



TECHNISCHE  
UNIVERSITÄT  
WIEN

# Fireflies

## Final Report

Master programme Visual Computing

Real-time Rendering

### Group Elements:

João Francisco Carvalho Neto - e1637651 - joao.neto.feup@gmail.com

João Pedro Miranda Maia - e1637652 - jpmmaia@gmail.com

16th January 2017

# Contents

<b>1</b>	<b>Project Description</b>	<b>3</b>
1.1	Technical Specifications . . . . .	3
1.2	Scene Description . . . . .	3
1.3	Features and Effects . . . . .	3
1.3.1	Lighting . . . . .	3
1.3.2	Frustum Culling and Instancing . . . . .	4
1.3.3	Terrain rendering . . . . .	4
1.3.4	Dynamic Cube Mapping . . . . .	5
1.3.5	Normal Mapping . . . . .	6
1.3.6	Specular Mapping . . . . .	6
1.3.7	Shadow Mapping . . . . .	6
1.3.8	Skydome and clouds . . . . .	7
1.3.9	Fog . . . . .	8
1.3.10	Billboarding . . . . .	8
<b>2</b>	<b>Resources used</b>	<b>8</b>
<b>3</b>	<b>Bibliography</b>	<b>8</b>

# 1 Project Description

## 1.1 Technical Specifications

This project is implemented using the following *APIs*:

- *DirectX11* for rendering;
- *Win32 API* to handle events of the operating system and the creation of windows;
- *DirectSound* for playing *WAVE* files;
- *DirectInput* for handling user input (mostly for testing purposes);
- *DirectXMath* for common linear algebra and graphics math operations;
- *DirectXTex* [Mic] for loading texture files;
- *nlohmann/json* [Loh] for writing and saving scene related information (such as a scene description and animations) on files;
- *RapidXml* [Kal] for writing and saving settings related with the application and chosen hardware (for example, which graphic card to use);
- *Assimp* [Sch+] for loading 3D models from *fbx* files.

The release was tested on a desktop with a NVIDIA GeForce GTX 960. The application runs in fullscreen mode. To exit it, simply press the ESC button.

## 1.2 Scene Description

The scene is composed of a procedurally generated mountain range and contains trees and foliage.

## 1.3 Features and Effects

On this section, we present a list of features and effects we implemented.

### 1.3.1 Lighting

The *lighting model* is implemented as a modified version of the *Blinn-Phong* shading as described in [Lun16, Chapter 8].

The *specular lighting* is calculated using a *Fresnel factor* and a *roughness factor*. The *Schlick's approximation* is used to approximate the *Fresnel factor*. To calculate the *roughness factor*, the *microfacet model* described in [Lun16, Chapter 8] is used.

The lighting model supports *directional*, *point* and *spot lights*.

### 1.3.2 Frustum Culling and Instancing

*Frustum culling* refers to rejecting entire groups of triangles from further processing that are outside the viewing frustum with a simple test [Lun16, Chapter 16].

Instancing refers to drawing the same object more than once in a scene, reducing significantly the overhead of the API [Lun16, Chapter 16].

In our implementation, every model has a bounding box in local object space. For each instance of that model, we transform the viewing frustum to the local space of that instance:

$$view\_to\_local\_space = view\_matrix^{-1} \cdot world\_matrix^{-1} \quad (1)$$

Then we check if the transformed viewing frustum intersects the bounding box of that instance. If it does, then we mark that instance to be drawn. For drawing using the hardware instancing provided by DirectX11, we use the method *ID3D11DeviceContext::DrawIndexedInstanced*.

### 1.3.3 Terrain rendering

We implemented a terrain rendering which uses the tessellation shaders to provide Level of Detail based on the distance to the camera.

