Report Contents:

1. Introduction

2. Previous Work & Research

3. Anatomy of Human Eye
   3.1 Iris
   3.2 Sclera
   3.3 Conjunctiva
   3.4 Cornea
   3.5 Limbus
   3.6 Tear Duct

4. Renderman Shader
   4.1 Eye.sl
      4.1.1 Function: Get Intensity
      4.1.2 Function: Get to position
   4.2 HeaderFiles
      4.2.1 Sclera.h
      4.2.2 Viens.h
      4.2.3 Iris.h
   4.3 TearDuct
      4.3.1 Eye&Duct.sl
      4.3.2 Duct.h
   4.4 Eye.rib
   4.5 Pupil.sh

5. Mel Scripts
   5.1 Intensity.mel
   5.2 Look Around.mel
   5.3 Blink.mel

6. Maya & Slim modelling, animation & rendering

7. Shader Development & Problems encountered
   7.1 Mapping the shader correctly onto the geometry
   7.2 Generating the Iris pattern
   7.3 Generating Sclera & Vein patterns
   7.4 Tear Duct
   7.5 Reflections & Environment Maps
   7.6 Generating the size of the pupil
   7.7 Anti-Aliasing
1. Introduction, Aims & Objectives

The human eye is incredibly complex and detailed. It consists of several different layers, all made up from different substances with each of these substances contributing to the overall appearance of an eye. Eyes are fundamental to a character’s expression, without realistic eyes the character can seem lifeless. Bearing this in mind, the aim of this project was to simulate a realistic human eye using the Renderman Shading Language (RSL).

The reason for choosing to do this was based on the idea of giving greater control to the animator over the appearance and behaviour of the eyes and by creating it with Renderman the eye shader would be entirely procedural, with parameters that can be
changed by the user, such as the colour of the iris, or the degree of veins on the surface. The reflections from the environment using an environment map, which is also passed in as a parameter, so when animating a character, the user can have greater control over the desired reflections in the eyes.

Another aim of this project was to have the eye responding to the amount of light in a scene – with the pupil changing size depending on the intensity of the light on the surface. This feature of the shader is not complete yet – it does work however the results are not always satisfactory. This is discussed in section 7.6. At the moment the pupil size can be controlled by the user, through the \textit{pupilsize} parameter.

The project also included generating some scripts to automate the animation of the eyeball and the blinking. These scripts are written in MEL scripting language for Maya.

---

2. \textit{Previous Work & Research}

Previous work done into writing a Renderman shader to simulate human eyes was done by L. Gritz\cite{7}. He seemed to be first to use the technique of using the centre of the pupil as the spheres “North Pole”, where the \( t \) value is 1.

In 2003, Dan Smollan completed his masters thesis also on the simulation of a human eye\cite{4}. The approach I have taken is slightly different, in that the patterns are all generated by different methods, and also the parameters going into the surface shader are different.

There has been lots of research into new methods for implementing sub-surface scattering in Renderman\cite{10}, which would ideally be the best technique to use when modelling a surface like an eyeball, however I choose to try and implement methods that were not as complex with reasonably short render times.

There are also many tutorials available on line to create eyes \cite{5,8,9} The one I found best and used for the maya model was Andrew Whitehurts “The Eyeball tutorial” \cite{5} which uses two nurbs spheres with the pole of the inside one pulled in to create more depth for the iris and pupil, and the pole of the outer one pushed out to create a lens effect.

---

3. \textit{Anatomy of Human Eye}

Before trying to simulate an eye, it was necessary to first examine the composition and materials of the eye. The eye consists of several different sections\cite{11}: 

---
The Iris:  The Iris is the central section of the eye that gives an eye its colour. It is made up of bands of connective tissue that give the iris its appearance.

The Sclera:  This is the white of the eye. It varies in colour from blue-ish to beige, and can be different sizes. It is composed of 3 layers with the outer layer which contains the small blood vessels we can see.

The Conjunctiva:  This is thin membrane that is attached to the front of the sclera. It is translucent and highly reflective so this is where the reflective nature of the eye comes from and what gives an eye its moist appearance.

