LiteWQ Report

Zhu Siyuan, Zhang Zheng

2023-01-01

1 Introduction

WolfQuest is a wolf simulation game, developed by eduweb. It is especially popular among teenagers. It strives to provide a realistic and immersive simulation of wolf life in Yellowstone National Park.

We want to create a lightweight version of WolfQuest, called **LiteWQ**. LiteWQ aims to deliver decent wolf simulation experience. It will be a very simple game, with only a few features of original WolfQuest. It will be an excellent final project for Computer Graphics course.

2 Features

LiteWQ offers the following features:

- powerful and robust OBJ parser and renderer
- include trees, bushes, grass and wolf model
- scent visualization with glowing effect and wind
- real-time shadow mapping
- real-time collision detection
- experimental SSAO support (not usable yet)
- generate terrain with height map and navigable

2.1 OBJ Parser and Renderer

OBJ parser support the following features:

- vertex, normal, texture coordinate
- mesh group
- material

The whole parsing process is quite simple thanks to the clean grammar of Wavefront `.obj` and `.mtl` file. In our testing cases, we can parsing the triangular mesh model files directly exported from Blender.

Some impressive difficulties include:

- Mutiple objects or smooth groups (keyword `o` and `s`) in a same .obj file. We solve this problem with a faces collection class `Geometry`. A `OBJParser` can produce mutiple Geometry.
- MTL libraries and `usemtl` keyword is loosely connected with faces and vertices. We design a FSMlike mechanism and postpone the MTL parsing.

In rendering part, our key class is `TriMesh`, holds the global vertex and indices information for sub-meshes (objects) in a single .obj file. After parsing is finished, all the vertex and index will be flattened into a large global array which is **directly** used for OpenGL `glBufferData` and `glVertexAttribPointer`. We will record the offset and number of indices in the global indices array for each sub-mesh. We firmly believe this lowering the cost of `VAO` switching and thus boosting the performance.

Later, each submesh will have a MTL library guided material parameters or a default material. They are used to create a wrapped `PhongMaterial` class for each submesh.

Remark: Wavefront .obj allow mutiple UV index and normal vector for a single vertex, which has no directly mapping method in GL. We duplicate the vertex as the work-around.

2.2 Scent Visualization

There are two kinds of scent in this game: grounded and airborne. Grounded scent is also known as trail, which is simply solid spheres. Grounded scent will emit floating scent randomly. The scent particles flow with wind. The scent particles will fade out after a while.

