# A Practical Investigation of RenderMan RIS API

MSc Computer Animation and Visual Effects
Thesis
Bournemouth University
NCCA

Martin Davies

August 2015

**Abstract**

Pixar have recently released a new modular rendering framework for RenderMan, called RIS. It was developed in order to meet demand for efficient high quality physically accurate rendering by introducing efficient and reliable ray tracing to RenderMan pipelines. RIS makes a transition from RSL to the more powerful and flexible C++ programming language for plugin development. It is very new there are minimal plugins available. Most plugins for RIS are produced by Pixar and shipped with RenderMan. There is an opportunity to explore the API that RIS provides and, since it meets the current "fashion" of generating realism in computer generated imagery it opens the door to explore extending RIS with non-photorealistic rendering techniques. This project grasps these opportunities by explaining some of the extensive API and describing the development of a non-photorealistic RIS integrator plugin.

# Contents

# Acknowledgements

Christos Obretenov from Lollipop Shaders for his unwavering support.

Dylan Sisson from Pixar for his help and advice.

The RenderMan team at Pixar for their support and help on the RenderMan forum and other avenues.

The NCCA.

My Family.

# Introduction

Recently RenderMan has undergone an evolution. Pixar have added a modular rendering architecture that brings an efficient and reliable raytracing solution to RenderMan. This new framework, known as RIS, was released with version 19 of RenderMan and will be available in future releases. Its predecessor, REYES, has numerous shaders developed for it but as RIS is extremely new, very little plugin development has occurred. This project aims to investigate the extensive API that is provided with RIS and, using this knowledge, a RIS integrator plugin has been developed.

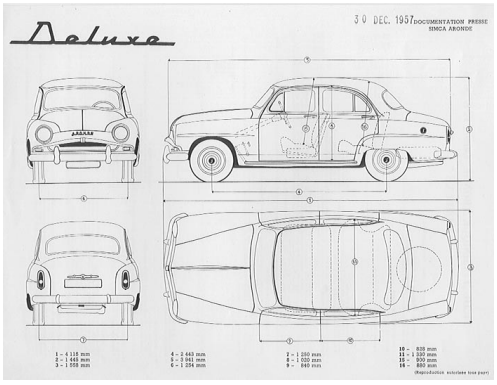
Currently it is the "fashion" to strive for realism in computer generated images. This area is covered well in RIS as Pixar supply the user with Direct Lighting, Path Tracing, and VCM(Bidirectional Path Tracing) integrators as well as realistic BXDF models, such as Disney's Principled BRDF. Therefore, creating a non-photorealistic integrator plugin is very appealing. Non-Photorealistic Rendering(NPR) is an area of Computer Graphics that has been extensively researched. It has applications in movies, tv, video games, medical imagery, art, technical manuals, and educational textbooks. The objective of NPR can change, depending on the look and feel that is to be achieved[31]. For instance, a movie or an artist might use an NPR technique to create a stylised look to their work, e.g. Figure 2.1 (a) and Figure 2.1 (b). Whereas a technical manual or medical imagery needs to be very detailed, accurate, and informative. e.g. Figure 2.2 (a) and Figure 2.2 (b).

As is evident, there are numerous styles of NPR. Some are selected for their artistic appeal whereas others for their simplistic nature which can help to convey information that can often be very complex. Creating an integrator that implements all NPR techniques is a hefty task. To reduce this task the integrator should implement an edge detection technique. This technique has many applications and is, potentially, a very useful tool. It can be used to create technical draws, such as Figure 2.2(a) or anime style images such as Figure 2.3.

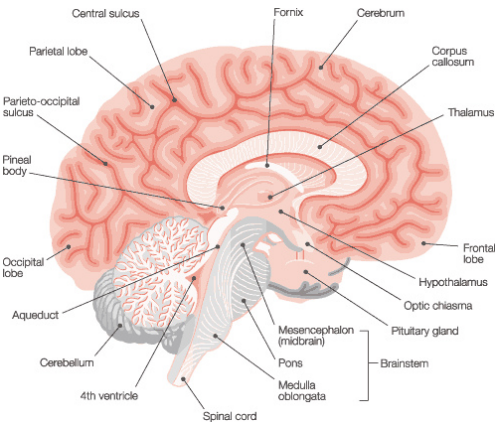
**(a)** Still from upcoming Peanuts movie. [27]

**(b)** Artist impression of woman using stippling. [12]

**Figure 2.1:** Artistic NPR examples.




**(a)** Technical drawing of car from four different angles. [21]




**(b)** Medical image of cross section of human brain. [7]

**Figure 2.2:** Technical NPR examples.

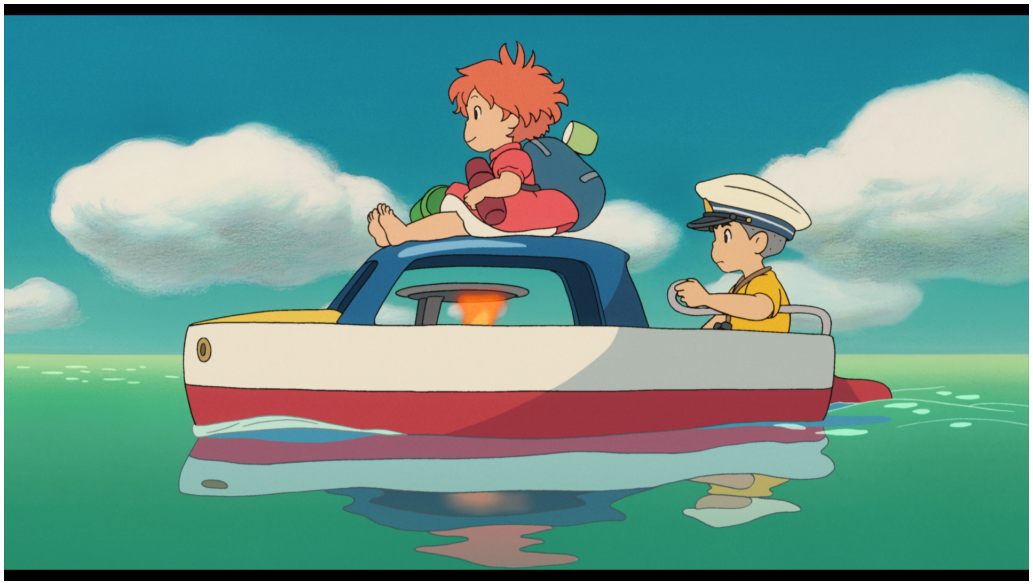


**Figure 2.3:** Still from Studio Ghibli's Ponyo. [8]

In chapter 3 an overview of the plugins that have been developed for RIS and REYES, and previous work in the NPR domain will be discussed. Chapter 4 consists of a technical background that studies the relevant technologies and methods that are required to achieve the goal. Chapter 5 presents the design of the system. This design has been informed by an investigation into the RenderMan RIS API. Chapter 6 describes the method of implementing the software. It also discusses some further design choices, the successes of the software, issues that arose during research and development, and limitations of the software. Finally, chapter 7 presents results of the implementation and concludes the findings of the research. Future work is also discussed in this section.

# Previous Work

This chapter focuses on previous plugins developed for RenderMan RIS and REYES and surveys previous work in the NPR research area. The focus of the NPR literature review is edge detection and line drawing. This covers various aspects from 2D image space to 3D object space methods.

## 3.1 RenderMan RIS

Many RenderMan based studios are moving to the RIS framework. For instance, The Moving Picture Company(MPC) has already made a full transition[11] and Industrial Light and Magic(ILM) are slowly migrating from REYES to RIS[10]. This transition is often a slow process as many studios have well defined and reliable pipelines in place for REYES which are often in use in large productions, e.g. The Star Wars Project at ILM[10]. Therefore, RIS plugin development either exists as secretive proprietary software or falls to the independent developer to pursue.

Currently, Pixar have released the majority of publicly available plugins, and with introduction of a free non-commercial version of RenderMan, it opens development opportunities to those who cannot afford a commercial RenderMan license. All of their plugins are available for use in RenderManStudio. Full functionality of this is available through Maya, Katana and Blender. There is limited functionality via other software packages such as Houdini and 3DSMax. All plugins can also be used via RIB files and command line rendering. As well as making all plugins available for use Pixar have also made a most of the plugin source code available as examples. These examples are available in $RMANTREE/lib/examples/RIS/plugins. Included are examples of Bxdf plugins, Integrator plugins, Pattern plugins and Projection plugins. All RIS plugins are developed in C++.

