Deferred Renderer Proof of Concept Report

Octavian Mihai Vasilovici

28 March 2013

Bournemouth University

1. Summary

This document aims at explaining the methods decide to be used in creating a deferred renderer in OpenGL 4.x with the help of the NGL library. It will focus on showing the methods, designs, algorithms and decisions that were made in order to create a Deferred Shading Renderer. It will consist of the initial research, algorithms and methods selected for the implementation, how certain drawbacks are planned to be resolved and the features that are going to be added later.

2. Introduction

The deferred rendering method was first introduced by Michael Deering et al. during SIGGRAPH conference in 1998, without explicitly using the "deferred" term in their presentation, according to Mark Harris from NVIDIA and Shawn Hargreaves from Climax (2004). Deferred rendering started to become viable with the introduction of the "programmable-pipeline" in the GPUs starting with the release of nVidia's GeForce 3 and ATI's Radeon 8500 back in 2001. (Villar J. R., p.3)

More and more modern games focus on creating real-life looking virtual worlds and in doing so, they use many light sources to lit many objects that eventually cover a high amount of pixels. Because of this, the traditional forward rendering technique becomes computationally expensive and depending on the complexity of the scene it might not be possible to render it in real time. For example, if an object is lit by 4 lights, it will be rendered 4 times, once for each light, in order to accumulate the light effect.

According to Koonce R.,(2007, c.19). the process of shading/lighting an object can be summarized in the following steps:

1. Compute the object's mesh shape
2. Determine the material characteristics
3. Calculating the incident light
4. Calculating the surface interaction with the light and creating the final image

Currently the most common approaches used for deferred rendering are: *deferred shading* and *deferred lightning*. In a deferred lighting renderer only the lighting and not the shading, calculations are postponed. In the initial pass over the scene only the attributes necessary to compute fragment lighting are saved. Since the second pass outputs only diffuse and specular lighting information, another pass must be made over the scene to read back the lighting data and output the final fragment shading. The advantage of deferred lighting is a massive reduction in the size of the stored data from the first pass. The cost is the need to render the scene meshes twice. (Wolfgang E., 2008)

In a deferred shading render, the lighting step is postponed and done in a second pass. This is done in order to avoid lightning the pixel more than once. (Wolff D., 2011, p.179) which means separating the first two steps from the last two in the above list. This report will present a deferred shading implementation.

3. Deferred Shading method

The two render passes in deferred shading consist of:

1. Rendering the scene's geometry in the first pass without evaluating the reflection model, but instead storing all the geometry information in an intermediate set of buffers that are commonly named the **G-buffer**.
2. In the second pass, the light evaluation is being done based on the information read from the G-buffer in order to compute the color for each fragment.

When using deferred shading the light evaluation is done only for the fragments that will be visible.(Wolff D., 2011, p.179)

The G-Buffer is a logical grouping of multiple a 2D textures with each texture bind per vertex attribute. The attributes are separated and written in the different textures all at once using a capability of OpenGL called Multiple Render Targets (MRT). Since the attributes are written in the fragment shader, the values that are stored in the G-buffer are the result of the interpolation performed by the rasterizer on the vertex attributes. This stage is called the **Geometry Pass**. Every mesh from the scene is processed in this pass. Because of the depth test, when the geometry pass is completed the textures in the G-buffer are populated by the interpolated attributes of the closest pixels to the camera. This means that all the pixels that have failed the depth test are dropped and the only ones that are left are those visible for which lighting must then be calculated.

For every pixel on the screen, the data from the G-buffer textures needs to be sample and the lighting computations to be done, but since that the geometry pass has finished the original objects' information is lost. To overcome this, the computations need to be done from the light source point of view:

-for directional light since all the screen pixels are affected by it a full screen quad will be used.

-for a point light a sphere model will be rendered with its center at the light source location.

-for a spot light a cone model will be rendered with the cone being located at the light source location.

This stage is called the **Lightning Pass**.

In order to overcome some of the undesired effects, like having all the pixels colored that are not inside the light volume but are covered in the screen space by the light volume, the **Stencil Buffer** must be used.

Another problem that must be taken into consideration is that the light will disappear as soon as the camera enters the light volume. This can be easily fixed by enabling front-face culling for the light volume geometry.
(Meiri E., 2012)
The minimum requirements for the G-buffer are to store the following attributes when doing the geometry pass:

- Position
- Normal
- Color
- Other attributes (*optional*) (Harris M., Hargreaves S., 2004).