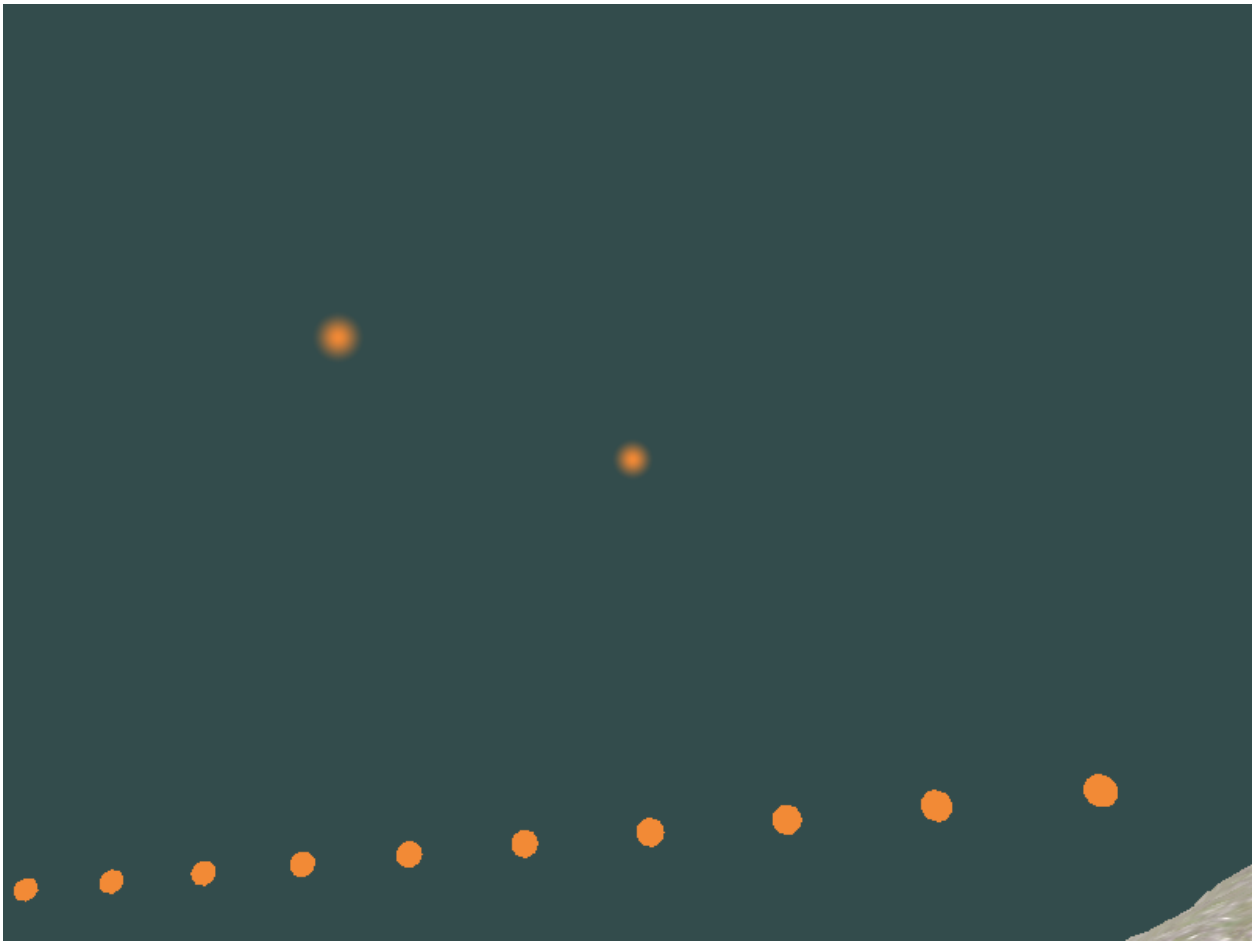

Figure 1: Scent Visualization

The scent particles are semi-transparent spheres, rendered with alpha blending in distance order. The gradual change in alpha is implemented in fragment shader, which computes the distance between the sphere center and fragment position **on screen space**. The fragment shader code is as follows:

```
#version 330 core
out vec4 FragColor;

uniform vec3 center;
uniform vec3 outer;
```

```
void main()
{
    vec3 color = vec3(0.95, 0.54, 0.21);
    vec2 uv = gl_FragCoord.xy - center.xy;
    float dist = length(uv);
    float radius = length(outer - center);
    float alpha = 1.0 - smoothstep(0, radius, dist);
    FragColor = vec4(color, alpha);
}
```