Bxdf plugins tell PRMan how light reflects off the surface of an object. In RIS each object has a Bxdf attached to it. The Bxdf is calculated per shading point attached to the object. This gives the user flexibility to apply different Bxdfs to each object, creating different reflection models in the scene. For example, this method makes it simple to have a diffuse and glass object in the same scene. Pixar has made the source code for all their Bxdf plugins available. These are:

- PxrBxdfBlend :
  Blends Bxdfs. This is going to be deprecated in future releases.
- PxrConstant :
  Constant reflectance model that inherits from the RixBxdfFactory class.
- PxrDiffuse :
  Diffuse light reflectance model
- PxrDisney :
  An implementation of Disney's Principled Shader[3].
- PxrGlass :
  Implementation of a glass Bxdf.
- PxrHair :
  Implementation of Marschner et al's light scattering model for human hair fibres [25].
- PxrLightEmission :
  Visualise the direct lighting response with respect to a direct light source.
- PxrSkin :
  Implementation of skin Bxdf with glossy reflections and subsurface scattering.
- PxrVolume :
  Applies a volumetric Bxdf to the model. Currently only path tracing and VCM integrators will be able to properly render an volumetric.
- PxrLM :
  Includes materials and layers. This is the building blocks of RIS's layering system.
- Simple Bxdf :
  Collection of well known Bxdfs models.

Integrator plugins perform the final shading calculations for PRMan. The shading algorithm used in an integrator is applied to every shading point in the scene. This means each

object will be shaded in exactly the same way and integrator shading can overwrite any. Pixar has made the source code available for all integrators except VCM. The integrators available are:

- PxrDebugShadingContext :
  Produces images that can be used in order to debug the shading context. For instance, normal and texture coordinates.

- PxrDefault :
  Evaluates the Bxdf assigned to each object and creates an image using this reflectance information. This is the only data in the final image, i.e. effects like shadowing are not present.

- PxrDirectLighting :
  Performs direct lighting where light rays only affect the shading point that they hit first. There is no indirect lighting component.

- PxrPathTracer :
  Unidirectional path tracing integrator. Rays are traced around the scene and colour values are added at each hit point for the final pixel colour.

- PxrValidateBxdf :
  Used as a debugging tool for Bxdf plugin developers. It has a 3 channel output that represents the luminance result for the hemispherical reflectance.

- PxrVisualizer :
  Includes numerous shading techniques for debugging scenes. This includes mesh, normal, and id visualisation and more.

Pattern plugins allow developers to create patterns on objects, such as noise, or manipulate the object in some way, such as displacement or mathematical calculations like the dot product. This allows the developer to control detail by varying the object's Bxdf parameters. Pattern calculations are performed for every shading point that is connected to an object. Therefore, different patterns can be applied to different objects in the scene. This aspect of RIS bares the most resemblance to RSL shaders. Pixar supply source code examples for a lot of patterns. This includes but is not limited to:

- PxrBump :
  Applies a bump map to and object from a file or algorithmically.

- PxrNormalMap :
  Applies a normal map to an object from a file or algorithmically.

- PxrRamp :
  Applies a colour ramp to an object.

- PxrMix :
  Mixes two colours together based on a defined mix percentage.

- PxrCross :
  Calculates the cross product between two vectors.

- PxrDot :
  Calculates the dot product between two vectors/

- PxrManifold :
  Version exist for 2D, 3D and 3D with normals. These allow transformations and arbitrary variable selection.

- PxrOSL :
  Evaluates and applies an Open Shading Language(OSL) shader to the object [26].

- PxrSeExpr :
  Evaluates and applies a Disney SeExpr script to the object [5].

- Third party patterns :
  A collection of third party patterns. Currently there is only an Ocean shader released with RenderMan.

Projection plugins manipulate how the rays released by PRMan behave and achieves certain camera effects, such as shutter speed. Ray behaviour can also be controlled by the integrator used in the scene. Pixar have released some projection examples, but they are not a commonly used within RIS. The examples are:

- PxrLightProbe :
  Creates scene as a light probe. i.e. as a HDR Map.

- PxrPerspective :
  Sets up camera perspective projection.

- PxrRollingShutter :
  Applies shutter effects to final shot.

There is also RIS plugins developed by independent companies available for purchase. For instance, Lollipop Shaders have developed an integrator that is available on their e-commerce website[15]. This integrator has a number of techniques in it including ambient

occlusion[16].

There is a lack of NPR extensions for RIS and a wealth of NPR research. Therefore, creating an NPR plugin for RIS will be a rewarding and useful tool. For a full description of how RIS works see chapter 4.

## 3.2 RenderMan REYES

REYES uses shaders in order to tell PRMan, or a RenderMan compliant renderer, how a scene should look. It does this by using shaders. There are five different types of shaders, written in the RenderMan Shading Language(RSL), that can be used in REYES. A surface shader describes the Bxdf of the object. This is how it reacts to light and therefore its appearance. A surface shader can also describe any patterns that may exist on the object. A displacement shader can alter the shading point position of an object. This allows the user to alter the shape and appearance of an object after RenderMan has diced the geometry. Light shaders are used to describe the direction, amount and colour of light that a light source distributes. Volume shaders describe how light is altered as it passes through a volumetric object, such as smoke. These can be used to create atmospherics. Imager shaders allow the user to alter pixel values after the render process has completed but before they are output to the final image[1].

There is a wealth of knowledge on how to write RSL shaders [1][30][9][23] and a lot of shaders have been developed in the many year REYES has been in use. It is trivial to create great looking objects from basic shaders; a plastic shader can be developed in 3-4 lines of RSL.

In terms of this project there is also a wealth of knowledge concerning NPR and edge detection shaders. For instance, in [1] there is a whole chapter on NPR using RenderMan. This chapter explains different camera settings to achieve non-photorealistic projections and shading. This covers drawing lines on objects to simulate the hatching rendering style[19] and lighting that make the line drawing effect more convincing. An interesting section, within the book, discusses shader architecture for edge and feature line detection. They use a pixel neighbourhood in order to find edges. By iterating through each pixel and evaluating its neighbouring pixels the algorithm acts as an image filter that detects silhouette and feature edges. For silhouette edges, the outline edges of the object, they consider "first- and second-order discontinuities of continuous tone data". This follows "Sobel's operator for first-order differentiation"[1] and is done using the following equations:

```
For first-order differentiation :
edge = (abs(a + 2 * b + c - f - 2 * g - h)
        + abs(c + 2 * e + h - a - 2 * d - f))/6.0

And second-order differentiation :
edge = (8 * x - a - b - c - d - e - f - g - h)/8
edge = abs(edge)
```

Where a, b, c, d, e, f, g and h are neighbouring pixels of some pixel in image space. For feature edges, edges within the object that occur due to curvature, they compare pixels to their neighbours to detect likeness. If there are neighbouring pixels that are unlike the pixel currently being evaluated then we may be on an edge. [1] work out this difference in pixel value, for a pixel x with neighbours a, b, c ,d, e, f ,g and h, using:

```
count = (a! = x) + (b! = x) + (c! = x) + (d! = x)
        + (e! = x) + (f! = x) + (g! = x) + (h! = x)
edge = count / 8.0
```

They also present an alternative method where a rule is imposed that an edge will most likely share four neighbours. The number of likely "same valued" neighbours, with respect to a bias is worked out using:

```
count = (a == x) + (b == x) + (c == x) + (d == x)
        + (e == x) + (f == x) + (g == x) + (h == x)
bias = 4
edge = 1.0 - abs(count - bias)/bias
```

As well as an edge detection algorithm [1] defines a cartoon shader. This style of shader alters light reflection so that there are sharp transitions into highly specular or shadowed areas. As well as this outline edges are usually bold and well defined. An example of this is Figure 2.3. This project is currently not interested in the light reflection of the cartoon shader but very interested in the process of outline generation. They use a camera based technique to work out the outline. This checks the angle between the shading normal and the view vector. If this angle meets some threshold then that point is an edge. This process is fairly accurate but can produce thick or sometimes missing outlines. To solve this issue [1] scale the angle by using the derivative of the angle. The RSL function filterstep() is then used to antialias the edges.

These ideas are still valid in the RIS framework as a shading grid is still accessible by some plugins. For instance, a RIS Pattern could make use of this method as it has a shading grid for the object it is assigned to. Unfortunately, this is harder at the integrator level as per object shading grids are no longer accessible.

## 3.3 Non-Photorealistic Rendering

In 2005 and 2008 researchers from Princeton and Rutgers Universities have organised delivered a SIGGRAPH class on line drawing[29][28]. This class covers differential geometry, perception of line drawings, algorithms for line drawing etc. An area of this class that is of particular interest is the algorithms for line drawing. Here they describe image and object space algorithms.

For image space they suggest two methods. Firstly, find the dot product of the normal and view direction and compare it to some threshold. If the region is darker than the threshold then it is set to the edge colour otherwise it is set to the background colour. This can produce undesirable thick lines. To reduce these they suggest loading the same width line into each mipmap level from a texture map indexed by the dot product of the normal and view direction. Other methods to control line thickness include a curvature dependent threshold. Secondly they suggest a depth based image processing method. This involves rendering the image as a set of colours that depend on depth. This data is then used in image processing, in conjunction with an edge detector, to find the changes in colour. This process makes the render simpler to perform but the image processing stage is more complex.

