MSc CAVE 2012/13
Bournemouth University - NCCA

Final report for the course : Animation Software Development

Topic: DEFERRED RENDERING
Fabio Turchet
i7266699

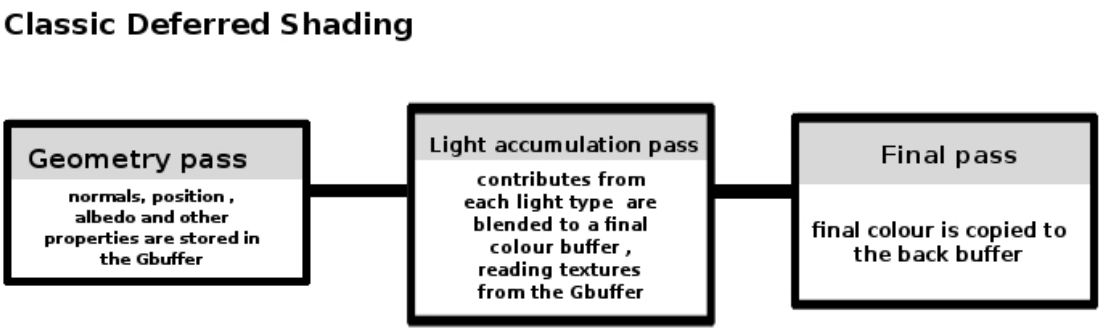
This report is an extension and integration of the initial. Here I will discuss in particular the implementation, the problems encountered and their solutions, the design and the algorithms used.

Introduction

Modern video games are using more and more a deferred approach at the core of their rendering pipeline and today it seems that the technique is widespread. Moreover it's an area of constant research and development and this makes it an even more interesting subject to study.
Here is presented an initial research for the development of a hybrid deferred renderer for real time visualization of medium complexity scenes : at the beginning a general overview will be given , then main advantages and drawbacks will be considered. In the middle part a possible initial design is discussed and finally the goals to achieve are presented.

Deferred Shading

The idea of deferred shading is not new and was first introduced by Deering (Deering 1988).
The core concept is that to produce the final image at least 2 passes are needed.
As explained by Möller et al. (2008, p. 279) in the first pass (that we can call “geometry pass”) all the geometric attributes needed for lighting of the objects in the scene are stored to a secondary buffer thanks to the availability on modern GPUs of multiple render targets , that is texture buffers. The geometry attributes stored are typically : position , z-depth , albedo (colour) , normals and all those attributes for the materials needed to compute a desired lighting model.
In the second pass , because all the geometry was rendered to a buffer , the lighting calculations can be done via shaders as well as many post processing effects such as motion blur , volumetric fog, depth of field , ambient occlusion . What is done is actually a decoupling of geometry from lights and this allows to lower the complexity when many lights are in the scene. The pixels affected by each light are obtained by performing intersection tests with bounding volumes : for a point light a sphere is used , for a spot light a cone and for a directional a full screen quad (all pixels are processed).



Actually after the geometry pass , everything becomes a post FX because the geometry is not rendered any more but read instead from the buffers.
The buffer storing MRTs is commonly known as “G-buffer” , a term first introduced by Saito and Takahashi (1990).

Advantages

The main advantage comes from a simple consideration in terms of complexity (Möller et al. 2008, p. 278)
Given m = number of objects and n = number of lights
For a multipass and single pass renderer the upper bound cost in the worst case is:
$complexity = O(m*n)$
In the deferred case instead :
$complexity = O(m+n)$
or , considering the number of pixels actually lit :
$complexity = O(numPixels*n)$

Two interesting presentations help clarify this a bit more. One is by Shawn Hargreaves from Climax (Hargreaves 2004) and the other by Michal Valient presenting the architecture behind Killzone2 (Valient 2007) .
It is explained there that in a single pass forward renderer in fact the loop goes through each object and finds the lights affecting it. Then for each of these lights the colour is calculated and accumulated.
In a multipass forward renderer the difference is that one loops through each light and for every object the colour is blended.

These approaches are good in the case of a low number of lights and materials. But they present various problems for example the fact that the number of shaders becomes very high , in a combinatorial manner. In the example showed by Möller , using four light types, six lights in the scene five material types , 25 shaders are required for multipass technique , 1050 for single pass and only 11 for deferred , allowing “strong separation between lighting and material definition” (Möller et al. 2008, p. 281)
The other big advantage is that light calculations are done once per pixel , avoiding this way the overdraw created by blindly forward rendering all the objects (even if a Z early pass helps a lot).

