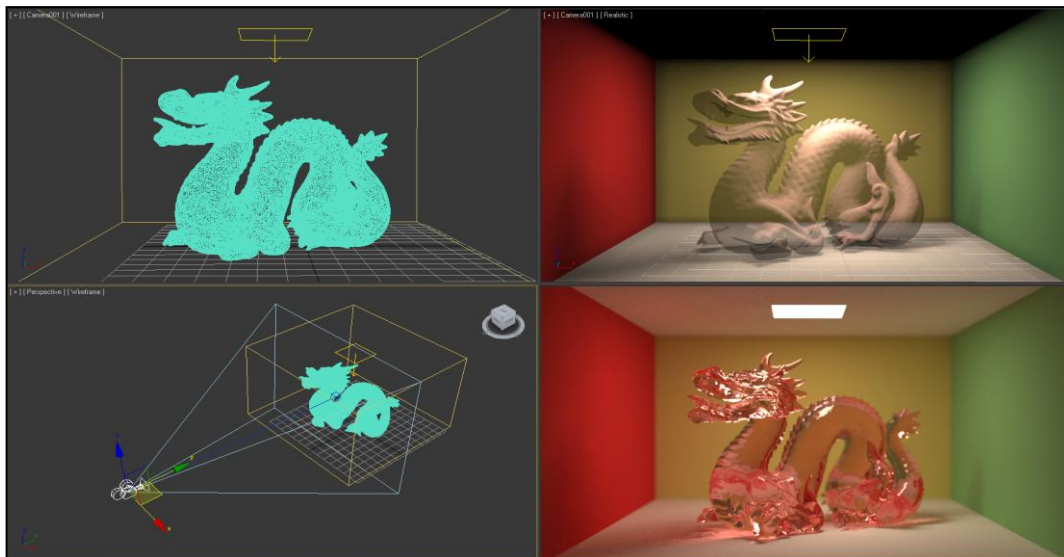


CHALMERS



Implementing a render plugin for 3ds Max using the Autodesk RapidRT path tracer

Master of Science Thesis in the Interaction Design and Technologies Programme

ERIK GUNNARSSON
MAGNUS OLAUSSON

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
Göteborg, Sweden, June 2012

The Author grants to Chalmers University of Technology and University of Gothenburg the non-exclusive right to publish the Work electronically and in a non-commercial purpose make it accessible on the Internet.

The Author warrants that he/she is the author to the Work, and warrants that the Work does not contain text, pictures or other material that violates copyright law.

The Author shall, when transferring the rights of the Work to a third party (for example a publisher or a company), acknowledge the third party about this agreement. If the Author has signed a copyright agreement with a third party regarding the Work, the Author warrants hereby that he/she has obtained any necessary permission from this third party to let Chalmers University of Technology and University of Gothenburg store the Work electronically and make it accessible on the Internet.

Implementing a render plugin for 3ds Max using the Autodesk RapidRT path tracer

ERIK. GUNNARSSON,
MAGNUS. OLAUSSON,

© ERIK. GUNNARSSON, June 2012.
© MAGNUS. OLAUSSON, June 2012.

Examiner: ULF. ASSARSSON

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Cover: The four default viewports of 3ds Max 2012 showing a scene of the Stanford dragon in a Cornell box. Top left viewport shows the camera view in *wireframe* shading mode. Top right shows the camera view in *realistic* shading mode. Lower left shows the scene setup from a perspective view. Lower right shows a camera view in ActiveShade (see Section 2.3.2) mode running RapidRT.

Department of Computer Science and Engineering
Göteborg, Sweden June 2012

Abstract

The aim of this thesis was to research the feasibility of implementing a render plugin for Autodesk 3ds Max 2012 using the ray tracer RapidRT, also developed by Autodesk. The focus was primarily on investigating RapidRT's capabilities as a render plugin, how to translate a 3D scene from 3ds Max to RapidRT, and also the visual quality of the rendered images. Two versions of the plugin were made using different versions of RapidRT. The resulting plugins, although not complete in functionality, shows that using RapidRT as a render plugin in 3ds Max is feasible. The results show that the plugins can be used to render images with visual quality on par with integrated renderers, and that they can translate 3D scenes from 3ds Max to RapidRT and produce images close in appearance to those produced by these renderers.

Sammanfattning

Syftet med denna rapport var att undersöka huruvida en implementering av en renderingsinsticksmodul – som använder sig utav RapidRT, en av Autodesk utvecklad ray tracer – till Autodesk 3ds Max lämpligen kunde genomföras. Fokus var först och främst på att undersöka 3D-scenöversättningen från 3ds Max till RapidRT, hur det senares funktionalitet presenteras i 3ds Max och den visuella kvaliteten på de renderade bilderna. Två versioner av insticksmodulen utvecklades som använder sig av olika versioner av RapidRT. Om än ej kompletta i funktionalitet så visar insticksmodulerna att RapidRT som insticksmodul till 3ds Max är ett genomförbart projekt. Resultatet visar att insticksmodulerna kan användas till att rendera bilder med visuell kvalitet på nivå med integrerade renderare och att de kan översätta scener från 3ds Max så att renderade bilder är utseendemässigt lika i jämförelse med dessa renderare.

Acknowledgements

This master's thesis project was done at the Autodesk Gothenburg office, and we, the authors, would like to thank everyone at the office for making us feel welcome. We would like to thank Christoffer Baar for giving us this opportunity, as well as Magnus Pettersson for helping making it happen to begin with. Thanks also go to Peter Rundberg, Per Svensson and Henrik Edström for their encouragement and help during the project. In particular, thanks goes to our supervisor, Oliver Abert, who gave us feedback and good counsel during the project, as well as for this thesis. Thanks to our examiner at Chalmers, Ulf Assarsson, for feedback on this thesis, and lastly we would like to thank our opponents Johan Elvek and Rickard von Haugwitz for the feedback, and a special thanks to Rickard for additional help with editing of the thesis.

Table of Contents

1. Introduction	1
1.1 Purpose.....	1
1.2 Problem Definition.....	2
1.3 Scope	2
1.4 Method	3
1.4.1 Use Cases	3
1.4.2 Programming Language and Framework.....	3
2. Background and Previous Work	4
2.1 3D Rendering	4
2.1.1 The Rendering Equation	4
2.1.2 BRDF	5
2.1.3 Rasterization.....	5
2.1.4 Ray tracing	5
2.2 RapidRT	8
2.2.1 Shaders	8
2.2.2 Geometry.....	9
2.2.3 Interactivity	10
2.3 3ds Max.....	10
2.3.1 SDK.....	10
2.3.2 Interactive renderer	14
2.3.3 Interface.....	14
2.3.4 Renderers.....	15
3. Analysis.....	16
3.1 Plugin Overview	16
3.2 Translating from 3ds Max to RapidRT	17
3.2.1 Cameras.....	17
3.2.2 Lights	17
3.2.3 Materials.....	19
3.2.4 Maps.....	24
3.2.5 Geometry Conversion	24
3.3 Interactive Rendering	29
3.4 Rendering time	30

3.5 Usability	31
3.5.1 User Interface	31
3.5.2 Interactive Rendering	33
4. Results	34
4.1 Scene translation	35
4.1.1 Geometry, Lights and Camera	35
4.1.2 Materials - Shader conversion.....	36
4.2 The Use Cases	39
4.3 Interactivity	40
4.4 Setup/load.....	40
4.4.1 The Robot Scene	41
4.4.2 The Car Scene	41
5. Discussion	42
6. Conclusion	44
7. Future Work	45
8. References	46
8.1 Bibliography.....	46
8.2 Image Credits	49
Appendix A	50
Appendix B	52

1. Introduction

It is common in the computer software industry to wish to integrate new technologies in existing products through extensions or plugins. These plugins can serve to increase the functionality of the product and provide more options for the user. This makes it interesting to study the opportunities and pitfalls that come with developing a plugin and how to utilize the new technologies' capabilities in enhancing an already existing product. In this case the product is Autodesk 3ds Max and the plugin is a renderer, RapidRT, which is a ray-tracing technology developed by Autodesk in Gothenburg.

RapidRT is a renderer which has seen recent changes in core functionality. There are some fundamental differences between the old RapidRT (version 3.0) and the new RapidRT (version 4.0 pre-release). Throughout this thesis, we will refer to both versions, since the plugin was developed for the old RapidRT to start with and later migrated to the new version. It should be noted that the new version was in development and evolved concurrently with the work done for this report. All RapidRT images in this report are rendered using the new version of the plugin, if not otherwise noted.

Autodesk 3ds Max¹ is a 3D modeling software with uses varying from making film and video game models, to architectural design. There are a few renderers that come packaged with 3ds Max, for example NVIDIA mental ray and NVIDIA iray. There are external renderers available as well, which hook into 3ds Max through the 3ds Max *Software Development Kit* (SDK). The SDK supports a range of different plugin types apart from renderers.

1.1 Purpose

The purpose of the project was to show that RapidRT is a suitable solution for rendering in 3ds Max. The focus was initially on determining if such a solution is feasible to implement and what challenges it presents. The project's focus later went beyond this to explore how to achieve good visual quality in the translation from 3ds Max to RapidRT, which became the main focus of the study.

¹ <http://usa.autodesk.com/3ds-max/>

1.2 Problem Definition

This thesis investigates how to integrate an external renderer into 3ds Max. Mostly, this will be approached from a technical point of view, but the usability of the plugin will also be touched upon. This involves looking into the challenges of translating the scene geometry and material properties of 3ds Max, to a representation that makes sense in RapidRT. The goal is to achieve good visual quality and an efficient and correct translation.

In order to gauge how successful the plugin is in what it tries to achieve, it will be compared to plugins for other renderers available for 3ds Max. To some extent, the plugin can be considered successful if it can highlight the positives of using RapidRT in conjunction with 3ds Max, and produce images that come close in appearance to those produced by integrated renderers.

1.3 Scope

Writing a render plugin for 3ds Max is quite an extensive task if completeness is strived for. Select features of 3ds Max and RapidRT were prioritized according to the goals and questions asked in the problem definition. This made it a logical choice to prioritize features that were believed to yield the greatest visual impact and not on translating very specific features that only may be useful for working with 3ds Max on a professional level and to a very specific end.

Also deemed to be outside the scope of this work is animation and particle effects; the focus is limited to translating a static scene. An additional example of this is that hair and fur have not been investigated.

Although there are many different 3D modeling software with rendering solutions on the market currently, for the context of this report, only renderers integrated in 3ds Max will be used for comparisons in terms of features and performance. It may well be useful to look at renderers for other software than 3ds Max; however, these are left out here to narrow down the focus of the study in order to keep comparisons as relevant as possible; this study is focusing on the 3ds Max software in conjunction with RapidRT.

1.4 Method

The scope and problem definition of the project very much influenced the methods used to develop the RapidRT plugin for 3ds Max. The approach taken was to work iteratively, deciding as the project work progressed what was to be done next, apart from having some basic milestones from the beginning.

At the early stages, time was spent on researching the 3ds Max SDK and studying the RapidRT *Application Programming Interface* (API). Several prototypes were made which explored different aspects of the work to be done and helped to clarify the vision of what lay ahead.

1.4.1 Use Cases

A couple of use cases were used to gauge how well the plugin performed. These were 3ds Max scenes that were used during the development of the plugin and used to see what needed to be developed next, in order to produce a good image of the scene. See Chapter 4 for images of the scenes used.

1.4.2 Programming Language and Framework

The plugin was written in C++ using the Visual Studio *Integrated Development Environment* (IDE), utilizing the RapidRT API, including RapidSL, and the 3ds Max SDK. There was not much choice involved in the aforementioned technologies since they are central to the project at large. The choice of IDE was made from a practical standpoint, since it was straight-forward to integrate the 3ds Max SDK and to link the debugging capabilities of Visual Studio to 3ds Max. To our aid we had the documentation of the old version of RapidRT and the 3ds Max SDK Programmer's Guide and Reference. Furthermore, we had the possibility to consult with the developers of RapidRT.

Several different versions of 3ds Max were used during the course of the project. The plugins developed are compatible only with 3ds Max 2012. Other versions were used to test and try out the already existing renderers. For example, 3ds Max 2013 was used to be able to compare the plugins with NVIDIA Iray renderer under fairer conditions, since the 2013 version comes with a later version of Iray.

