# Real-Time Graphics Programming: Deferred Rendering in Vulkan

Manuel Pagliuca - 975169
October 19, 2022

**Abstract**
The following paper will abstractly describe how it implemented a **deferred renderer** using the **Vulkan** graphic API for the course project of *Real-Time Graphics Programming* started on 13/03/2021, @UNIMI, A.Y. 2020/2021.

## 1 Introduction

The **deferred rendering** is a rendering technique that involves two GPU pipeline passes, the aim of that it's to obtain a strong optimization for the computation of the lights in a scene (it was first suggested by Michael Deering in 1988 [4]).

This technique involves the use of **MRT** (Multiple Render Targets), which are **auxiliary buffers** for the storing of the results from the first *fragment shader*. This is done during the first pipeline pass, which is also called **geometry pass**, on these auxiliary buffers are saved information like *depth, positions, normals, albedo, ...*.

The second pipeline pass, which is also called **lighting pass** will access these buffers which contain these different textures of the scene computed in the first pipeline, afterward will apply the lighting computations only over these fragments instead of calculating that for all the objects (even the not visible objects) in the scene.

The **advantage** of using this technique is given by the fact that the lighting computations are only done for the fragments that will actually go to the screen, so you can use the lights **intensively** in the scene.

The **disadvantage** is that if I use this technique in a scene with low lighting computations I will use two pipelines for a too simple a calculation, and therefore the rendering technique will prove counterproductive (i use two pipelines when I can do everything with one).

## 2 Development tools

### 2.1 Vulkan

**Vulkan** [5] is the latest cross-platform API for 3D graphics and computing released by the Khronos Group on 16 February 2016. It is considered the successor of **OpenGL** [3], and its aims are to provide high-performance 3D graphics applications (like video games and interactive media). Unlike the predecessor, writing an application that uses this API is much more verbose since it leaves you complete possibility and responsibility towards the graphics card.

### 2.2 Visual Studio 2019

The Microsoft IDE[6] is an extremely powerful tool for the development of a graphic application thanks to an infinity of tools that it provides (*debugger, IntelliSense, extensions, Nsight, ...*).

### 2.3 External libraries

- **GLFW**, *Graphics Library Framework* [10] - For providing a window and a surface to Vulkan.
- **GLM**, *OpenGL Mathematics*[2] - Provides essentials data structures and functions for dealing with 3D math.
- **PCG**, *A Better Random Number Generator* [9] - For generating random numbers for initial light colors and positions.
- **stb image**, *a reading and writing images library* [7] - For loading external textures.

### 2.4 NVIDIA Nsight Graphics

**NVIDIA Nsight Graphics** [8] is a standalone developer tool that enables you to debug, profile, and export frames built with Direct3D (11, 12, DXR), Vulkan 1.2, OpenGL, OpenVR, and the Oculus SDK. At the moment the Nsight Graphics debugging tool it's integrable on Visual Studio through an apposite extension.
I used the **framebuffer debugger** during the implementation of the deferred rendering for checking if the state of the auxiliary buffers was correct.

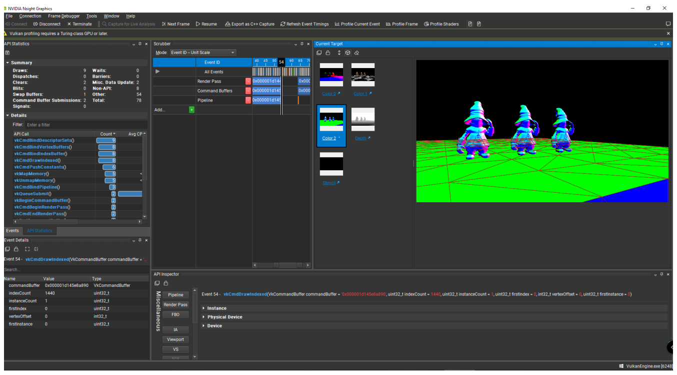


Figure 1: Frame Debugger from NVIDIA Nsight Graphics (Visual Studio extension)

## 3 Scene

The scene it's composed of three models of Vivi Orunitia (a character from *Final Fintasy*) and a floor with a chess pattern.