Choosing the render target format is also very crucial since a good precision must exists for the attributes like Color or Depth but in the same time it must be taken into consideration not to waste memory. The format of the render targets are also platform specific, for example, a 128-bit format is not available on the X-box platform.
(Zima C., 2008)

Also, since the G-buffer is being populated per frame, a higher precision will increase the data traffic on the memory bus. Instead of writing to a single render target, this is done to multiple targets. This means the number of bytes written is multiplied by the amount of the render targets. During the lighting pass, the buffers are sampled which also shows an increase in the bytes read. Memory bandwidth (fill rate) is the only factor that determines the performance of a deferred shading engine on any GPU and thus deciding the precision of the G-buffer is a key aspect (Koonce R. 2007 c.19.8.2).

All the render targets must follow these basic rules:

- Must have the same number of bits.
- Render targets with multiple channels can be mixed

Example:

This will work	This will not work
RT0 = R32f RT1 = G16R16f RT2 = ARGB8	RT0 = G16R16f RT1 = A16R16G16B16f

(Harris M., Hargreaves S., 2004).

Other problems arise when using deferred shading like the impossibility to render any translucent (transparent) objects since it cannot handle alpha-blended geometry. Alpha blending is not supported partly because of hardware limitations, but it is also not supported by the method since

it stores only the material attributes of "only" the nearest pixel. To overcome this limitation, the forward rendering method is used for translucent objects after the deferred rendering passes have ended (Koonce R. 2007 c.19.8.1).

Another problem with deferred shading is the inability to support hardware anti-aliasing like MSAA (Multi Sampling Anti-Aliasing). There are numerous other techniques that can still be used for anti-aliasing in conjuration with a deferred rendering method, like FXAA (Lottes T., 2009), MLAA (Morphological Anti-aliasing) (Reshetov A., 2009) or using an *edge detection mechanism and blur filter* (Shishkovtsov O. 2005, c.9)

The method of anti-aliasing will not be in the scope of this report, but it was thought of as a future addition to the renderer.

One main advantage of the deferred shading method is its ability to be integrated very easily and fast with all the popular shadow techniques. The most common approach is the use *Shadow maps.* This technique works very well for directional or spot lights but can be a bit trickier with point lights. A solution for point light shadows is to use a *Virtual Shadow Depth Cube map.* (Harris M., Hargreaves S., 2004).

The method of creating shadows will not be discussed further in this report, but it will be implemented in the final renderer at a later stage.

4. Results based on the research

Based on the research a deferred shading renderer was selected rather than a deferred lighting one since it is more used in the industry, and it is not required to render the scene geometry twice.

Since the difference, at render time, from a forward renderer is adding a shading pass while the geometry pass is rendered in a normal way, the mechanics for the renderer were thought as it follows:

- Load all the shaders during the initialization step and keep them all in a dynamic array.
- To each shader attach multiple lights.
- To each shader attach multiple meshes.
- To each mesh load one or more textures.
- To each mesh/object attach one or more materials.

This approach ensures that after the initial setup, where all shaders, meshes, lights are loaded/created, the renderer can simply iterate through the shaders and draw all the objects attached to the currently selected shader. The advantage of this method is that the GPU does not change its state very often and can provide a performance boost compared to a method where all the objects are stored in an

array and the drawing is done by traversing this array. This will lead to a lot of GPU state changes since theoretically each object has its own shader. This takes us to the next topic.

Starting with Shading language 3.0 the creation of "Uber" shaders is allowed, which means create one shader that can be used to draw all the objects in a scene, lights (Harris M., Hargreaves S.,2004).

According to Dudash B., this is not a good approach since it can stress the GPU and other problems might arise. It also proves difficult to maintain, update and implement from a programming point of view. Based on this, the decision to use multiple shaders, split in two (fragment and vertex) for each light type was taken.

Another key factor was to use the NGL library as much as possible which includes the *Camera, Light, ShaderLib* and *ShaderProgram* classes, the transformation stack, the existing types, like *ngl::Vec3*, etc. The only code that would be written is the OpenGL functions for the deferred shading along with the classes that expand the preexisting functionality.

4.1. The G-buffer layout

Since it is known that the size of the G-buffer is the ultimate drawback of the deferred shading pipeline, a close consideration for the precision must be made.

In the prototype accompanying this report each render target was selected to have 32bit floating point precision, which in a real world application is considered inappropriate since it allocates too much memory. Since the complexity of the tested scene is rather low, no "eye" performance loss was seen. In the final version of the renderer, some of the render targets precision will be lowered to provide a less memory bandwidth.

