Demo My Sponza:

Performance Evaluation

Patryk Szylin
M2125007

Contents

Sponza Overview

The deferred renderer consists of four passes to generate the scene including models and lights, and one post-processing pass to implement Fast Approximate Anti-Aliasing. The application runs at acceptable frame rate, however it is evident that implementation of FXAA caused the performance to drop drastically, from 60fps to 30+/-fps. In following sections I will identify bottlenecks and list techniques used in this project for GPU and CPU optimizations to increase performance, after which I will compare them to forward renderer and to a model that has no CPU or GPU optimizations. Optimizations for volumetric stencilling is absent therefore light volumes disappear when inside of them, however the fragment computations is present that time. Implementation of Physically Based Rendering is minimal with evident effects of specular highlights and material colours.

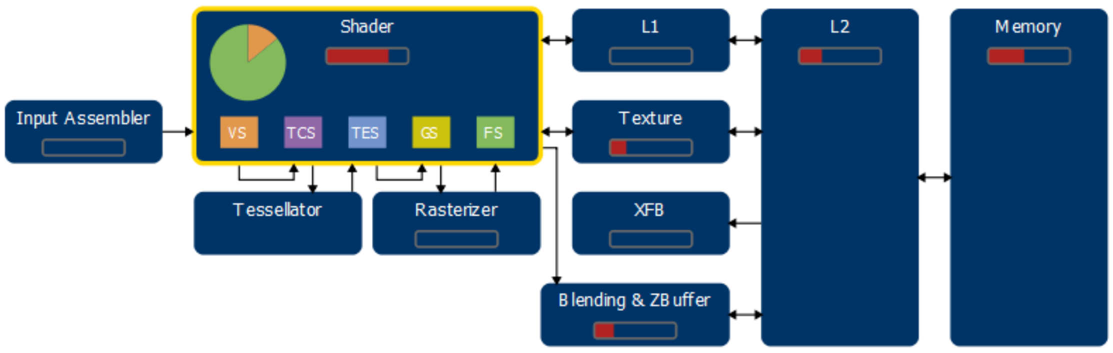


Figure 1. Overview of Sponza scene with BRDF lighting and shadows including FXAA before any optimizations.

Bottlenecks

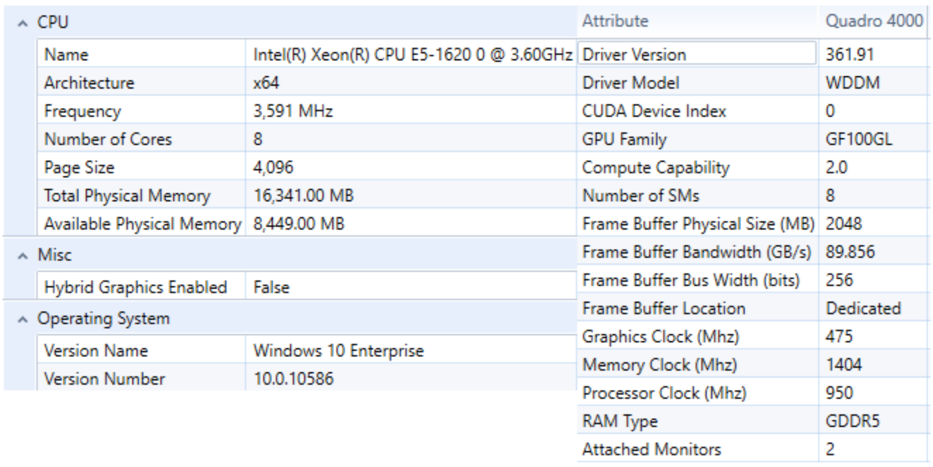
Most evident bottleneck in the application is the *fragment shader*, that's due to the complex calculations used for physically based shading. The fragments are being calculated every frame regardless whether they're being affected or not, thus decreasing the performance drastically.

The current use of off-screen *framebuffers* such as G-Buffer ad Colour Buffer have almost 50% negative impact on the application's performance. Current texture format that goes into the G-Buffer is the size of 32bit Float, containing high precision data which puts a lot of pressure on the framebuffer *bandwidth* as well as memory management.