2. Background and Previous Work

In this chapter, the software and concepts involved in the project are presented in more detail. This includes an overview of computer graphics and rendering, and details regarding 3ds Max and RapidRT functionality.

2.1 3D Rendering

Generating images from 3D models is called 3D rendering. From mathematical descriptions of objects and lights on the computer, a 3D renderer computes visibility and lighting conditions and produces an image based on this data. From a 3D modeling standpoint there are a few famous examples of programs that do this: Autodesk 3ds Max, Blender², and LightWave 3D³ are some of these. They all let the user define a 3D scene and render images using different supplied renderers. Regarding for example games, 3D models are created using modeling software such as those just mentioned, and are then used and rendered in real-time game engines.

2.1.1 The Rendering Equation

The rendering equation aims to describe how much light is transferred (radiance) from one point to another. There are different variants of the rendering equation (Kajiya, 1986; Akenine-Möller *et al.*, 2008, p.327) and they all describe radiance in different ways. More precisely, the radiance is the emitted light from the surface point plus the light reflected along the outgoing direction towards another point. For most materials the reflected radiance depends on the incoming radiance from all direction in the hemisphere above the surface. For a perfect mirror, the reflected radiance is equal to the incoming radiance in the reflection direction. This means that solving the equation fully requires an infinite amount of recursions. Because of this, 3D rendering techniques try to give an estimate of this equation. By solving an approximation of the equation between the position of the eye and points in a 3D scene, it is possible to render realistic 3D images. Below is the rendering equation given by Kajiya (1986).

² www.blender.org

³ www.newtek.com/lightwave/

$$I(x, x') = g(x, x') \left[e(x, x') + \int_S p(x, x', x'') I(x', x'') dx'' \right],$$

where $I(x, x')$ is the intensity of light from x' to x , $g(x, x')$ is a geometrical term used to test the visibility of the surface point, $e(x, x')$ is the emittance and $p(x, x', x'')$ is the intensity of scattered light from x'' to x through x' , and S is the set of all surface points.

2.1.2 BRDF

Given the incoming light direction and the outgoing view direction, the *Bidirectional Reflectance Distribution Function* (BRDF) tries to describe the interaction of light at a surface point (Akenine-Möller *et al.*, 2008, p. 223). There are many different BRDF solutions proposed, many of them useful in different situations. Lambertian reflection for example describes the reflectance of diffuse surfaces (Akenine-Möller *et al.*, 2008, pp. 227-228). Other BRDFs describe more involved lighting situations with more complex material properties, for example the Cook-Torrance model (Cook and Torrance, 1982).

2.1.3 Rasterization

There are a few different approaches to 3D rendering. The fastest one, very much due to the specialized *Graphics Processing Units* (GPU) available, is rasterization. Rasterization is the process of filling in the pixels that a triangle in a 3D scene overlaps. The triangle is divided into fragments, and each fragment is shaded and tested for visibility, and the resulting color is the one used for the pixel (Gregory, 2009, p.441). This technique is the most common when it comes to real-time applications such as games, due to its speed.

Scanline rendering is a rasterization technique that works per pixel instead of per polygon, working its way horizontally line by line of pixels (Birn, 2005).

2.1.4 Ray tracing

A different approach to 3D rendering is ray tracing, where the aim is to simulate light by tracing rays through the three-dimensional scene. In ray tracing, rays are traced from the eye per pixel, into the scene (Whitted, 1980; Kajiya, 1986; Akenine-Möller *et al.*, 2008, p 412); these are usually called primary rays. This is the opposite of what happens in nature where the rays (photons) originate from the sun. When a ray hits a surface, a number of different new rays are spawned, depending on which algorithm is used. Typically, one *shadow ray* is spawned per light in the scene. The shadow rays test whether the given point is in shadow

or not. If the material is reflective, a ray is also shot in the reflection direction. If the material is transparent, a ray is shot in the refraction direction. These rays are then followed in the same way as the primary rays.

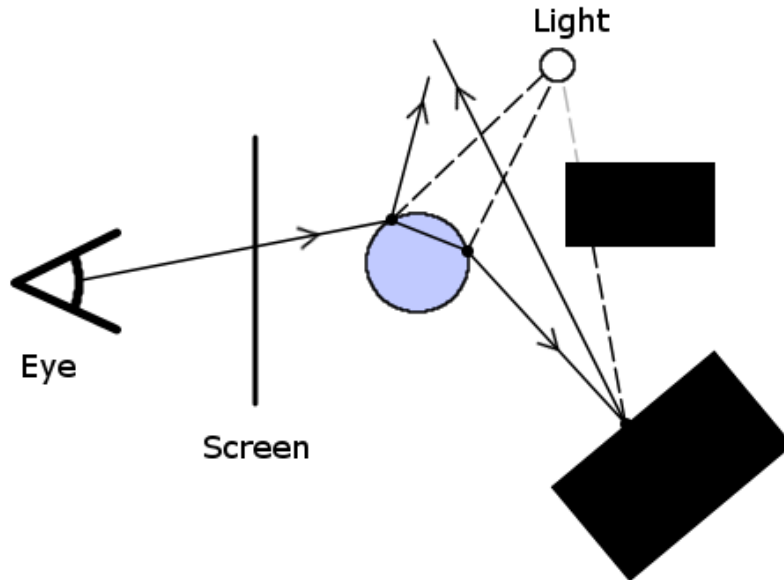


Figure 1 – Illustration of ray tracing. The blue, larger circle is a transparent sphere and the black rectangles are opaque objects. Dashed lines represent shadow rays.

To completely solve the rendering equation, it is not enough to shoot reflection, refraction, and shadow rays. The outgoing radiance depends on the integral of incoming radiance from the entire hemisphere. A method that improves on these early ray tracing algorithms is Monte Carlo ray tracing. Two different types of Monte Carlo ray tracing are covered in Section 2.1.4.1 and Section 2.1.4.2.

One problem with ray tracing is how to decide when the recursion of spawning new rays should stop. There are a few different ways of doing this. One alternative is to decide on a fixed number for the maximum number of recursions. Another alternative uses the fact that for each recursion the contribution to the final image is lowered, unless the surface is a perfect mirror. One could decide on a threshold, and when the final contribution falls below this threshold the recursion is terminated.

However, both these alternatives introduce bias (Arvo and Kirk, 1990). To avoid bias a method called Russian roulette can be used. This method works by randomly terminating the recursion according to some probability P . To avoid bias, the weight of the rays that are chosen not to be terminated in a step of the algorithm can be increased by $1/(1 - P)$, as suggested by Arvo and Kirk (1990).

Rendering a scene with ray tracing requires a fast way to find closest intersection points between rays and the geometry of the 3D scene. To increase the speed of this operation it is beneficial to store the scene geometry in a spatial acceleration data structure. Otherwise, each ray has to be tested for intersection against every triangle in the scene, which yields a complexity of $O(n)$. With a tree-like data structure the complexity drops to $O(\log n)$. However, this means that movement or deformation within the scene for interactive rendering requires this data structure to be rebuilt, which can be slow (Wald *et al.*, 2007). Attempts have been made to do this in real time both on the *Central Processing Unit* (CPU) (Wald *et al.*, 2006) and on the GPU (Roger *et al.*, 2007).

2.1.4.1 Distributed ray tracing

Distributed ray tracing is a type of Monte Carlo ray tracing (Akenine-Möller *et al.*, 2008, pp. 414-415). It captures effect such as glossy reflections, penumbras, motion blur, and depth of field. It works by shooting more rays in many different directions at each surface interaction/bounce (Cook *et al.*, 1984). The rays are shot according to some given distribution and the distribution depends on the material (BRDF). Unless depth of field and motion blur are sought after, usually only a few rays are shot per pixel.

2.1.4.2 Path tracing

Path tracing was introduced in Kajiya, 1986. In path tracing, only two rays are shot per surface interaction; one reflection ray and one shadow ray to sample a light source. The reflection ray is, just as in distributed ray tracing, shot in a random direction given some distribution decided by the material's BRDF. This essentially creates a path through the scene for each primary ray.

If a material is half reflective and half diffuse, there could be a large variation in the resulting color of a path hitting the object. Because of this, more primary rays need to be generated compared to distributed ray tracing, in order to reduce noise. The benefit of this method is that the number of higher generation rays is reduced, as these rays contribute less to the final image. For a thorough guide on implementing a physically based renderer, such as a path tracer, see Pharr and Humphreys, 2010.

2.2 RapidRT

RapidRT is a CPU-driven ray tracing technology developed by Autodesk. It has been used in a few different Autodesk products including Showcase and Opticore Studio. The old version of RapidRT was based on a modified version of distributed ray tracing, but has recently taken a direction towards path tracing and also an emphasis on physical correctness. When using RapidRT, a user gets hold of features such as global illumination, caustics, depth of field, and other features that would be expected from a rendering solution. It also supports progressive rendering, which means that RapidRT will output images while continuously rendering. In this way, a useful but noisy image will be output quickly and then refined over time. This, together with cluster support, makes it viable to do interactive rendering using RapidRT. Rendering using a cluster means that the work on an image is spread out on several work stations, speeding up rendering.

To connect to and render images with RapidRT, its API is used. Through the API, a scene, cameras, different options for rendering, and framebuffers can be set up. These are combined into a so called frame definition. When everything is setup correctly, a frame call is issued and RapidRT starts to render the given frame definition. Multiple frame calls can be issued before any frame is complete if double buffering is used, which means that RapidRT can continue to render the next frame immediately when the previous frame is completed. When a frame is complete, it can be retrieved via a call to a synchronization function. If a frame is complete, a callback given to the framebuffer will be called. If no frame is complete, the synchronization function returns immediately.

The scene consists of a tree structure of entities called the scene graph. An entity can be a group, object, camera or light. A group is simply a collection of more entities. An entity has a transform that moves, scales, and rotates the entity and all its children. To define a scene, you supply a group entity to act as the root of the scene graph.

2.2.1 Shaders

Shaders for RapidRT are written in RapidSL, which is a C-style shading language. Shaders are compiled into separate *Dynamic-Link Library* (DLL) files and are either material, light, or environment shaders.

For material shaders, there are four functions that can be implemented, each used by RapidRT in different stages of the path tracing algorithm. If any of them are left out, a default behavior is used. The main function is called **sample**, which is called when a new direction is needed after a

surface has been hit. It should return a new direction for the path, the type of interaction that occurred (i.e. diffuse reflection, specular reflection or refraction), the probability for the event, and how much light that is transferred from the returned direction to the direction given as input to the function. By generating a pseudorandom number between zero and one (a sample) and then, with the information about the probability of reflection, refraction and the value of the sample, it can be determined how the interaction at the surface point should be handled.

The second function is **transfer**, which is called when a shadow ray is shot and a light is sampled. As input, the shader is given both an in and out direction. It should return how much light is transferred from the in direction to the out direction, as well as the probability that the given out direction was generated.

The third function is **transmission**. It is called when a shadow ray hits an object between the light and point being shaded. The function should return the amount of light that is transmitted through the object. If the object is completely opaque it should return zero. However, because of varying index of refraction of objects, it is not physically correct to treat shadow rays in this manner. Therefore, it is possible to turn transmission off completely with a setting in RapidRT.

The fourth function is called **emission**. It is used by emissive objects. The only value returned by this function is the amount of emitted light.

In the old version there was also a **shade** function. It performed the full shading function for a hit and was responsible for shooting any new rays necessary to do the shading. The other functions were called as a result of what the shade function did. All lights in the scene could be iterated in the shade function, and when this was done, the lights were sampled with shadow rays, which resulted in transmission being called on any objects in the path of the shadow ray for each light. The shade function was very much in control over how the rendering was done.

2.2.2 Geometry

RapidRT accepts geometry in a format similar to the one in OpenGL, described in the OpenGL specification (Segal and Akeley, 2012). RapidRT has one array of indices where three indices are paired together to form triangles. These indices points into arrays of vertex attributes, such as position, normal and texture coordinate.

2.2.3 Interactivity

In general, modifying the scene interactively using ray tracing is expensive due to rebuilding of the speed-up structure if objects in the scene change (see Section 2.1.4). Properties of the old version of RapidRT make it faster to redraw the scene if objects that are commonly moved in the scene are tagged as dynamic. In the new version, all objects are tagged as being dynamic.