For object space they suggest two methods to find outline edges. The first method involves looping over each edge in the object mesh. For each edge the adjacent faces are tested. If one face is front facing and the other is back facing then this edge is a silhouette edge. See Figure 3.1.

The second method they suggest involves iterating over all the faces in the mesh. For each face the dot product between the normal and view direction is calculated for each vertex. If this equals zero then this vertex lies on the silhouette edge. If the values of this dot product are not zero for any of the vertices then the zero crossing needs to be calculated. The zero crossing is the point on two of the edges where the dot product between the normal and view direction equals zero. To find the zero crossing the vertex with the different sign needs to be isolated. Then interpolate from the isolated vertex to the other vertices, along the edges of the face. At each interpolation point the dot product is recalculated

until a point is found where it equals zero. This is the zero crossing for that edge. Once all the zero crossings have been collected a line can be drawn through all the points. This line represents the silhouette outline. See Figure 3.2.

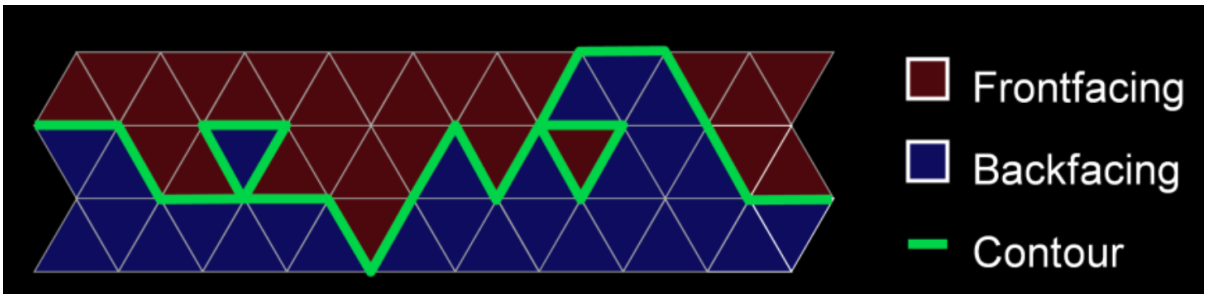



**Figure 3.1:** Description of silhouette edge detection by iterating edges in a mesh [28]

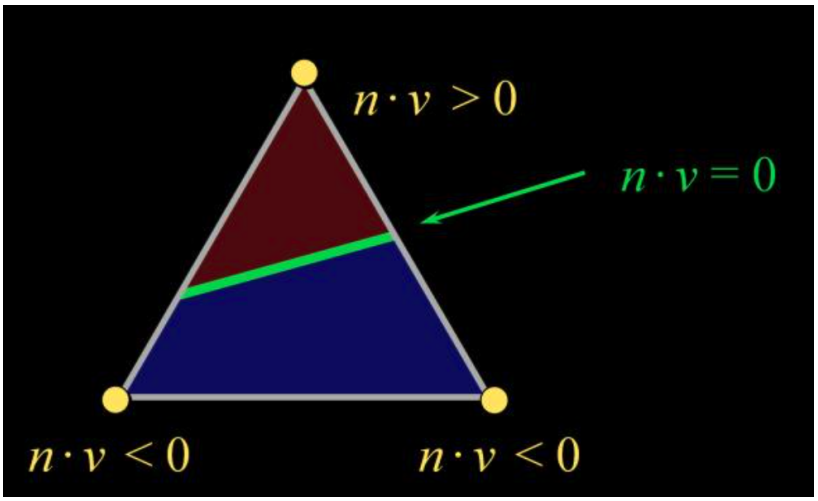



**Figure 3.2:** Description of silhouette edge detection by iterating faces in a mesh and finding the zero ccrossings[28]

As well as a complex rendering system Pixar produce world leading research. They have a few published papers that are concerned about NPR techniques. Particularly, this project is interested in [20] and [2]. In [2] toon shading techniques are explored, including edge and feature line detection. This paper seems to be designed for an artist for a quick solution for toon shading. It suggests the usage of Maya rigging tools to split object into regions. The outline of these regions is computed and a thickness map is applied in order to stylise the lines. Finally the silhouette is applied and the image is composited with colour.

[20] has more of a technical focus. They use ideas set out in the SIGGRAPH line drawing class and improve on them by ensuring contour consistency is maintained. This is done by reconstructing the mesh as a smooth surface and creating a Catmull-Clark subdivision surface of triangles that ensure contour consistency.

# Technical Background

This chapter investigates how the RIS framework operates and compares it to the REYES framework. It also discusses important NPR algorithms for edge detection and line drawing and other technology required by the project.

## 4.1 What is RenderMan?

Contrary to belief RenderMan itself is not a production renderer. RenderMan is in fact a way to tell the renderer what and how to render an image. Therefore, it is possible to develop a renderer that follows the RenderMan specification.

Classically, this can be done by using RIB files to describe scenes and RSL shaders to describe how things should look or behave. RIB files are still used with RIS but C++ plugins are used alternatively to RSL shaders.

## 4.2 RenderMan REYES

REYES is the original rendering framework provided with RenderMan[24]. It is very versatile and can render complex scenes with ease. It works by dicing objects into micropolygons. It evaluates each of the micropolygons with respect to the applied shaders and lights. REYES gets most of its versatility by employing rendering "cheats". These are not so much cheats but clever work arounds to render complex surfaces, light effects etc quickly. For example, a RSL developer can create a caustic map in a replica scene before the final render is invoked. Once the caustic map has been created it can be used in the final render to apply caustics to the scene.

An extension is the hybrid REYES architecture. This combines classic rasterisation techniques with modern day ray tracing techniques. This allows RSL developers to create

complex scenes with physically accurate reflections, refractions etc. that render quickly. Due to the physical accuracy of hybrid REYES there is no worry about debugging images with incorrect reflections from HDR maps.

## 4.3 RenderMan RIS

The RIS data flow is quite simple, see Figure 4.1. When PRMan, or a RenderMan compliant renderer, starts RIS mode it loads its plugins. Like REYES a RIS renderer will also dice the objects into micropolygons. An integrator is used for final shading calculations and other plugins, such as Bxdfs are invoked. Rays are shot into the scene and shading data is compiled, this forms the current shading context. A loop is then performed where Bxdf's and Patterns are evaluated against light vectors. From this weighted RGB lobes, PDFs and weighted vectors are generated for final shading calculations by the integrator.

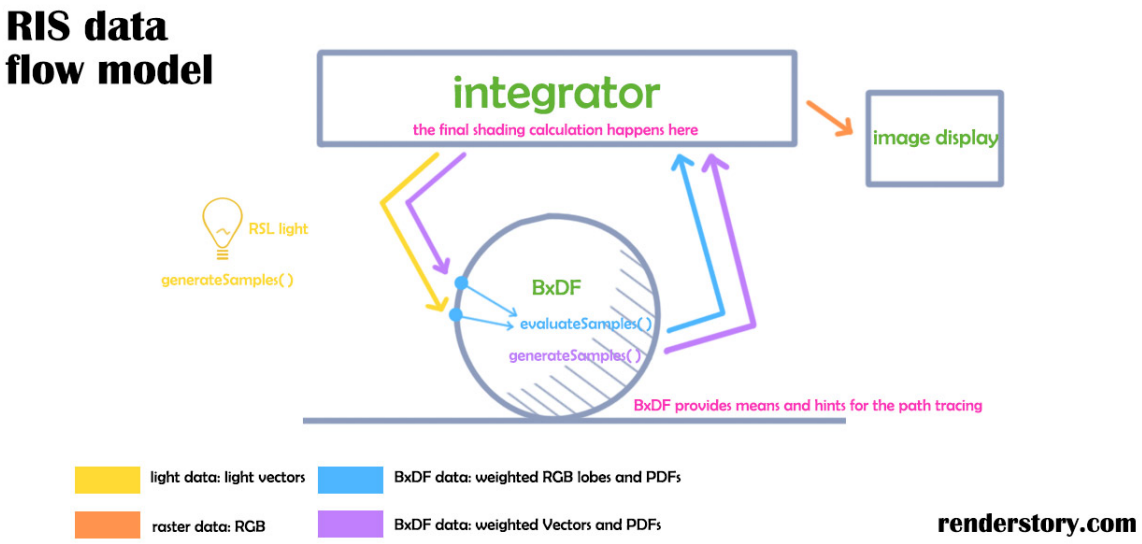



**Figure 4.1:** RenderMan RIS Data Flow Model [6]

A RenderMan RIS developer is able to make a variety of plugins. These are all constructed using the header only library that Pixar ship with RenderMan Pro Server. This type of development environment has been accessible to RenderMan developers for many years but with RIS Pixar introduced the Rix system. The Rix is composed of :

- RixBxdf.h :
  Creates a Bxdf from the shading context and the set of patterns connected to an object.