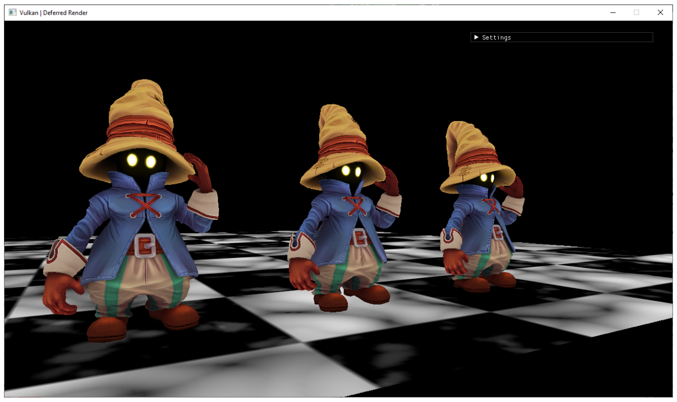


Figure 2: The scene

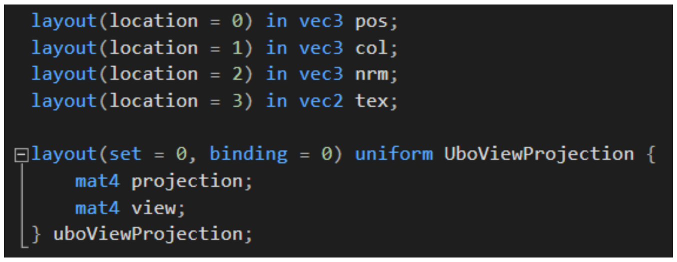
## 4 Implementation

### 4.1 First pipeline

#### 4.1.1 Vertex Shader

This will receive the following external resources :

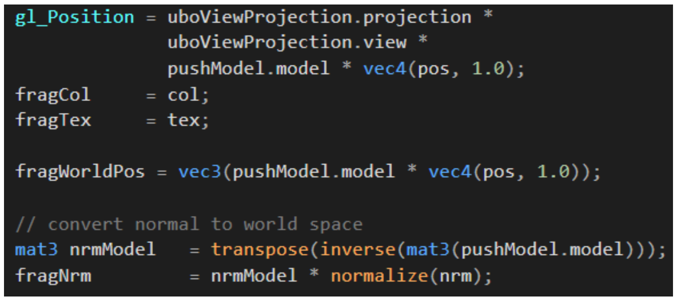
- The *colors, normals, texels and position* vectors (from *vertex input*).
- An **Uniform Buffer Object** containing a struct with the *view matrix* and the *projection matrix*.
- The *model matrix* as **Push Costant** (inside a struct).



```
layout(location = 0) in vec3 pos;
layout(location = 1) in vec3 col;
layout(location = 2) in vec3 nrm;
layout(location = 3) in vec2 tex;

layout(set = 0, binding = 0) uniform UboViewProjection {
    mat4 projection;
    mat4 view;
} uboViewProjection;
```

Figure 3: *Vertex input* and *UBO* of the first pipeline *vertex shader*

It calculates the correct position of the input vertices using the correct ordering of the *projection, view* and *model* matrices, the result is stored inside the **gl_Position** variable.
The texels (*tex*) and the colours (*col*) are directly passed to the *fragment shader*. Even the positions (*pos*) are passed to the *fragment shader* but they have to be converted to **world space** with the *model matrix* first. A similar computation happens for the normals (*nrm*) too, we need to