2.3 3ds Max

Autodesk 3ds Max is a 3d modeling, animation, and rendering tool used in several different industries such as game development, film, and architectural design.

2.3.1 SDK

In order to extend 3ds Max and provide, for example, more alternative renderers, the 3ds Max SDK is used. The SDK grants access to the necessary data needed for rendering a scene within 3ds Max. Through the SDK it is possible to iterate over the scene graph and get access to the nodes of the structure. Each node in the scene graph can have pointers to objects that are in the scene, and holds a transform describing the position, rotation, and scale of the object. The node also keeps track of the number of child nodes. The object connected to the node can be for example a geometry, light, or camera object. For geometrical objects, the mesh data is accessible, as well as the assigned material. In the case of light objects, depending on the type of light, there are different values describing it, for example color, intensity, and direction. The camera object gives access to for example *Field of View* (FOV) and whether the camera is orthographic or not.



Figure 2 – A screenshot from 3ds Max 2012, showing parts of a scene graph where a sphere and a box is grouped together, and lit by a directional light with a target.

With the 3ds Max SDK it is possible to write different types of plugins. Among these are Utility, Material, Shader, File Export and Renderer plugins. For many of these plugins, a 3ds Max plugin wizard supplies a shell to start from, including a few functions that should be implemented in order to have the plugin work as intended. For a render plugin, this includes functions that will be notified by 3ds Max when the scene should be rendered and how. The three main functions are Open, Render

and Close. In the Open function the initialization of rendering is supposed to be done. The actual rendering is done in the Render function and clean up is done in the Close function.

2.3.1.1 Parameter Blocks

Values, or parameters, used by a plugin in 3ds Max can be stored in a parameter block. Parameter blocks help with the setup of the user interface, with animation of parameters, with saving and loading of plugins, and to provide access to the parameters from other parts of 3ds Max than the plugin itself (Autodesk, 2011f). In short, it is a way to for 3ds Max to keep track of the plugin and its parameters.

2.3.1.2 Geometry

There are different types of geometrical objects within 3ds Max, such as triangular meshes, polygon meshes, patch meshes, splines, and NURBS (Autodesk, 2011a). All of them can be converted to a TriObject, the triangular mesh representation within 3ds Max.

The base for accessing the geometrical data from a TriObject is a class called Mesh. The format is made for easy editing and is a variant of face-vertex mesh (Autodesk, 2011b). However, unlike models supplied to an OpenGL-like API, 3ds Max does not store normals for each vertex. Instead, normals can be calculated based on the faces.

The Mesh class contains a list of faces, vertices, texture faces, and texture coordinates (see Figure 3). It also contains a few other things that are not covered here. One face contains three indices into the list of vertices, and those three vertices forms a triangle. The face has a material identifier and also keeps information about which smoothing groups it belongs to. The material identifier determines which material should be applied to the face if the object is assigned multiple materials through the Multi/Sub-Object material (Autodesk, 2011x) of 3ds Max.

There are 32 different smoothing groups in total and a face can belong to any number of these. If two adjacent faces share any smoothing group, the normals for the shared vertices should be smoothed, interpolated, between the faces' normals.

For each face, there is one texture face. The texture face contains three indices into the list of texture coordinates, which defines how textures should be mapped to the surface.

Face	Texture-face	Vertex
0: 0,4,5	0: 0,1,2	0: 0,0,0
1: 0,5,1	1: 0,2,3	1: 1,0,0
2: 1,5,6	2: 0,1,2	2: 1,1,0
3: 1,6,2	3: 0,2,3	3: 0,1,0
4: 2,6,7	4: 0,1,2	4: 0,0,1
5: 2,7,3	5: 0,2,3	5: 1,0,1
6: 3,7,4	6: 0,1,2	6: 1,1,1
7: 3,4,0	7: 0,2,3	7: 0,1,1
8: 0,1,2	8: 0,1,2	uv-coordinate
9: 0,2,3	9: 0,2,3	0: 0,0
10: 4,7,6	10: 0,1,2	1: 1,0
11: 4,6,5	11: 0,2,3	2: 1,1
		3: 0,1

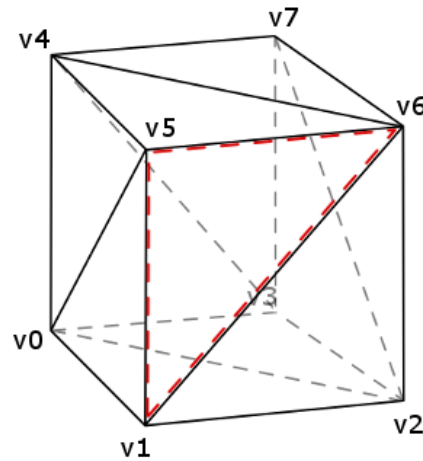


Figure 3 - Simplified illustration of the structure of data kept in the Mesh class. A list of faces points into the list of vertices, and the list of texture-faces points into a list of uv-coordinates. Face 2 is highlighted and shown in the cube. Vertex 1 has uv-coordinate (0,0), vertex 5 has (1,0), and vertex 6 has (1,1).

2.3.1.3 Cameras

When you render a scene in 3ds Max, it is possible to either use a camera object or simply render using the view in one of the viewports, which both can use either perspective or orthogonal projection. The render plugin is told which alternative is used through the parameters of the Open function. Either a scene node with an attached object of the class CameraObject or a ViewParam object is provided. They both contain information about how projection should be done, indicating whether the projection should be orthographic or perspective. It is also possible to access the field-of-view angle, near and far clip plane distances, and target distance for depth of field. The ViewParams also contains a transform matrix from world space to view space. For the CameraObject class, the transform for the node to which it is attached is used instead, whose inverse becomes the view transform matrix. In addition, the ViewParams contains a zoom parameter. It controls the size of the viewing plane when using orthographic projection.

2.3.1.4 Lights

3ds Max has a number of different lights by default and it is possible for developers to add their own types of lights through plugins (Autodesk, 2011c). Through the 3ds Max interface, two different categories of lights are exposed. These are photometric and standard lights. Any of these lights belong to one of the following classes of lights: ambient, directional, spot, and point lights. For each light, there are different parameters such as color and intensity. For spot lights, there are also the falloff and hot-spot angles.

2.3.1.5 Materials

3ds Max uses materials to decide how objects should be shaded. Basically, a material defines how the light should interact at a surface point with that material. A material in 3ds Max defines parameters for how the interaction should behave. For example, the Arch&Design material, which is a material in the mental ray material library, defines parameters for reflections and refractions – be it glossy or not – as well as options for the BRDF, self illumination, and other properties. The Arch&Design material can, with different values for these parameters, emulate material properties for glass, concrete, copper, chrome, and others. By default, 3ds Max ships with a multitude of different materials, from very specific to all-purpose materials. It is also possible to write custom materials with the plugin system.

Materials are edited in the material editor, where it is also possible to combine materials (see Figure 4). There are a number of special materials, called compound materials, which take the output from other materials as input and uses them in some way to produce a new output. For example the Blend material that takes two materials and blends them together by some given ratio. It also optionally accepts a third input in the form of a texture, which is used as a mask deciding how the materials should be blended together.

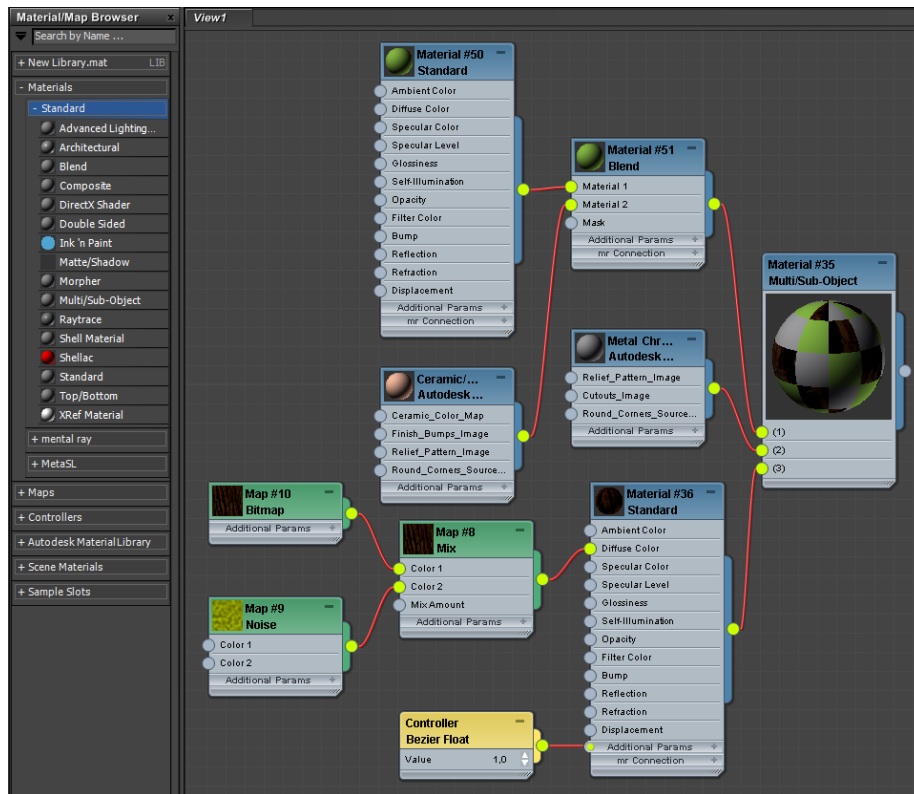


Figure 4 – A screenshot of the 3ds Max 2012 material editor. This figure shows how materials and maps can be combined in the material editor.

Textures are also assigned using the material editor. Textures are called Texmaps in 3ds Max and include both procedural textures and bitmap textures. In 3ds Max, bitmap textures are a subtype of Texmap, which is simply called Bitmap. Texmaps can be combined in ways similar to materials. There are Texmaps that take multiple Texmaps as input to produce a different output. One of them is called Mix. It takes two Texmaps and mixes them together according to some given function.

2.3.2 Interactive renderer

A feature supported in 3ds Max that involves the concept of interactive rendering is ActiveShade. Through the render setup menu, a renderer for ActiveShade can be selected and used to render the scene either in a viewport or in a separate window. The idea with this is to be able to manipulate the scene and have the ActiveShade renderer render the scene to reflect the changes on the fly. Manipulation cannot be done in the ActiveShade window however. For example, to move the camera, it has to be moved using a tool such as the *Select and Move* tool in another viewport.

2.3.3 Interface

The interface of 3ds Max looks basically the same from one yearly iteration to another in recent years, with often only small changes to the interface design. 3ds Max is a program that is usually used maximized, taking up the whole screen (see Figure 5). As default, the main view is split into four viewports and provides toolbars and menus with a multitude of tools for the users, both for the regular *intermediate users* and the *expert users* (see Cooper *et al.*, 2007, pp. 47-48). Specifically for rendering, there is a render setup menu from which the renderer to use can be specified and tweaked using common render parameters as well as render-specific options (see Figure 6).

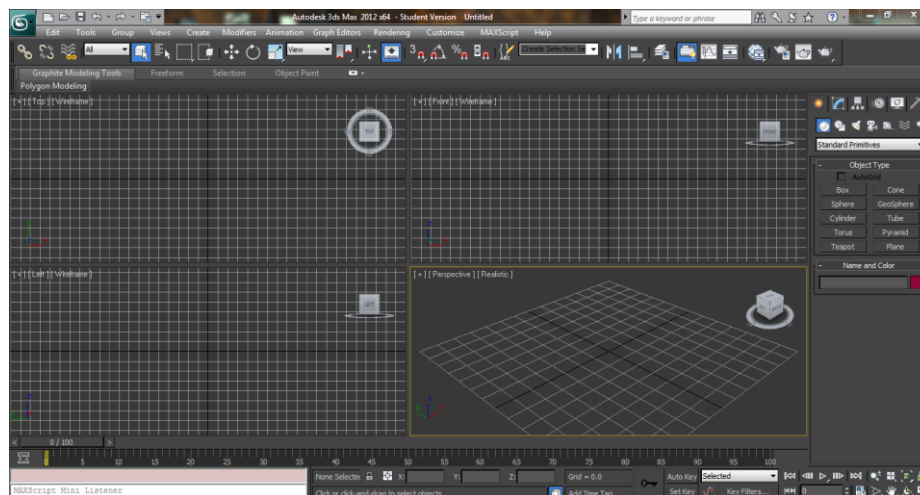


Figure 5 – A screenshot of the Autodesk 3ds Max 2012 interface.