Disadvantages

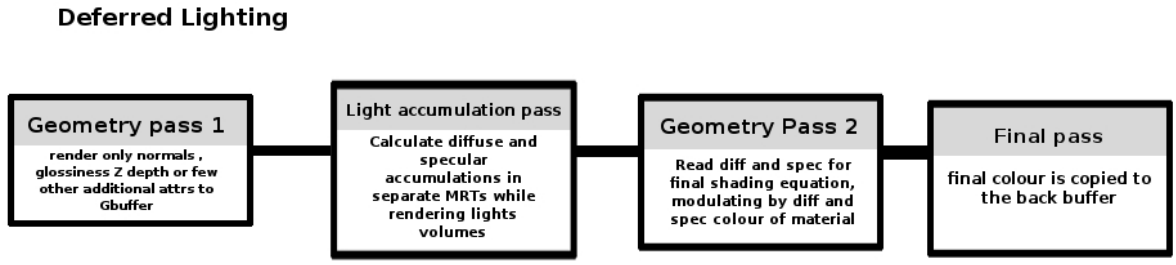
The step forward towards lower complexity in computations per pixel comes at a cost that is an higher complexity in space.
It is clear that the more information is stored in the Gbuffer , the more data has to be transferred per frame through the pipeline and this can be a problem if the bandwidth is saturated with 4 or 5 render targets at once. Despite this bandwidth “problem”, deferred shading was adopted by Unreal 3 and 4 engine (2012).
Another major drawback concerns transparent objects. In on order to render those , a post pass is needed with sorting back to front.
Also, the variety of materials is limited by the attributes packed in the render targets and often the limitation extends to a single lighting model.
Finally, anti aliasing operations are not straightforward and require additional techniques.

Deferred Shading vs Deferred lighting

A variant of the classic technique that is used in more and more modern engines makes use of Deferred Lighting .
Hoffman (2009) explains this process as being constituted of 3 passes instead of 2.
This technique presents a different approach to the calculation of the lighting equation that is split into its 2 major components of diffuse contribution and specular contribution:

$$L = \text{object diff} * L_{diff} + \text{object Spec} * L_{spec}$$

this means that is possible to calculate separately the 2 factors , accumulate them in buffers and read them back in a subsequent pass.
The following image illustrates the concept better :



The major advantage is a significant reduction of the Gbuffer size allowing multiple materials at the cost of adding an extra geometry batch.

Anyway , there is not collective agreement that this technique is actually better even if it was adopted by Crytek and is more and more popular. In fact at the beginning of the research, it looked like light pre pass was the way to go because in theory was the solution to the excess of bandwidth. A more in depth research showed that for example in terms of bandwidth the gain is not that much (5X vs 6X) (Kaplanyan 2010 , p. 33) This is confirmed also by some considerations done by Stone

(2009) .
It looks like the advantages of deferred lighting are not that important , also considering the target platform of this project (PC) in fact the second geometry pass could heavily hit performance on non hyper specialized hardware such as the one on consoles or just when the number of triangles in the scene are of a considerable amount.

Another variant is the light pre pass introduced by Engel (2008), but according to Hoffman (2009) : “the deferred lighting approach that also stores the specular spread in the initial pass is vastly preferable, especially since m adds only a byte to each pixel in the G-Buffer. The shading controls have a physical basis, control is straightforward, and the algorithm is efficient and compact” ,where “m” is the glossiness of the material.

Finally one of the most recent versions is called indexed deferred lighting (Trebilco 2008) , but it was not analysed in depth to understand the real benefits.

Design

In terms of design , the first consideration to be taken into account concerns the type and structure of the Gbuffer. The more attributes are contained here , the bigger it becomes and thus potentially affecting negatively the fill rate.
In particular a trade-off has to be searched between precision in terms of floats used to identify the attributes and final desired quality.
Because the Gbuffer is so critical several techniques have been developed in order to pack the attributes in a Render Target limiting the memory bandwidth consumption.
According to Shishkovtsov (2005) the best performance was obtained by not packing position and normal attributes because the overhead computation in the shaders is minimized. At the same source is possible finding a table showing the cost in terms of extra operations of each type of encoding for normals and position attributes. In any case , as showed by Evans
and Kirczenow (2009) the Z component reconstruction of the normal in view space can lead to a negative value so it's a myth and should be avoided.
In order to provide a way to compare the different possibilities of Gbuffer configurations, the design of the renderer will attempt to support Gbuffers with RenderTargets having different sizes (32 and 64 bits) and different options of encoding the information. While this would be an interesting comparison test , the practical utility could be the possibility for the application to run also on (relatively) older machines.