#### Terrain resources creation on CPU

In the first place, we read the height map values from a 16-bit file. As each value has 16-bits, it means that the height can have 65536 different values. Then, we calculate the normal and tangent maps using the values read from the height map, by calculating the partial derivatives using the central differences method:

$$pixel(x, z) = (x, height(x, z), z) \quad (2a)$$

$$\frac{\partial height(x, z)}{\partial x} \approx \frac{height(x + 1, z) - height(x - 1, z)}{2} \quad (2b)$$

$$tangent(x, y, z) = \frac{\partial pixel(x, y, z)}{\partial x} = (1, \frac{\partial height(x, z)}{\partial x}, 0) \quad (2c)$$

$$\frac{\partial height(x, z)}{\partial z} \approx \frac{height(x, z + 1) - height(x, z - 1)}{2} \quad (2d)$$

$$bitangent(x, y, z) = \frac{\partial pixel(x, y, z)}{\partial z} = (0, \frac{\partial height(x, z)}{\partial z}, 1) \quad (2e)$$

$$normal(x, y, z) = tangent(x, y, z) \times bitangent(x, y, z) \quad (2f)$$

On [Lun12, Chapter 19], they decided to calculate the normal and tangent directly on the shader. However, we decided to precompute the normals and tangents on the CPU, and store the results on the normal and tangent maps which are then sampled by the terrain pixel shader.

As described in [Lun12, Chapter 19], we created a quad mesh which provides the control points necessary to feed the input assembler stage and draw the terrain.

## Shaders implementation

The terrain Vertex shader is just a normal pass-through shader. The control points are passed to the Hull shader. The Hull shader computes the tessellation factors based on the distance to the camera [Lun12, Chapter 19]:

$$\text{blend\_factor} = \frac{\text{min\_tessellation\_distance} - \text{distance\_to\_eye}}{\text{min\_tessellation\_distance} - \text{max\_tessellation\_distance}} \quad (3a)$$

$$\text{tessellation\_expoent} = (1 - \text{blend\_factor}) \cdot \text{min\_tessellation\_factor} + \text{blend\_factor} \cdot \text{max\_tessellation\_factor} \quad (3b)$$

$$\text{tessellation\_factor} = 2^{\text{tessellation\_expoent}} \quad (3c)$$

These values are fed into the Tessellation shader which tessellates the geometry according to the tessellation factors. In this case, the geometry near to the camera will become more tessellated than that that is far away. This is an example of Level of Detail.

The Tessellation shader outputs vertices which are input to the Domain shader, which works as a vertex shader for the generated vertices. In this stage, we offset the position of the vertex in y component according to the value sampled from the height map.

On the pixel shader, we also implement procedural texturing, an idea based on [Ras, DirectX 11 Terrain Tutorials - Series 2 - Tutorial 13: Procedural Terrain Texturing]. We compute the slope at that pixel (which is basically one minus the y-component of the normalized normal vector). If the slope is high enough, the output color is a computed rock color. If the slope is low, then we interpolate it with computed grass or snow colors, depending on the altitude of the terrain at the given pixel.

Finally, to draw a path, we use a alpha map texture. These alpha values are used for interpolating the calculated color until that moment with the path color.

The computation of the shading of each type of material (rock, grass, snow and path) uses normal maps and specular maps which are described on sections 1.3.5 and 1.3.6, respectively.

### 1.3.4 Dynamic Cube Mapping

For implementing cube mapping, we guided ourselves by the lecture slides and by [Lun16, Chapter 18].

The whole process consists in creating a cube map texture by rendering the scene 6 times from the same position but in different directions, having a 90 degrees field of view. Therefore, whenever the object moves, 6 view matrices need to be created, one for each axis direction: +X, -X, +Y, -Y, +Z, -Z.

During the rendering process, the cube map texture needs to be bound as a Render Target. All objects are rendered into the texture, except the object itself. Then, the object itself is rendered, and the cube map texture is bound as a Shader Resource View and sampled in the pixel shader.

For sampling the cube map texture, we use the reflection vector which is commonly used in light shading calculations. Following the advice of [Lun16, Chapter 18], we multiply the sampled value additionally by the shininess and fresnel factor (which was also used in the lighting calculation, section 1.3.1). The shininess multiplication allows for the reduction of reflection for rough materials, whereas the fresnel factor multiplication determines how much light is reflected from

the environment into the eye based on the material properties of the surface and the angle of the reflection vector and the normal.

