Graphics Programming with Shaders

AG0904A – Academic Year 2015/2016

Alberto Taiuti – 1300250

Introduction

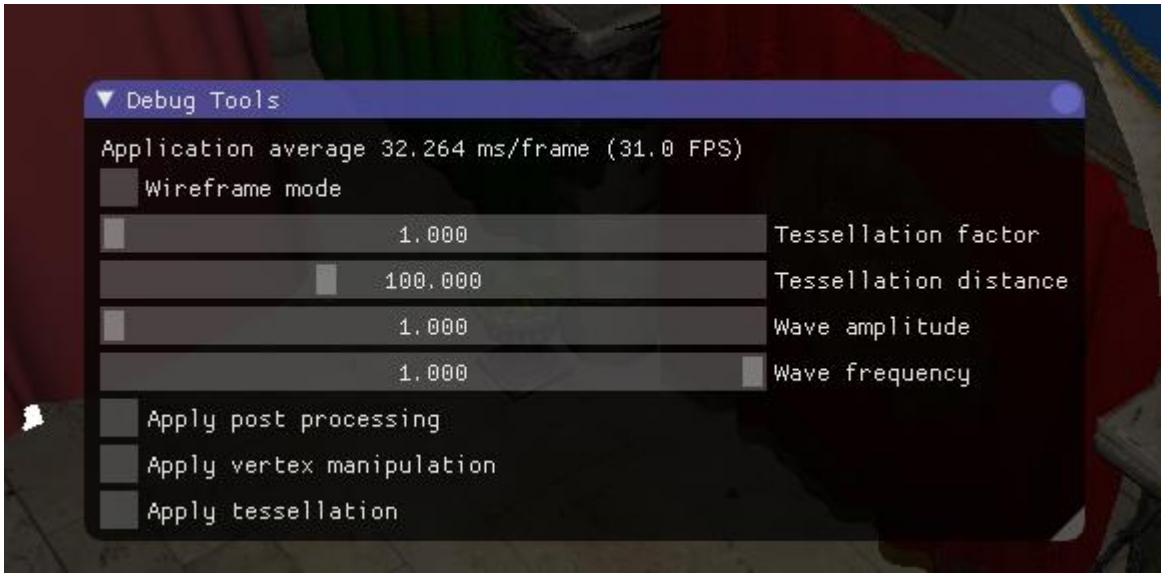
The purpose of the application is to show the ability of programming 3D graphics applications using the modern programmable pipeline. In the specific, the graphics rendering of this application was developed using Microsoft's latest graphics API, DirectX 11.The scene rendered is the 3D model named Sponza (Crytek 2010), a famous model for graphics programming benchmarking.

Figure 1: The scene as rendered in the application

Figure 1 shows the scene as it were before the lights values were tweaked. The scene coming with the submission now shows shadows and shading in a better way. Discussion of why this happened can be found in the next paragraphs.

User controls

- WASD: Move the camera along the XYZ axis.
- Arrow Keys: Rotate the camera's view direction.
- GUI: Select the tickboxes and move the sliders to see the changes in real time. The GUI window can be hidden by double clicking on its title bar, and moved around by dragging it from the title bar.



Source usage

The application was subdivided in folders as requested. However, in order to correctly build and run from the VS environment the contents of *exe.zip* must be in the same folder as the *.sln* file.

The HLSL shaders were not all loaded in the application's solution since they were edited with an external editor. Hence, in order to review them, they can be opened with a text editor of choice from the folder *shaders* found in *exe.zip*.

Techniques used

Main new additions to the framework

The given tutorial framework (Robertson 2015) was deeply rewritten in order to accommodate for the rendering of Sponza.

The main completely new additions are:

Shader classes manager

Figure 2: The GUI

A class which manages instances of classes whose objective is to describe a shader. It ensures that there is only one instance of a type of shader class which can be reused and shared across different materials.
In order to achieve this, it uses an STL map, where the CRC of the name of the shader class instance is the key and the value is the pointer to the instance of the shader class in dynamic memory.

Texture resource manager

A class which achieves as the same result as the shader classes manager but with respect to textures loaded in memory. The functioning is very similar to the one of the shader class manager.

Renderer classes

A class, "Renderer", describes a generic rendering mode, which is inherited from in order to create specific rendering modes, such as Forward Rendering. Forward Rendering is the technique used for this application and hence this class has been created. This structure makes it easy to expand to a Deferred Renderer in the future.