Modifications after the initial design

The configuration of the Gbuffer didn't change much as shown below.

R16	G16	B16	A16	
Normal X	Normal Y	Normal Z		RT0
Position X	Position Y	Position Z		RT1
Diffuse R	Diffuse G	Diffuse B		RT2
Spec Col R	Spec Col G	Spec Col B		RT3
Depth32		stencil8		RT4
Final Col R	Final Col G	Final Col B	Final Col A	RT5

I decided to use 16 bits per channel Render Targets as 32 bits was an accuracy not really necessary, Moreover the Specular RT is present but not used at the moment. It is there to allow a quick extension in case of specular maps read from file.
An optimization that is generally done in the industry is to store only the depth in the Gbuffer (encoding it) and then reconstruct the other coordinates starting from the fragment position in the shader. I am well aware of this technique, but I found that the performance was not a big issue so I skipped it.
It looks like that openGL now supports render targets of different format so it is possible for example to store the normal as 16F and the colour as GL_RGBA8 . This is really helpful to reduce the size of the Gbuffer that can become very quickly fat. With today's hardware and recent graphics cards bandwidth is not a big issue, but optimization in real time has to be always the priority.

The core class diagram also didn't change much, proving the initial design was quite extensible for the project. In particular the extensive use of abstract classes made things a lot easier and avoided duplication of code. A good example is the RenderPass class.
The design is taking into account the possible future implementation of the renderer with a different API, so the Factory pattern was used, adapting the example by Reddy (2011 , pp 88-91).
The new classes added are essentially extensions of NGL classes , like Obj, Camera, AABB. These extensions were needed for example to implement frustum culling based on AABB.
As suggested in the course feedback, the LightManager class was completely removed and its members (essentially std::vector's) moved to the World class.

Program Flow

Shown below is the show of renderer. Most of the job is done in the draw() function of the Renderer where pre , normal and post passes are performed sequentially. An early Z pass was considered as very first pass but not implemented. It should provide a bit of speed up in case of many objects in the scene.

After openGL context initialization, the main camera is created as well as the RenderGL object. The World class takes care of initializing textures and objs to load.
The very first pass of the renderer performs a sorting of all the objects in the scene. At the moment only by depth, but it is easy to pass a custom function. This way we could order sequentially groups of objects for example by shader and by geometry type to avoid in particular overdraw and GPU state changes especially for shaders that are expensive.
Also, every pass performs a frustum culling operation in order speed up the rendering.

After the sorting, we generate the shadow maps. I didn't want to create a new FBO just for that. The reason why I was considering the idea was because it is not possible to have two depth buffers at the same time attached, which makes sense. Also I didnt want to perform blits or texCopy every frame as it is slow. The solution I adopted was to just create a detached texture in depth format and attach it/ detach it dynamically when needed to write the maps qithout destroying the object. This way the depth buffer of the Gbuffer was not affected or overridden and we have a texture ready to use later in the light pass to calculate the shadows.

Once this is done for just the lights that cast shadows (this check is not yet implemented though) , we can perform the geometry pass and store the info we need in camera space into the textures.