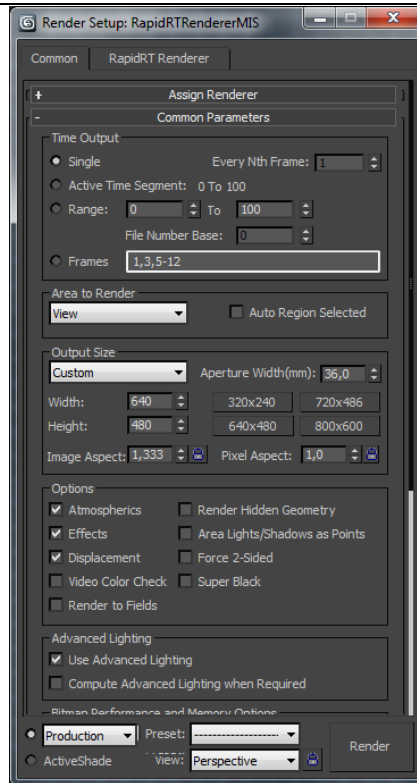


Figure 6 – Screenshot of the Render Setup menu in 3ds Max 2012.

2.3.4 Renderers

When it comes to rendering, 3ds Max 2012 ships with a default scanline renderer, NVIDIA mental ray, NVIDIA iray, among a few others. The first is a renderer using mainly the scanline rasterization technique, while the two latter renderers both utilize ray tracing techniques. Images can be rendered either through a production render (offline render) or through ActiveShade (interactive render).

2.3.4.1 NVIDIA mental ray

Developed by mental images and now owned by NVIDIA is the mental ray renderer. It uses several different algorithms to speed up rendering in different situations: scanline rendering, a rasterizer with support for motion blur among other things, and ray tracing for effects such as reflections and refractions (NVIDIA, n.d.1a).

2.3.4.2 NVIDIA iray

The NVIDIA iray renderer is a physically based renderer which is specifically developed to take advantage of GPUs. iray uses progressive rendering in order to refine the image in iterations (NVIDIA, n.d.2a). The number of iterations can be specified, set to unlimited, or the user can specify the time to spend on rendering the image. The fact that iray, like RapidRT, is physically based and progressive makes it a good basis for comparison for the context of this work. It also supports the ActiveShade feature of 3ds Max, enabling interactive rendering of the scene.

3. Analysis

During the course of the project, two different versions of RapidRT have been used. For convenience we have chosen to call them the old and the new version (see Chapter 1), which uses version 3.0 and 4.0 (pre-release), respectively. The plugin migrated from using the old version to the new, and so the plugin has seen two versions. The two versions of the plugin differ not only due to being implemented in succession, but due to differences between the two versions of RapidRT. The old-and-new notation will also be used when talking about the plugin.

3.1 Plugin Overview

The plugin implements the three functions required by a render plugin for 3ds Max:

- In *Open*, a RapidRT device is created and the 3ds Max scene graph is iterated. While iterating the scene graph, all the objects in the scene are converted and copied to RapidRT.
- When *Render* is called, the render loop starts. Here, RapidRT is asked for new frames and the plugin receives new frames, which are displayed in the render window of 3ds Max.
- In *Close*, the device is released and the renderer is shut down.

The plugin also supports ActiveShade, which is implemented via an interface called InteractiveRender. Here follow some of the functions that need to be implemented:

- *BeginSession* is similar to *Open* and uses the same functions for setting up the scene. When the device is created and the scene is set up, *BeginSession* starts the render loop using a timer (more on this in Section 3.3).
- If during rendering, 3ds Max receives instructions to shut down rendering, the *IsRendering* function is called. It returns false (even though we are always rendering), telling 3ds Max that it is OK to shut down.
- In *EndSession*, which is called when the renderer is to be shut down, the timer is killed and the renderer destroyed.

3.2 Translating from 3ds Max to RapidRT

The most fundamental aspect of the task of implementing a rendering plugin, with the exception of actually writing a renderer, is the translation from the 3ds Max scene representation to that of the external renderer. Hence, the majority of work was to correctly convert scenes from 3ds Max to RapidRT. This includes translating cameras, geometry, lights, and materials, and making them appear as expected.

3.2.1 Cameras

Converting the camera from 3ds Max to RapidRT is mostly a matter of reading the values for field of view and transform from 3ds Max, and setting the values for the RapidRT camera. However, as with many aspects of the translation, there are parameters that do not match correctly and which demands some decision making. As an example, the clip values supplied by 3ds Max are ignored. It was simply not deemed necessary to support. Another example is that RapidRT has a projection matrix for the camera and a transformation matrix for the node to which it is attached. The 3ds Max SDK, however, supplies a view transform, so the transformation matrix needed for the RapidRT camera is calculated as the inverse of the view transformation matrix.

The orthographic camera is supported in the plugin. However, when it comes to rendering not from a camera but from a viewport in 3ds Max, the orthographic view is problematic. This is because the parameters that are made available by 3ds Max are different when no camera is involved in the rendering (see Section 2.3.1.3). The lacking information from 3ds Max in this case, led to viewport orthographic rendering not being prioritized. Also, it was not a feature that was strictly needed since the camera-orthographic rendering was already supported by the plugin.

3.2.2 Lights

Regarding the lighting of the scene, the old and new version of the plugin supports different kinds of lights; from the old to new, some features were lost and some gained.

3.2.2.1 Standard lights

The standard lights in 3ds Max are not physically based. For example, no lights in real life are either infinitely small like point lights (for example Omni lights in 3ds Max), or infinitely far away like directional lights. The old version of RapidRT was able to represent the standard lights by default, through a type of entity. Hence, the plugin supports standard lights, and uses light shaders to control how the light behaves in terms of direction and attenuation. When the scene graph is iterated

and a light object is found, the type of the light is evaluated and handled accordingly: creating a light in RapidRT and connecting it to an appropriate shader.

In the new version of RapidRT (at least the version used in our implementation), there is no standard way of simulating these types of lights, since the light-entities are no longer supported. However, some of the standard lights should be fairly straight-forward to implement an approximation of with the new RapidRT. One example is that the new version supports emissive geometry, so a point light could be simulated by making a very small sphere emit light. Directional lights could perhaps be baked into the environment (see Environment section below).

3.2.2.2 Area Lights

Area lights are lights that have an area and so can cast shadows that can have a penumbra (see Figure 7). In the new version of the plugin, geometry can have an emissive material applied to them. This means that any object in the scene can be turned into a light source. In the photometric lights rollout in the 3ds Max interface, mr Sky Portal (mr is short for mental ray) can be found. mr Sky Portal is treated as a light by 3ds Max and so it is discovered when the scene graph is iterated in the plugin and created as an emissive plane in RapidRT. Apart from mr Sky Portal, lights can be created by tagging geometrical objects as lights in RapidRT, and so they will be sampled as such. Via plugin functionality, the geometry is tagged as emissive by clicking the self-illumination checkbox in the 3ds Max Arch&Design material. Since 3ds Max does not recognize the object as a light, if such a geometric object is found when iterating the scene graph, it is considered and treated as a special kind of light and is set up using the luminance and filter color as intensity and color respectively, in a trivial shader. This offers a translation that is not identical but still usable. The color-temperature parameter of the material is not involved in the calculations done by the plugin, so the lighting will differ.

A problem with the emissive geometry is to adjust the intensity. The amount of light reaching a certain point depends on how large area the emissive geometry covers of the point's hemisphere (see Section 2.1.1). Hence, both the distance between the light and the point, and the size of the light, plays a part in the intensity calculation. The setting in 3ds Max for light intensity for geometry adjusts how much luminance the light emits in total, with no regard to its size. To counter this in the plugin the light intensity in the shader is scaled with the surface area. RapidRT automatically calculates the surface area and supplies this value as an input to the shader.

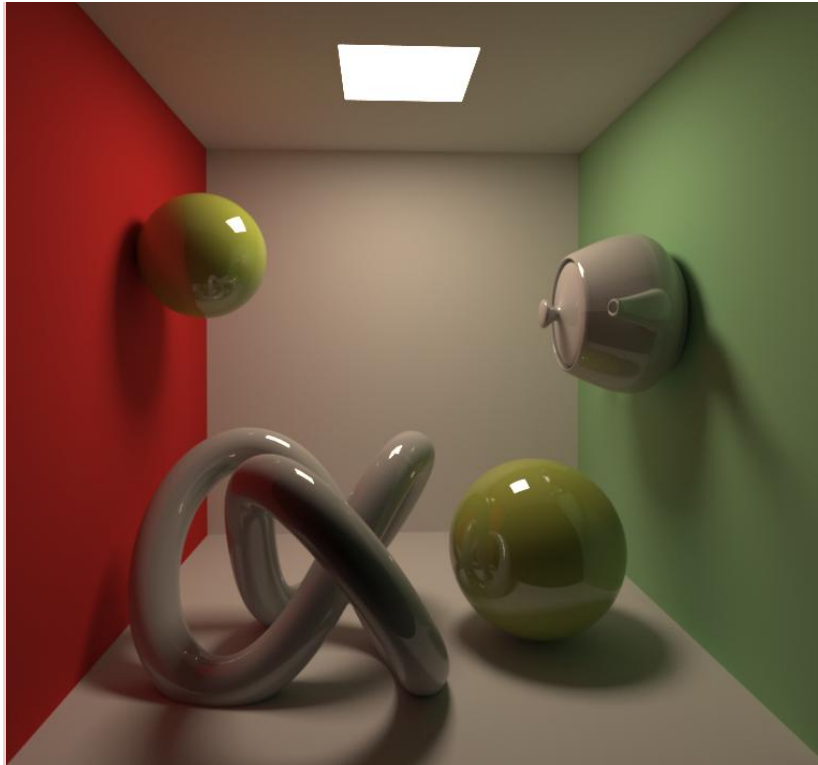


Figure 7 – A screenshot of a 3D scene rendered using RapidRT in 3ds Max 2012. An area light source illuminates the scene and the shadows show a soft transition from a completely shadowed surface to a completely lit one.

3.2.2.3 Environment

Another way to light the scene, which is implemented in the new version of the plugin, is by using an *High Dynamic Range* (HDR) image as the background image (environment). When a ray hits the environment, the intensity of the background image's pixel will help determine the lighting conditions in the scene. The light sources in the background image can be manipulated, using image editing programs, to be much brighter than the other pixels, causing these light sources to cast shadows in the scene. For a result that actually casts shadows, it can be useful to note that linear values of the pixels need to be extracted from 3ds Max when reading the texture to set it up in RapidRT. Failure to do so will result in the color values being remapped to the zero-one range and the high intensity set for the light sources will be nullified.

3.2.3 Materials

3ds Max has a material system where materials can be combined and linked together in many different fashions. For RapidRT, which is using pre-compiled shaders, a shader has to be made for each material that is to be supported. As the material is loaded in the plugin, the parameters of that material are supplied to the corresponding shader. Since 3ds Max has hundreds of unique materials, including the Autodesk Material Library, it was not feasible to implement all of them. One alternative was

to implement only the standard materials, which are about 16 in number, and includes the Standard material. This sounds reasonable to do, however; many of them are materials rarely used or at least not essential for achieving good visual quality in many cases. The final alternative was to implement only select materials; materials that was deemed to be useful for the evolution of the plugin. Since the aim never was completeness in all areas, but rather visual quality, the latter alternative was deemed to be the most useful approach.

One of the standard materials is the Raytrace material (Autodesk, 2011y). Since RapidRT uses ray tracing this material is a good candidate for implementation. However, the Standard material was chosen instead. The primary reason for this was that scenes used for testing used the Standard material. In fact, during the project, no scenes that were used utilized the raytrace material; this vindicated the decision to focus on the standard material.