The framework's main application holds a pointer to the base class Renderer and uses it to render the scene by calling its *render* method. This allows to swap with potential different rendering methods at run-time.

The Forward Renderer implemented in the application keeps performance in mind. In fact, methods which were previously contained in the shader and lab classes have been moved and unified under the renderer class to achieve better performance.

For example, the vertex data for a model is sent all in one take to the GPU, and the model's meshes are rendered using their index offset with respect to the beginning of the buffer rather than sending their meshes one at a time.
Also, the shaders' buffers and parameters are set only once per material, so that meshes sharing the same material are renderer all in one take after the shaders' parameters for that given material are set. The renderer then moves onto the next material and renders all the meshes which make use of it, repeating the process until all the meshes are rendered.

The rendering also recognises the difference between buffers which can be set only once per frame and the ones which need to be set once per material, setting the formers (e.g.: light buffers, camera buffers) at the beginning of the rendering of a particular frame.

Post processing classes

Similar to what done with rendering techniques, this set of classes describe potential rendering techniques. Gaussian Blur is the post processing chosen for this application. The advantage of having a base class with inherited classes is that it can be changed at runtime and the fact that it accepts a render target and it returns it after processing it makes it very easy to use.

The base class *PostProcess* holds data which is to be shared across any type of post processing, and also defines the interface. It is the duty of the child classes to define attributes and personal methods for obtaining the post processing required.

In the case of the Gaussian Blur post processer, the class holds methods and shaders necessary for its purpose:

- Methods for downsampling, upsampling, etc
- Two downsample render targets
- Ortho meshes for upsampling, downsampling, and applying the blurring
- A reference to the shaders manager object from which to obtain the horizontal and vertical Gaussian blur shaders after their creation

This way, the whole process is opaque to the user: all the user has to do it pass a render target to the ApplyPostProcess method of the base pointer class and have it returned after the processing.

The Gaussian Blur shaders use a specific constant buffer which describes the size of the render target.

Extensive usage of render targets

The scene, on its own, is always rendered to a render target, which is then either output as it is to the back buffer or is further processed by the post processing class instance one or more times. This

allows for flexibility on what to do with a rendered scene and, potentially, allows for easy implementation of Deferred Rendering (Saito, Takahashi 1990).

Materials

The material class describes a material for a given mesh. It also holds a reference to which shader class is used for rendering the material so that it can be readily used when necessary. It contains attributes which mimic the ones described in the material files .mtl.

There is also a relative structure and constant buffer which are used in pixel shaders to apply shading. These buffers are set once per material, and the subsequent draw calls for meshes use these values.

Meshes ordered by material

The meshes are ordered by material for rendering. This highly improves the performance of the application as there are less state changes when rendering the meshes. Meshes can be added for rendering to the renderer, which looks up their material and orders them correctly internally. The rendering procedure is explained in the paragraphs above.

Proprietary model file format

A parallel application was created in order to generate a proprietary file format of a given model, which can then be loaded in the application much more faster than if parsing an .obj file every time. The model class, loaded this way, already has all the information needed to be rendered: list of materials, necessary textures, etc. The loading system uses the Boost C++ libraries for achieving serialization and deserialization, while the model loader application uses both Boost and the TinyObj library (Fujita)

ImGui implementation

In order to show the capabilities of the application, the ImGui library (Cornut) was integrated in the application. This allows the user to easily tweak program and see how this influences it in real-time.

Normal mapping

The application makes use of the normal mapping rendering technique (Krishnamurthy and Levoy 1996, Cohen et al. 1998) in order to properly render Sponza. In fact, Sponza makes extensive use of normal maps.