The Cornea:  This is an ellipse shape, that’s highly refractive and transparent. It is attached to the sclera at the limbus and this is where it is thickest. It has a refractive index of 1.38.

Limbus:  This is where the sclera and cornea join. This is where you can see the change in whiteness between the sclera and the iris.

Tear Duct:  This is a small area of skin that contains the tear glands. It is moist and refractive with a few slender hairs. It is red/pink and if you look closely you can see tiny blood vessels.

4.  Renderman Shader
The shader for the eye has been broken down into several parts. *Eye.sl* is the main body of the shader, which generates *Eye.slo*, the shader object file. This .slo file is then imported into Slim to attach the shader to a surface in Maya. The functions written for generating each section of the eye are included as .h files.

There are also several .h files included that are taken from the “Advanced Renderman”, by Larry Gritz[1]. The functions used from there are different noise functions, and some filtered versions of Renderman functions such as *smoothstep* and *abs*.

### 4.1 *Eye.sl*

The main body of the shader is contained in the file “*Eye.sl*”. It generates an *Eye.slo* file, which is the name of the surface shader. This is referenced in the rib file with the geometry to attach it to.

This surface shader is passed in several parameters. These include:
- All the specific lighting co-efficients required to calculate the effect of light on the surface, such as $K_d$ and $K_s$ (for diffuse and specular light);
- *IrisColor*(1-6) → to control the colours going into the iris pattern;
- *VeinVal*, a parameter which controls the amount of veins on the eye;
- *EnvName*, a string to hold the name of the environment map for reflection calculations:

```c
Surface Eye(float Ka= 1; float Ks = 1; float Kd = 1; float Kr = 0.5;
    float roughness = 0.1; float specularcolor = 1;
    color IrisColor1 = color "rgb" (0.478, 0.353, 0.200);
    color IrisColor2 = color "rgb" (0.780, 0.404, 0.089);
    color IrisColor3 = color "rgb" (0.384, 0.271, 0.114);
    color IrisColor4 = color "rgb" (0.337, 0.486, 0.314);
    color IrisColor5 = color "rgb" (0.827, 0.918, 0.655);
    color IrisColor6 = color "rgb" (0.384, 0.271, 0.114);
    float VeinVal = 0.73; string EnvName = "hill1.tx";)
```

The colours to be used in the shader are then declared. Following this a call to the sclera function is made to set up the colour to be used for shading the sclera:

```c
scleraColor = sclera(basecolor, white);
```

Next, the surface calculations are carried out.
- The shader is transformed to the appropriate shading space.
- *ss* and *tt* are declared (using the built in texture co-ordinates s and t) to be the texture co-ordinates used when shading the current point on the surface. *tt* is set to 1-*t* so we can work from the bottom to the top of the surface.

```c
point pp = transform("shader", P);
float tt = 1-t;
float ss = s;
```

- The surface normal is normalized and ensured it is facing forwards
- Vector V is calculated – the direction back towards the observer
- Vector R is calculated which is the direction of the reflected ray. It is used in ray tracing for environment map reflection calculations.

```cpp
normal Nf = faceforward (normalize(N), I);
vector V = normalize(I);
vector R = normalize(reflect(I, N));
```

Once the surface calculations are made, the variables to calculate the distances from the pupil to the point we are shading and the light are set up. The point we are current shading is held in the variable `here`:

```cpp
point PupilCentre = point(1, 0, 0);
point here = point(ss, tt, 0);
float dist, fuzz2 = 0.015, fuzz3 = 0.005;
```

Following this a call is made to the function `ProceduralIris` which will return the iris pattern as a colour value. It takes in the 6 colours that were passed into the shader by the user. The first four are used to generate the top colours, and the last two are to generate the noise underneath.

```cpp
Ct = ProceduralIris(Nf, IrisColor1, IrisColor2, IrisColor3, IrisColor4, IrisColor5, IrisColor6);
```

The Veins function is called next to generate the pattern for the veins on the surface. It takes three parameters, the sclera colour (for underneath) the vein colour, and the vein value to determine the amount of veins.

```cpp
veins = Veins(scleraColor, TDpink, VeinVal);
```

Next, if we are using the intensity of the light in the scene to generate the size of our pupil, the calculations are carried out here.