An example of a real time game engine uses the following layout for the G-buffer:

R8	G8	B8	A8
Depth 24bpp			Stencil
Light Accumulation RGB			Intensity
Normal X (FP16)		Normal Y (FP16)	
Motion Vectors XY		Spec-Power	Spec-Intensity
Diffuse Albedo RGB			Sun-Occlusion

(Valient M. 2007)

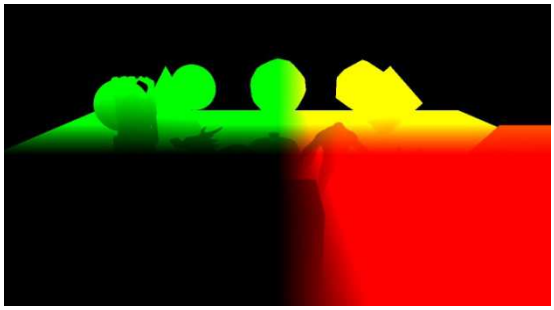
The render targets from the above table differ from the ones from the current prototype but it offers a good example on what is tried to be achieved.

Upon closer inspection regarding the lighting models to be implemented, the G-Buffer layout was selected to contain the following render targets:

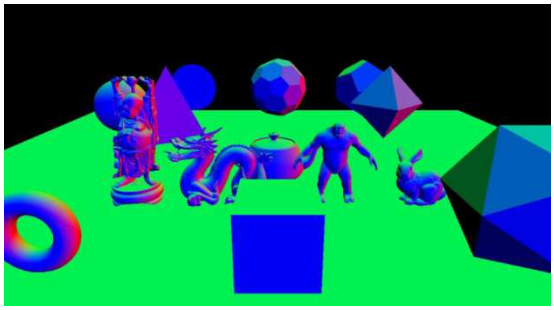
- **Position –** for storing the position of the objects.
- **Color –** for storing the diffuse and ambient material color of the objects.
- **Normal –** for storing the normals information of the objects.
- **Specularity –** for storing the specular color of the light.
- **Tangent –** for storing the tangent information read from the obj file.
- **Binormal –** for storing the binormal information that is calculated.
- **NormalMap –** for storing the normalMap so it can be later used during the light pass
- **Depth** – needed in the Stencil pass and for all the lightning calculations.
- **Final frame** – the texture containing the final output.

The format for each render target texture was selected as *GL_RGBA.* The decision was based on the following article which suggests an increase in performance when a *GL_RGBA* format is used rather than *GL_RGB.* This will force a cast from *GL_RGB* to *GL_RGBA* at the cost of extra operations. (OpenGL 2012)

An example of G-buffer textures from the prototype using a directional light:

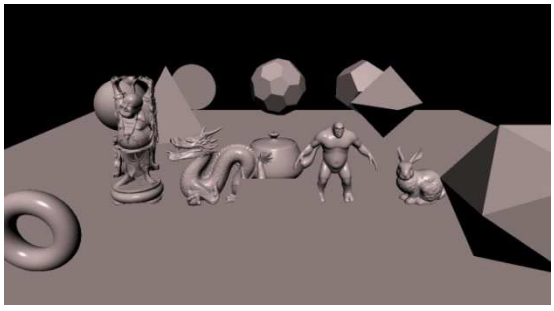


1. Position texture



2. Normal texture

3. Diffuse texture



4. Final Image

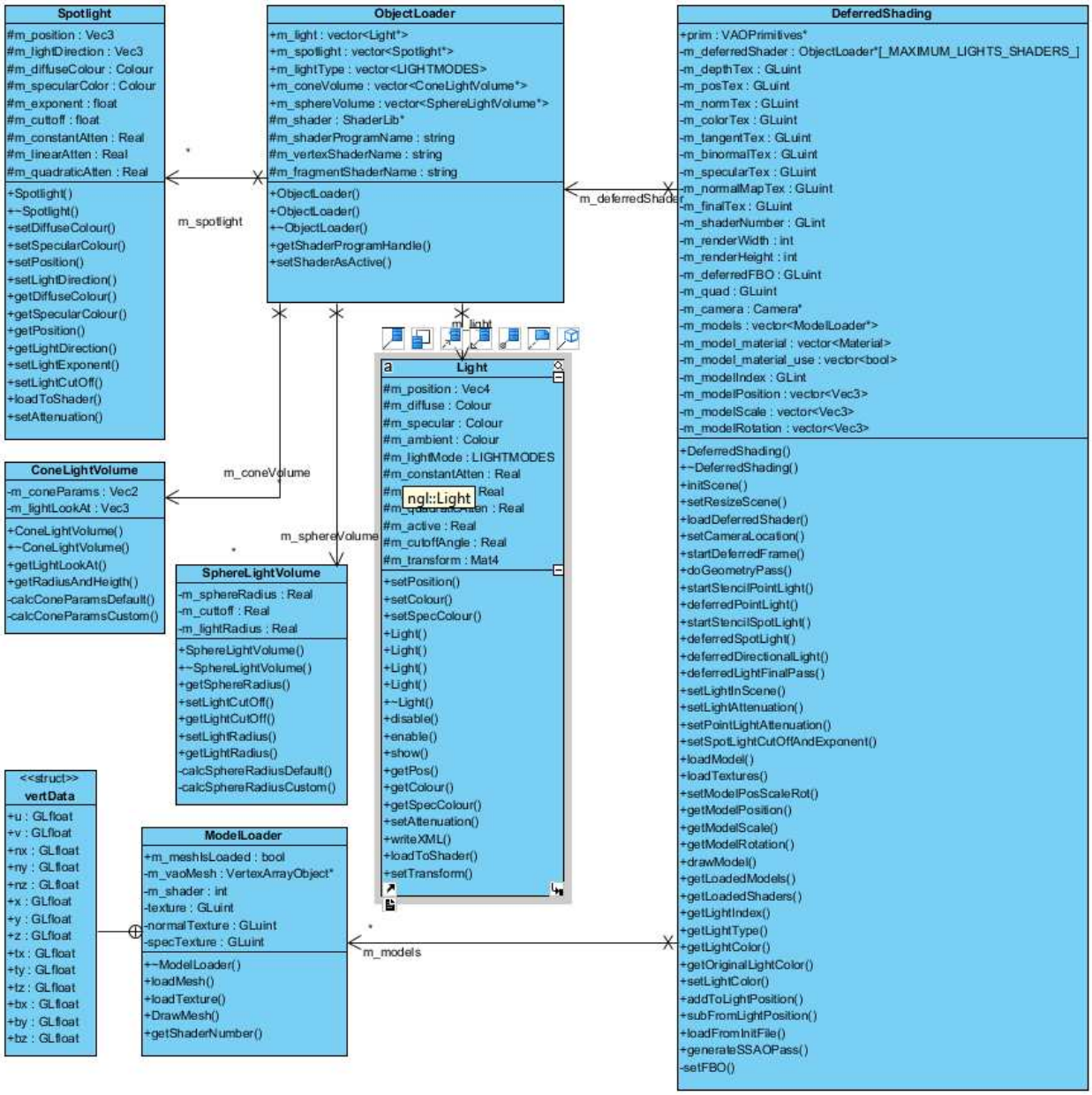
4.2.Deferred shading pipeline

The deferred pipeline was split in the following passes:

- Setting up the rendering targets and deferred frame buffer object (FBO).
- Loading all the deferred shaders that are going to be used for light calculation.
- Adding one or more lights to each shader.
- Preparing the renderer for a new frame, binding the deferred FBO and the final frame texture to be written.
- Start the geometry pass, by binding the render textures and storing all the needed information in them.
- Render the scene's objects with their respective shaders and textures.
- For each loaded deferred shader:
    - For each light, do the stencil pass by drawing the light volume to determine the boundaries in which the light must be calculated.
    - For each light do the light calculation by inside the light volume and determine the fragment color.
    - Blend the textures together to achieve light accumulation.
- Bind back the default FBO (the back-buffer), and copy from the deferred FBO the final image texture.
- Repeat the above steps for the next frame.

4.3.Class Design Diagram

The initial class diagram design is as follows:



4.4.DeferredShading Class

This class is the core of the renderer. Currently it contains all the methods needed for preparing the renderer for a new frame, binding the deferred frame buffer, setting up the render targets, doing the stencil pass and light pass for:

- Directional light
- Point light
- Spot light

The light volume for rendering a directional light is a quad formed by two triangles. The vertices are stored in an array and are normalized to screen space. The quad is declared here and not in a class of its own because essentially is just an array containing a small number of vertices and no other methods are needed for it. A screen filling quad is rendered since a directional light actually affects all the pixels in a scene. Another method suggests that instead of a quad a box can be used (GPU gems2). In this case a quad was used for simplicity.