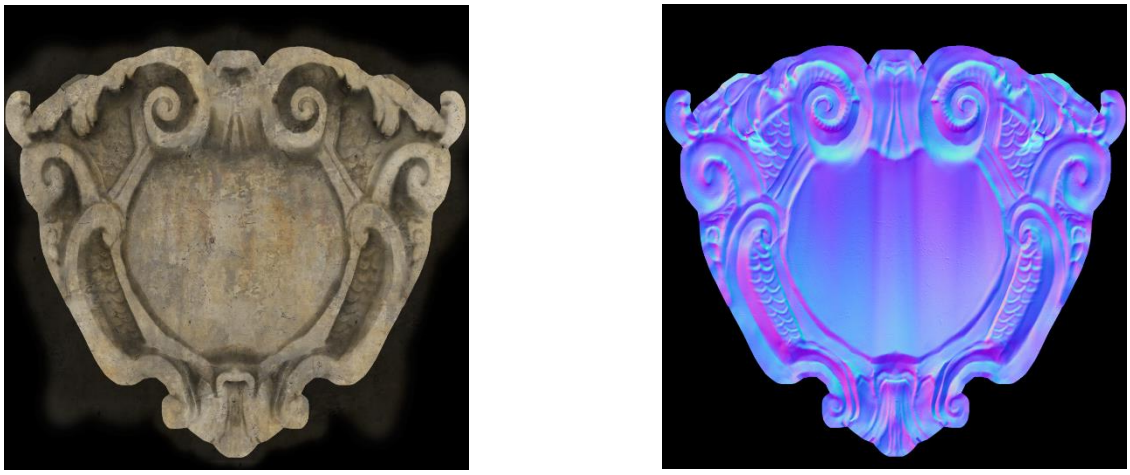


Figure 3: Diffuse texture from Sponza and its relative normal texture

The version implemented in this application entails transforming the entities external to the vertex which are involved in shading into tangent space and perform light calculations in this space (Lengyel 2011).
The advantage of normal mapping is that it makes a geometry appear very detailed without having to use a very high number of faces in it by increasing the shading detail rather than the topology detail.

Specular mapping

Where possible, such as for the Sponza model, specular mapping is also performed by reading a specular texture map in the pixel shader and using that value for the specularity calculations.

Other major features

The software also improves over the techniques presented during both the lectures and the tutorials (Robertson 2015).

Distance-based dynamic tessellation

The application shows the usage of tessellation by implementing tessellation based on the position of the camera.

When tessellation is activated, the renderer switches the shaders being used by all the materials: it replaces their vertex shaders with basic vertex shaders which pass the data down the pipeline to the hull and domain shader, and it activates the usage of the domain and hull shaders. It also changes the topology from triangle to 3 points patch list.

The hull shader receives a special buffer which defines the maximum tessellation factor, parameter which can be tweaked from the GUI, and also the camera's position. In the constant function, it calculates the tessellation factor for every edge and centre of the patches based on the distance of the edge or centre from the camera position. If the edge/centre is too far away from the camera, it defaults to a tessellation factor of 1. If it lies within the range for tessellation, then it's tessellated depending on the distance.
The point function of the hull shader doesn't apply any further computation and simply passes the data down to the domain shader.