To determine the size of the pupil the intensity at the current surface point is calculated. Then the distance from this point to the centre of the pupil is calculated and the intensity is scaled by this length. This new intensity value becomes the intensity at the pupil centre and a selection of if statements are used to determine the size of the pupil. At present this will only work if there is one light being used to determine the pupil size, as the `GetToPos` function returns one position.

Get the intensity at the point we are shading by calling the `GetIntensity` function. This function goes through an illuminance loop to get the intensity value at this point from all the lights in the scene.

```cpp
intensity = GetIntensity(Nf);
```

The position the light is being directed to is then calculated in the function `GetToPos`, which goes through another illuminance loop and gets the position the light is directed at.

```cpp
LightPosition = GetToPos(Nf);
```
Next calculate the distance from this point to the pupil and use this distance to scale up the intensity at the pupil to then use to determine what size the pupil should be.

```cpp
float dist = distance(PupilCentre, here);
float lightDist = distance(LightPosition, here);
float pupilDist = distance(PupilCentre, here);
```

We need to determine if we are closer to the light than the pupil. If we are the intensity is scaled up by the length to the pupil, but if the light is closer the intensity is doubled then scaled up by the length because we are in a shaded section.

```cpp
if (pupilDist < lightDist)
    { 
        dist=dist*1.5;
    }
```

```cpp
PupilIntensity = intensity*dist;
```

The final size of the pupil is determined by a series of if statements – if the intensity value is high, then the pupil is small and if it is low then the pupil is large.

```cpp
if (PupilIntensity >=0.1)
    {
        pupilsize = 0.08;
    }
else if (PupilIntensity <0.1 & & PupilIntensity >= 0.07)
    {
        pupilsize = 0.06;
    }
```

Use the `pupilsize` to get a min and max pupil size.

```cpp
float pupilmin = pupilsize-0.001;
float pupilmax = pupilsize+0.001;
```

Next, shader goes through the series of Mix statements to layer up all the various sections of the eye from the pupil to the sclera, with each mix function blending a different layer.

**Sclera → OuterRim, OuterRim → InnerRim, InnerRim → IrisColours, IrisColours → InnerIrisRim, InnerIrisRim → Pupil.**

```cpp
OuterBrown = mix(OuterBrown, brown1, fBm(pp, filterwidthp(pp), octaves, lacunarity, gain));
iris3 = mix(OuterBrown, veins, smoothstep(0.11, 0.12-fuzz2, tt) );
outerIris = mix(brown1, green1, fBm(pp, filterwidthp(pp), octaves, lacunarity, gain));
iris2 =mix(outerIris, iris3, smoothstep((0.11-fuzz), (0.11+fuzz),tt) );
Iris1 = mix(Ct, iris2, smoothstep((pupilsize-fuzz2), (0.11+fuzz),tt) );
```
\texttt{InnerIris = mix(black, Iris1, smoothstep(0.027, (pupilsize+fuzz2), tt) );}
\texttt{Ct = mix(black, InnerIris, smoothstep(0, 0.027, tt) );}

Next we call the function to calculate the reflection on the eye, using the environment map that was passed in and this is added to the final lighting calculations.
\texttt{color Reflection = MyReflection(EnvName, Nf);}

There is also a second reflection function included which will calculate accurate reflections using ray tracing
\texttt{C2 = Environment (EnvName, envspace, envrad, P, R, blur, alpha);} ;

Finally the overall surface colour is calculated including the interaction with the lights.
\texttt{Ci = Oi*Ct*( Kd*diffuse(Nf) + MyReflection(EnvName, Nf) +Ka*ambient() + specularcolor * Ks*specular(Nf, V, roughness));}

4.2 \texttt{Sclera.h}

The Sclera function generates the “white” of the eye pattern.