In the *InitScene()* method we create the geometry for all the lights: a quad for directional light, sphere for point light since it is an omnidirectional light and a cone for a spot light. The geometry for the cone and sphere it is done by using the "ngl::VAOPrimitives" class from the NGL library.

The method *loadDeferredShader()* is using an object of *ObjectLoader class* and stores all the deferred shaders that are will be used for light calculations. The difference between each light is made by *ngl::LIGHTMODES* which is an enumeration in class *Light* from NGL. Unfortunately, the current version of NGL lib has all the light modes equal to 1. In order for the selection to work properly, this enumeration was changed inside the original header file, and a copy added to the project.

The method *setLightInScene()* is the only method responsible to add a new light for a specified deferred shader number. The scope of this method was to be as generalized as possible.
It requires the following parameters:

- Shader number to which the light will be added to.
- Light position.
- Light color.
- Light specular color.
- Light type: directional, point light, spot light.
- Cone/sphere radius and height for calculating the light volume. For the sphere only the radius parameter is taken into consideration. If both values are set to 0.0, a default radius and height will be used.
- Rotation in degrees along x, y, z axis. It will only be taken into consideration for the cone light volume since for a sphere it does not make sense.

The purpose of having one, super generalized method capable of handling all light types is that eventually a technique of reading a scene setup from a file will be added to the renderer and all the information passed to one method instead of multiple.
From this method the light attenuation factors and spot light format parameters are missing. It is possible to have a point light, directional light or spot light with default light attenuation, light

exponent and cutoff. However methods allowing the user to adjust these parameters were added: set *LightAttenuation(), setLightCutOffAndExponent()*.

The methods, *startDeferredFrame(), doGeometryPass(), startStencilPointLight(), startStencilSpotLight(), deferredPointLight(), deferredSpotLight(), deferredDirectionalLight()* and *deferredLightFinalPass()* are responsible for preparing the renderer to process a new frame and calculate the light.

All the other methods are needed in order to be able to access the required information from the vector of loaded lights: getLightIndex() and used in the rendering pipeline: *getLightType(), getLoadedShaders()*.

The method *setCameraLocation()* is responsible for making a local copy of the general camera. This approach was selected instead of passing the camera by reference in all the methods where is needed in order to avoid passing one extra parameter in these methods.

All the methods are **public** so they can be accessed by any object of *DeferredShading* class. The only **private** method is *SetFBO()* which is responsible for creating the deferred frame buffer and set the render targets. As this is done only one per runtime, at the beginning of the program during the initialization stage, this method must have private access.

The overall class suffered multiple iterations to the current design. Initially it was thought to use a *Singleton* pattern. The reason for this is that a renderer should exist only once per application. Later on the class was changed to be a normal class based on the following logic looking towards to more real life applicability:

- In the main application a pointer is crated and memory allocated.
- All the shaders are being loaded and lights attached.
- All the objects (with their materials, textures, etc) for the current scene are being rendered.
- The light calculations are done.

The need to render a new complete scene appears (a different level for example). At this point if a *Singleton* pattern is used, a mechanism to delete all the loaded shaders, lights is needed. Since the deferred frame buffer and render targets are already created it would mean the vectors containing the loaded shaders and lights must also be purged, and a new initialization stage must begin.

Another approach would be not to make a *Singleton* class. This way when a new stage is needed to be rendered, the current pointer to the renderer class can be deleted which calls the destructor of the class and all the previously allocated memory is freed. The downside of this method is that in the class destructor, all the pointers created within the class must also be deleted. Failure in doing so will result in a memory leak. An advantage to this method is that a new “renderer” can be created in advance (before the current level finishes) and at the end of the current scene the new render to be used. After the old renderer pointer is not used anymore, it can be deleted to free up the allocated

memory space. The obvious advantage is that the intermediate “loading level” messages are not needed since the new data is prepared in advanced and thus creating the illusion of a continuous virtual world. A disadvantage to this method could be the need to allocate more memory (RAM memory) and possible it is viable only on computer platforms where the memory limitations are easier to be resolved. The second method is only theoretical with no background evidence to support it at the moment. This is one of the main reasons it was selected for the deferred renderer. Another advantage to this method is that each scene could use its own specialized G-Buffer. More tests are needed in order to confirm the viability of this method, both from the system memory management point of view and CPU to GPU load and utilization.