- RixBxdfLobe.h :
  A set of functions that allow Bxdf lobes to be probed and manipulated.

- RixColorUtils.h :
  Includes functions that allow colour manipulation

- RixDeepTexture.h :
  Allows the the user to load, create or modify Pixar deep textures.

- RixIntegrator.h :
  Integrator plugins must inherit from this. It allows the user access to integrator specific features, such as setting ray geometry.

- RixInterfaces.h :
  All RIS plugins must include this file. It gives access to helpful methods like RixMessages for message passing.

- RixLightFilter.h :
  Modifies the lighting contribution to a surface point after the light has been sampled.

- RixLighting.h :
  Allows the generation, emission and evaluation of light samples.

- RixLPE.h :
  Light Path Expression. Helps support LPE AOV display channels in integrators.

- RixLPEInline.h :
  Inline version of RixLPE.h.

- RixPattern.h :
  Characterises pattern for purpose of RixBxdf parameterization.

- RixProjection.h :
  Define camera and lens projections.

- RixRIB.h :
  Allows generation of RIB file.

- RixRIBParser.h :
  Used to parse a RIB file for use in standalone software

- RixRiCtl.h :
  Control for the Ri interface. This is used in the classic RenderMan C API to generate RIB files.

- RixRNG.h :
  Random number generator for use in Monte-Carlo integration and multiple-importance sampling.

- RixRNGInline.h :
  Inline private methods of RNG. This is included for use by RixInterfaces.h and should never be included directly in other applications.

- RixRNGProgressive.h :
  Progressive RNG. This is included for use by RixInterfaces.h and should never be included directly in other applications. Lookups in a pre-generated tables of progressive samples.

- RixShading.h :
  Opens up shading methods and defines the RixShadingContext. This delivers the built in shading variables to the developer.

- RixShadingBuiltin.h :
  Methods included for shading of volumes and subsurface scattering.

- RixShadingParam.h :
  RixShadingParam encapsulates the type, detail and value associated with an individual parameter of a RixShadingPlugin.

- RixShadingUtils.h :
  Supplies developer with shading utilities like RixMix and RixSmoothStep.

- RixSloInfo.h :
  Used to inspected compiled RSL shaders (.slo files)

- RixSubdEval.h :
  Used to load in and access data about a subdivision surface.

So how is RIS similar to REYES? From a shading point of view it is easy to compare. REYES controls the look of an object using a surface shader. This is controlled by RIS using a combination of Bxdf and Pattern plugins. For displacement REYES uses a displacement shader. RIS requires Pattern plugins to calculate bumps or displacement. Volume shaders are a versatile way to render volumetric data in REYES. RIS does not have this versatility, instead it relies on the integrator to cope with volumetric rendering. Both frameworks use RSL light and imager shaders.

## 4.4 Technology

Some other technology is required in order to develop a plugin. Technology used in this project would be helpful in other plugin development projects for both RIS and REYES. The hardware and software used is:

### 4.4.1 Hardware

- Late 2013 MacBook Pro. Intel i7, 8GB RAM.
- 1TB external hard drive to backup data.

### 4.4.2 Software

- Qt Creator and Sublime Text text editors and development platforms.
- TexShop LaTeXdevelopment package.
- RenderMan Pro Server 20.0
- RenderMan Studio 20.0 for Maya 2015
- it (image tool) : Pixar's image display and editing software.
- Maya 2015
- Houdini : MPlay was used in order to compile final videos.

# Design

In this chapter a design is described to create a RIS Pattern and Integrator that uses camera based techniques to calculate the edges of an object in 3D space. The system will use a camera based method in order to calculate silhouette and feature edges.

During the implementation process a RIS Pattern was created. The pattern helped to fully understand how to solve the problem in an integrator plugin and they share the same shading logic. Therefore, this section describes two designs, one for a edge detecting RIS Pattern and one for a RIS integrator.

Most of the system design is dictated by the type of RIS plugin. Each plugin has a minimum amount of standard classes to work. These classes can be extended by other classes in order to solve a problem. All of the code design follows the Pixar plugin coding standard that is evident in the examples provided with RIS. Although it could be confusing at times this standard includes all classes in a single file. It seemed this design decision was made for simplicity of release.

## 5.1 RIS Pattern

This section refers to the UML Class Diagram in Figure 5.1.

Firstly there is only one instance of PRMan, this is the RenderMan compliant renderer of choice. If the user has more than one license for PRMan then this design could be extended to include more than one instance of PRMan. However, for a single license it is impossible to have more than one instance of PRMan running at once. When PRMan evaluates a RIB file it may encounter zero RIS Patterns. This is fine, it will continue to produce the final image without a problem. It could also encounter numerous RIS Patterns throughout the scne and even have more than one pattern applied to a single object.

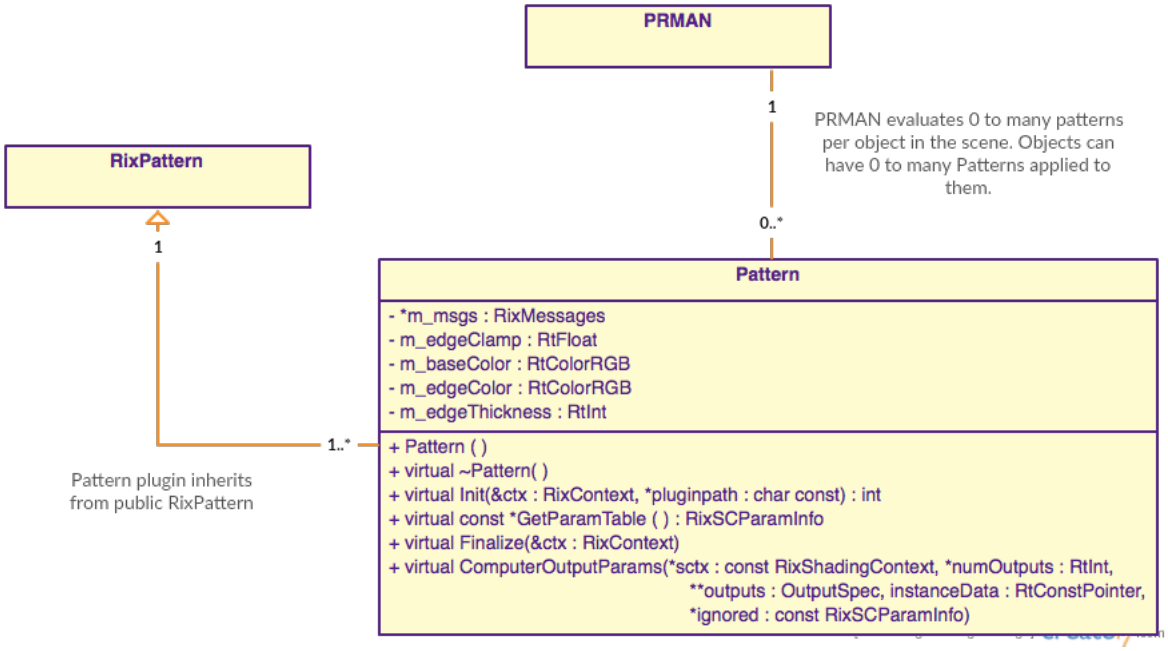



**Figure 5.1:** RIS Pattern UML Diagram

A RIS Pattern also needs to inherit from the RixPattern class. This supplies it with methods useful for pattern developers. In a single RIB file there may be multiple patterns. Each pattern will inherit from the same instance of the RixPattern class.

The pattern design has six methods. These represent the required RIS methods. There are no extra methods or classes needed in this design to solve some problem. The Pattern( ) method is the constructor of the pattern. This class sets the default values for the class attributes. The Pattern( ) method is the destructor. This is called when the class exits. The Init () method sets up all of the shared attributes of the pattern. This pattern does not require many globally shared resources. Therefore, only the Rix Messaging system is setup here. The GetParamTable( ) method returns RixSCParamInfo. Simply, it sets the output parameters ready for data and it retrieves input parameters. This data is stored in a static table that can be accessed using the EvalParam( ) method included in the shading contexts used in this plugin. Finalize( ) is a method that is invoked just before the destructor is called. In here any preliminary memory management must be performed here. For instance, if a pattern uses mutexes then Finalize( ) is where they will be deleted. Finally, the ComputeOutputParams( ) method is the heavy lifter of the plugin. This is where the pattern logic is implemented or other methods are invoked in order to perform the pattern logic. In this method, the pattern described here, deals with its user defined parameters first. It creates local pointers that point to each parameter in the table that

was created in GetParamTable( ). It then allocates memory for the OutputSpec. This is a struct that holds output data. Once it has allocated memory it retrieves built in shading variables from the shading context. Since this is a camera based edge detection method it will need to get the shading normal and view direction. Now the pattern can implement the shading logic and store the pixel output colour in the output parameter that was set in the GetParamTable( ) method.