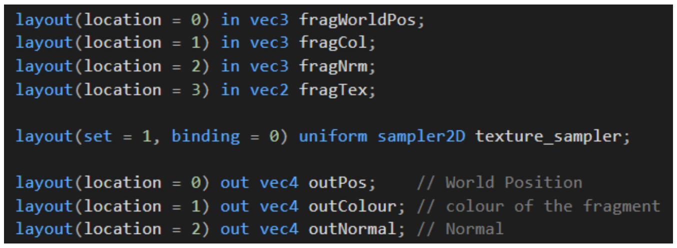
```
gl_Position = uboViewProjection.projection *
              uboViewProjection.view *
              pushModel.model * vec4(pos, 1.0);
fragCol     = col;
fragTex     = tex;

fragWorldPos = vec3(pushModel.model * vec4(pos, 1.0));

// convert normal to world space
mat3 nrmModel    = transpose(inverse(mat3(pushModel.model)));
fragNrm          = nrmModel * normalize(nrm);
```

Figure 4: Computation of the first *vertex shader*

convert normals to world space, but we have to use a different *model matrix* which is the **transpose** of the **inverse** of the original *model matrix*.

#### 4.1.2 Fragment Shader

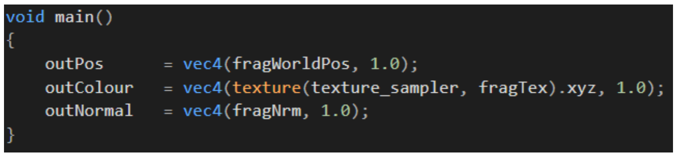
It receives the data sent from the previous *vertex shader* and the only external resource it's a *sampler2D* for sampling the texture images.



```
layout(location = 0) in vec3 fragWorldPos;
layout(location = 1) in vec3 fragCol;
layout(location = 2) in vec3 fragNrm;
layout(location = 3) in vec2 fragTex;

layout(set = 1, binding = 0) uniform sampler2D texture_sampler;

layout(location = 0) out vec4 outPos;    // World Position
layout(location = 1) out vec4 outColour; // colour of the fragment
layout(location = 2) out vec4 outNormal; // Normal
```

Figure 5: *Fragment shader* external resources and passed variables

This *fragment shader* has been set up to save outputs in three different **offscreen framebuffers**: *positions, normals* and *colors*.



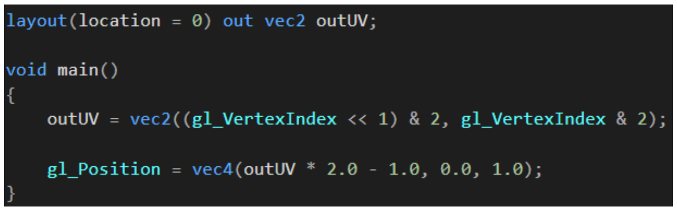
```
void main()
{
    outPos      = vec4(fragWorldPos, 1.0);
    outColour   = vec4(texture(texture_sampler, fragTex).xyz, 1.0);
    outNormal   = vec4(fragNrm, 1.0);
}
```

Figure 6: *Fragment shader* saving the result in the MRT

### 4.2 Second pipeline

#### 4.2.1 Vertex Shader

It generates the UV coordinates for sampling the textures from the auxiliary buffers and passes it to the *fragment shader*.
This is a trick for rendering a fullscreen quad by using a big triangle that is bigger than the far plane (more efficient since we are using just three vertices), then the portions of a triangle that are out of screen boundaries will get clipped (without bandwidth loss since the out of screen portions



```
layout(location = 0) out vec2 outUV;

void main()
{
    outUV = vec2((gl_VertexIndex << 1) & 2, gl_VertexIndex & 2);

    gl_Position = vec4(outUV * 2.0 - 1.0, 0.0, 1.0);
}
```

Figure 7: *Vertex shader* of the second pipeline

of triangle won't be sampled at all) [11]. The **bitwise** manipulation operations are used to obtain three pair of vertices (0,0), (2,0) and (0,2).
To do this we don't want to pass any vertex in input, this translates into passing an empty *VertexInputState*.
The number of vertices to be used will be subsequently explained with the following function :