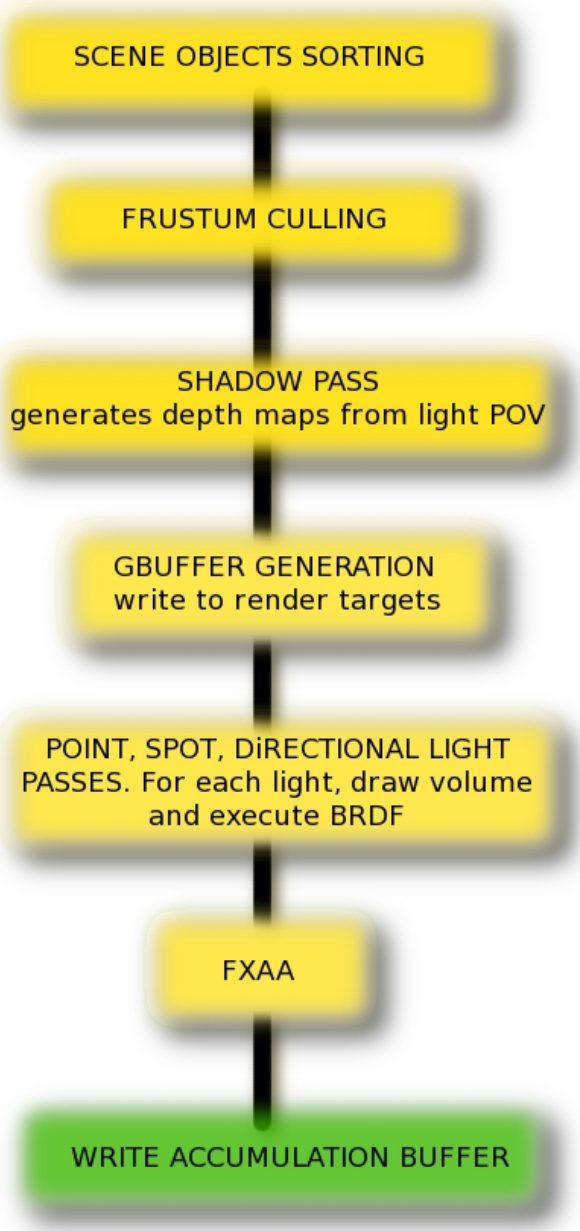


Illustration 1: Deferred Renderer workflow

The next step is drawing the light volumes. In order to pass to the shader only the vertices inside the geometry, a combination of depth test and stencil test is performed. First the stencil buffer is populated with just a vertex shader and then based on that and the depth test it is possible to perform the pixel detection . A case that still has to be implemented is when the camera is inside the

light volume, which equals checking if the near plane is inside the bounding box. For lack of time I couldn't add this simple feature. For each light we draw its shape : a sphere for point lights, a cone for spot lights and a full screen quad for directional.

The shaders of each light fetch the parameters stored in the geo pass and perform the light calculation. One of the problems I had was that in this pass the geometry has to be semless with no overlapping faces. So I couldn't use ngl primitives and use instead objs exported from Maya. Another problem of ngl primitives is the lack of bounding box calculation for collisions and intersect tests, needed for culling. Also in that case I used normal objs. A normal Phong shader was adapted for the use in deferred, taking into consideration that normals and positions were calculated in eye space so all the lighting had to be done relative to the camera, as it is common for a renderer (The other option is doing it in world space).
For the shadow shader see dedicated section .
Every light pass contributes to the final result (GL_BLEND) and writes to the accumulation buffer.

Once the light pass is done, the final result can immediately be output to the backbuffer.
But one of the cool advantages of a deferred approach, besides supporting a very high number of dynamic lights and keeping performance, is that we have stored on the GPU all the textures we need to perform post processing FX , like a compositing software. SSAO, DOF, fog, AA are quite easy to achieve in deferred rendering. So as pre last pass is possible to blend in all these effects.

This pipeline is not so complicated as the one in a modern renderer, like the one showed by Engel (2008), the “inventor” of the deferred lighting variant . In fact, a real renderer needs anyway a forward pass to draw transparent objects , that would have been a nice extension to do, if time allowed.

At the very end, we output the accumulation buffer out , binding the main frame buffer.

Techniques and Algorithms

Frustum culling
I have implemented the check for bounding boxes that was missing from NGL. The Camera and AABB classes were extended to do that.

Object sorting
It is performed in the RenderGL::draw() . It is a for loop using std::sort. What is useful is that the user can pass any boolean function to extend the sorting methods , like
RenderCommon::sortByDepth()

Shadows
Shadows have been implemented , even if they are a bit buggy and need to be reviewed.
The technique is the classic one of shadow maps, I used different articles as reference such as the one by CodingLabs (2010)
The trick to have shadows is considering that the positions we need are already in the textures, but they are in camera space. So in order to go to light projected space, they first need to be transformed into world space. This is done in the Spotlight fragment shader :

```
// go back to world pos
vec4 posWorld = Vinv*vec4(eyeSpacePos,1.0);
// go to light texture space
vec4 LightSpacePos = textureMatrix*posWorld;
```

```
    float ShadowFactor = CalcShadowFactor(LightSpacePos , TexCoord);
```

where *Vinv* is the inverse of the camera view matrix and *textureMatrix* is the transformation to go from world space to the light point of view space (projection included)

I also tried the optimization for openGL using textureProj and sampler2DShadow
with luminance test but didn't work as expected.
Shadows present some nasty banding that I couldn't get rid of even by lowering the threshold.