The attributes that are defined in this design are mainly to store the data passed through by user parameters. The pointer m_msgs is an instance of the RixMessages class. This pointer is used for message passing. For instance, it can be used to throw error messages if a parameter is incorrectly defined. The remaining attributes are all reserved to store user parameter data. m_edgeClamp sets the clamping amount for the edge detection algorithm. m_baseColor sets the base colour for the object that is using the pattern. It may be necessary to set this to white if a Bxdf is assigned to the object; it is set to white by default. m_edgeColor defines the colour of the edges. To be more precise, it sets the colour that is mixed with the base colour in the final shading calculation. Finally, m_edgeThickness is a scalar that is applied to the final edge result and affects the "boldness" of the edge line. This is poorly named and is renamed m_edgeBoldness in the RIS Integrator.

The design of this edge detection RIS Pattern has informed the design of the edge detecting RIS Integrator.

## 5.2 RIS Integrator

This section refers to the UML Class Diagram in Figure 5.2.

Like the RIS Pattern, it will use one instance of our chosen RenderMan compliant renderer, PRMan. Unlike the RIS Pattern there is only one instance of the RIS Integrator. There can only be one integrator used per shot. To use different integrators within the same RIB file the Display settings must be defined per frame, i.e. within FrameBegin and FrameEnd. The declaration of an integrator tells PRMan to run in RIS mode. If no integrator is defined then PRMan will default to REYES mode. If this occurs and RIS plugins are defined throughout the scene, e.g. Bxdfs, then an erroneous image will be produced. It is also similar in that the RIS Integrator must inherit from the RixIntegrator class. This provides the developer with useful integrator methods. For instance, this class allows a developer to set ray geometry attributes, such as ray origin, using the RtRayGeometry struct present in RixIntegrator.

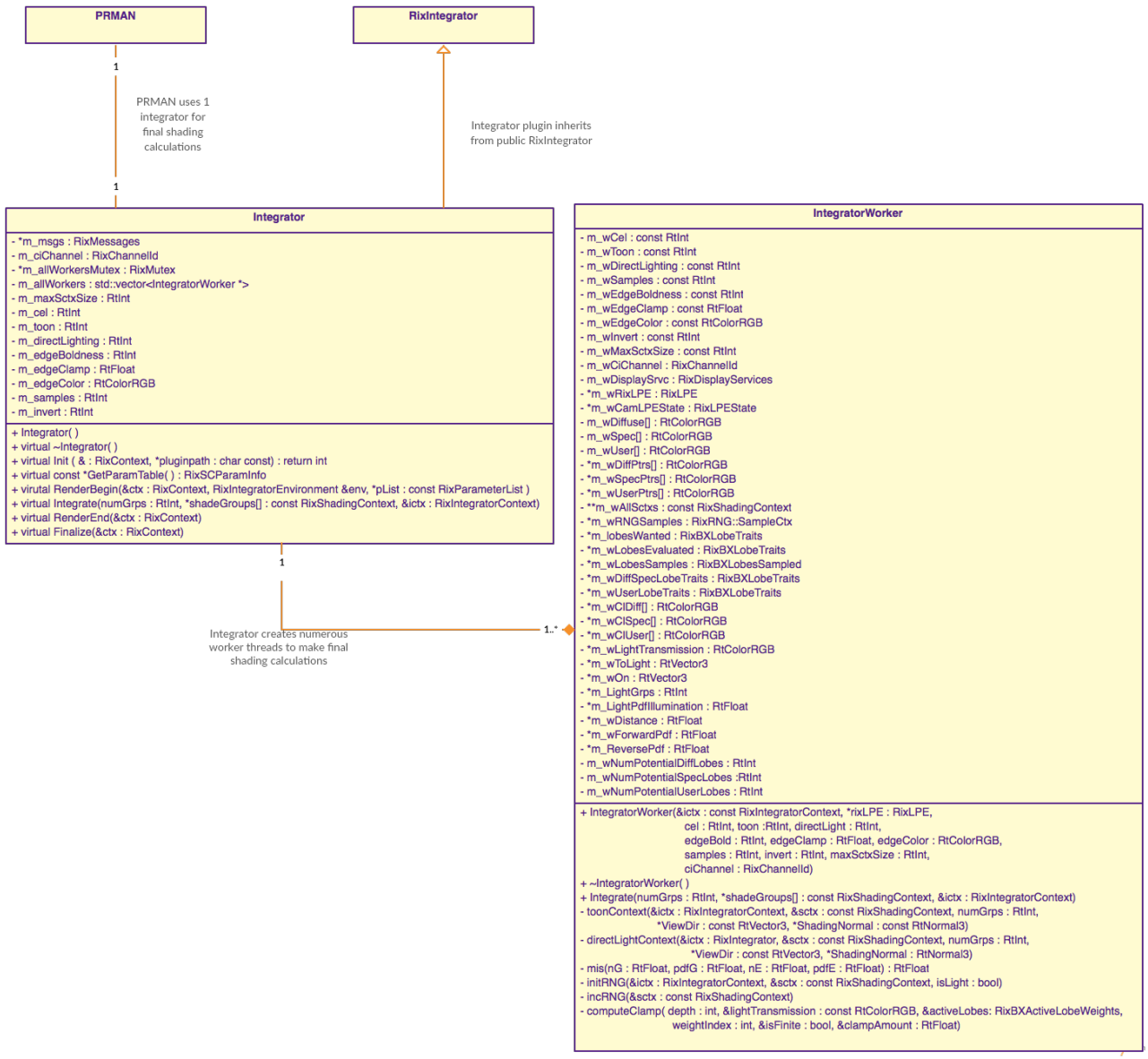



**Figure 5.2:** RIS Integrator UML Diagram

The RIS Pattern design had methods that must be defined, as is dictated by RIS. An integrator plugin also has methods that must be defined in order to operate. This design will only implement the required methods. The Integrator( ) method is the constructor of the integrator. It sets the default values for the attributes that are defined in the class. Integrator( ) is the destructor method. This is called just before the class terminates and deletes any memory dependent aspects of the code, in order to avoid memory leaks. The RIS Integrator and RIS Pattern share three methods. These are Init( ), GetParamTable( )

and Finalize( ). Init( ) sets up shared resources. In the integrator this method sets up the Rix Messaging system and mutexes that are used with worker threads. GetParamTable( ) sets output and input parameters. It is rare that an integrator will create an output parameter. In our design the integrator uses this class to retrieve user parameters set in the RIB file. Finalize acts the same as in the RIS Pattern. In the integrator Finalize will ensure the mutexes are cleaned up and avoid memory leaks.

The integrator has three unique methods, RenderBegin( ), Integrate( ) and RenderEnd( ). RenderBegin( ) sets up everything ready for the Integrate( ) method to perform the shading calculation. It retrieves the parameter data set by the user and populates a global RixIntegratorEnvironment. After RenderBegin( ) has completed the Integrate( ) method is called. This method is similar to the RIS Pattern method ComputeOutputParams( ), where it allocates memory, retrieves built in shading variables and performs the shading calculations. In this design Integrate( ) sets up worker classes, using the RixMutex locking system, and runs them if the user parameters has been set to run the workers. If not it will run a local shading algorithm to calculate the edges. The user can also set the integrator to not calculate edges. If the case is set where no worker threads and no local edge shading is instructed to operate then a black image will be created. After Integrate has finished the final shading calculations RenderEnd( ) is called. This method performs preliminary deletion, shuts down the render and calls Finalize( ). In this design each worker class is deleted in RenderEnd( ).

The attributes defined in this class represent shared resources and user defined parameters. Like the RIS Pattern design the pointer m_msgs is an instance of the RixMessages class. This is used for message passing. Specifically, it is used to notify the user of a badly constructed parameter value. m_ciChannel sets up a colour channel id. The pointer m_allWorkersMutex points to an instance of the RixMutex class. This mutex can be used to lock the workers vector when adding workers. This will ensure vector coherence. The vector m_allWorkers stores pointers to each worker class. This isn't used in the shading calculations but makes deletion of each worker simple. The attribute m_maxSctxSize is an integer value that stores the maximum size of the shading context. This value is assigned in RenderBegin( ) and is retrieved from the RixIntegratorEnvironment class.

The remaining attributes are all user definable parameters. m_cel indicates that the edges should be drawn. m_toon indicates that toon lighting should be employed. m_directLighting indicates that direct lighting should be calculated. m_edgeBoldness, m_edgeClamp and m_edgeColor are the same as in the RIS Pattern design, where m_edgeBoldness is equal to m_edgeThickness. m_samples is an integer that indicates how many samples to use in direct lighting calculations. Finally, m_invert tells the integrator to invert every pixel colour value.