2.3 Real-time Shadow Mapping

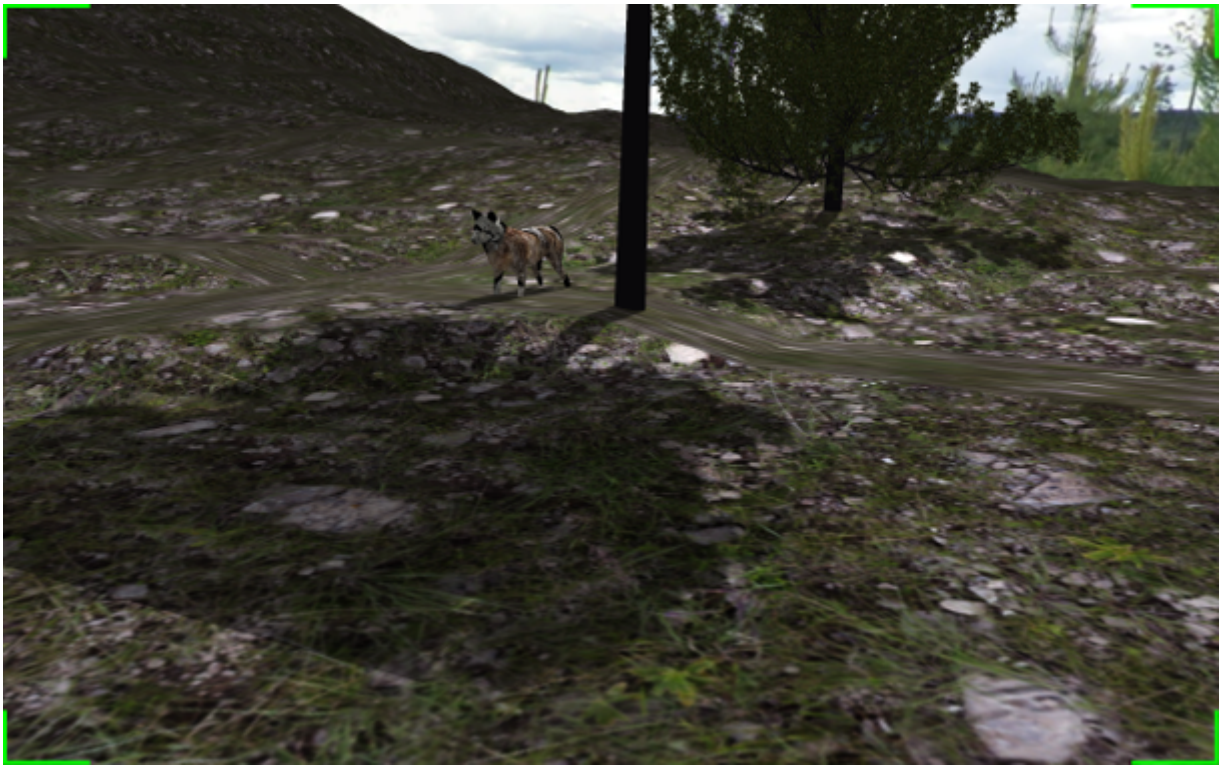

Figure 2: Shadow Mapping

Shadow mapping is a technique to render simple hard-edge shadow. Referring to LearnOpenGL, with Bling-Phong lighting model.

This is done by delay rendering. First we make transformation from World to Light, then rendering into a frame buffer. We then discard the color and leave the depth info. This depth mapping will saved in a texture bound to frame buffer. In the next step, we render to screen frame buffer and utilize the depth texture to draw shadows.

We adapt the `shadow bias` tricks for anti-aliasing.