4.5.ObjectLoader Class

This class is responsible for loading all the needed assets in a scene, like shaders meshes, textures, etc. The present design for this class is not its final form. At the moment it contains the necessary information for manipulating lights: a vector of *ngl::LIGHTMODES* for point light, directional lights and a vector of *SpotLight* type. These vectors were created in this class because in the *DefferedShading* class a pointer to the current class in order to load the shaders used in the light calculations. It also contains vectors for storing all the light volume information (per light).

4.6.SphereLightVolume Class

This class is very basic. It contains three methods: *calcSphereRadiusDefault(), calcSphereRadiusCustom()* and *getSphereRadius().* The first two methods store the sphere radius calculated based on either the maximum value of the light color’s channels or the user provided radius. The last method simply returns the radius value.

4.7.ConeLightVolume Class

The scope of this class is to store all the necessary information for the cone:

- Radius
- Height
- Rotation on X, Y, Z axes
- Look at position

The previous implementation that was used in the prototype stage was completely removed. The algorithm that was initially used was doing the following:

By default if a cone VAO primitive is drawn at a location (x, y, z) this will be oriented like this:

- The center of the base of the cone will be the specified location (x, y, z).
- The cone will be oriented at 180 degrees along the Z axis pointing in the +Z.

If the cone is then rotated -90 degrees (270 degrees) on the X axis the rotation will take place around the specified location (x, y, z). Since this location is actually the light position that is being passed, in this case the cone needs to be translated on –Y axis with the value of the specified cone's height so that the given location (light position) is situated in the top of the cone.

For a -45 degrees rotation on the X axis, the value by which the cone needs to be translated on -Y will be (height/2) and on –Z will be the same (height/2).

An algorithm was deducted based on these observations that calculate the new position where the cone needs to be drawn in order to take into consideration the rotation. At the moment it is incomplete since only the X rotation is being handled.

The current implementation uses an Euler rotation matrix. Since the rotations are affecting only one specific piece of geometry that is used as a light volume the decision was made not to use a Quaternion matrix since the up orientation is irrelevant in this case.

For the cone light volume a custom mesh was generated in Autodesk Maya converted to a header file using the provided tool found in *NGL* library and loaded as a predefined Vertex Array Object. This method was selected based on two considerations: the pivot of the cone is located in the apex of the cone rather than at the base as the cone from the standard *NGL* library is. This saves two extra operations of translating the pivot, rotating and translating back. The second consideration is that the generated cone has a full base unlike the *NGL* one that had no base, so an extra circle was needed to be rendered. The base is extremely important in order for the stencil operation to succeed and the light to be visible as it is projected on it.

4.8.Spotlight Class

This class is used to model a simple spot light type. The declared attributes are color, specularity,the light cutoff with a maximum angle of 90 degrees and the light exponent factor. The light attenuations factors were also added to this class in order to try and model a realistic light.

All the attributes of this class are declared as **protected** so a more complete model of the spot light to be created starting from this class and in conjuration with Runtime polymorphism the model from either the base or derived class can be selected based on the provided parameters.

4.9.ModelLoader Class

This class is used to handle the loading of models as OBJ files. It is also responsible for loading three types of textures: diffuse color, normalMap and specular color. The textures are loaded only once, in the beginning of the application. The information is stored in proper render targets so it can be sampled during the light calculation pass. The class is also responsible for storing all the information that is read from the object files in a structure: the vertex, normals and UV coordinates and computing the tangent and binormal factors. Other methods implemented in the class are the global draw method that will render all the models attached to a shader. The algorithm for loading OBJ and texture files was developed and implemented by Jon Macey. The shaders for applying the diffuse, normal and specular maps were adapted to the deferred renderer pipeline.