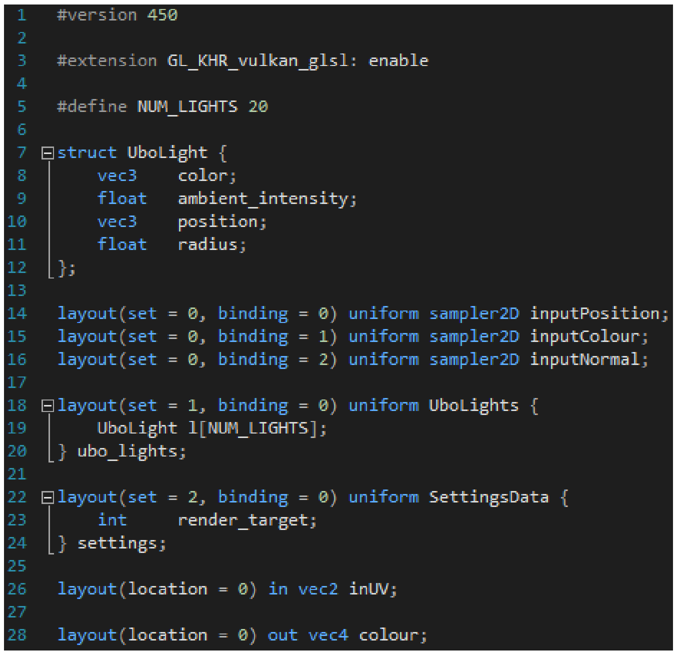
Listing 1: Drawing call for the second pipeline

```
vkCmdDraw(commandBuffer, 3, 1, 0, 0);
```

### 4.2.2 Fragment Shader

The following shader will receive many external resources, for clarity, I show them in a list.

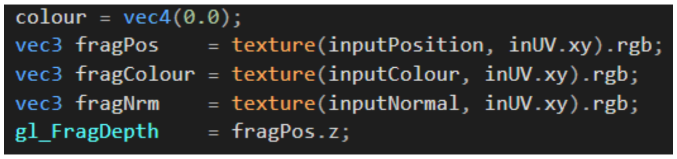
- The **MRT** (as *sampler2D* since the content is a 2D texture).
- The lights (as *descriptor set*).
- User input settings (as *descriptor set*)
- **UV** coordinates (from the *vertex shader*).



```
#version 450

#extension GL_KHR_vulkan_glsl: enable

#define NUM_LIGHTS 20

struct UboLight {
    vec3    color;
    float   ambient_intensity;
    vec3    position;
    float   radius;
};

layout(set = 0, binding = 0) uniform sampler2D inputPosition;
layout(set = 0, binding = 1) uniform sampler2D inputColour;
layout(set = 0, binding = 2) uniform sampler2D inputNormal;

layout(set = 1, binding = 0) uniform UboLights {
    UboLight l[NUM_LIGHTS];
} ubo_lights;

layout(set = 2, binding = 0) uniform SettingsData {
    int     render_target;
} settings;

layout(location = 0) in vec2 inUV;

layout(location = 0) out vec4 colour;
```

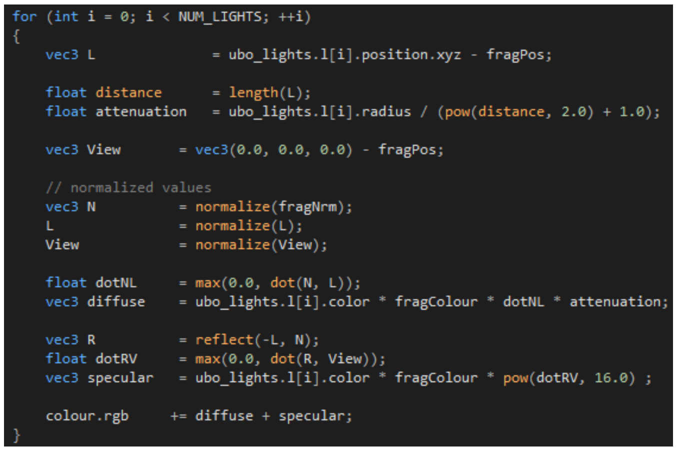
Figure 8: *Fragment shader* of the second pipeline