It takes in two parameters, the basecolour, which is generally a blue-ish tint and an off white colour.
\texttt{color sclera(color basecolor; color white;)}

The fBM function from Advanced Renderman [1] is used to return a value to control the mix between these two colours, which will generate the colour returned by the function.
\texttt{return mix(white, basecolor, fBm(pp, filterwidthp(pp), octaves, lacunarity, gain));}

4.3 \texttt{Veins.h}

The amount of veins visible in an eye is an important detail. If there are a lot of veins visible, the character could be tired or hungover, or if there are a huge number, it could be used to simulate a black eye. So, the amount of veins visible is also adjustable through a parameter to the eye surface shader which is then passed into this vein function. It takes 2 other parameters which are the colour of the veins and the sclera colour.
\texttt{color Veins(color TDbink; color scleraColor; float VeinVal)}

A turbulence function is used to generate the vein pattern, and the value of this is used along with \texttt{VeinVal} to mix between the basecolour of the sclera and the veins. This colour is returned from the function.
```cpp
float TurbVal = turbulence(pp, filterwidthp(pp), octaves, lacunarity, gain);
return mix(scleraColor, TDpink, smoothstep(0, VeinVal, (TurbVal*0.75) ));
```

4.4  
---

Iris.h

The function to generate the iris pattern takes in 7 parameters – the surface normal of the eye, and 6 colours. The first 4 colours are for generating the colours of the lines in the iris, and the last 2 are for the turbulence underneath.

```cpp
color ProceduralIris(normal Nf; color Color1; color Color2; color Color3;
    color Color4;
    color Under1;
    color Under2; )
```

Next the texture co-ordinates ss and tt are changed. The area being shaded is split up into a grid. The decimal value of ss and tt is then found using the mod function which will throw away the integer of a number and return the decimal.

```cpp
float ss = mod( (s*sin(5)), RepeatT);
float tt = mod(t*2+0.1*sin(s*RepeatT), 1);
```

Next the colour values needed are declared:

```cpp
    color Ct1, Ct2, Ct3, Ct;
```

The cellnoise function is then used to generate a random value for each cell in the grid

```cpp
    tt = cellnoise(ss)*tt;
    float lines = 10;
```

Turbulence function is used to generate noise values, which are then used with the ss co-ordinate to generate a value that is passed through a sin wave.

```cpp
    float TurbS = (ss*lines) + (3.0*(turbulence(pp, 0.65, 0.2, 0.25, 1)) );
    float SinS = sin(TurbS*PI);
```

Next the floor function is used to get the whole number in the function to see which line we are on

```cpp
    float whichLine = floor(ss * lines);
```

Use the filtered smoothstep and abs functions to prevent aliasing, (from “Advanced Renderman” [1]) when blend between 0.1 and 0.9 using the distance value.
float dist = filteredabs(SinS-0.2, 0.5);
float inLine = filteredsmoothstep(0.1, 0.9, dist, 2);

Then calculate the different layers of colour in the lines of the iris using these values just calculated
  Ct1 = mix(Color1, Color2, inLine);
  Ct1 = mod(whichLine, dist)*Ct1;
  Ct2 = mix(Color3, Color4, (inLine-0.5));
  Ct2 = mod(whichLine, dist)*Ct2;

To calculate the pattern underneath these lines a turbulence function is used to mix the two underneath colours together

float TurbVal = turbulence(pp, filterwidthp(pp), octaves, lacunarity, gain);
Ct = mix(Under1, Under2, smoothstep(0, 1, TurbVal ) );

Once both top and bottom patterns of the iris are generated, then they are simply added together
  color MyCt = Ct + (Ct1+Ct2);

This is the value that is returned as the iris colour.
  return MyCt;

4.3. **Tear Duct**

The tearduct is a small mound skin attached to the inside of the eye. This is an important part of simulating a realistic eye. This project includes a shader for the surface of the tearduct, however there are two possible ways of attaching this to the eye. The first method is to paint it onto the surface of the eye using a texture map, and then displacing it. However this method is not suitable if the eye is going to be moving, because the tearduct will move with it, and in a real eye the duct doesn’t move. The preferable method would be to apply this shader to separate geometry modelled with the face.