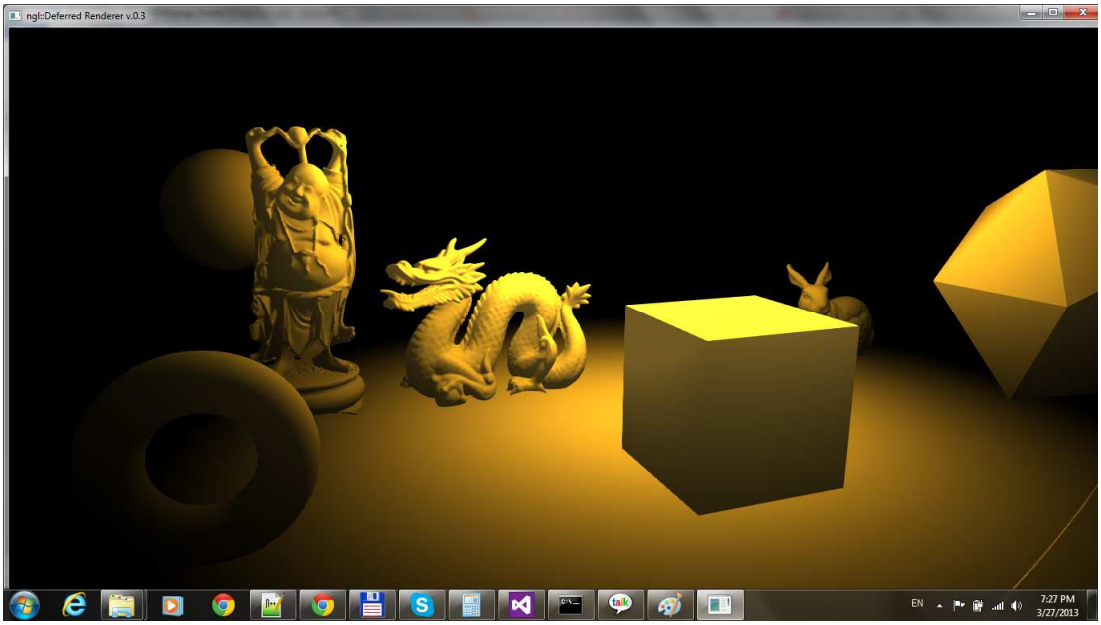
5. SSAO (Screen Space Ambient Occlusion)

The renderer has been fitted with an extra pass to calculate the ambient occlusion in order to mimic the global illumination. The classic method contains in developed by Crytek uses a depth buffer, it samples fragments around it in 3D space, projects them back to screen and compares the depth of the sample with the depth of the position in order to find out if the sample is in front (not occluded) or behind a surface (occluded). An occlusion buffer is then generated based on the average distance of the samples.

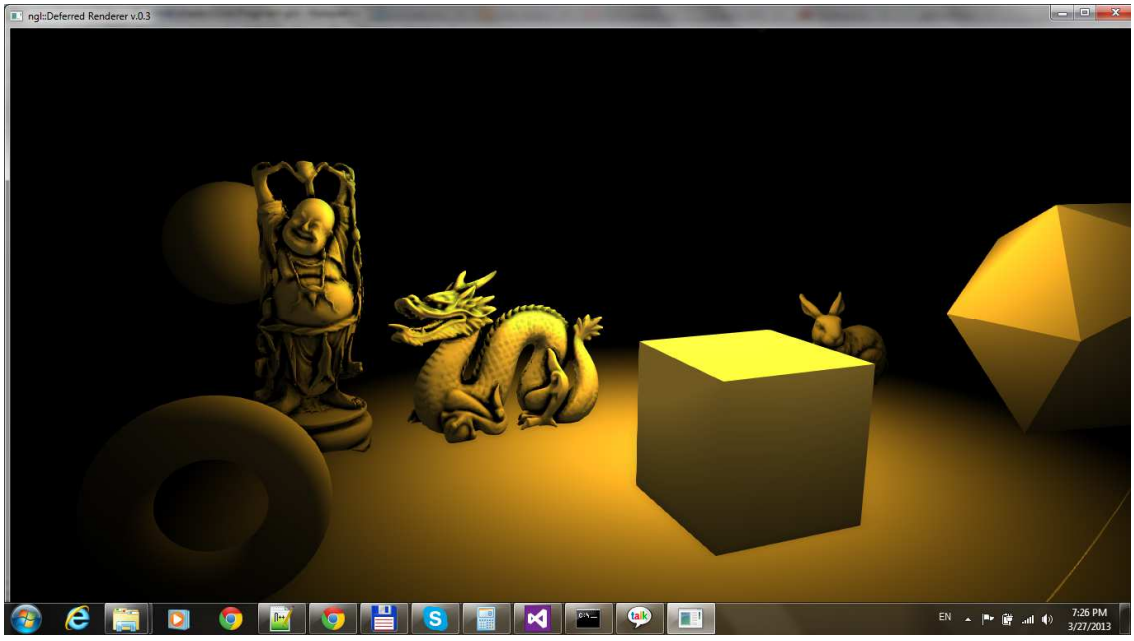
SSAO (Screen Space Ambient Occlusion) Pass Output

The method that was selected to be implemented however doesn't use this algorithm. Instead the position is used along with the normal buffer. The occlusion will be generated as one component per pixel and stored in a buffer. (Jose Maria Mendez, 2010) This value is then rendered directly in the final texture on the alpha channel to save up space, but also to blend it with information already present in final texture render target. The resulted image contains all the geometry, light and ambient occlusion information and is ready to be rendered to the screen. In order for this to work, the SSAO must be calculated after the light pass. This approach was selected in order to save an extra render target.