There is a macro that tells me the number of lights on which I have to iterate, its value must coincide with its copy on the application (used to define the lights).
Inside the main function, the first thing that occurs is the retrieving of texels from the MRT.



```
colour = vec4(0.0);
vec3 fragPos    = texture(inputPosition, inUV.xy).rgb;
vec3 fragColour = texture(inputColour, inUV.xy).rgb;
vec3 fragNrm    = texture(inputNormal, inUV.xy).rgb;
gl_FragDepth    = fragPos.z;
```

Figure 9: Retrieving *texels* from MRT



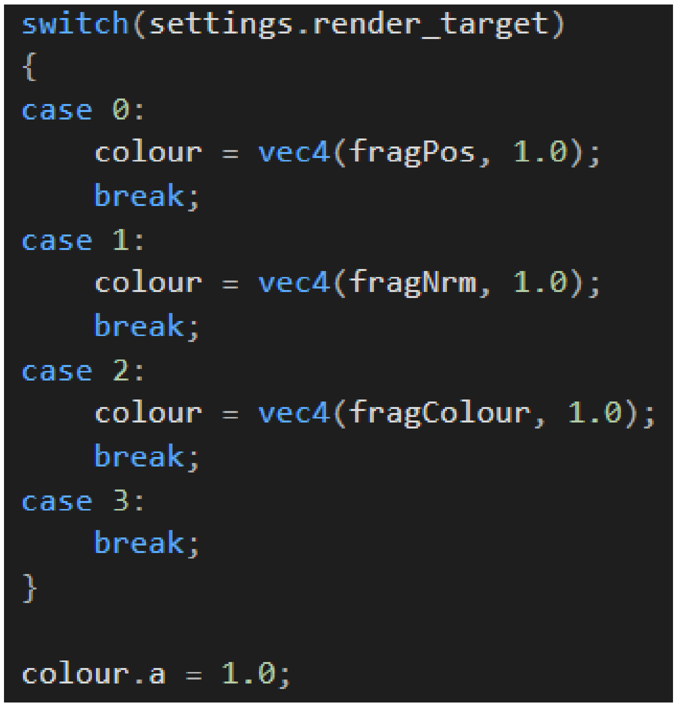
```
for (int i = 0; i < NUM_LIGHTS; ++i)
{
    vec3 L              = ubo_lights.l[i].position.xyz - fragPos;

    float distance      = length(L);
    float attenuation   = ubo_lights.l[i].radius / (pow(distance, 2.0) + 1.0);

    vec3 View       = vec3(0.0, 0.0, 0.0) - fragPos;

    // normalized values
    vec3 N          = normalize(fragNrm);
    L               = normalize(L);
    View            = normalize(View);

    float dotNL     = max(0.0, dot(N, L));
    vec3 diffuse    = ubo_lights.l[i].color * fragColour * dotNL * attenuation;

    vec3 R          = reflect(-L, N);
    float dotRV     = max(0.0, dot(R, View));
    vec3 specular   = ubo_lights.l[i].color * fragColour * pow(dotRV, 16.0) ;

    colour.rgb     += diffuse + specular;
}
```

Figure 10: Lighting computation

The lighting computation portion of the code, it's substantially a *for loop* which iterates over all the 20 lights and compute the lighting model using the **MRT** data.
The first thing calculated is the lighting factor for the **diffuse light**, i will need the normalized vector $\boldsymbol{L}$ which goes from the fragment to the light source, then i compute the dot product between the normal vector $\boldsymbol{N}$ (retrieved from the **normal map**) and $\boldsymbol{L}$. The result of this dot product will explain to me how much the angle between these two vectors is wide, then it is **clamped** between the range $[0, +\infty]$, in this way it is possible to reuse the output as a valuable lighting factor (by considering 0 the negatives dot products, where the angle between vectors it is greater of 90°).