2.4 Real-time Collision Detection

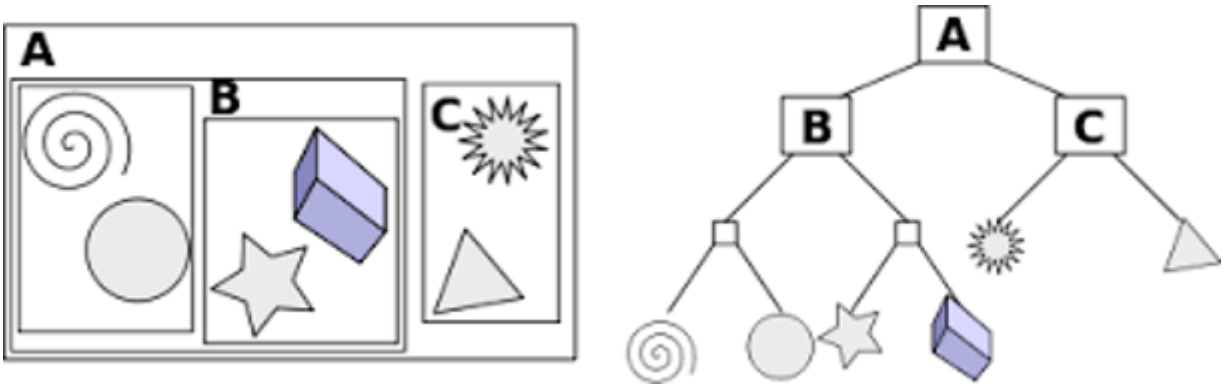

Figure 3: BVH Tree

We choose BVH rather than Quadtree or Octree for the original purpose for acceleration ray-tracing. But we soon found that OpenGL shader pipeline seems not compatible with BVH (CUDA or OpenCL is better). So we utilize BVH for collision detection.

In advance we implements a `AABB` bounding box class and its utility functions. And geometry primitives type `Shape`. We must point out again that ray-tracing is very awful to implement as OpenGL shader prefer plain vertices and indices buffer data to draw triangles but ray-tracing prefer strutuced data which recording the detailed information of each primitive rather than vertices. This lead to a situation that we often are bothered to transform the same data between different forms.

Once we imports a 3D model, we are prepared to build BVH for it as the pipeline below:

- Transform the OpenGL vertex and index information into `Triangle` primtives.
- Recusively build BVH with SAH optimization in the following steps.
- For each dimension split into $\mathbf{B}(=12)$ buckets, and choose the best split position with SAH cost estimation.
- After get the best dimention and its split position, sort the primitives by this dimension. Generate left-hand side primitives and right-hand side primitives, then build BVH recursively.

In collision detection:

- From root of BVH, if the bounding box not intersect, means the 3D model not hit the camera.
- Otherwise, recusive into left and right sub-BVH.

In pratice, we build BVH for our wolf model with 200,000 triangles. The game can deal with the collision smoothly without much fps cost.

2.5 Experimental SSAO Support

Screen Space Ambient Occlusion (SSAO) is a technique to simulate ambient occlusion in screen space. It is a very popular technique in modern game engines. SSAO works together with deferred shading. We heavily referenced LearnOpenGL to implement SSAO in our game. However, as of now, SSAO is not usable in our game.

The SSAO algorithm is as follows:

- Render the scene to a G-buffer, which contains position, normal and color information of each fragment.
- Also render the scene to a depth buffer.
- Compute ambient occlusion factor for each fragment in screen space with random samples.
- Apply blur to the ambient occlusion factor to reduce “banding” effect.
- Apply lighting to the scene with ambient occlusion factor.

Here is a screenshot visualizing occusion factor:

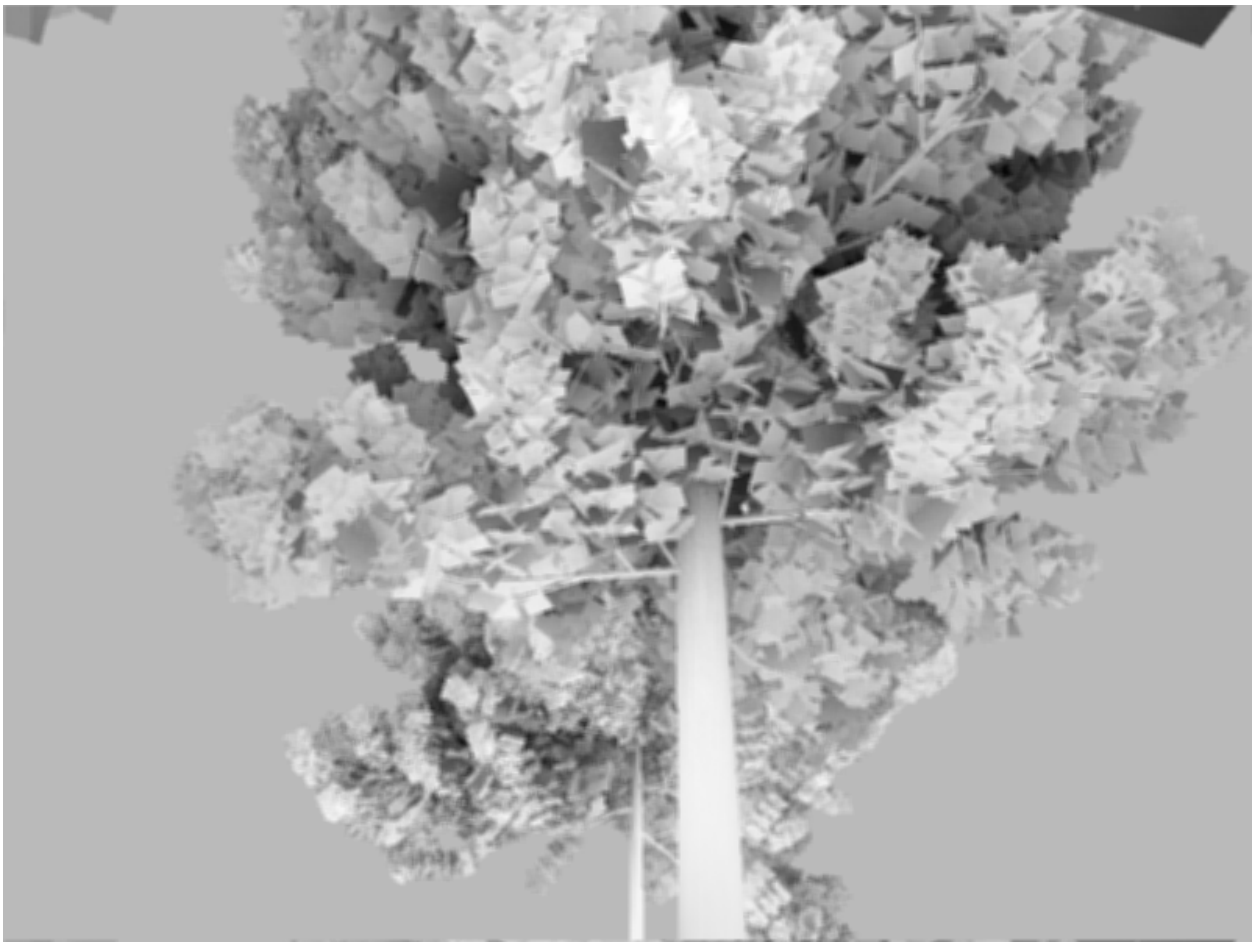

Figure 4: SSAO

It seems pretty good, but somehow when combined with lighting, the result is simply garbage. We are still working on it.

2.6 Generate Terrain with Height Map

Given a bitmap with varying color on various height, we can generate a terrain with it. Simply create a mesh with vertices on the bitmap, and use the color as the Y coordinate. X and Z coordinates are mapped to the bitmap coordinates. The mesh is pre-computed on CPU with uniform grid. This can be improved by utilizing GPU tessellation.

Moreover, to navigate on the terrain, we don't necessarily need to use collision detection. Instead, we can simply rely on the height map, and compute the Y coordinate with bilinear interpolation on the fly. This is much faster than collision detection. The camera will be constrained to the terrain, and will not be able to fly.

Figure 5: Generated Terrain

3 Conclusion

We mainly refer to LearnOpenGL to implement the features of LiteWQ. The game is still not complete yet, so we can work on proper SSAO, first and third person camera, and more gameplay features. We will also try to improve the performance of the game.

In the proposal, we mentioned that we will create a mobile version, but due to the lack of time, we only attempted it at an early stage and abandoned it. We will also try to create a mobile version in the future.