The algorithm was created by Jose Maria Mendez and was implemented in HSLS. The shader presented in his article was adapted to GLSL and fitted in the current deferred renderer pipeline.



Final Buffer image using a point light and no SSAO (Screen Space Ambient Occlusion) Pass



Final Buffer image using a point light with SSAO (Screen Space Ambient Occlusion) Pass

6. Conclusions are future work

The deferred renderer uses the deferred shading technique. The implementation was done based on the above design. While the results so far are pretty impressive some known issues still exist, that were found during extensive periods of testing and also a lot of features can be added , or current methods swapped with more efficient ones.

The future work will mainly consist of:

- fixing a design problem for the directional light so it supports positioning in the world
- implementing the a projected light
- refining the current code
- implementing a more efficient model loader.
- implementing other formats support for loading model and textures
- implementing methods/shaders for rendering transparent objects after the deferred stage
- possible implementation of post process effects like bloom
- possible implementation of an anti-aliasing method like the FXAA or even MSAA

7. Reference:

- Dudash B. *Next Generation Shading and Rendering*. Nvidia Corporation. Available from: https://developer.nvidia.com/sites/default/files/akamai/gamedev/docs/English_Advanced_Shading.pdf. [Last accessed 10 December 2012]

- Hargreaves S., Harris M., 2004. *Deferred Rendering*. NVIDIA Corporation. Available from: http://developer.download.nvidia.com/assets/gamedev/docs/6800_Leagues_Deferred_Shading.pdf. [Last accessed 10 December 2012]

- Koonce R. and Nguyen H., 2007. *GPU Gems 3*. Addison-Wesley Professional.

- Lottes T., 2009. *FXAA*. NVIDIA Corporation. Available from: http://www.ngohq.com/images/articles /fxaa/FXAA_WhitePaper.pdf . [Last accessed 10 December 2012]

- Meiri E., 2012. *Tutorial 35 - Deferred Shading - Part 1*. Available from: http://ogldev.atspace.co.uk /www/tutorial35/tutorial35.html. [Last accessed 10 December 2012]

- Meiri E., 2012. *Tutorial 36 - Deferred Shading - Part 2*. Available from: http://ogldev.atspace.co.uk /www/tutorial36/tutorial36.html [Last accessed 10 December 2012]

- Meiri E., 2012. *Tutorial 37 - Deferred Shading - Part 3*. Available from: http://ogldev.atspace.co.uk /www/tutorial37/tutorial37.html. [Last accessed 10 December 2012]

-OpenGL documentation. 2012. *Common Mistakes*. The Industry's Foundation for High Performance Graphics. Available from: http://www.opengl.org/wiki/Common_Mistakes#Texture_upload_ and_pixel_reads. [Last accessed 10 December 2012]

- Mendez J.M., 2010 *A simple and practical approach to SSAO.* Available from: http://www.gamedev.net/page/resources/_/technical/graphics-programming-and-theory/a-simple-and-practical-approach-to-ssao-r2753 [Last accessed 28 March 2013]

- Reshetov A., 2009 *MLAA*. Intel Labs. Available from: http://visual-computing.intel-research.net/publications/papers/2009/mlaa/mlaa.pdf [Last accessed 10 December 2012]

Shishkovtsov O. and Pharr M., 2005. *GPU Gems 2*. Addison-Wesley Professional.

- Villa J.R. *OpenGL Shading Language Course*. TyphoonLabs Available from: http://www.opengl.org/sdk/docs/tutorials/TyphoonLabs/Chapter_1.pdf [Last accessed 10 December 2012]

- Wolff. D and Iyer K., 2011. *OpenGL 4.0 Shading Language Cookbook*. Birmingham: Packt Publishing Ltd.

- Wolfgang E.,2008. *Light Pre-Pass Renderer*. Available from: http://diaryofagraphicsprogrammer. blogspot.co.uk/2008/03/light-pre-pass-renderer.html [Last accessed 10 December 2012]

- Zima C., 2008. *Creating the G-Buffer*. Available from http://www.catalinzima.com/tutorials/deferred-rendering-in-xna/creating-the-g-buffer. [Last accessed 10 December 2012]