### 1.3.5 Normal Mapping

For the terrain, the normals and tangents are calculated from the height map as described in section 1.3.3. These vectors are then accessed in the pixel shader by means of a corresponding normal and tangent maps where the values were stored. Then, the texture normal map is sampled and the result is transformed from the range  $[0, 1]$  to  $[-1, 1]$ . Thereafter, the texture normal sample is transformed from texture space to world space by multiplying it with a orthonormal basis composed by the tangent, binormal and normal vectors of the terrain. The bitangent is calculated on the fly as the cross product of the normal and tangent. This implementation took as reference [Lun16, Chapter 19] and the lecture slides.

### 1.3.6 Specular Mapping

For implementing specular mapping, we simply multiply the specular component of light by the a specular factor which is sampled from a specular map.

### 1.3.7 Shadow Mapping

We implement shadow mapping with the help of [Lun12, Chapter 21] and the lecture slides. The first step is to render the scene's depth from the light's perspective and store that information. Then, when we render the scene from the camera's perspective, we project the light's depth texture and use it to compare the distance to the light, and find if the given pixel is in shadow or in light.

For each frame, we compute the view and projection matrices of the main light that casts shadows. These two matrices when multiplied together, along with homogeneous perspective divide, transform a point from world space to normalized device coordinates (NDC) space. The homogeneous perspective divide is performed inside the pixel shader, at the time the shadow factor is calculated. In NDC space,  $[x, y] \in [-1, 1]$ , therefore to transform to texture space,  $[u, v] \in [0, 1]$ , we just need to use the matrix described in equation 4.

$$ndc\_to\_texture\_space = \begin{bmatrix} \frac{1}{2} & 0 & 0 & 0 \\ 0 & -\frac{1}{2} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ \frac{1}{2} & \frac{1}{2} & 0 & 1 \end{bmatrix} \quad (4)$$

The y-coordinate is multiplied by -1 because the positive y-axis in NDC space goes in the opposite direction to the positive v-axis in texture space.

The shadow matrix is computed as a multiplication of the view and projection matrices of the main light and the `ndc_to_texture_space` matrix.

In the vertex shader, we multiply the world position of the vertex by the shadow matrix to transform from world space to the shadow texture space. When the homogeneous perspective divide is performed, these coordinates are transformed into the shadow map texture coordinates.

In the pixel shader we calculate a shadow factor. In the first place we apply the homogeneous perspective divide to the projective texture coordinates. These coordinates, which are used to sample the shadow map, usually don't coincide with a texel in the shadow map. As it is incorrect to

interpolate depth values, we use a method called *percentage closer filtering* (PCF). This method can be implemented via the `SampleCmpLevelZero` method that is supported by Direct3D 11+ graphics hardware. By setting the comparison function of the comparison sampler to `LESS_EQUAL`, this method samples 4 neighbor texels, checks if they are less or equal than the depth of the central texel and finally interpolates these results. On our pixel shader we perform a 4-tap PCF for each texel and its 8 neighbors to achieve a softer shadow. The shadow factor is computed as the sum of all 4-tap PCF divided by 9. This shadow factor is then multiplied by the result of the light intensity computed for the shadows source light.

### 1.3.8 Skydome and clouds

#### Skydome

The skydome implementation is based on [Ras, DirectX 11 Terrain Tutorials, Tutorial 10: Sky Domes]. It involves rendering a sphere centered on the scene camera, with a rasterizer state set to not cull faces. Since we want to see everything in the scene, and not just what is inside the sphere, we turn off the depth buffer, render the dome, turn it on again, and render the remaining objects.

The vertex shader consists in positioning the sphere so that its center corresponds to the position of the camera. As the sphere model is centered in the origin and has a radius of 1, its coordinates vary between -1 and 1. Therefore, we compute a blend factor as equation 5, in order to have the value in the range [0, 1].

$$blend\_factor = \frac{local\_position + 1}{2} \quad (5)$$

In the pixel shader, we simply interpolate two given colors using the blend factor calculated in the vertex shader.