The subsequent calculation is carried out for the reflected radius using GLSL reflect with the $\boldsymbol{L}$ vector of opposite sign and the $\boldsymbol{N}$ vector unit. Finally, the dot product between the reflected vector $\boldsymbol{R}$ and the vector that goes from the fragment to the origin is calculated (the one who looks at the fragment). The same previous clamping is performed, this means that the value will be 0 if the angle between the person looking at the reflected beam is greater than 90° (outside the field of view), instead, the result of the dot product will reach the maximum value if the vector of the chamber looking at the fragment and the vector of the reflected ray coincide.

Ultimately the specular light is calculated based on the light color, fragment color, and shininess (since you are using a **Blinn-Phong model** [1]).
The fragment shader will output different fragments with respect to the user input.



```
switch(settings.render_target)
{
case 0:
    colour = vec4(fragPos, 1.0);
    break;
case 1:
    colour = vec4(fragNrm, 1.0);
    break;
case 2:
    colour = vec4(fragColour, 1.0);
    break;
case 3:
    break;
}

colour.a = 1.0;
```

Figure 11: Switch output in the base of settings

The latest portion of the code is self-explanatory, it is just a simple switch that changes the output between the MRT and the deferred scene with lights.

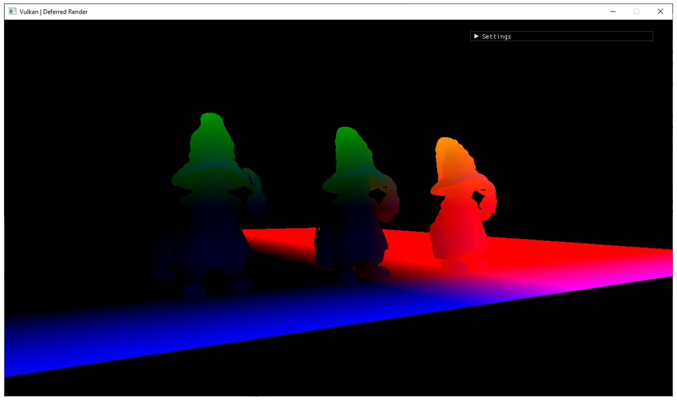


Figure 12: Positions

## 5 Performance metrics

The resolution used for performance analysis it's a 1280x760, the benchmark consists of measuring the average *frames per second* with a different number of lights. The average **FPS** is not limited by the monitor Hz, this is for having a more heterogeneous pool of samples.