The worker class is more complex. This is where lighting and final edge detection calculations are performed. IntegratorWorker( ) is the constructor of the worker. It gets a lot of data passed from the initial integrator class. Most of this data is user defined parameters. The constructor assigns these values to class attributes for use throughout all methods defined in the class. As well as this it uses the amount of samples and the max shading context size calculated in the parent class(m_maxSctxSize) in order to assign memory for each of the pointer attributes that require it. Integrator( ) is called when the worker terminates. This method deletes each of the pointers that was assigned memory by the constructor.

The workhorse of the worker class is the Integrate( ) method. This is initiated from the parent integrator class. It makes use of the other methods in the class and initiates the lighting and shading calculations. If the user has selected toon lighting then the method toonContext( ) will be called by Integrate. If the user has selected direct lighting then directLightContext( ) will be called by Integrate. Each of these methods also include the edge shading algorithm to combine edge shading if the user selects that style. The remaining methods are helper method for the direct and toon lighting calculations.

# Implementation

The purpose of this project is to investigate the RenderMan RIS API and generate a working knowledge of the new rendering framework. A further goal is to create a RIS Integrator that is capable of detecting edges. Since the integrator is concerned about NPR techniques then visual realism is not a concern. However, edge detection can be used for diagrams in technical manuals and educational textbooks where detail, and sometimes realism, is desirable. See Figure 2.2. Therefore, realism can be useful and can not be useful depending on the context that the system is being used. This chapter explains the implementation process.

## 6.1 Implementation Process

At the start of the implementation the PxrDirectLighting and PxrVisualizer integrator examples, supplied with RenderMan 20, was used as a guideline. It was from here that the standard RIS Integrator layout and Pixar coding standard was discovered. Once a simple constant colour integrator was constructed derivatives in u,v space were investigated. RenderMan supplies the developer with some built in shading variables, via the header file RixShading.h, these are described in Chapter 4. It was suggested that the built in variables could be used to calculate the rate of change in the normal vectors at each shading point[4]. Unfortunately, RIS does not have a versatile function that allows the calculation of arbitrary derivates. This is however, available in RSL, in the form of the Du( ) and Dv( ) functions. It does however supply dPdu and dPdv. These are precomputed and give analytical tangent at the point P. Du(P) and Dv(P) perform a calculation on neighbouring point information for and approximation of the same value as dPdu and dPdv[22]. Using this data inside a derivative step algorithm, see code 6.1, edges were detected but a lot of unwanted noise was present in the image, see Figure 6.1.

---

`Code 6.1`

```
for(j = 0; j < numsteps; j++)
{
        add dPdu to Vec3
        add dPdv to Vec3
}
Divde Vec3's by numsteps
Dot Product of Vec3's
Add Dot Product results
Clamp between 0 and 1
Smoothstep result
Assign to pixel color
```

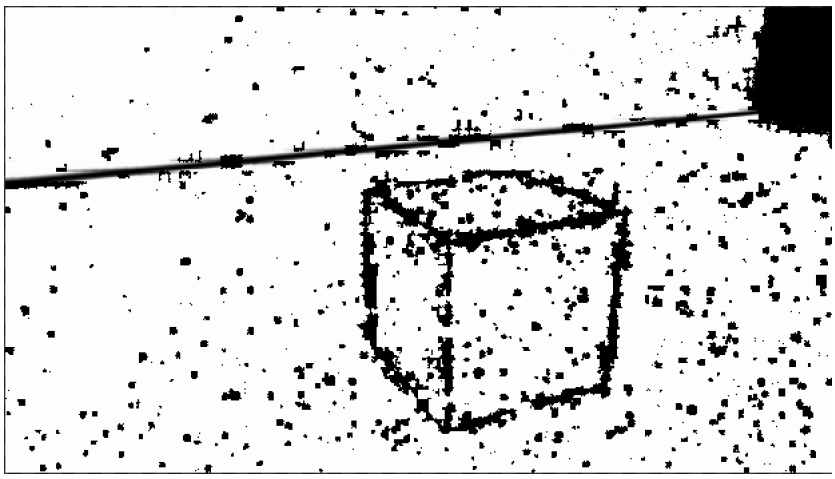


**Figure 6.1:** Erroneous Image using dPdu and dPdv in derivate step algorithm.

Having access to dPdu and dPdv provided a good method of point comparison, however, normal comparison was required. Extending RIS to perform arbitrary normal derivative calculations was attempted. Usage of the library Adept[32] was investigated in order to calculate the Jacobian of the normal vectors. This worked well in single thread applications but, in the integrator, it ran into very severe memory issues. Using Adept was abandoned.

Another option was a mesh based approach. Initially it seemed that mesh data could be accessed via the RixSubdEval.h header file. This gives you access to vertex edge and face data of a Catmull-Clark Subdivision Surface. Upon further investigation the mesh data is not available at the integrator stage. This method is reserved for the creation of librix tools. librix is an interface for tool creation from outside of the RenderMan interface.

Other mesh based methods were also explored. Reconstruction of the mesh from shading data was attempted. This consisted of finding shading points that were near the vertices of the underlying geometry. Once these vertices were discovered they were stored in a std::vector. Whilst iterating over the vertex points a triangular mesh was constructed and methods describe in the 2008 SIGGRAPH Line Drawing class[28] were employed. This method stored a lot of local data and enforced the integrator to perform a lot of calculations per shading point. This completely stalled the rendering process and often forced PRMan to crash.

At this stage an investigation into RIS Patterns began. The algorithms presented in the 2008 SIGGRAPH Line Drawing class[28] states that edges are at points where the dot product between the normal and the view direction is 0. By using this logic a shading algorithm was produced, see code 6.2. This works out the dot product and compares it to some threshold to detect edges as the first image space algorithm describes in [28]. See Figure 6.2.

Code 6.2

```
dot(viewDir, Norma)
Work out threshold
if(dot < threshold)
        set dot to edge colour
else
        set edge to background colour or white
edge = Smoothstep(threshold, 1, dot)
Mix(1,0, edge)
```

The logic was placed within the barebones constant shading integrator and early results were successful. After this user parameters were added allowing them to control aspects such as edge colour and threshold amount. The next step was the addition of direct lighting was added following the PxrDirectLighting example. See Firgure 6.3.

## 6.2 Final System

### 6.2.1 Usage

The RIS integrator and RIS Pattern can be used via RIB files. In a RIB file and integrator is defined in the display, or camera, settings section. It is included using the following command:

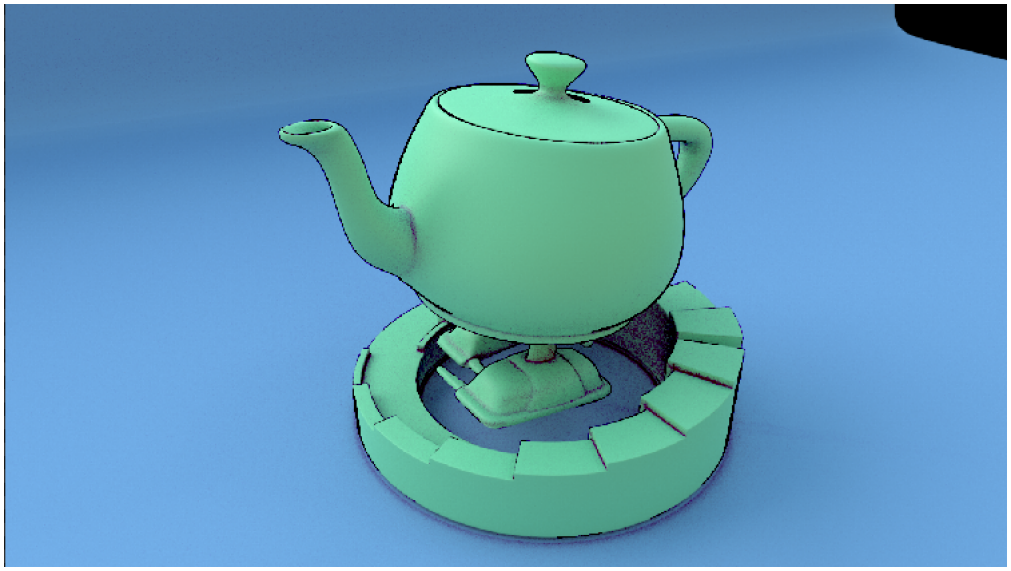


**Figure 6.2:** Edge Detection using RIS Pattern and PxrPathTracer

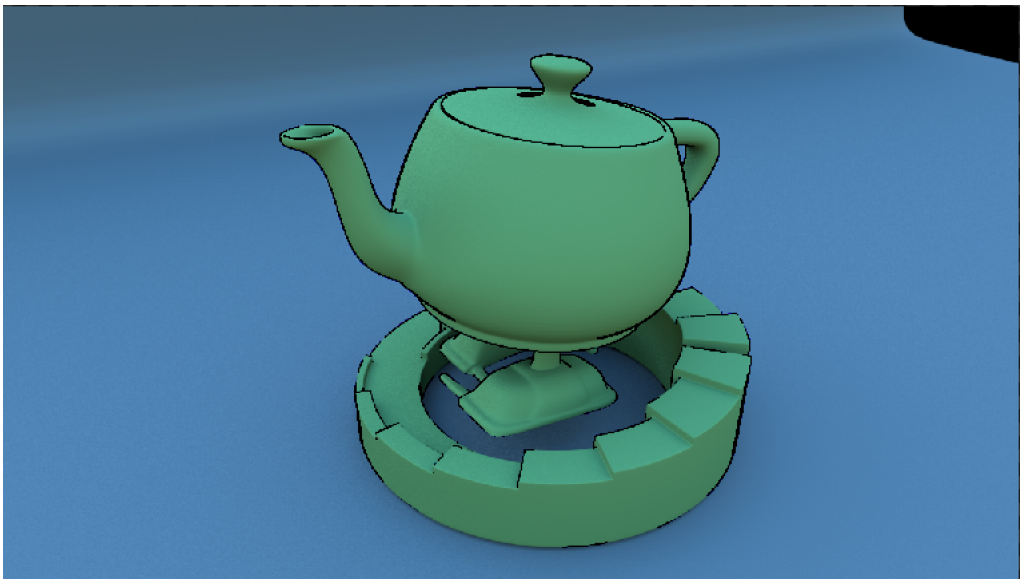


**Figure 6.3:** Edge Detection using RIS Integrator and custom direct lighting

```
Integrator "name of integrator" "id" "parameter list"
```

To ensure that PRMan can find the integrator the compiled .so must be in the directory $RMANTREE/lib/RIS/integrator.

Many RIS Patterns can be defined in a single RIB file. These are applied directly to the object and therefore must be within the WorldBegin and WorldEnd tags. The following command can be used to add a RIS Pattern to an object in a RIB file.

```
Pattern "name of pattern" "id" "parameter list"
```

The compiled pattern must also be placed in a specific directory in order for PRMan to find it. This is : $RMANTREE/lib/RIS/pattern.

Both plugins can also be loaded into Maya for use with RenderMan Studio. Lollipop Shaders have a good tutorial on how to do this and have recently created a Maya module to simplify the process. The tutorials can be found at [13][14]. The Maya module can be downloaded with a trial of their ambient occlusion integrator at [17]

### 6.2.2 Issues

As previously explained this project has had many issues. The final iteration did not have many issues and performed well when testing. The biggest issue that was encountered was a memory issue. This would not affect the final image but the software would not exit cleanly and probably resulted in memory leaks. The error was caused by a pointer being freed that was never allocated. This was due to worker classes being created with no data and was caused by a malformed if statement. This was simply fixed but difficult to locate.

The camera based method of finding the edge is not always reliable. Sometimes large outlines are produced where faces of the mesh can be considered completely shaded black. This degrades the final image as it reduces the amount of detail. This method can also miss detail if the clamping threshold is set too low. This could be detrimental if the integrator was being used to create accurate line drawings for technical manuals or educational texts

### 6.2.3 Efficiency and Performance

The software was tested on a number of complex objects, these are included in the appendices and the images supplied with this document. Each time, the software performed well managing to identify and draw many of the silhouette and feature edges and create some nice images. For instance Figure 6.4.

An image like Figure 6.4 would take approximately 50 seconds to fully render, this is at a resolution of 1280x720. This is quick considering the detail in the model. If multiple frames are required and lazy render is turned on then the first frame will take approximately 50

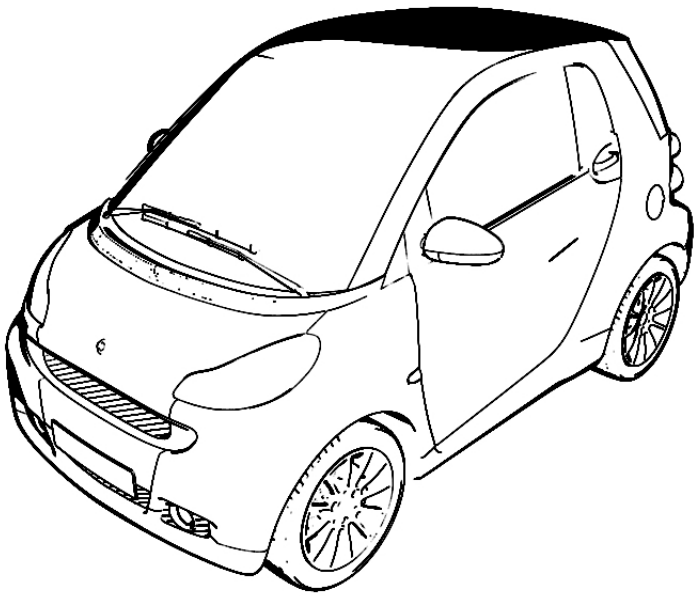


**Figure 6.4:** RIS Integrator of Smart Car edges

seconds and the remaining frames will take approximately 10 seconds to fully render.

For images that require direct lighting, like Figure 6.3, a frame at the resolution 1280x720 will take about 30 minutes to fully render. If multiple frames are required and lazy render is turn on then the first frame will take approximately 30 minutes and the remaining frames take approximately 15 minutes to fully render.

### 6.2.4 System Review

The development of an edge detection RIS Integrator has been a success. It successfully finds edges and scales well for complex objects. As part of the testing process the integrator was applied to a shot of an environment from a short film "Cygnus"[18]. It managed the complex scene well and produced very satisfying results, see Figure 6.5.

It is an efficient and accurate tool that can be used in a number of ways. It is possible to create a whole animated movie with it, create accurate line drawings for technical

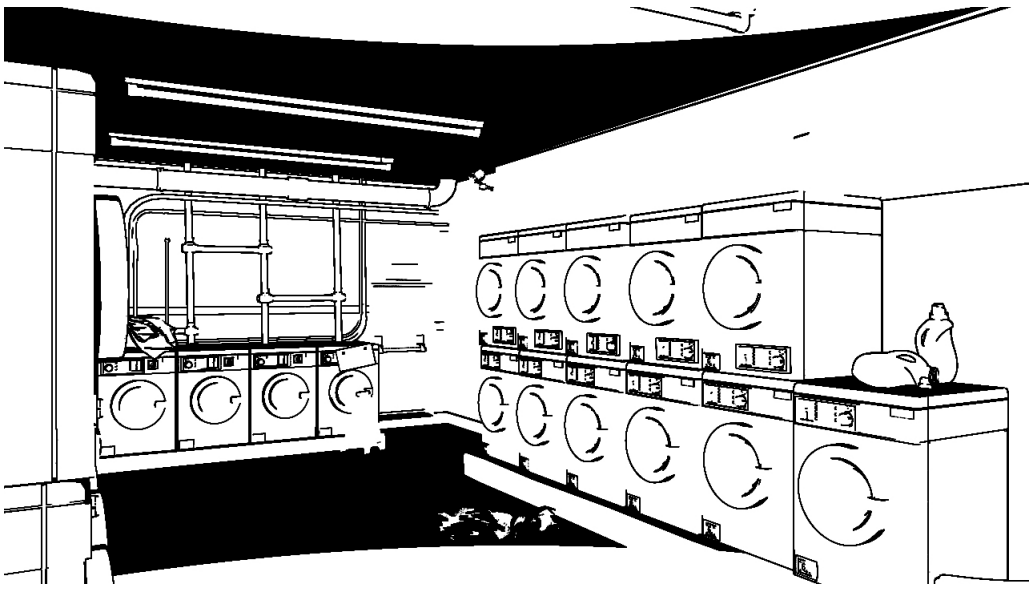


**Figure 6.5:** RIS Integrator applied to a scene from "Cygnus" [18]

manuals and educational textbooks, create artistic stills that can be altered using the user parameters and other plugins, such as Bxdfs.

# Conclusion

The focus of this project has been very specific but also broad. It aimed to investigate the new rendering framework provided by RenderMan and NPR techniques for edge detection and line drawing. A key objective of the project was to display the knowledge gained, by investigating into the previously mentioned areas, by developing a RIS plugin that achieves an established style described in the NPR research.

Although the software is fast and efficient is does not meet all goals. The camera based method employed in the integrator can sometimes miss or obfuscate detail. The original design called for toon shading model, which was not implemented, due to time limitations. Not all of the goals were achieved but many were successful. For instance, this project set out to develop a single RIS plugin. It has surpassed this and harboured the development of two RIS plugins. It has allowed an investigation into the new rendering framework provided by RenderMan and NPR techniques. It has implemented silhouette and feature edge detection, and direct lighting. Both of which were requirements in the design.