The methods to swap the depth buffer in order to not override the existing depth information of the shadow maps are
`GeoBuffer::resetDepthTarget()`
and
`GeoBuffer::setCurrentDepthTarget(TextureGL *_newDepthTarget)`

Stencil check and light Volumes

The volumes automatically shrink or grow based on the intensity set and based on the attenuation. The orient the cone properly I used an aim matrix forged properly and put on the stack together with the scaling factor. I implemented a method SpotLightRenderPass::getConeVolumeRadius
to automatically get the proper scale given the height (distance to target). This avoids regenerating vaos and new geometry.

The idea to detect the pixels inside the volume is to use the correct stencil functions

```
glStencilFunc(GL_NOTEQUAL, 0, 1);

glStencilOp( GL_KEEP, GL_INCR, GL_KEEP);

Essentially we have to imagine a ray cast from the camera that intersects the
volume and increase or decrease the stencil value by checking the depth value
generated previously.
```

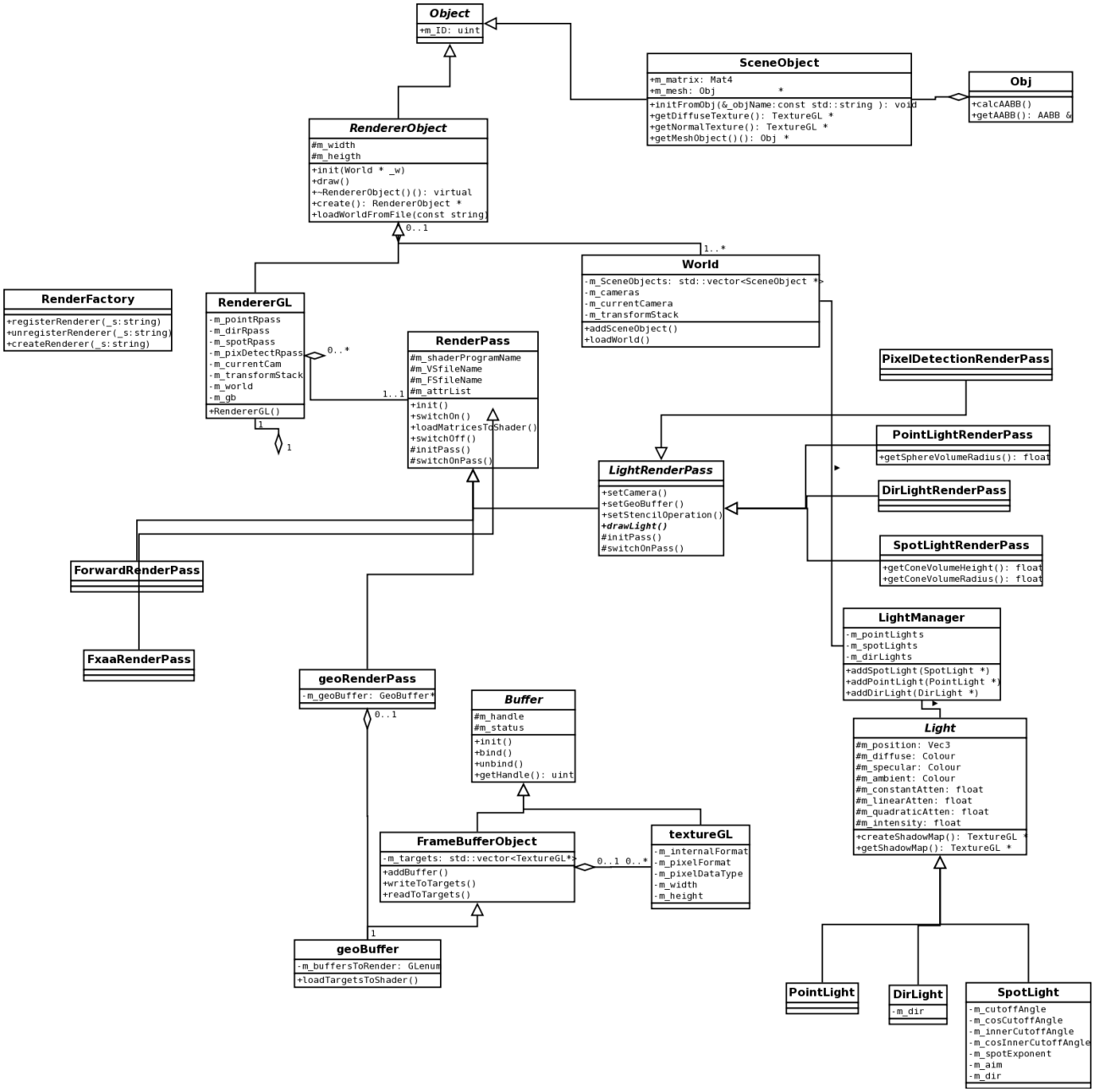
Anti Aliasing

As a test for post processing effects, I added edge antialiasing adapting the technique and shader by NVIDIA called FXAA that is fast and performs the operation in screen space like a normal image processing algorithm (Lottes 2009)
To show the affected pixels only there is the define called
`#define FXAA_DISCARD 1`
which proved to be really useful for debugging (Nvidia cards only) .
I researched and studied the algorithm which proves to be a very interesting technique.
The most important thing to know is how to pass the luminance values and knowing if the input texture is sRGB. In my case I didn 't have time to pack the luminance of the pixels in the alpha channel so i m using the
`#define FXAA_GREEN_AS_LUMA 1`

I've stripped out from the original code all the extra checks and not needed code. I ve tried to tweak the parameters and using the presets, but the results were not extraordinary so it is something to look into better in the future.

In terms of shaders, I took care to reck the shaders and make sure they were not binding the attributes application side, but in the new way offered by openGL that is via the layout (location=X)

