

It calculates the distance from the center of the screen and the current uv-coordinate. With the smoothstep function it interpolates between where the vignette should start and where it should end depending on the distance.

Color Settings Postprocessing Effect:

Implemented in: screen.glsl

Source: Lectures; Unity Shaders and Effects Cookbook

Adjusts some color settings of the rendered texture (see 4a: PostprocessingEffect).

Brightness (line 30-34): simply multiply a factor to the current color.

Contrast (line 36-39): multiply contrast to current color (as with brightness) and add a value depending on the contrast value. For contrast < 1, the image gets brighter, For contrast > 1 the image gets darker again. For contrast=1 it does the same as brightness.

Saturation (line 41-50): It calculates the intensity of the black in the color with the luminance coefficient (like when calculating a greyscale image). Then it interpolates between the greyscale and color. With saturation = 0, the image is greyscale.

Random Noise Postprocessing Effect:

Implemented in: screen.glsl

Source: <https://thebookofshaders.com/10/>

Created in line 19-22. Creates a pseudo-random number between 0 and 1 (for black and white) with the help of some seeds (12,12; 67,67; 12345,12345). To generate movement, the global time of the running program is used inside the sin and fract method to have new values every frame

Particle System for smoke and fire:

Implemented in: particle_vert.glsl & particle_frag.glsl

Source: Lectures HS Ravensburg-Weingarten; Lectures TU Wien; <http://www.opengl-tutorial.org/intermediate-tutorials/billboards-particles/particles-instancing/>

The class implementation can be found in chapter 4a.

The vertex shader gets all information from the layout locations and generates its output values and the positions. Because it is a billboard, the position is multiplied with the view and projection matrix, but not with the model matrix. The orientation of the particle depends on the user's view, because it is always facing the camera.

The fragment shader simply applies the used texture to the particle. To have them also influenced by fog, the same calculations as in the fog calculations for the models are applied here.

Fog:

Implemented in: fragment2.glsl; particle_frag.glsl

Source: Lectures; <http://in2gpu.com/2014/07/22/create-fog-shader/>

Created in line 56-66 in fragment2.glsl. A color for the fog is set. For the horror scene the color black is used. Then the distance from the vertex to the camera is calculated. The fog should appear exponentially, so the distance is multiplied with a chosen density and exponentially calculated. The formula is: $f = \frac{1}{e^{dist^2}}$, where f is the clamped factor between 0 and 1, and dist is the distance multiplied with the density. The value f is then used to interpolate between the fog-color and the vertex color. So, the higher the distance to the camera, the more fog is seen.

Depth Map:

Implemented in: depth_vert.glsl; depth_frag.glsl

Source: <https://learnopengl.com/Advanced-Lighting/Shadows/Shadow-Mapping>

Nothing much happens here. The gl_position is set based on the mvp matrix of the light from which the scene is rendered. Since only the depth of the scene is rendered, we don't need to do anything in the fragment shader.

Shadow:

Implemented in: fragment2.glsl

Source: Lectures, <https://learnopengl.com/Advanced-Lighting/Shadows/Shadow-Mapping>

To calculate shadows, we use the shadow map. The light-space position is given from the vertex shader (same as in the depth map shader).

In line 33-36 we transform this position to a range from [0;1], so we can make texture lookups for the depth map and get the depth for the current fragment.

For a smoother shadow, we don't only sample the texel where we are currently, but also some surrounding one. In the end, to get the median shadow value, we divide the accumulated shadow value by the total texture lookups.

The decision, if the fragment is in a shadow, can be found in line 48&49. We make a texture lookup of the depth map and compare it to the depth of the current fragment. If it is smaller, it is in the shadow. We also subtract a small bias from the current depth to avoid samples beneath the meshes surface and generating a shadow line pattern.

5. Test Environment

The demo was written on a notebook with the following specifications:

Asus ROG GL502VS

Intel Core i7-6700HQ , Quad-Core, 2,6GHz

16 GB RAM

NVIDIA GeForce GTX 1070

On this machine, the current demo runs with 60 fps;

6. Model and Sounds

All 3D-models are downloaded from the websites <https://www.turbosquid.com/> and <https://free3d.com/>. When needed, the models were edited in the modeling tool Maya.

The used sounds have the following source with the named creator:

- Scary Scream - <http://soundbible.com/1548-Scary-Scream.html> - rutgermuller
- Horror background - <https://www.reverbnation.com/forrestwilson4estbestproduction/song/16187334-creepy-halloween-horror-music-box> - Forrest Wilson

7. Screenshots