To include the tearduct on the eye geometry as a painted texture it was necessary to modify the surface shader. Here a brief outline of the section relevant to this.

4.3.1 **Eye&Duct.sl**

The *EyeAndDuct.sl* is essentially the same as the *Eye.sl*, however there are a few modifications.
A painted texture is created, with the area of the geometry to hold the tear duct shader painted black on white. This is then converted to a texture file and read into the EyeAndDuct.sl as a float value.

```cpp
float DuctTexture = float texture ("Duct.tx");
```

The texture is inverted and made into a separate texture file which is also read in as float.

```cpp
float EyeTexture = float texture ("Eye.tx");
```

A call is made to the tear duct function to generate the tear duct pattern, which is returned as a colour.

```cpp
color TDuct = TearDuct();
```

Finally the surface calculations for the tear duct with the lights are made, and this value is then multiplied by the texture.

```cpp
Ci = (Oi*Ct*( Kd*diffuse(Nf) + MyReflection(EnvName, Nf) +Ka*ambient() + specularcolor * Ks*specular(Nf, V, roughness)) * DuctTexture);
```

The calculations for the surface of the eye with the lights is then calculated, and this is multiplied by the inverse of the texture. This value is added to the tear duct value and returned as the shader result.

```cpp
Ci += (Oi*TDuct*( Kd*diffuse(Nf) + MyReflection(EnvName, Nf) +Ka*ambient() + specularcolor * Ks*specular(Nf, V, roughness)) * EyeTexture);
```
The tear duct is a small mound of skin joined to the eye. It is usually dark pink in colour with a few tiny but noticeable veins running through it. It is moist and reflective, similar to the eye.

This function is simply called, it doesn’t take any parameters. It returns a colour.

```latex
color TDUCT = TearDuct();
```

It first uses the turbulence function to generate a subtle noise between red and pink, then it mixes this colour into an fBM noise to create the veins.

```latex
Ct = mix(TDlightRed, TDlightPink, turbulence(pp, filterwidthp(pp), octaves, lacunarity, gain) );
Layer1 = mix(Ct, TDDoubleLightPink, fBm(pp, filterwidthp(pp), 6,3, gain) );
```

### 4.4 Eye.rib

`Eye.rib` is the rib file used to generate the spherical geometry to test the Eye shader. It was also used to apply the shader to a polygon when calculating surface positions when designing the shader.

### 4.5 Pupil.sh

`Pupil.sh` is a simple shell script generated to output values from 0.09 → 0.02 to be passed into the rib file as the input values to the `pupilsize` parameter for the surface shader.

It also generates a tiff per value passed in, with the frame number incremented each time.

### 5. MEL Scripts

I have written several MEL scripts to accompany this project, to automate as much of the animation for the eye as possible.

#### 5.1 MEL Script to animate the intensity of lights in Maya

This script changes the intensity of a point light in Maya. The slow change in intensity over the animation should correspond to the change in pupil size on the eye. However as it is a MEL script it only works for Maya lights, though I still wanted to write it to get practice at writing MEL scripts, and so I can convert it to a TCL script at a later date.
It starts by creating a point light and setting the intensity to 0.

\[\text{pointLight -n PLight1 -i 0;}\]

Next the number of frames the animation runs for is declared

\[\text{$NoFrames = 100;}\]

Next the number of frames is divided up so the animation can be broken down over sections of time.

\[\text{$NoFrames1 = $NoFrames/2;}\]

The following variables are declared, to hold the current intensity, the max intensity and the change in intensity.

\[\text{float $CInten = 0.0, $Inten = 5.0;}\]
\[\text{float $ChgInten = $Inten / $NoFrames1;}\]

The first loop in the script will change the intensity of the light from 0 up to the max value over half the time of the entire animation

\[\text{for ($FrameNo = 0; $FrameNo<$NoFrames1; $FrameNo++)}{}\]
\[\text{\hspace{1em} $CInten = $CInten + $ChgInten; }\]
\[\text{\hspace{1em} setKeyframe -time $FrameNo - value $CInten PLight1.intensity; }\]
\[\text{}\]