keyword .


Object
+m_ID: uint
SceneObject
+m_matrix: Mat4
+m_mesh: Obj *
+initFromObj(&_objName:const std::string): void
+getDiffuseTexture(): TextureGL *
+getNormalTexture(): TextureGL *
+getMeshObject()(): Obj *
Obj
+calcAABB()
+getAABB(): AABB &
RendererObject
#m_width
#m_heigth
+init(World * _w)
+draw()
+~RendererObject()(): virtual
+create(): RendererObject *
+loadWorldFromFile(const string)
0..1
1..*
World
-m_SceneObjects: std::vector<SceneObject *>
-m_cameras
-m_currentCamera
-m_transformStack
+addSceneObject()
+loadWorld()
RenderFactory
+registerRenderer(_s:string)
+unregisterRenderer(_s:string)
+createRenderer(_s:string)
RendererGL
-m_pointRpass
-m_dirRpass
-m_spotRpass
-m_pixDetectRpass
-m_currentCam
-m_transformStack
-m_world
-m_gb
+RendererGL()
0..*
1
1
RenderPass
#m_shaderProgramName
#m_VSfileName
#m_FSfileName
#m_attrList
+init()
+switchOn()
+loadMatricesToShader()
+switchOff()
#initPass()
#switchOnPass()
1..1
PixelDetectionRenderPass
PointLightRenderPass
+getSphereVolumeRadius(): float
LightRenderPass
+setCamera()
+setGeoBuffer()
+setStencilOperation()
+drawLight()
#initPass()
#switchOnPass()
DirLightRenderPass
SpotLightRenderPass
+getConeVolumeHeight(): float
+getConeVolumeRadius(): float
ForwardRenderPass
FxaaRenderPass
LightManager
-m_pointLights
-m_spotLights
-m_dirLights
+addSpotLight(SpotLight *)
+addPointLight(PointLight *)
+addDirLight(DirLight *)
geoRenderPass
-m_geoBuffer: GeoBuffer*
0..1
Buffer
#m_handle
#m_status
+init()
+bind()
+unbind()
+getHandle(): uint
Light
#m_position: Vec3
#m_diffuse: Colour
#m_specular: Colour
#m_ambient: Colour
#m_constantAtten: float
#m_linearAtten: float
#m_quadraticAtten: float
#m_intensity: float
+createShadowMap(): TextureGL *
+getShadowMap(): TextureGL *
FrameBufferObject
-m_targets: std::vector<TextureGL*>
+addBuffer()
+writeToTargets()
+readToTargets()
0..1 0..*
textureGL
-m_internalFormat
-m_pixelFormat
-m_pixelDataType
-m_width
-m_height
1
geoBuffer
-m_buffersToRender: GLenum
+loadTargetsToShader()
PointLight
DirLight
-m_dir
SpotLight
-m_cutoffAngle
-m_cosCutoffAngle
-m_innerCutoffAngle
-m_cosInnerCutoffAngle
-m_spotExponent
-m_aim
-m_dir