Major performance drawbacks during runtime are caused by the OpengGL blending. When multiple light passes are being blended the fragments are being calculated and combined together, resulting in new fragment colour. This process is increasing fragment calculations and OpenGL calls, thus reducing performance.

Optimizations

Performance optimization and statistics have been tested on standalone machine with following specs:



CPU	
Name	Intel(R) Xeon(R) CPU E5-1620 0 @ 3.60GHz
Architecture	x64
Frequency	3,591 MHz
Number of Cores	8
Page Size	4,096
Total Physical Memory	16,341.00 MB
Available Physical Memory	8,449.00 MB
Misc	
Hybrid Graphics Enabled	False
Operating System	
Version Name	Windows 10 Enterprise
Version Number	10.0.10586

Attribute	Quadro 4000
Driver Version	361.91
Driver Model	WDDM
CUDA Device Index	0
GPU Family	GF100GL
Compute Capability	2.0
Number of SMs	8
Frame Buffer Physical Size (MB)	2048
Frame Buffer Bandwidth (GB/s)	89.856
Frame Buffer Bus Width (bits)	256
Frame Buffer Location	Dedicated
Graphics Clock (Mhz)	475
Memory Clock (Mhz)	1404
Processor Clock (Mhz)	950
RAM Type	GDDR5
Attached Monitors	2

CPU

There were only two CPU optimizations used for Demo My Sponza; one used to decrease the number of draw calls using instancing, and the second to pack data using uniform buffer objects, that remain accessible by all shaders throughout the span of a program.

Geometry Instancing

This technique allowed me to draw multiple copies of the same mesh with a single draw call and two vertex streams: mesh to be instanced and per-instance data (vertex positions, normals, texture coordinates and model matrices including world transformations). Each instance or a "copy" is then drawn into different locations in the scene using a draw call specifying how many instances of a mesh I want to draw.

Using instancing I have reduced draw calls from 84 to 29, significantly reducing CPU workload, while each draw call took 1.5ms +/-.

Uniform Buffer Objects

Throughout the program, I was required to use the same objects across multiple programs without changing their values. This redundant procedure has caused unnecessary increase in *uniform* OpenGL calls, thus enlarging the CPU workload. Objects like projection and view matrices have never changed, yet they needed to be passed to individual programs multiple times.

I used uniform buffer objects to allocate global uniform variables that stored a structure of matrices, point light and spot light variables, and whenever needed, I accessed them through their index.

GPU

Fragment Shader

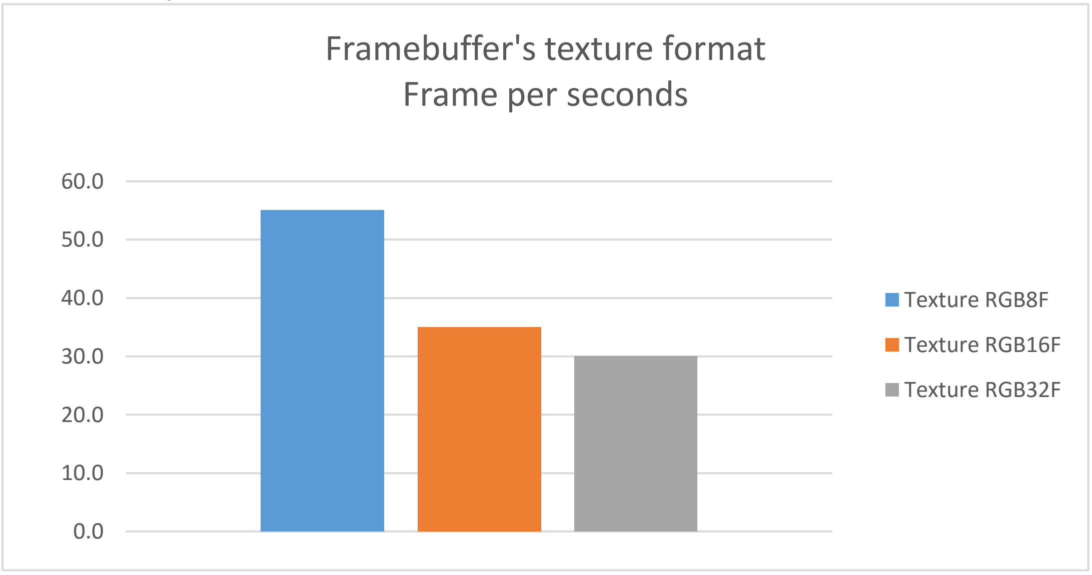
To optimize fragment shader calculations I have downsized textures to 1280x720 from 1280 x 1024, thus reducing the amount of pixel calculations per texture.