3.2.3.1 The Standard Material

The first material that was implemented was 3ds Max's Standard material. Information about the Standard material can be extracted from 3ds Max in a straight-forward way; its parameters can be accessed through well-documented functions. It is possible to model many different effects using only this material. However, there is no regard to physical correctness and energy conservation in the Standard material. Parameters of the Standard material include diffuse, ambient, and specular color as well as opacity, index of refraction, specular glossiness, and bump map (Autodesk, 2011z). This raises the question of how to weigh together all the different parameters in the RapidRT shader.

In the old version of RapidRT, it was possible to do shading much like an ordinary GPU shader. The shader was responsible for outputting a final color from the shade function, which means that it was possible to simulate effects such as specular highlights. The first version of the plugin supported specular color, along with diffuse color or map, reflections, bump maps, and refractions. In the Standard material, the type of shading to use can be chosen. This has no impact in the plugin implementation though, since it uses Blinn-Phong shading regardless.

Later when the plugin moved on to using the new version of RapidRT, the Standard material translation had to be changed. Support for the index-of-refraction parameter was added. However, it no longer supports specular color. Also, a paradigm shift in terms of shader code happened between the two version of RapidRT, which highlighted the question of how to read the different parameters and weigh them together. If the

material is 80% reflective and 60% opaque, there is a question of how to decide which distribution of rays to shoot from the surface point. In this new implementation, the percentages of reflectiveness and transparency are checked to see if they exceed 100%. If they do not, they are left unmodified. If they do exceed 100%, they are weighted against each other. In the example above, the reflection probability would be 67% ($80 / (80 + (100 - 60)) \approx 0.67$) and the remaining percentages of the rays are refractive.

3.2.3.2 The Arch&Design material

For the new version of the plugin, there was a request from Autodesk that we would have a look at the Arch&Design material. Apparent was that this was a material often used in practice and was more physically correct than the Standard material, having constraints for energy conservation. The parameters of the Arch&Design material are not as easy to access as the ones for the Standard material. In fact, it is very hard to find anything about it in the 3ds Max SDK Programmer's Guide and Reference. A reason for this might be that it is actually a mental ray material and, as default in the material editor, is hidden for the default scanline renderer, as well as for the RapidRT plugin. The material's parameter block parameters (see Section 2.3.1.1) had to be accessed and written out, so that the necessary information needed to find and extract the parameters in the setup phase (the *Open* function), was revealed.

The translation of the Standard material already implemented for the new version of RapidRT in the plugin was reworked into a translation of the Arch&Design material. The solution supports diffuse color/map, bump/normal map, reflection (Fresnel or user specified curve), transparency and index of refraction.

The shaders of the new version of RapidRT use probabilities to decide the shading. In the sample function of the shader (see Section 2.2.1), a sample (a pseudo-random value between zero and one) is generated. If the sample is below the refraction probability, the sample is considered to be refractive and will influence the final color as such. If the sample is lower than the probability for refraction and reflection combined, the sample is considered reflective. If the sample is not lower however, the sample is considered to be diffuse. The following simplified code of the sample() function from the Arch&Design shader that was implemented for the plugin, shows in a little more detail how the shading is approached.

```

void sample()
. color = getDiffuseColor();
. normal = applyBump();
. reflectProb, refractProb = calculateProbabilities();
.
. s = sample(); //random number between zero and one
. if (s < refract) //REFRACTION
. . outDirection = refract(rayDirection, normal, indexOfRefraction);
. else if (s < refractProb + reflectProb) //REFLECTION
. . if (completelyGlossy)
. . . outDirection = reflect(rayDirection, normal);
. . else //GLOSSY REFLECTION
. . . outDirection = wardSampleDir(reflect(rayDirection, normal),
. . . . glossiness);
. else //DIFFUSE
. . outDirection = sampleCosineHemisphere(normal);

```

Figure 8 – Simplified code from one of the RapidRT shaders written for the project.

The reflection curve of the material (BRDF), which controls how much light is reflected at different grazing angles and controls the reflection probability in the shader, can be specified by the user. Alternatively the *by IOR* option can be selected, which enables Fresnel reflections using the specified index of refraction value instead of the custom curve (see Figure 9).

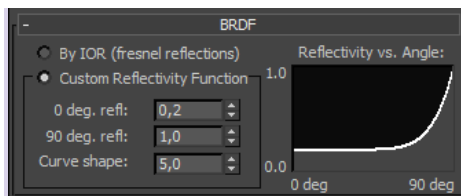


Figure 9 – This screenshot from 3ds Max 2012 shows the BRDF rollout of the Arch&Design material, where a custom reflectivity function can be specified.

The curve used by the plugin for the user customizable curve is Schlick's approximation of Fresnel reflectance (Schlick, 1994); here, taken from Real-Time Rendering by Akenine-Möller *et al.*

$$R_F(\theta_i) \approx R_F(0^\circ) + (1 - R_F(0^\circ))(1 - \overline{\cos \theta_i})^5,$$

where θ_i is the angle of incident, R_F is a reflectance function and $R_F(0^\circ)$ is the value of the reflectance function when the incident light is perpendicular to the surface. In the plugin shader, this becomes:

```
float RdotN = -dot(rayDirection, normal);
reflectionProbability = (minRef + (maxRef - minRef)
    * pow(1.0 - RdotN, refShape)) * reflectiveness;
```

Here, $RdotN$ is the dot product of the two normalized vectors of the ray direction and surface normal, which is equal to the $\overline{\cos \theta_i}$ term. $minRef$ and $maxRef$ are the minimum and maximum reflection values, respectively, specified by the user and ranging from zero to one. $refShape$, also specified by the user, is the power of five exponent. $reflectiveness$ is a multiplier used to scale the reflectivity value, and is set by the user as well.

When a sample is considered to be reflective, it can be more or less glossy. The Arch&Design material uses a glossiness parameter to control how glossy a surface is, whilst the shader made for RapidRT uses a roughness value. Some kind of conversion has to be made and only a simple conversion is done in the current version of the plugin:

$$roughness = 1 - glossiness;$$

If not reflected, the light can be refracted. In the shader, the probability for this is calculated in the following way:

$$refractProbability = (1.0 - reflectionProbability) * transparency;$$

The $reflectionProbability$ is calculated as described earlier in this section, and $transparency$ is a value specified by the user. The rationale behind this calculation is that the light which does not reflect either refracts or does not.

Even though the interface of the Arch&Design material provides options for glossy refractions, these are not supported by the plugin. This is because it has not been investigated for this work, due to prioritization reasons. The reasons are the same for the different map inputs to the Arch&Design material that lack support in the plugin. For diffuse interactions with a surface, the shader uses the Lambertian reflectance model.

3.2.3.3 Matte/Shadow Material

The Matte/Shadow material was implemented in the old version of the plugin, since it was needed for a use case. It can be seen in Figure 23. It is not a complete implementation of the Matte/Shadow material. The only supported feature is that it captures shadows. This is done by first calculating the amount of light reaching a point. To do this, shadow rays and rays to sample the indirect diffuse light are shot. The resulting light is then clamped to a value between zero and one. Finally, a refraction

ray is traced in the same direction as the incoming direction. The resulting color of the refraction ray is then multiplied by the amount of light at the point. This achieves the effect of seeing through the plane while still having shadows cast on it. This *shadow plane* functionality was not implemented for the new version of the plugin, since it was not prioritized at that point in the project.

3.2.4 Maps

Maps/Textures, or Texmaps, as they are called within 3ds Max, can be combined similarly to materials as mentioned in Section 2.3.1.5. In order to support any combination of Texmaps, the 3ds Max API function to render a Texmap to a Bitmap is used. If a Bitmap is used, it is simply converted to a RapidRT texture. If any other Texmap is used, the hierarchy of Texmaps is traversed to find a suitable size for a Bitmap that it will be rendered to. For example, if a Mix Texmap would use a Bitmap and a Noise Texmap, the size of the input Bitmap would be used. If there are only Texmaps without defined sizes in the Texmap hierarchy, a default size of 256x256 pixels, which is defined by the plugin, is used.

Bump maps are treated a bit differently. There is a Texmap within 3ds Max called Normal Bump. If a Texmap is attached to the bump map on any of the supported materials it is checked to see if it is a Normal Bump. If this is the case the Texmap attached to the Normal Bump is copied like any other texture. All other Texmaps attached to a bump map is assumed to be a height map. These are converted to a normal map by the plugin. This is done to reduce the work needed in the shader.

To avoid loading the same texture multiple times, the textures are kept in a hashmap. If the same texture is found in the scene again, the RapidRT instance of this texture that is already loaded is used and no loading takes place. Before this solution was implemented a big performance impact was noted in certain scenes with many (more than 100) Texmaps used by multiple materials.

3.2.5 Geometry Conversion

As mentioned in Section 2.3.1.2, 3ds Max stores meshes in a format that is made for easy editing, which requires some conversion to be made rather than just copying the data straight into RapidRT. The list of vertices in 3ds Max has to be flattened, which means that vertices that do not share all attributes (normals, uv-coordinate, etc) have to be duplicated for each combination of attributes.

It is not known when starting the conversion how large each output buffer should be, since it is not known how many new vertices will be created. There are also no normals; they have to be calculated. 3ds Max provides functions for calculating normals, and even expanding the list of vertices. However, the normals created are not weighted based on the size of the face, which creates issues for certain meshes, as can be seen in Figure 10.



Figure 10 – Figure A shows a tire rendered with the default scanline renderer, figure B show the same tire rendered with the RapidRT plugin using normals from 3ds Max functionality, figure C shows our own implementation rendered with the RapidRT plugin, which is based on a solution suggested in 3ds Max SDK (Autodesk, 2011d).

The process of setting up the vertex, normal, and index buffer for RapidRT is not trivial and essential to get the geometry to show up correctly. To convert from the 3ds Max format to RapidRT, the list of faces is iterated to find out how many different materials the geometry uses. This is because one RapidRT object will have to be made for each material, due to the fact that RapidRT cannot use different shaders on different parts of the same object.

The list of faces is iterated yet again. Since we are now trying to make several different geometrical objects for RapidRT out of one 3ds Max object, vertices may belong to several of these objects. Hence, there could be several copies of the same vertex, but with different attributes. The aim now is to set up a structure where, if a vertex is asked for, the correct one is returned; the correct one being the vertex with the matching material, texture coordinate, and smoothing group.

The setup of this structure proceeds as follows: for each vertex there is a linked list containing unique copies of that vertex. The attributes that make it unique are the following: smoothing group, texture coordinate, and material. For each copy the smoothed normal is also stored. In this iteration of the faces, vertices are added to these linked lists. If there is a possibility to merge vertices they are merged. This is possible if they share all of the attributes that made them unique. Smoothing groups are considered equal if the vertices share at least one group. When two vertices are merged together the resulting smoothing groups are the union of the two vertices' smoothing groups.

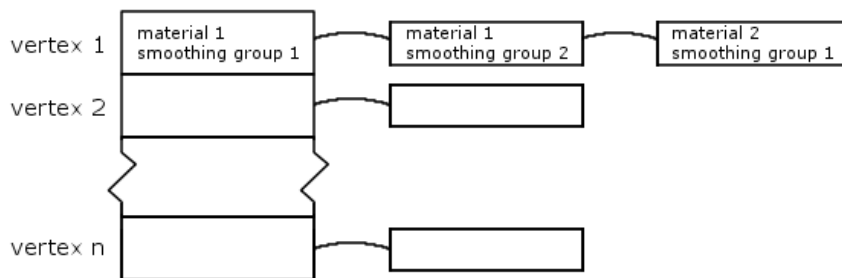


Figure 11 – This figure shows the structure used to manage the creation of geometry. Vertically and to the left is a list of all vertices. For all vertices there is a list of copies of that vertex, each with unique properties.

Even though vertices are merged as much as possible in the previous step, there is still a possibility that some of these copies still need to be merged. If, for example, one vertex is shared by three faces, let us call them A, B and C. A belongs to smoothing group one, B belongs to smoothing group two, and C belongs to both smoothing group one and two (see Figure 12). Suppose that they all use the same material and texture coordinates. They should be merged since they share the same attributes, but depending on which order these are added to the list of copies, they may be merged or not. If A and B are added first, they will not be merged, since their smoothing groups do not overlap. Then C will be merged with A. However, if C is added first, all of them will be merged, since both A and B individually share smoothing group with C. To solve this, the next step is to iterate the vertices gain; each combina-

tion of vertex attributes in the linked list is checked in order to see if they can be merged.