Implementation and main Classes

RendererGL
The openGL implementation of the RendererObject. This is the main object that draws the scene and manages all the different phases of the pipeline. It accepts a scene description in the form of a World pointer , but in the future will support also loading from file.
In terms of design the extensible factory paradigm will be applied , adapting the example by Reddy (2011 , pp 88-91).
The important point here is that objects don't striclty have a draw method in them. This is because in terms of design, the object is just a container of information. What it needs are the material , the transformation and the mesh to use and then it's the renderer that takes care of ordering internally the objects and deciding through various techniques (instancing for example) how to draw them SO the responsibility and the “power” are not inside the object but rather are given to the centralized renderer. The main reason for this is optimization.

```
RendererGL::drawObject(SceneObject *_so, bool _drawAABB)
```
gives tthe possibility to draw an object with boudinig box, for debug purposes

World
The class that contains the array of SceneObjects , array of lights , array of cameras , in other words it s the scene representation. For the lights, the choice is to use different arrays for each light type. The reason for this is that the light pass needs to loop for each type. If an array of Light* was used instead, an extra cost for ordering by type and dynamic casting (small) would be added.

SceneObject
Implementation of the Object interface that adds the attributes characteristic of a 3D entity , including the model matrix that can be queried when constructing the MVP by the renderer.

This class was extended quite a bit since the prototype to add texture initialization and support to attach an obj mesh or a ngl primitive. Core methods are

```
SceneObject::initFromObj()
and
SceneObject::initFromPrimitive()
```

FrameBufferObject , Gbuffer , TextureGL
A Gbuffer is an FBO so it s deriving from it. But it has some specific methods and a particular initialization so it needs to be a separate class. The paradigm of aggregation applies here as the class contains one or more RenderTargets (no need to differentiate it as separate class because they are essentially textures) .

RenderPass (and derived)
A renderpass is a class that in order to perform a task at render level needs to be interfaced with a shader. The light passes and the post Effects inherit from it. An easy way to think of renderpasses is as modules that the renderer combines to build it s pipeline.
If the user wants to create a new effect, it is simply a matter of deriving from this class.

Obj
Extension of NGL::Obj to support AABB calculation

Camera

Extended to provide Camera::isBoxInFrustum(AABB &_bbox)

Light classes
Using the ngl lights as a starting point I implemented reworked versions to have more freedom with the light creation. All the light types derive from an abstract Light class implementing class polymorphism.
Light contains methods to create the shadow maps

```
TextureGL * createShadowMap(unsigned int _width , unsigned int _height);
```

RenderCommon
Here I put all those utilities and functions commonly used by a multitude of classes such as the ones to load matrices to shader.

Optimizations

The following list contains the main optimization areas and techniques for a deferred renderer.

The following table summarizes some of the possible optimizations presented by Placeres (2005)

Problem	Possible solutions
Too many material attributes needed	Store ID of material for subsequent indexed look up or texture fetch
Reducing the number of light sources looped	Occlusion/visibility culling tests . Discard far/ too small lights. Assigning a fixed number of sources that can affect a pixel or screen region
Reducing the number of pixels passed to the shader	Discard back facing polygons of bounding volumes. Scissor test and clipping planes. Use of stencil pass to tag in volume pixels.
Gbuffer too heavy	Packing attributes and encoding
Lower per fragment computations	Don't calculate full lighting equation based on LOD and blending by distance . Calculate lighting on render targets half resolution of the original, but keeping the colour texture full resolution
AntiAliasing	MLAA , DLAA , post MSAA , FXAA
Transparent objects	Render them after the opaque ones in a forward way. Use depth peeling algorithm (Bavoil and Myers 2008)
General optimization to reduce gpu state change	Batching the polygons of the objects and the polygons of the light sources in order to draw for example 30 point lights with one call. Possibly an ubershader instead of 3 or more light shaders.

Were possible , in terms of shader performance , the attribute locations are pre registered.

Goals achievements / Future improvements

I wanted to add many features but time didn't allow it. I had to scrap the Sponza demo as implemented the obj support too late. In replacement in the demo scene I am using models with normal maps and diffuse texture. In particular a chapel , a detailed video game character and a plane to use as terrain.
I wanted to implement transparency of object. This would have been the last pass and done in a forward manner just for specified lights.
I also scrapped the Gbuffer format idea that could have been useful. Also, a proper debugging system would be a must feature to have in the future.
In summary the result achieved is satisfactory and it proved that it would not be so difficult to extend the program with new effects. A more modular approach would have been more beneficial though.
A config file is missing and would be useful to load the scene. Also, a system integrated with the operating system to intercept system calls when shaders are modified so they get recompiled in real time without exiting the program would optimize dramatically the debug time.