When this loop is complete, the intensity is now set at the maximum value. The second loop will bring it gradually back down to 0 over the course of the animation.

\[\text{$CInten = $Inten;}\]
\[\text{for ($FrameNo = 0; $FrameNo <$NoFrames1; $FrameNo++)}{}\]
\[\text{\hspace{1em} $CInten = $CInten - ($Inten/$NoFrames1);}\]
\[\text{\hspace{1em} $FrameNo2 = $NoFrames1 + $FrameNo;}\]
\[\text{\hspace{1em} setKeyframe -time $FrameNo2 - value $CInten PLight1.intensity;}\]
\[\text{}\]

5.2 **MEL Script to animate an eyeball looking from left to right and up and down**

MEL script to automate the animation of an eye looking from left to right or up and down.

The up and down procedure changes the value of rotation around the X axis.

The over and back procedure changes the value of rotation around the Y axis.

Both procedures are defined by taking in 3 arguments through the command line:

\[\text{proc LookLR (float $StartFrameNo, float $EndFrameNo, float $NoLooks)}\]
\[\text{proc LookUD (float $StartFrameNo, float $EndFrameNo, float $NoLooks)}\]
The required variables are defined and the user entered values are used to calculate the number of frames in the animation, and the number of frames per “look”.

```plaintext
float $EyeBall = 0;
float $NoFrames = $EndFrameNo - $StartFrameNo;
float $NoFramesLook = $NoFrames / $NoLooks;
float $FrameNo = $StartFrameNo;
float $z = $StartFrameNo;
```

The first loop to be defined is the loop for the number of “looks”
```
for ($b = 0; $b < $NoLooks; $b++)
```

Next its the loop for the number of frames per look
```
for ($i= 1; $i <= $NoFramesLook; $i = $i +1)
```

The loop is gone through each for each look, with it being broken up into 4 quarters. The first one is the eye looking to the left. The speed the eyeball moves is determined by the number of frames in the block
```
if($FrameNo < ($z + ($NoFramesLook / 4)))
{
    setKeyframe -time $FrameNo -value $EyeBall LeftEyeBall.rotateX;
    setKeyframe -time $FrameNo -value $EyeBall RightEyeBall.rotateX;
    $EyeBall = $EyeBall - (45 / ($NoFramesLook/4));
}
```

The second quarter is the eye returning to the centre from the left:
```
else if ($FrameNo > ($z + ($NoFramesLook / 4)) && $FrameNo < ($z + ($NoFramesLook / 2) ))
{
    setKeyframe -time $FrameNo -value $EyeBall LeftEyeBall.rotateX;
    setKeyframe -time $FrameNo -value $EyeBall RightEyeBall.rotateX;
    $EyeBall = $EyeBall + (45 / ($NoFramesLook/4));
}
```

The third quarter is the eye looking to the right:
```
else if ($FrameNo > ($z + ($NoFramesLook / 2)) && $FrameNo < ($z + ($NoFramesLook /2) + ($NoFramesLook / 4)) )
{
    setKeyframe -time $FrameNo -value $EyeBall LeftEyeBall.rotateX;
    setKeyframe -time $FrameNo -value $EyeBall RightEyeBall.rotateX;
    $EyeBall = $EyeBall + (45 / ($NoFramesLook/4));
}
```

The final quarter is the eye returning to the centre from the right:
```
else if ($FrameNo > ($z + ($NoFramesLook /2) + ($NoFramesLook / 4)) && $FrameNo < ($z + $NoFramesLook ))
{
    setKeyframe -time $FrameNo -value $EyeBall LeftEyeBall.rotateX;
```
setKeyframe -time $FrameNo -value $EyeBall RightEyeBall.rotateX;  
$EyeBall = $EyeBall - (45 / ($NoFramesLook/4));
}

5.3 **MEL Script to automate the blinking of an eye**

MEL script to automate the animation of an eye blinking. User enters the start frame, the last frame, and how many times they wish the eyes to blink. The script uses these values to calculate the rate of blinking over the course of the animation. The blinking only applies to spherical based geometry – so would be more suited towards something like non-realistic animation, however the script could be adapted to take into account specific models.