#### Clouds

The clouds were implemented similarly to [Ras, DirectX 11 Terrain Tutorials, Tutorial 12: Perturbed Clouds].

First, we load a 3D model of a curved plane we previously created on Blender.

Then, on the vertex shader, the plane is positioned so that its center corresponds to the position of the camera, as done before with the skydome.

On the pixel shader, two textures are used: one representing clouds and another perlin noise. The effect consists in sampling the noise texture first. Then, this noise sample is scaled by a function (6) which varies through time. This way, the clouds slightly vary in shape through time.

$$scale\_noise\_value = 0.1 + 0.4 \cdot \cos(0.00005 \cdot total\_time\_ms) \quad (6)$$

After that, the noise value is added to the original texture coordinates (which are also translated through time), resulting in perturbed texture coordinates. These are used to sample the clouds texture.

Finally, the sampled color is decreased by half, as we are using additive blending to blend the clouds with the background skydome color.

### 1.3.9 Fog

Fog was originally implemented as described in [Lun16, Chapter 10]. However, the implementation suffered significant changes to be compatible with other effects.

The fog intensity is value that varies between 0 and 1 and is calculated as described in equation 7.

$$fog\_intensity = \frac{distance\_to\_eye - fog\_start}{fog\_range} \quad (7)$$

One optimization we're able to perform in the pixel shader is to return as soon as possible the color of the fog if the fog intensity is greater than 0.99.

We use the alpha channel of the fog color as a measure of how dense the fog is. Since we also implemented shadows, we multiply the shadow factor by  $1 - fog\_color_{alpha}$ . This way, when the fog is very dense, the shadow factor is low and consequently the scene is darker and the shadows are not so noticeable.

Finally, the final color of the pixel is the result of the interpolation between the calculated color and the fog color using the fog intensity value.

Another optimization we do is to adjust the far plane of the perspective matrix in order to frustum cull objects occluded by a dense fog.

### 1.3.10 Billboarding

For implementing billboarding, we based on [Lun16, Chapter 21]. In the first place, we store the center and extents as a single vertex in a vertex buffer. Therefore, we set the primitive topology to be a point list. The vertex shader is a simple pass-through shader. In the geometry shader, we expand the single point to a rectangle. As we are rendering foliage, we also rotate the billboard (in local space) around the Z-vector, to give it the effect of wind. The pixel shader is equal to the standard shader we use for every other object.

## 2 Resources used

The tree model was downloaded from [ala].

The height map was download from [Ras, DirectX 11 Terrain Tutorials - Series 2 - Tutorial 8: RAW Height Maps].

The cloud and noise textures were download from [Ras, DirectX 11 Terrain Tutorials, Tutorial 12: Perturbed Clouds].

The foliage billboards were download from [reinerprokein].

All other textures were downloaded from the Total Textures Repository.

The background music is the Cloud Atlas: Original Motion Picture Soundtrack, composed by Tom Tykwer, Johnny Klimek & Reinhold Heil.

## 3 Bibliography

[Mic] Microsoft. *Microsoft/DirectXTex DirectXTex texture processing library*. URL: <https://github.com/Microsoft/DirectXTex>.



- [Loh] Niels Lohmann. *nlohmann/json JSON for Modern C++*. URL: <https://github.com/nlohmann/json>.
- [Kal] Marcin Kalicinski. *RapidXml*. URL: <http://rapidxml.sourceforge.net/>.
- [Sch+] Thomas Schulze et al. “Open asset import library (assimp)”. In: *Computer Software*, URL: <https://github.com/assimp/assimp> ().
- [Lun16] Frank Luna. *Introduction to 3D game programming with DirectX 12*. David Pallai Mercury Learning and Information, 2016.
- [Lun12] Frank Luna. *Introduction to 3D game programming with DirectX 11*. David Pallai Mercury Learning and Information, 2012.
- [Ras] RasterTek. *Raster Tek*. URL: <http://www.rastertek.com/>.
- [ala] alanthegamer. *tree\_alan-2 - 3d model - fbx*. URL: <http://tf3dm.com/3d-model/tree-alan-2-68600.html>.