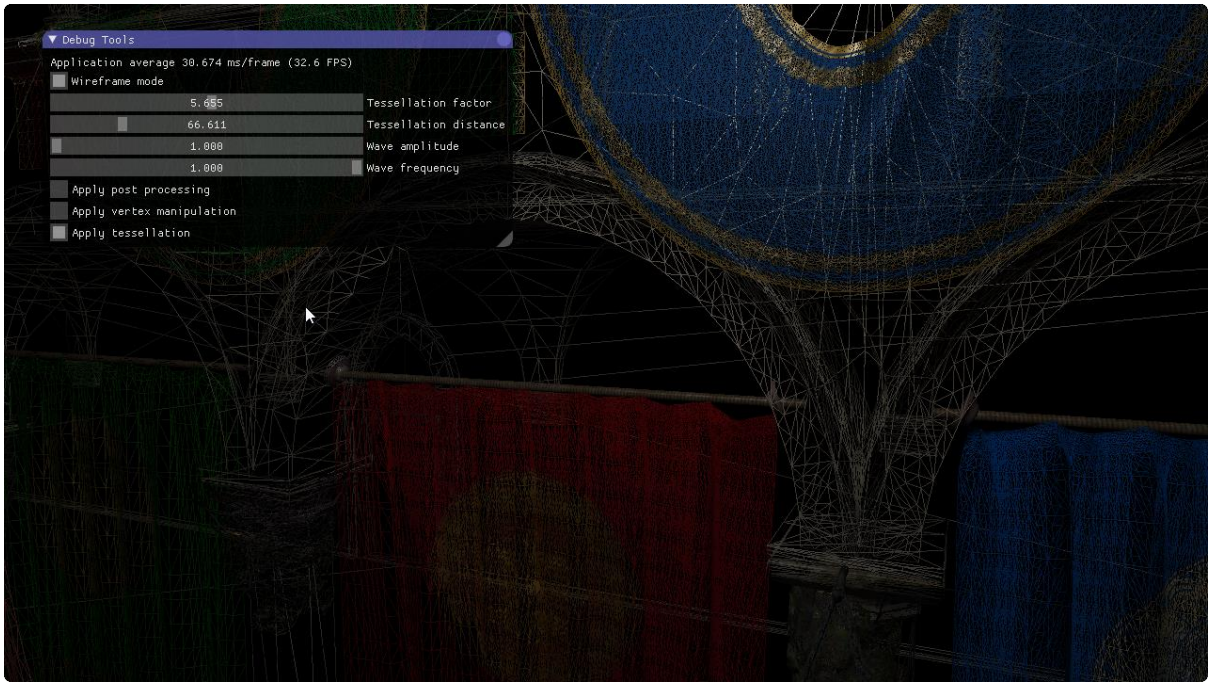


Figure 4: Distance-based tessellation

The domain shader then assumes the function that the vertex shader previously had, and in fact receives as the same buffers as a non-tessellation vertex shader. For normal-mapped materials, it calculates the tangent space coordinates for lights and camera, while for non-normal mapped materials it calculates the light direction, etc, normally.

Multiple lights shading

The application makes use of up to 4 lights. Their functioning is very similar to the one of the fixed function pipeline in OpenGL 1.2.

Multiple lights shadow mapping

It is possible to have up to four light sources for shadow mapping. For each light, the scene is rendered from the light's point of view using a depth material, which only outputs the depth values, onto separate render targets (Rastertek).

Figure 5: Shadow mapping

Then these render targets are used as texture sources by the pixel shaders in order to perform correct shadow mapping.

Position of the lights in the scene

In order to show the position of the lights in the scene, the renderer mainly uses the geometry stage of the pipeline.

The position of each light is passed as a point primitive to the pipeline, and using a special shader class these primitives are used in the geometry shader to create a cube around them at the position of the point.

The geometry shader outputs 32 new vertices which surround the point at equal distances.

Vertices manipulation

Vertex manipulation in the program entails manipulating the position of the vertices of the curtains in Sponza via a sine wave by changing the vertex shader in use by the appropriate shader classes. The user can then modify the amplitude and frequency of the waves via the GUI.

In order to achieve this a special constant buffer is used, which contains the frequency and amplitude parameters.

Shortfalls and solutions

The application doesn't use soft shadows; these could be very easily implemented by passing the render targets to the post processing class or by using more sophisticated techniques such as the Stratified Poisson Sampling. Soft shadows were not implemented as the more advanced technique used is multiple lights shadow mapping. Moreover, due to the lack of more advanced shading techniques such as HDR, the shadows are not too visible when a point light is active in the scene; however this is just a matter of tweaking the light settings which is out of scope of the module. The application also lacks shadow mapping for point lights; this technique was not implemented

because it wouldn't have added any value to showing the capabilities of the programmer, as it would only have involved expanding what is being done for spot lights on cube mapping.

There are some shading artefacts when tessellation is active, but those are not errors; instead, they exist because of the topology of Sponza, which was not created with tessellation in mind. In fact, when the topology density is increased by means of tessellation, the points between which the rasterizer has to interpolate are closer and hence the calculations appear different from when the vertices are sparser.

Another drawback is having to use a global override of the delete and new operators, which can be found on the top of the main function in Main.cpp; not having this was creating problems in Release mode related to the DirectX math library. A solution would be to abandon this library and switch to another one, such as GLM.

The issue above introduces further problems which stop the application from being able to modify the values of the light at run-time, either in Debug or Release mode. Again, the solution would be to re-write the application to use another maths library.

Bibliography

COHEN ET AL., 1998. Appearance-Preserving Simplification. *SIGGRAPH,* .

CORNUT, O., *ImGui.*

CRYTEK, 2010. *Sponza.*

FUJITA, S., *Tiny Obj Loader.*

KRISHNAMURTHY AND LEVOY, 1996. Fitting Smooth Surfaces to Dense Polygon Meshes. *SIGGRAPH,* .

LENGYEL, E., 2011. *Mathematics for 3D Game Programming and Computer Graphics.* Third edn.

RASTERTEK, *Multiple Lights Shadow Mapping.* http://www.rastertek.com/dx11tut41.html edn.

ROBERTSON, P., 2015. *Lectures, Graphics Programming with Shaders.*

SAITO, T. and TAKAHASHI, T., 1990. Comprehensible rendering of 3-D shapes, *SIGGRAPH '90 Proceedings of the 17th annual conference on Computer graphics and interactive techniques*, 1990 1990, ACM, pp. 197-206.