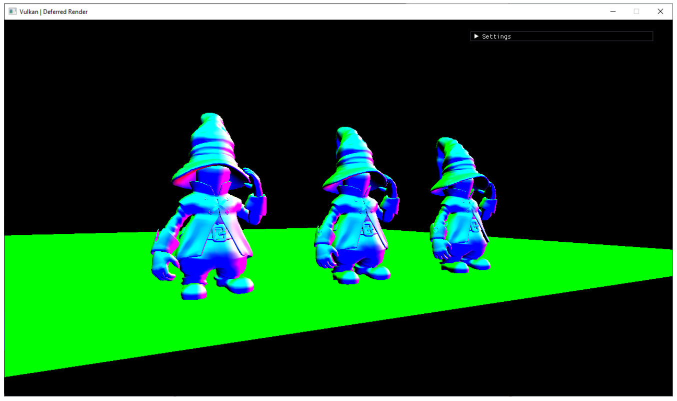


Figure 13: Normals

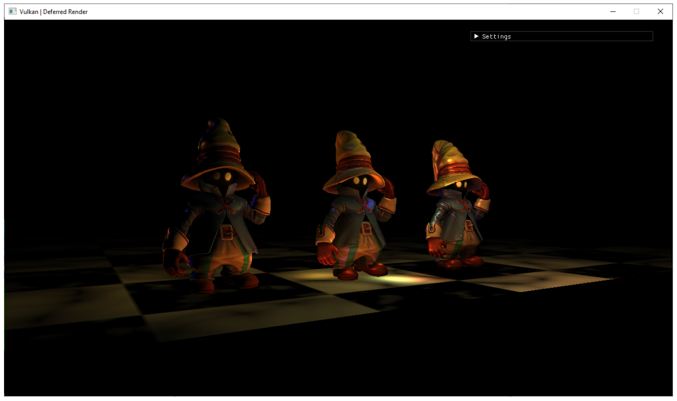
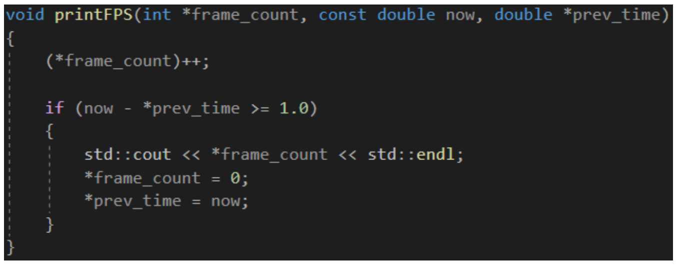


Figure 14: Final result

The scene got executed 5 times with a different number of lights: 5, 10, 20, 50, and 100. For each execution of the sceen I took 10 different samples of frames (this was implemented from code) and then averaged the samples with another really simple parsing software that I made.



```
void printFPS(int *frame_count, const double now, double *prev_time)
{
    (*frame_count)++;

    if (now - *prev_time >= 1.0)
    {
        std::cout << *frame_count << std::endl;
        *frame_count = 0;
        *prev_time = now;
    }
}
```

Figure 15: Function used for sampling the FPS

To make sure that the **FPS** calculated internally (and then stored externally) was correct, I checked the correctness of the values with the overlay provided by the **Frame Debugger** from Nsight Graphics.

Table 1: FPS results from the profiling

| Number of Lights | Avarage FPS |
|---|---|
| 5 | 668.4 |
| 10 | 639.5 |
| 20 | 600.2 |
| 50 | 575.556 |
| 100 | 406 |
| 1000 | 57.5294 |

The drop in framerate was inevitable, but it is easy to deduce that the amount of FPS remains amply high, even in the extreme case where 1000 lights were used (always taking into account that I used the overlay to achieve greater data integrity, without the latter the performance would increase by about ten FPS) the framerates are released at an excellent value for the GPU on which the benchmark was performed.

## 6 Conclusions

In conclusion, this has been a forging experience in terms of acquiring knowledge regarding the new graphics API from Khronos, but not only that, due to the verbosity and complexity involved in writing the code a significant amount of challenges during the implementation of this long project which took four months. Not only that, it was an educational experience in practical but also cultural terms for exposure to more sophisticated rendering techniques that involve the use of multiple graphics pipelines, such as deferred rendering.

## References

[1] Jim Blinn Bui Tuong Phong. **Blinn–Phong reflection model**, `https://doi.org/10.1145%2F563858.563893`.

[2] christophe lunarg. **GLM, OpenGL Mathematics**, `https://github.com/g-truc/glm`.

[3] Khronos Group. **OpenGL, Open Graphic Library**, `https://www.opengl.org/`.

[4] Khronos Group. **The triangle processor and normal vector shader: a VLSI system for high performance graphics**, `https://doi.org/10.1145%2F378456.378468`.

[5] Khronos Group. **Vulkan, Graphics Library Framework**, `https://www.vulkan.org/`.

[6] Microsoft. **Visual Studio**, `https://visualstudio.microsoft.com/it/`.

[7] nothings. **stb image library**, `https://github.com/nothings/stb`.

[8] NVIDIA. **Nsight Graphics**, `https://developer.nvidia.com/nsight-graphics`.

[9] Melissa E. O'Neill. Pcg: A family of simple fast space-efficient statistically good algorithms for random number generation. Technical Report HMC-CS-2014-0905, Harvey Mudd College, Claremont, CA, September 2014.

[10] The GLFW Development Team. **GLFW, Graphics Library Framework**, `https://www.glfw.org/`.

[11] The GLFW Development Team. **Vulkan tutorial on rendering a fullscreen quad without buffers**, `https://www.saschawillems.de/blog/2016/08/13/vulkan-tutorial-on-rendering-a-fullscreen-quad-withou`