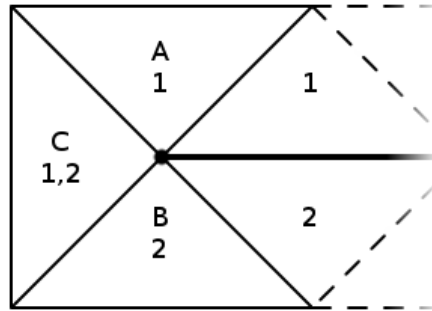


Figure 12 – In the image above, faces A, B, and C shares a common vertex. Imagine that the polygons form the shape of a gable roof seen from above; where the vertices on the edge are located lower than those that make out the ridge of the roof. Face C will be smoothed with A and B to form a cone-like smoothing at the end on the roof. This is not a good way to model a roof, and the result is far from satisfactory, yet the example serves to show where a smoothing situation such as this may arise.

The next step is to merge normals of vertices of the same smoothing group but with different texture coordinates. If this is not done, artifacts such as in Figure 13 can be seen, where a seam appears between faces that share vertices that have different texture coordinates. Merging is done by iterating each list of vertex copies. First, all elements before the current element is examined; if the smoothing group overlap, the normal is copied to the current element, since it has already been merged with the other vertices in previous steps. If no such overlap exists we continue with examining the elements after the current element, and merge the current one with the ones that it can be merged with.



Figure 13 – A seam can be seen in the reflections on the sphere where normals have not been merged and vertices share smoothing group but have different texture coordinates. The seam can be seen clearly in the reflection on the sphere where the lines in the road make a sudden skip vertically.

At this point we have a linked list for each vertex that contains all the necessary copies, and enough information about them to be able to separate them. It is possible to find a unique vertex given a face with associated material, smoothing group, and texture coordinate.

Finally, the list of faces is iterated again for each material. Each face with the correct material will be added to the final buffers. The indices into the vertex array is used to find the new vertex; in order to filter out copies that are used by other materials or faces with other smoothing groups, these attributes are included in the query.

```

for each face f
.   increase counter for material f
.
for each face f
.   for index i 1-3 in f
.   .   append to vertex list i a vertex with attributes from face f
.   .   .   (smoothing group, mtl id, texture coordinates)
.
for each vertex list l
.   for each vertex v in l
.   .   normalize v and find new index given its mtl index
.
for each material m
.   make buffers of correct size, given last index for mtl
.   for each face f
.   .   if material of f == m
.   .   .   for each index i 1-3 of f
.   .   .   .   get vertex v in vertex list for vertex i that match all
.   .   .   .   .   attributes for f
.   .   .   .   .   append new index of v into new index array
.   .   .
.   .   for each vertex list l
.   .   .   for each vertex v in l
.   .   .   .   if material for v == m
.   .   .   .   .   put v into attribute arrays
.   .   .
.   .   create RapidRT geometry given new index array and attribute arrays

```

Figure 14 – Pseudocode for converting the 3ds Max geometry to the representation used in RapidRT.

However, this solution has problems. Imagine you have three copies of a vertex, A, B, and C. All of them have different texture coordinates, so they will not be merged. A belongs to smoothing group one, B belongs to smoothing group two, and C belongs to smoothing groups one and two. First A will be processed; it will be merged with C and its smooth-

ing group updated to belong to both one and two. Then B will be processed, it will copy the normal from A, since they share smoothing group two. However, B's normal will be discarded. And finally C will also copy the normal from A. This is not correct, since the normal is now a combination of only A and C, whereas it should be the combination of all three. However, in most applications this is not a problem; it is rare to see more than two different smoothing groups at one vertex.

Currently, the hashmap solution described in Section 3.2.4 for maps is not implemented for geometry. Each copy of a geometric object will be loaded twice, which means that RapidRT will use up more memory than necessary and that the setup of the scene will take longer time. However, the impact of this is not as severe as with the maps, as no substantial performance impact was noted when loading many objects. Note also that geometry is split into more objects if they use the multi-material, which makes implementing the hashmap solution a little bit trickier.

3.3 Interactive Rendering

Both the old and the new version of the plugin supports the ActiveShade functionality of 3ds Max. Due to the old version of RapidRT being optimized for modifications of certain objects in a scene (see Section 2.2.3), the old version of the plugin supports moving and adjusting lights as well as moving the camera. The new version, with its current light limitations (see Section 3.2.2), supports only adjustments of the mr Sky Portal. Emissive geometry is only possible to move, not to adjust in any other way. The adjustments of the emissive geometry are done through the material editor, which the interactive renderer of the plugin is not aware of.

Even though moving emissive objects and non-emissive objects are essentially the same, the latter is not supported by the plugin. There are different reasons for this. In the old version, editing of the scene was limited to cameras and light for reasons mentioned above. When geometric lights were added in the new version, to increase the number of available lights, no considerable amount of work was needed in order to allow these new types to be moved as well. Doing the same for ordinary geometry would require more work as they are treated differently within the plugin. Also, if moving geometry is supported, it could be expected that editing the geometry would be allowed. This could be problematic, because it means that the whole mesh would have to be sent to RapidRT again, and that a rebuild of its acceleration structure would be needed. It was therefore not prioritized or implemented.

The old version of RapidRT had built-in support for sub-sampling. The old version of the plugin therefore could be told to sub- or super-sample, which could increase interactivity or produce better-looking images per iteration, respectively. Support for this does not come built-in in the new version of RapidRT. However, sub-sampling could be done manually by rendering the image at for example half the resolution and then scale the image to be twice as big.

The interactive renderer in the RapidRT plugin basically uses the exact same code as the production renderer. It differs mainly in the render loop. The interactive renderer requires frame calls to be issued to RapidRT with regular intervals. Two options were considered: either using a separate thread or a timer that would trigger the frame calls. For simplicity, a timer was chosen. One of the reasons the timer was considered simpler was that issues with threading was avoided; 3ds Max is not thread safe and has limited support for synchronization (Autodesk, 2011e).

3.4 Rendering time

The biggest impact on total rendering time the plugin has is the time it takes to convert the 3ds Max scene to a RapidRT scene. When the scene is loaded, it is mostly up to RapidRT to render the scene. The only interaction between the plugin and RapidRT while it is rendering is that the plugin asks RapidRT to continue to produce more frames, and it also receives finished frames from RapidRT. There could possibly be some timing issues where the plugin lets RapidRT idle before asking for a new frame, but since RapidRT supports double buffering (see Section 2.2) it continues working on the next frame as soon as it has finished the previous frame. The plugin would have to wait two frames before asking for a new one, for RapidRT to start idling.

In production rendering, this is unlikely to happen, since the main thread is blocking on a synchronization call for most of the time. It does so with a timeout however, and when it times out, the main thread updates the user interface to keep it from appearing frozen. In addition, when a frame is complete it also copies the frame to a 3ds Max Bitmap, and once again updates the user interface.

Interactive rendering on the other hand uses a timer, which increases the risk of letting RapidRT wait. The timer is set to ten milliseconds and only if RapidRT produces frames at a rate faster than this it will idle. In fact, since the synchronization with RapidRT is hard-coded to happen roughly 100 times per second, this is also the maximum number of

frames that can be output per second; and 100 frames per second is so fast that the user most probably would not notice if the frame rate were any higher. This means that the rendering is still interactive.

3.5 Usability

Even though it was not the main focus of this work, usability concerns comes with the territory when developing tools and features that people are expected to use. This section discusses usability concerns that naturally emerged when handling support of the interface features of 3ds Max.

3.5.1 User Interface

When it comes to the user interface of the plugin, the subject is split in twain. One part concerns the interface of the render plugin, and the other the interface of 3ds Max more generally.

3.5.1.1 Renderer Interface

When creating a render plugin for 3ds Max, the plugin is given a certain screen space in the render setup menu to place any controls that the renderer needs. Within this screen space, the plugin can decide the look and behavior of the user interface.

Many of the controls that decide how the scene should be rendered are located outside the location given to the renderer plugin; such as settings for the lights, camera and materials. If a renderer absolutely needs certain input, such as special controls for the depth of field from the camera, it is possible to make a camera plugin.

The rest of the user interface is mostly out of the plugin creator's control. For example, the rendering is handled by 3ds Max as follows: there is a render settings window for setting up render options and when an image is rendered it is presented in another window. Whether this is optimal or not can be argued. However 3ds Max is a software with a legacy, with several previous releases, and its users probably expect it to work in a certain way. If the rest of the application follows a standard that the users are used to it is not necessarily a good idea to deviate from it, as reasoned by (Cooper *et al.*, 2007, p.317). This is the reasoning behind not implementing graphical interface functionality for the plugin, beyond the standard functionality provided.

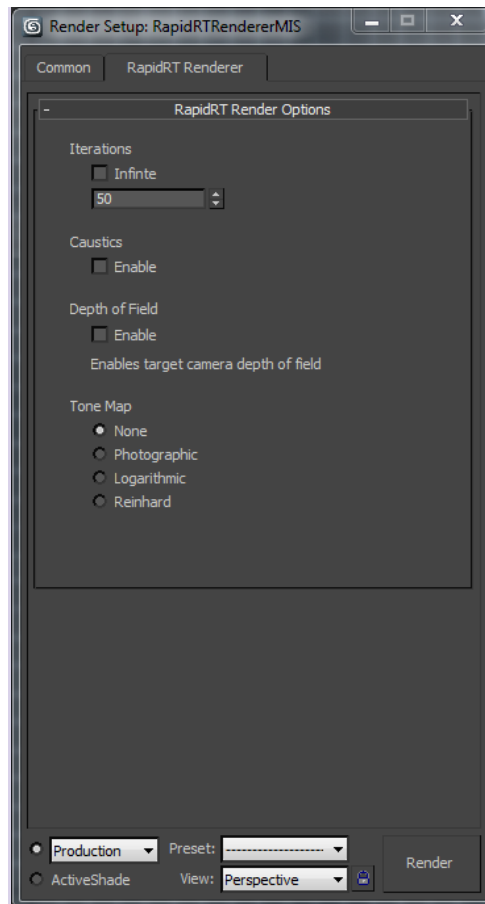


Figure 15 – A screenshot from the Render Setup menu of the RapidRT plugin in 3ds Max 2012. Note that the Tone Map options are not fully supported in the plugin.

3.5.1.2 3ds Max Interface

In the translation from 3ds Max to RapidRT there are a lot of values that needs to be extracted from 3ds Max to be able to make a sensible translation, and hopefully one that matches as closely as possible.

In many instances, 3ds Max supplies a lot of options for its users, and many times this causes complications in the translation. Often the question becomes where to draw the line; one option should be supported in the plugin and not another. Implementing all of these options has already been established to be infeasible in the context of this project and the main guideline was to look at features that would improve visual quality of the rendered image (see Chapter 1). This renders many interface elements without function, but still visible and apparently tweakable for the user. Furthermore, the functionality will be spread out all over the 3ds Max interface. This is not good, since it makes user interaction with the product a guessing game. Although it is unfortunate, there is not much to be done about this in many cases. In some situations, for clarity, when providing limited or different translation of a feature, the interface elements can be placed in the render settings menu; an example of this is the depth of field option in the plugin (see Figure 15).

3.5.2 Interactive Rendering

Interactive rendering also adds to the usability of the plugin. If the user wants to make some type of modification to the scene, such as the intensity of a light source, the interactive renderer will update the image to reflect the change. Without the interactive feature the user would have to do the change and then start a production render. Depending on the renderer, it can take a considerable amount of time before the first frame becomes visible (See Section 4.4).

4. Results

In order to evaluate the results, a measuring stick is needed. A simple approach, which is chosen here, is to compare to similar solutions. During the course of the project, the two renderers NVIDIA mental ray and NVIDIA iray were used for this purpose. This choice was made since they both are included with 3ds Max 2012 and 2013. mental ray was primarily used for making sure that the geometry was translated the way it should, while iray was used at later stages when comparing visual quality.