1280x720 texture	1280x1080 texture
921,600 pixels	**1,310,720 pixels**

389,120 pixels were reduced by this procedure, the resulting performance was significant. To further optimize fragment shader performance I ensured that no excessive filtering is needed and the textures are at the lowest precision necessary.

Framebuffer bandwidth

Memory usage is one of the biggest bottlenecks in the application and framebuffers are the largest consumers. To reduce the buffer's memory usage I have decided to use 16bit floats in comparison to 32bit floats, simply because the visual effect is not noticeable and the trade-off in terms of performance is great. This has increased framerate by 40%.

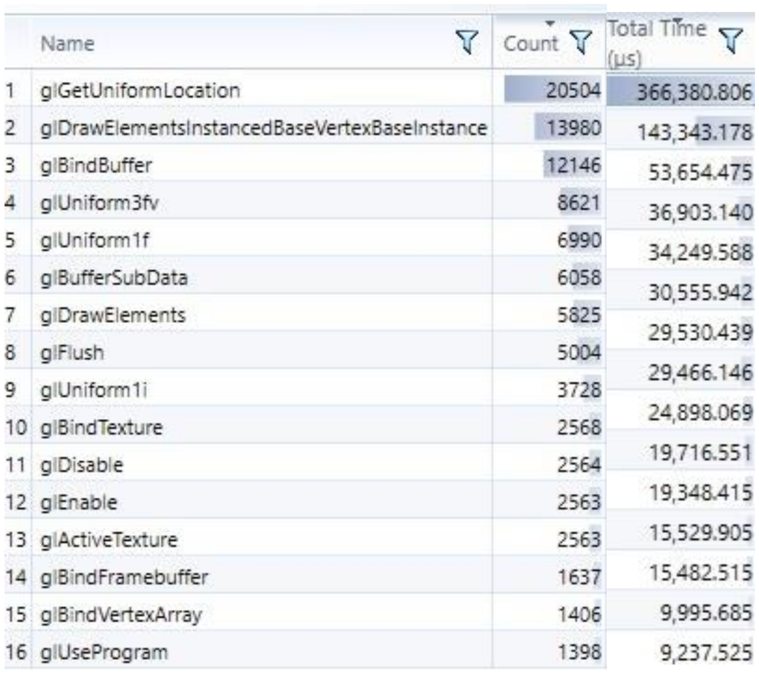


Conclusion

The overall performance of the application runs at acceptable rates ranging between 40-55 frames per second, including animated objects. Fragment shader still remains a big problem in current renderer, taking most of the processing power away. Some unnecessary fragment calculations are discarded using stencil operations, however this could be extended using volumetric stencilling to further discard fragments that are computed but not visible.

Instancing has reduced the number of draw calls substantially and resulted in efficient way of reducing CPU usage. CPU optimizations were minimal because the entirety of application was affected by GPU, fragment calculations had a big impact and it was hard to implement sufficient ways to overcome this issue.

Statistics



	Name	Count	Total Time (µs)
1	glGetUniformLocation	20504	366,380.806
2	glDrawElementsInstancedBaseVertexBaseInstance	13980	143,343.178
3	glBindBuffer	12146	53,654.475
4	glUniform3fv	8621	36,903.140
5	glUniform1f	6990	34,249.588
6	glBufferSubData	6058	30,555.942
7	glDrawElements	5825	29,530.439
8	glFlush	5004	29,466.146
9	glUniform1i	3728	24,898.069
10	glBindTexture	2568	19,716.551
11	glDisable	2564	19,348.415
12	glEnable	2563	15,529.905
13	glActiveTexture	2563	15,482.515
14	glBindFramebuffer	1637	9,995.685
15	glBindVertexArray	1406	9,237.525
16	glUseProgram	1398	

Figure 2. General statistics of OpenGL calls, illustrating that passing data to a shaders took the longest time.