The procedure is called from the command line, and takes in 3 arguments:

```
proc Blinking (float $StartBlinkFrame, float $EndBlinkFrame, float $NoBlinks)
```

The rest of the variables needed are declared at the start of the procedure

```
float $FrameNo = 1;
float $LidTop = 45; //value top lid starts at
float $LidBottom = -45; //value bottom lid starts at
```

Next, the number of frames in total is calculated, and using the number of blinks required, the amount of frames per blink is calculated.

```
float $NoFrames = $EndBlinkFrame - $StartBlinkFrame;
float $NoFramesBlink = $NoFrames/$NoBlinks;
$FrameNo = $StartBlinkFrame;
float $z = $StartBlinkFrame;
```

The first loop is declared – this one is for the number of blinks wanted.
```
for ($b = 0; $b < $NoBlinks; $b++)
```

The next loop declared is for the amount of frames per blink.
```
for ($i = 1; $i < $NoFramesBlink; $i = $i+1)
```

First this loop checks to see if the frame number falls in the first or second half of the blink – so if the eyes are open or closed.
```
if ($FrameNo < ($z + ($NoFramesBlink / 2)) )
```

A Keyframe is set for each eyelid to rotate along the X axis by a certain degree each time. This small amount is calculated using the amount of frames per blink.
```
setKeyframe -time $FrameNo -value $LidTop LeftEyeLidTop.rotateX;
setKeyframe -time $FrameNo -value $LidTop RightEyeLidTop.rotateX;
setKeyframe -time $FrameNo -value $LidBottom LeftEyeLidBottom.rotateX;
setKeyframe -time $FrameNo -value $LidBottom RightEyeLidBottom.rotateX;
```
$LidTop = $LidTop + (45/ ($NoFramesBlink/2));
$LidBottom = $LidBottom + ( -45/ ($NoFramesBlink / 2) );

If the frame number is in the second half, then the opposite number is calculated to bring
the eyelid back to the position it was in.

else if ($FrameNo > ($z + ($NoFramesBlink /2) ) && $FrameNo <= ($z +
$NoFramesBlink ) )
    {
        setKeyframe -time $FrameNo -value $LidTop LeftEyeLidTop.rotateX;
        setKeyframe -time $FrameNo -value $LidTop RightEyeLidTop.rotateX;
        setKeyframe -time $FrameNo -value $LidBottom
            LeftEyeLidBottom.rotateX;
        setKeyframe -time $FrameNo -value $LidBottom
            RightEyeLidBottom.rotateX;
        $LidTop = $LidTop - (45/ ($NoFramesBlink/2 ));
        $LidBottom = $LidBottom - (-45/ ($NoFramesBlink/2 ));
    }

6. **Maya & Slim modelling, animation & rendering**

To test the shader in Maya, I created a NURBS sphere, and selected the control points
along the top. I moved these control points inwards towards the sphere to create a small
dent in the surface. This lines up with the iris and pupil to help give the eye more depth.

I modelled simple geometry around the eye spheres to represent eyelids, and used the
MEL scripts to generate some blinking and looking around animation.

To shade the eyelids I created a layers shader in Slim, layering up the skin shader, a
velvet shader and a clay shader to give the skin some grooves. The resulting animation is
more toon-like than realistic but this is mostly due to the shape of the eyelids and the skin shader.
7. Shader Development & Problems Encountered

This section is a breakdown of the development of the shader, along the problems I encountered and how I dealt with them.

7.1 Mapping the texture correctly onto the geometry

The initial problem I had was how to get the shader to generate the pupil and iris patterns in the right position. I ended up using a similar method to Larry Gritz in his eye shader [*******], where the pole of the sphere is used as the centre of the pupil — so the layers of the eye will appear as lines on a polygon and when wrapped around a sphere they create a loop.
The next issue I had was how to generate a realistic iris pattern that would give depth to the eye. The first method I tried was using a texture file as the iris pattern. I created some sample texture files using sections from images of real eyes. I then mapped this onto the area of the iris, and blended it into the pupil and the outer rim. It gave acceptable results, but it didn’t achieve the look I was aiming for.