In this section, the focus is on visual quality and likeness in the comparisons, since performance has been out of our control, apart from the time it takes to set up the scene. The data supplied with the images is meant to help the reader get a feeling for the fairness of the comparison in terms of visual quality and speed. Even if the focus is not on the speed of the rendering, which is very much an aspect of the renderers themselves and not the plugins, such numbers are given anyway occasionally, purely because the reader might find them interesting. If not otherwise noted, the setup for rendering is a machine with an Intel Core2 Quad CPU 2.67 GHz, 8 GB RAM and an NVIDIA Quadro FX 4600 GPU with 768 MB of memory.

Since both iray and RapidRT are progressive renderers, it is convenient to specify the number of iterations that has progressed. However, there are most surely differences in the algorithms used in each iteration between the renderers, so this does not guarantee any fairness. We present this metric anyway, since it at least says something of how far the image has progressed. Another metric that can be used is time. The time value however, is very dependent on hardware and since RapidRT and iray are CPU and GPU based respectively, a fair such comparison is hard, if not impossible, to make.

4.1 Scene translation

The main part of the work was the scene translation: making sure that the camera angles, lighting and scene sculpting created by the user of 3ds Max was transported to and rendered by RapidRT correctly.

4.1.1 Geometry, Lights and Camera

As can be seen in all the comparison renders in this report, most notably in Figures 18 and 20, the geometry, environment light, and camera are translated in a manner which can be considered to produce correct results.

In Figures 17 and 18 below, the scene is lit with the same HDRI in both RapidRT and iray and the lighting is matching. The emissive geometry, however, differs somewhat, since all parameters are not taken into consideration (see Section 3.2.2.2). Differences in emissive geometry lighting can be seen in Figure 16.

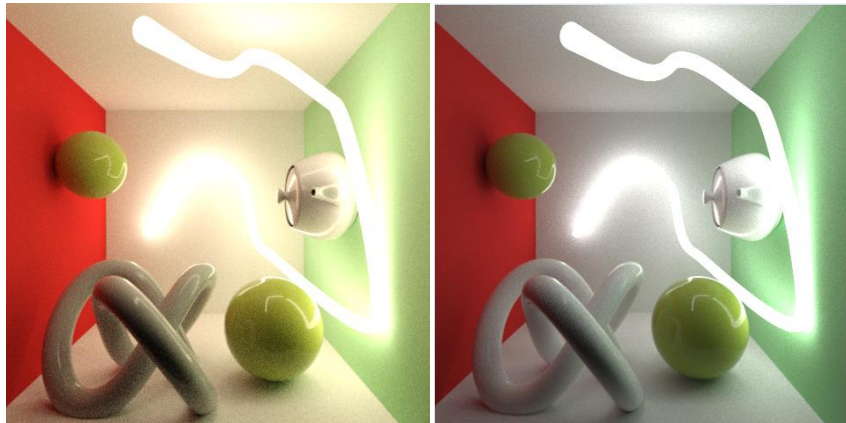


Figure 16 – These images show the difference in the lighting with emissive geometry between iray, to the left, and the RapidRT plugin to the right. The temperature of the light is not considered in the RapidRT plugin.

Also a difference in look between iray and RapidRT in 3ds Max is the background image. For a reason unbeknown to us, iray chooses to downsample the image, and so it is blurry, which can be seen in side-by-side comparisons with RapidRT (see Figure 17).



Figure 17 – iray. The background image is blurry even though no depth of field is used.



Figure 18 – RapidRT plugin. The background is in correct resolution and the lighting is very similar to the iray image above.

Both Figure 17 and 18 above are rendered on different hardware than the other images in this report. For more information on this and these images, see Appendix B.

4.1.2 Materials - Shader conversion

The shaders that have been written for RapidRT in order to match the standard and the Arch&Design materials of 3ds Max are of importance if the rendered images are to look similar. Similar, in this case, means in regard to what the user would expect from rendering a material in for example iray or mental ray.

The scene in Figure 19 and 20 shows a render of a kitchen, which is a rather complex scene. It is lit entirely by the environment. In the render done with the RapidRT plugin, 1 086 iterations passed. The iray render was allowed to run until the image had a similar level of noise with the

RapidRT image; the number of iterations is not known. As can be seen in the RapidRT render, there are a few white pixels. These overly bright pixels are due to a bug which has not been located and the prime suspect is the shader code that was written for the project. A larger image is available in Appendix A.



Figure 19 – iray render of a kitchen scene lit by an HDRI via 3ds Max 2013.



Figure 20 – RapidRT render of a kitchen lit by an HDRI via 3ds Max 2012. The reflections are tweaked slightly via the Arch&Design material as to match the iray image above.

One of the major differences between iray's and the plugin's implementation of the Arch&Design material is how glossy reflections are treated. The test scene in Figure 21 consists of a fully reflective teapot, a background environment, and a plane behind the teapot to make it stand out from the background. The difference is due to a mismatch between the roughness value used in the shader, and the glossy parameter of 3ds Max (See Section 3.2.3.2). Clearly, some other conversion needs to be done in order to produce a more correct result.







<i>3ds Max Glossy Value</i>	<i>iray render</i>	<i>RapidRT render</i>
0.10		
0.40		
0.90		

Figure 21 – The figure shows how a material is affected by different levels of glossiness in iray and RapidRT, respectively. In this case, where the object is fully reflective, a glossy value of one means that the object should be a perfect mirror.

Another difference is the fact that the plugin shader currently does not have any reflection color, which means that certain materials, such as metals, are difficult to match. Figure 22 shows a dragon with the copper preset of the Arch&Design material.

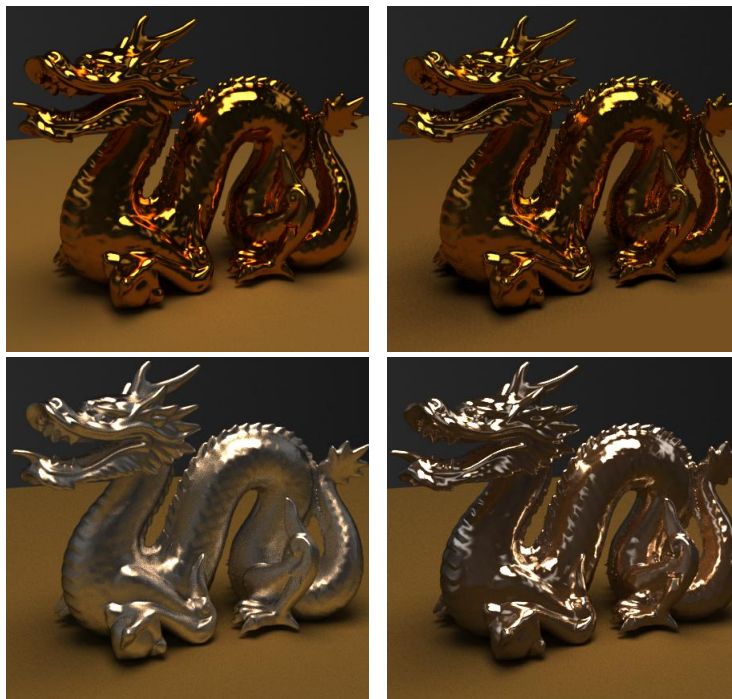


Figure 22 – The image shows the Stanford dragon rendered by three different renderers. The top left image is rendered using iray. The top right image shows a mental ray render and the lower left shows the RapidRT plugin with identical settings as the two previous renders. The lower right image shows the RapidRT plugin tweaked to a closer match. The glossiness and color have been changed.

4.2 The Use Cases

The first use case used to gauge the progress was the one in Figure 23. The image shows a car on a football field.



Figure 23 – The old version of the RapidRT plugin. The car is standing on a shadow plane to simulate throwing shadows on the environment.



Figure 24 – The new version of the RapidRT plugin. The car is standing on a green plane.

The render with the old version in Figure 23 uses a directional light. A background image has been setup to try and match the directional light in the new version in Figure 24, though it is a lower resolution image. Also note that the shadow plane used in the old version is not supported in the new version.

Other scenes that were used were several different Cornell boxes, among them the scene in Figure 25, which are showing the caustic effects. Depth-of-field effects can be seen in Figure 20.

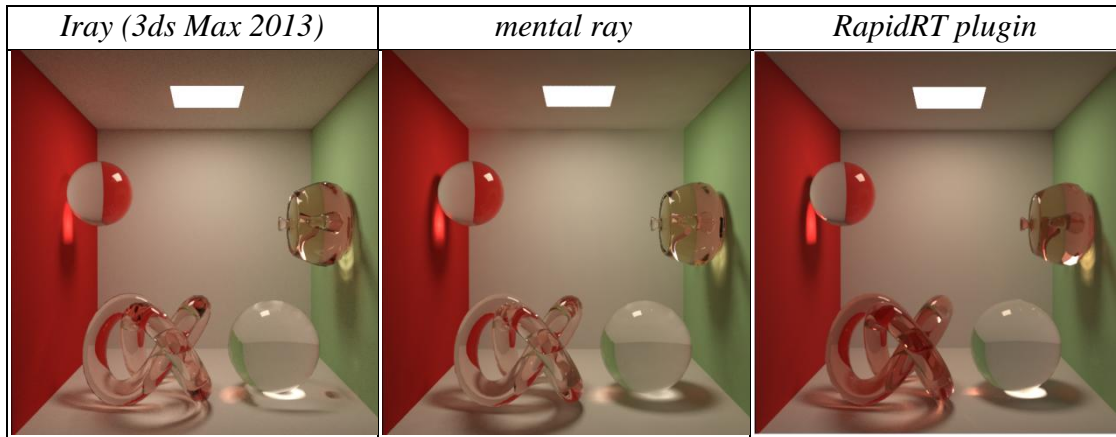


Figure 25 – These images show the caustic effects that the different renderers produce.

The image comparison above shows that the shaders written for the plugin produce similar caustic effects to those of the integrated renderers. In the leftmost image, iray can be seen to have some issues with the caustic produced by the sphere on the floor; something is obviously wrong. Where the RapidRT plugin, to the right, differs most from the others, is on the torus knot on the floor. The torus knot appears redder than in the other renderers, as well as showing reflections of the light source on inside surfaces, which can be attributed to effects that have not been implemented in the shader for the material.

4.3 Interactivity

The interactive rendering capabilities of the RapidRT plugin are hard to demonstrate in a paper. Also, the degree of interactivity in terms of speed depends on the hardware, which makes it difficult to compare (see Section 4). Another aspect of the interactivity that is easier to discuss is what can be manipulated in the scene and be updated by the renderer. In the RapidRT plugin, cameras and lights can be moved. However, only the mr Sky Portal light can be manipulated in any other way; which includes changing the intensity of the light.

4.4 Setup/load

Here follows results from a couple of tests where the time to image was measured, from that the render button was pressed. The data shows measurements from the new version of the RapidRT plugin. The times are clocked manually and so are not exact. It should be noted that iray performs baking of the environment before starting to render. Exactly

what the baking entails we cannot say, we can only conclude that it takes quite a lot of time for the baking to finish, and that a big part of the time to image consists of baking in iray's case.

4.4.1 The Robot Scene

The first scene used for comparison consists of a robot standing on a table, and is lit by the environment. The scene is rendered with iray from 3ds Max 2012, which is not the latest version of iray. The scene contains 5 078 992 polygons, 2 568 899 vertices, and 108 objects. The resolution of the image produced was 640x480 pixels. The plugin took 9 seconds on average using two results, while iray took 54 seconds for the first render, and if the scene had already been rendered once, 27 seconds.

One thing the test scene lacks is the use of textures. An explanation of the difference could be that the plugin handles texture in a different manner than iray. Also, objects in the scene use unique geometry, and not copies. If they were copies it would be possible to cache load these, which is something the plugin does not do (see Section 3.2.5). Hence, a scene with many copies might give iray the advantage. Regardless of these issues, the results are significant enough to conclude that the plugin at least performs well.

4.4.2 The Car Scene

The car scene from Figure 24 was also used to test the setup time (or rather, the time to image) for the plugin and iray. The iray version used is the one from 3ds Max 2013. The scene contains 2 577 128 polygons, 1 321 110 vertices and 230 objects. The scene has been rendered five times with each renderer and the numbers presented is the averages.

Resolution	RapidRT plugin	iray (3ds Max 2013)
1920x1200	10 seconds	29 seconds
640x480	6 seconds	40 seconds

The numbers above are interesting, since they seem very counter-intuitive in iray's case. With the lower resolution, the time to image in the RapidRT plugin's case, is lower. This seems logical, since the image to render is smaller and fewer rays have to be shot. In iray's case, however, it actually takes longer time to image, when rendering with smaller resolution.