Known issues

Shadows are buggy.
Need to fix sending of parameters to shader in order to avoid ngl print errors.
A lot of code optimization possible and a lot of warning to fix at compile time.

Usage

Use mouse controls to zoom, pan and rotate the view.

References

Bavoil , L., Myers K. , 2008 . “Order Independent Transparency with Dual Depth Peeling ”. NVIDIA document. Available from *http://developer.download.nvidia.com/SDK/10/opengl/src/dual_depth_peeling/doc/DualDepthPeeling.pdf* [Accessed 20 November 2012]

Codinglabs, 2010 ,“Deferred Rendering Shadow Mapping” . Available from *http://www.codinglabs.net/tutorial_opengl_deferred_rendering_shadow_mapping.aspx* [Accessed 24 March 2012]

Deering, M. , Winner S., Schediwy B., Duffy C. and Hunt N. Aug. 1988 . "The triangle processor and normal vector shader: a VLSI system for high performance graphics". *ACM SIGGRAPH Computer Graphics* (ACM Press) 22 (4), 21–30

Engel, W. 2008 “Renderer Design for Multiple lights” . Available from http://pisa.ucsd.edu/cse125/2008/lectures/engel-renderers-sp08.ppt [Accessed 10 December 2012]

Evans , A. , Kirczenow , A. 2009 .“Regressions in RT Rendering: LittleBigPlanet Post Mortem” Presentation at SIGGRAPH2009 New Orleans. Available from http://prelight.googlecode.com/files/final_alex_siggraph.ppt [Accessed 7 December 2012]

Möller, T. , Haines, E. , Hoffman , N. , 2008. Real Time Rendering. 3rd edition. Wellesley : A K

Peters, Ltd.

Hargreaves , S. , 2004 . “Deferred Shading” Presentation at GDC-
Available from : http://www.talula.demon.co.uk/DeferredShading.pdf
[Accessed 1 December 2012]

Hoffman , N. , 2009 , “Deferred lighting approaches” Available from :
http://www.realtimerendering.com/blog/deferred-lighting-approaches/
[Accessed 1 December 2012]

Kaplanyan , A. , 2010 , “CryENGINE 3: reaching the speed of light” . Available from :
http://www.crytek.com/download/AdvRTRend_crytek.ppt [Accessed 1 November 2012]

Klint , J. , 2008 , “Deferred Rendering in Leadwerks Engine ” , online document available at
http://www.leadwerks.com/files/Deferred_Rendering_in_Leadwerks_Engine.pdf [Accessed 15 October 2012]

Lottes, T. 2009 , “NVIDIA FXAA” , Available from
http://developer.download.nvidia.com/assets/gamedev/files/sdk/11/FXAA_WhitePaper.pdf
[Accessed 6 March 2012]

Mittring , M. , 2012 , “The Technology Behind the Unreal Engine 4 Elemental demo” . Presentation at SIGGRAPH2012 for the Advances in Real-Time Rendering in 3D Graphics and Games Course . Available from
http://www.unrealengine.com/files/misc/The_Technology_Behind_the_Elemental_Demo_16x9_(2).pdf [Accessed 15 November 2012]

Placeres , F. , 2007 . “Overcoming Deferred Shading Drawbacks” , pp. 120– 130, in ShaderX5: Advanced Rendering Techniques edited by Engel, W. Boston: Charles River Media

Reddy, M. , 2011. “API Design for C++” , Burlington : Morgan Kaufmann.

Saito, T. , Takahashi , T. , 1990 . "Comprehensible rendering of 3-D shapes". *ACM SIGGRAPH Computer Graphics* (ACM Press) 24 (4): 197–206

Shishkovtsov , O. , “Deferred Shading in *S.T.A.L.K.E.R.* “ in GPU gems 2. Available from
http://http.developer.nvidia.com/GPUGems2/gpugems2_chapter09.html [Accessed 1 November 2012]

Stone , A. , 2009 , “Deferred Shading Shines. Deferred Lighting? Not So Much.” . Available from :
http://gameangst.com/?p=141 [Accessed 7 December 2012]

Trebilco , D. , 2008 , “Light Indexed Deferred Lighting ” , Available from *http://lightindexed-deferredrender.googlecode.com/files/LightIndexedDeferredLighting1.1.pdf* [Accessed 1 December 2012]

Valient, M. , 2007. "Deferred Rendering in Killzone 2." Presentation at Develop Conference in Brighton, July 2007. *Available from: www.guerrilla-games.com/publications/dr_kz2_rsx_dev07.pdf [Accessed 30 October 2012]*