Finally the integrator produces high quality fast NPR edge renders which have been praised by industry professionals. Therefore, the project has been extremely successful and has the potential to be extended to include the features it lacks.

## 7.1 Feedback from industry

Most of the response from industry has been via the RenderMan Forum. Here are a few quotes from the forum (Where their real name is unknown their forum username has been used):

- bsavery - Pixar RenderMan Team : “Looks good!”
- tonytrout : “Good job :) I saw a great short done in this style, but cel shaded.”

- MWY3510 : “Great job. I think that the edging is fantastic.”
- Christos Obretenov - Lollipop Shaders : “Looking great, Martin, good to see your progress! I like the black/white one with the edges.”
- Wayne Wooten - Pixar RenderMan Team : “Nice work!”

## 7.2 Future Work

The RIS integrator is very successful, however, it can be improved and extended. As mentioned before the camera based method can be unreliable and fail to render objects in high detail. To overcome this problem a mesh based solution could be employed or a the derivative step algorithm could be perfected and included. This could be done by using the local curvature to set a clamping threshold. The local curvature is supplied to the developer by RenderMan RIS as a built in shading variable.

To improve the final look more lighting models can be included. The original design called for a toon lighting model and a direct lighting model. The direct lighting model is the only one to be implemented in the final integrator. It could be extended by making the toon lighting model available to users and could be extended even further by including a path tracing lighting model.

To make the tool more diverse more NPR styles could be added to the integrator. This is a nice idea but it would be a bad design choice. As more styles are added to the integrator the harder it needs to work. Therefore, the efficiency of the software would be compromised. A similar effect has been witnessed during the implementation process. An attempt to reconstruct a mesh from shading points resulted in the integrator stalling and PRMan crashing. A better approach to this problem would be to create a suite of integrators that each create a single NPR style well.

# Bibliography


[1] A. A. Apodaca and L. Gritz. *Advanced RenderMan: Creating CGI for Motion Pictures*. San Diego, CA: Morgan Kaufmann Publishers, 1999.

[2] A. Harvill, "Effective Toon-Style Rendering Control Using Scalar Fields" Pixar Technical Memo #07-08, May 2007.

[3] B. Burley. "*Physically-Based Shading at Disney*". Disney Research Library[Online]. Available: https://disney-animation.s3.amazonaws.com/library/s2012_pbs_disney_brdf_notes_v2.pdf. [Accessed: Jul. 8, 2015]

[4] Christos Obretenov . *Finding edges with the derivative step algorithm*. (By Communication). 2015.

[5] Disney Animation Studios. "*Se.Expr()*". Disney Animation Studios[Online]. Available: http://www.disneyanimation.com/technology/seexpr.html. [Accessed: Aug. 23, 2015].

[6] E. Gabbassoff. "*RIS Data Flow*". renderstory.com[Online]. Available: http://renderstory.com/what-bxdfs-and-closures-are/. [Accessed: Aug. 17, 2015]

[7] flickr. flickr.com. Available: https://c1.staticflickr.com/1/150/429949624_3e198c96e9.jpg. [Accessed: Aug. 14, 2015]

[8] Director : Hayao Miyazaki, Ponyo, 2008. Japan: Studio Ghibli and Disney.

[9] I. Stephenson. *Essential RenderMan*. London, UK: Springer, 2007.

[10] Ian Cromley. *Shading at ILM*. (By Communication). 2015.

[11] Interview with MPC Shader Writers. *Shading at MPC*. (By Communication). 2015.

[12] J. Esteve. "*Stippling and Blue Noise*". Jose's Sketchbook[Online]. Available:http://www.joesfer.com/?p=108. [Accessed: Aug. 14, 2015]

[13] Lollipop Shaders. "*Custom Integrator & BxDF Install Tutorial for RenderMan19 RIS*". YouTube[Online]. Available: https://www.youtube.com/watch?v=OY9lIrigpB4. [Accessed: Jul. 16, 2015]

[14] Lollipop Shaders. "*Occlusion Integrator: Maya Module Install*". YouTube[Online]. Available: https://www.youtube.com/watch?v=RTDJsU1RnGQ. [Accessed: Aug. 7, 2015]

[15] Lollipop Shaders. "*Our Story*". Lollipop Shaders[Online]. Available: http://lollipopshaders.com/aboutus/. [Accessed: Aug. 24, 2015]

[16] Lollipop Shaders. "*RIS Integrator Occlusion RM20*. YouTube[Online]. Available: https://www.youtube.com/watch?v=JadqWmIbsnQ. [Accessed: Aug. 20, 2015]

[17] Lollipop Shaders. "*RIS Integrator*". Lollipop Shaders[Online]. Available: http://lollipopshaders.myshopify.com/products/ris-integrator. [Accessed: Jul. 1, 2015]

[18] M.Davies, et al. "*Group Project - Cygnus"*. YouTube[Online]. Available: https://www.youtube.com/watch?v=_LoIkpaVm7I. [Accessed: Mar. 31, 2015]

[19] M. Fussell. "*Pen and Ink Drawing Techniques*". TheVirtualInstructor.com[Online]. Available: http://thevirtualinstructor.com/penandink.html. [Accessed: Jun. 30, 2015]

[20] P. Benard, et al., "Computing Smooth Surface Contours with Accurate Topology " ACM Transactions on Graphics, vol. 33, no. 19, pp. 1-21, issue 2, March 2014

[21] P. Seed. "*Aronde 1300 (1956-1958)*". Phil Seed's Virtual Car Museum[Online]. Available: http://www.philseed.com/simaronde1300.html. [Accessed: Aug. 14, 2015]

[22] Pixar Animation Studios. "*What's the difference between dPdu and Du(P)?*". Pixar Animation Studios[Online]. Available: http://renderman.pixar.com/view/DP21880. [Accessed: Jul. 13, 2015]

[23] R. Cortes and S. Raghavachary. *The RenderMan Shading Language Guide*. USA: Thomson Course Technology, 2008.

[24] R. L. Cook, et al. "The Reyes Image Rendering Architecture". In Proceedings SIGGRAPH 1987, New York City, NY: ACM, 1987. pp.95-102.

[25] S. R. Marschner, et al., Light Scattering from Human Hair Fibers." In Proceedings SIGGRAPH 2003, New York City, NY: ACM, 2003. pp. 780-791.

[26] Sony Pictures Imageworks. "*Open Shading Language*". GitHub[Online]. Available: https://github.com/imageworks/OpenShadingLanguage/. [Accessed: Jul. 4, 2015]

[27] Director : Steve Martino, The Peanuts Movie, 2015. USA: Twentieth Century Fox and Blue Sky Studios

[28] S. Rusinkiewicz, et al., "*SIGGRAPH 2008 Class: Line Drawings from 3D Models*". Princeton University[Online]. Available: http://gfx.cs.princeton.edu/proj/sg08lines/. [Accessed: Jul. 22, 2015]

[29] S. Rusinkiewicz, et al., "*SIGGRAPH 2005 Course 7: Line Drawings from 3D Models*". Princeton University[Online]. Available: http://gfx.cs.princeton.edu/proj/sg05lines/. [Accessed: Jul. 22, 2015]

[30] S. Upstill. *The RenderMan Companion: A Programmer's Guide to Realistic Computer Graphics*. Boston, MA: Addison-Wesley, 1990.

[31] T. Strothotte and S. Schlechtweg, *Non-photorealistic Computer Graphics: Modeling, Rendering and Animation* San Francisco, CA : Morgan Kaufmann Publishers, 2002.

[32] The Clouds Group, "*Adept: A fast automatic differentiation library for C++*". University of Reading[Online]. Available: http://www.met.reading.ac.uk/clouds/adept/. [Accessed: Aug. 16, 2015]

# Appendices

# Final Images and Videos

All high resolution final images have been supplied with this document.

Four videos have been supplied with this document. These are turntables of objects rendered using the RIS Integrator described in this document. They can also be found at:

`Smart Car : https://youtu.be/nGV3wkmZ4Q8`

`Walking Teapot : https://youtu.be/uEd2qk7_P6I`

`Direct Lighting : https://youtu.be/XAJh86dv50E`

`Inverted Pixels : https://youtu.be/Bw41nXttoLs`

`Blue Edges : https://youtu.be/y74i4iSP2n4`

# RenderMan Forum

The RenderMan forum was extensively used for feedback and support. A single thread follows most of the problems and successes that occurred during this project. Viewing the thread will give the reader an insight on how the project progressed during the allotted time. The thread can be accessed by going to the following webpage (this requires a full RenderMan forum account and can not be accessed with a non-commercial account):

`https://renderman.pixar.com/forum/showthread.php?s=&threadid=28261`