5. Discussion

Implementing a render plugin for an application like 3ds Max is a daunting challenge. Because of 3ds Max's history, there are many features that are kept in the application from older versions so that users do not lose functionality they relied on. Even if this means that some functionality is kept through many generations of the application and eventually turns out to be rarely used. For example, the newly added integrated renderer, iray, does not support the Standard material. The multitude of different parameters, some physically correct and some not, makes it tricky to know what is best left at the sideline when trying to prioritize features for a project such as this.

The approach we took with iteratively evaluating features and deciding whether to try to implement them or not turned out to be a good choice. In this manner, we made sure that we did not overreach in terms of the scale of the project, while in the beginning realizing that we could not do all that a fully functioning plugin implementation would entail. The iterative approach helped when deciding what features were to be implemented by showing what was missing and what we needed to add. Some features that were implemented for the old version of the plugin turned out to not be used in the new version. However, this was mostly due to the fact that we switched from one version of RapidRT to another; a version which worked in a different way.

The 3ds Max SDK Programmer's Guide and Reference turned out to be a good help at some times and vexing at other. Sometimes, extracting parameters from 3ds Max, which could be set in the interface, was impossible to do in a straight-forward way. This caused us to go via detours; some things did not even show up in the Programmer's Guide and Reference. The documentation of the RapidRT API was useful during the course of the project, although especially so during the initial stages of learning RapidRT. However, we started using the new version of RapidRT – which shared only a limited amount of functionality with the old version, and of which we had no documentation – in later stages. Working with the new version was helped by having access to the source code and the people developing RapidRT, who were gracious to provide aid when needed.

In many places in the translation of 3ds Max to RapidRT, things can be improved by simply putting more time on adapting more of the features. Concerning the material translations for example, more of the parameters, preferably all, could be implemented and so help improve the one-to-one translation of the material. The things that we focused on, how-

ever, turned out to provide a good enough translation in many cases, or could be tweaked to match the integrated renderers quite well.

The newer version of RapidRT presented more challenges in terms of matching parameters from the 3ds Max interface than the old one; much due to the new version's physically based approach. As already has been established, the 3ds Max interface is filled with options and parameters, and it is not always straight-forward to make sense of them in a physically correct context.

In some of the images we have rendered and included in this report shows a few visual defects in the form of overly bright pixels (see Appendix A). As previously noted, the reason for these was not found, even though we spent quite some time on trying to locate the issue. Even though they appear under some circumstances, these pixels do not prevent the images to compare well overall to the integrated renderers, in terms of visual likeness.

6. Conclusion

The original question regarding the feasibility of integrating RapidRT in 3ds Max via the latter's plugin interface would we consider to have been answered to some capacity. The results show that as it stands, the RapidRT plugin developed during the project can achieve visual likeness with high-quality visuals on par with integrated rendering technology. It has to be noted, however, that the functionality and support of features in the plugin is limited. Mainly this is due to the scope of the project, but also due to translation issues such as parameters that do not match in a good way.

7. Future Work

The resulting plugin, however good results it produces under certain circumstances, is far from complete in its implementation. Many things can be improved or added to the plugin.

The interactive rendering aspect is a useful one from the users' point of view since the result of the rendering can be quickly evaluated with no need for waiting a long time just to check image composition or lighting. Even more useful than the current version of the plugin's implementation of interactive rendering, would be if the user could manipulate all parameters impacting the scene and that the image would reflect the changes as they are made. For example, moving geometry around and changing materials in the ActiveShade window would be interesting to look at.

Also very interesting and related to the previous paragraph is the possibility of manipulating the scene through the ActiveShade viewport. This would improve the usability of the plugin considerably since the users could then do manipulation and rendering in the same viewport, and in that way see the impact of the changes they have made directly and strongly, spatially connected.

In order to keep the frame rate up during interactive rendering on slower machines, sub-sampling could be done during navigation.

As it is, the current plugin does not work well using a cluster to aid the rendering. It was out of our reach to make it work, but it should work and scale well in theory.

Another aspect of a rendering plugin is the matter of animation. 3ds Max has support throughout its SDK for retrieving, for example, the objects' transformation matrices at specific times. In fact, a time value is often a parameter to functions used in relation to nodes that can change in the scene and can therefore be animated. We did not look closer on this since it was decided to be outside the scope of the project and we needed to narrow down the work that we set out to do. However, it seems that it should be feasible to do at least a coarse implementation without huge obstacles.

Also possible to do is tone mapping, which should help in create images with good lighting. This should perhaps also help in comparing with other renderers, if scenes are already set up with exposure control (tone mapping).

8. References

8.1 Bibliography

Tomas Akenine-Möller, Eric Haines, and Natty Hoffman (2008). *Real-Time Rendering 3rd Edition*. A. K. Peters, Ltd., Natick, MA, USA.

Robert L. Cook and Kenneth E. Torrance (1982). A reflectance model for computer graphics. *ACM Trans. Graph.*, vol 1, no 1, pp. 7--24, June.

Gregory Jason (2009). *Game Engine Architecture*, A. K. Peters, Ltd., Natick, MA, USA.

James T. Kajiya (1986). The rendering equation. *SIGGRAPH Comput. Graph.*, vol 20, no 4, pp. 143--150, August.

Turner Whitted (1980). An improved illumination model for shaded display. *Commun. ACM* 23, vol 23, no 6, pp. 343--349, June.

James Arvo and David Kirk (1990). Particle transport and image synthesis. *SIGGRAPH Comput. Graph.* vol 24, no 4, pp. 63--66, September.

Robert L. Cook, Thomas Porter, and Loren Carpenter (1984). Distributed ray tracing. *SIGGRAPH Comput. Graph.* vol 18, no 3, pp. 137--145, January.

Jeremy Birn (2005). *Digital Lighting and Rendering (2nd Edition)*. New Riders Publishing, Thousand Oaks, CA, USA.

David Roger, Ulf Assarsson, and Nicolas Holzschuch (2007). Whitted Ray-Tracing for Dynamic Scenes using a Ray-Space Hierarchy on the GPU. *Rendering Techniques 2007 (Proceedings of the Eurographics Symposium on Rendering)*, pp. 99--110, June.

Ingo Wald, William R. Mark, Johannes Günther, Solomon Boulos, Thiago Ize, Warren Hunt, Steven G. Parker and Peter Shirley (2007). State of the Art in Ray Tracing Animated Scenes. *Eurographics 2007 State of the Art Reports*.

Ingo Wald, Thiago Ize, Andrew Kensler, Aaron Knoll and Steven G. Parker (2006). Ray Tracing Animated Scenes using Coherent Grid Traversal, *ACM Trans. Graph.*, vol 25, no 3, pp. 485--493, July.

Mark Segal and Kurt Akeley (2012). *The OpenGL[®] Graphics System: A Specification (Version 4.2 (Core Profile) - April 27, 2012)*. The Khronos Group Inc. pp. 20--42, URL: <http://www.opengl.org/registry/doc/glspec42.core.20120427.pdf> (2012-05-23)

Alan Cooper, Robert Reimann, and Dave Cronin (2007). *About Face 3: The Essentials of Interaction Design*. John Wiley & Sons, Inc., New York, NY, USA.

Christophe Schlick (1994), An Inexpensive BRDF Model for Physically-based Rendering. *Computer Graphics Forum*, vol 13, no 3, pp. 233--246, August.

Matt Pharr and Greg Humphreys (2010). *Physically Based Rendering, Second Edition: From Theory to Implementation* (2nd ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Autodesk, Inc (2011). 3ds Max SDK Programmer's Guide and Reference 2012.

a: *Modeling*. URL:

http://download.autodesk.com/global/docs/3dsmaxsdk2012/en_us/index.html?url=files/GUID-52AEE4CD-1E6E-4021-AB7C-A911BA780F2-197.htm,topicNumber=d28e17575 (2012-05-23)

b: *Meshes*. URL:

http://download.autodesk.com/global/docs/3dsmaxsdk2012/en_us/files/GUID-031890D6-CF79-488C-A204-B512698DB10-240.htm (2012-05-23)

c: *Overview: Plug-ins*. URL:

http://download.autodesk.com/global/docs/3dsmaxsdk2012/en_us/index.html?url=files/GUID-031890D6-CF79-488C-A204-B512698DB10-240.htm,topicNumber=d28e19963 (2012-05-24)

d: *Computing Vertex Normals*. URL:

http://download.autodesk.com/global/docs/3dsmaxsdk2012/en_us/index.html?url=files/GUID-9B83270D-1D20-4613-AB35-5EE9832C162-247.htm,topicNumber=d28e20218 (2012-05-24)

e: *Thread Safety*. URL:

http://download.autodesk.com/global/docs/3dsmaxsdk2012/en_us/index.html?url=files/GUID-610CE507-CD6B-4D82-A248-50BCB7F9CD4-99.htm,topicNumber=d28e11251 (2012-05-24)

f: *Lesson 6: Parameter Blocks*. URL:

http://download.autodesk.com/global/docs/3dsmaxsdk2012/en_us/index.html?url=files/GUID-610CE507-CD6B-4D82-A248-50BCB7F9CD4-99.htm,topicNumber=d28e11251

Autodesk, Inc (2011). *3ds Max Reference 2012*.

x: *Multi/Sub-Object Material*. URL:

<http://download.autodesk.com/us/3dsmax/2012help/index.html?url=files/GUID-D968CDD9-4C5D-489D-A311-ED7486FCD4A-2019.htm,topicNumber=d28e392409> (2012-05-23)

y: *Raytrace Material*. URL:

<http://download.autodesk.com/us/3dsmax/2012help/index.html?url=files/GUID-06832470-021B-47A4-B912-E347156B74A-2005.htm,topicNumber=d28e389192> (2012-05-24)

z: *Standard Material*. URL:

<http://download.autodesk.com/us/3dsmax/2012help/index.html?url=files/GUID-78973040-1D59-491E-8440-30B786641AD-1987.htm,topicNumber=d28e385840> (2012-05-24)

NVIDIA (2012). Features mental ray Technical Specifications.

1a: URL: <http://www.mentalimages.com/products/mental-ray/about-mental-ray/features.html> (2012-05-24)

NVIDIA (2012). About iray Physically Correct GPU Rendering Technology.

2a: URL: <http://www.mentalimages.com/products/iray/about-iray.html> (2012-05-24)

8.2 Image Credits

Figures 13, 17-21, 23, 24, 26-31: Images are using HDRIs from <http://www.hdrilabs.com/sibl/archive.html> (2012-05-24)

Figure 27: 3ds Max model *new room*, by morfik, uploaded 2009-08-11 <http://www.evermotion.org/download/show/122/new-room#> (2012-05-25)

Figure 22 and Cover image: The Stanford dragon is taken from The Stanford 3D Scanning Repository, <http://graphics.stanford.edu/data/3Dscanrep/> (2012-05-25)

Figures 17-20, 28-31: The scene depicted in these images are based on a kitchen scene found at: <http://cg3dmodels.com/interior-3d-scenes/kitchen-interior-3ds-max-scene-models/> (2012-06-07)

Appendix A



Figure 26 – A kitchen scene lit by an HDRI. The image is rendered with the RapidRT plugin.



Figure 27 – A living room scene lit by an HDRI. The image is rendered with the RapidRT plugin.

Appendix B

The following images, as well as in Figures 17-18, are rendered using the following setup:

CPU: Intel Core i7-3930K @ 3.2GHz

GPU: 2x NVIDIA GeForce GTX 580

Power consumption:

iray rendering on 2 GPUs: 730W for the whole system. Idle: 125W.

RapidRT rendering (with a small GPU in place instead of the 580s): 185W for the whole system. Idle: 60W

The small GPU used in the RapidRT renderings was a NVIDIA Quadro FX 1700.

iray bundled with 3ds Max 2013 was used.

RapidRT 4.0 pre-release with MIS support used (plugin for 3ds Max 2012).



Figure 28 – iray. 10 seconds.



Figure 29 – RapidRT plugin. 10 seconds.



Figure 30 – iray. 300 seconds.



Figure 31 – RapidRT plugin. 300 seconds.