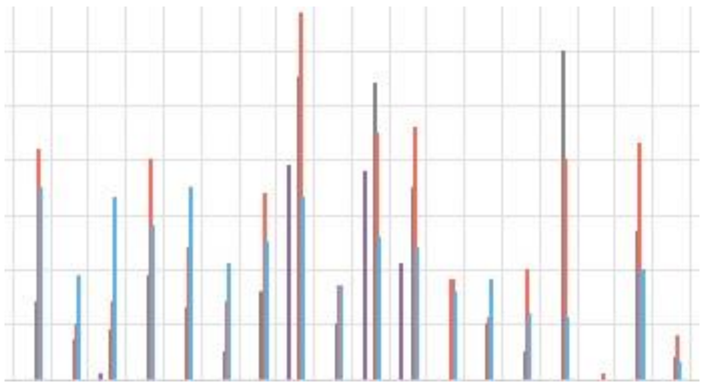


Figure 3. Framebuffer is used at high cost when drawing Sponza geometry.

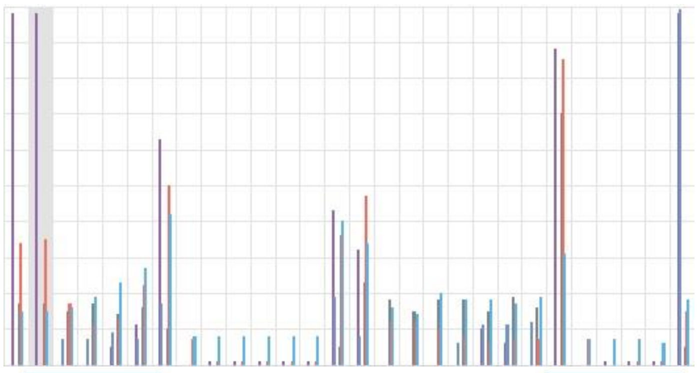


Figure 4. Shader peaking at 90% bottleneck when drawing full screen quads.

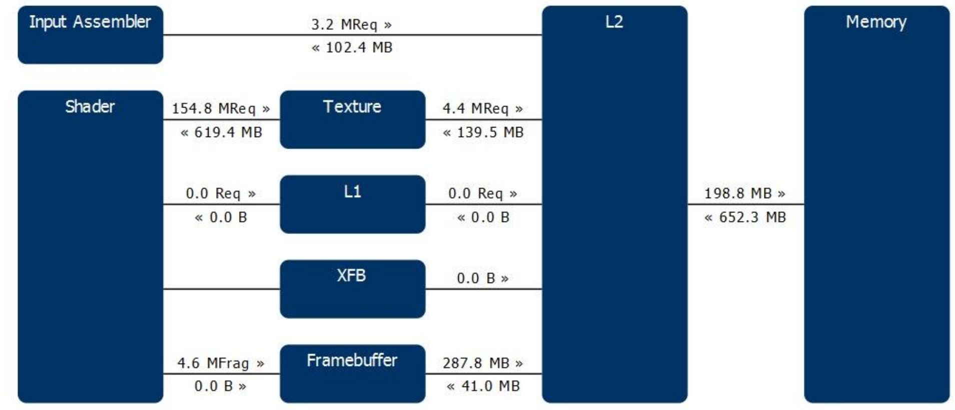


Figure 5. Pipeline performance and workload, indicating that most workload exists when passing and retrieving data from memory.

Future Improvements

Few ways to reduce fragment's cost calculations would be to store all complex and expensive calculations of functions into a texture. The benefit derives from the usefulness of texture lookups. For instance it would allow me access a texture value for the cost a single texture look up.

Further improvement relates to discarding even more pixels at runtime. Rendering geometry front-to-back comes with fragment shading advantages as most pixels will fail the depth-test and result in fewer depth and colour writes to framebuffers.

Additionally, volumetric stencilling would enable me to discard certain pixels drawn by spot lights. Volumetric stencilling enables users to discard any fragments that are outside a volume, in this case light.

Most textures use RGB colours, adding alpha component to the textures would allow me to store 4 floating point data, and with a cost of one single texture lookup to access it instead of passing this data through uniforms. This is useful for specifying material attributes e.g. roughness, shininess, normal mapping etc.