Next I started to create lines along the area of the iris pattern, just along the t direction. I wanted to break these up into different random colours so I broke up the geometry into a grid and worked in squares that repeated. I then added noise and passed the values through a sin wave to generate a more wavy, irregular pattern.
I built on this, adding more noise and trying to get the pattern more irregular, but still retaining the lines that are characteristic of an iris.

Next I worked on the idea of creating an underneath layer but with a much smoother noise. I did some tests and found the turbulence function created a good wiggly pattern. When I was happy with the pattern I added this to the “lines” of the iris.

For the outer layers, an fBM function.[1] is used to generate a very subtle transition between the iris and noise around the edge, before it blends in with the sclera, to simulate the limbus area of the eye.

### 7.3. Generating the Sclera and Vein patterns

Next I had to simulate the veins and the sclera (white) of the eye. I tried a few tests with different types of noise, but I found that for the sclera it was best to stay subtle, so I used
fBM to get a soft mix between white and blue-ish/beige colour. The veins in an eye are difficult to simulate – they have to be subtle but distinct and they can’t be too irregular. Again, I experiment with different blends and noise values, finally settling with a sharp turbulence noise.

![Early Veins Test](image1) ![Final Vein Pattern](image2)

7.4 **TearDuct**

The tear duct needed some thought. Because the tear duct is not actually part of the eyeball, but it is attached it was difficult to know whether to try and incorporate it into the eyeball shading model or create one specifically for it.

I ended up creating a separate shader, that can be painted onto the eyeball using a texture map or onto separate geometry.

The tear duct is small but yet there are still tiny veins visible on the surface. To simulate this I carried out some tests and eventually found a suitable noise function to generate small enough lines.

![TearDuct Test 1](image3) ![TearDuct Test 2](image4) ![Final TearDuct](image5)

7.5 **Environment Maps and Reflections**
A human eye has highly shiny quality – like that of glass or fluid. However, I feel that giving an eye created in CG a prominent specular highlight only works against it. In order to create a subtle reflection on the eye I decided to use environment maps.

There are two functions I ran tests with, both are extracts from the Advanced Renderman book. [1].

The first one takes in the environment map texture name and the surface normal. It simply looks up the map and gets the point that will be reflected in the point we are shading.

Using this function with the following texture and the eye shader I got the following results:

![Environment Map](image1.png) ![Function1 Test](image2.png)

The second method is based on ray tracing. It looks up the environment map, and ray traces against a sphere of known, finite radius. These are the results from using this function with the same texture map:

![Function2 Test](image3.png)

I have used the first one as it is not quite as it doesn’t affect the surface as much, however I have left the call to the function in the code for future reference.

### 7.6 Generating the size of the pupil
One of the methods I decided to explore in developing this shader was using the amount of intensity on the surface of the pupil as a measure for the size it should be, much like a real eye. I began by looking at different methods to determine the intensity, the first being Lamberts law.

Lamberts law states that the intensity of illumination on a surface is proportional to the cosine of the angle between the surface normal and the light source direction.

I created an illuminance loop to go through all the lights in the scene and get the light source direction, and then use this with the surface normal to get the intensity at the current point. I then tested methods of using this to generate the pupil size.

The major issue with this is the nature of shaders: you only know information about the current point being shaded. You cannot reference other points, or find other points information and relate them to the current point, and the intensity at the pupil needs to be determined for each point being shaded, whether it is in the pupil or not. The intensity value also needs to be the same for the pupil each time too to give a smooth pupil circle.

The best solution to this issue I decided upon was based on the idea that you always know the centre position of the pupil, and the current point. Get the intensity at the current point, get the distance we are from the pupil, and use the distance to scale up the intensity. However then we must consider that we might be closer to the light than the pupil, and if we are our intensity will be greater than the pupil. We also need to consider the number of lights on the scene. Currently, the method I have employed to use an illumence loop to get the “to” position parameter of the light works, however it only works if there is 1 light in the scene, and it still produces results that are not smooth enough. I am continuing work on this area of the shader.

7.7 Anti-Aliasing.

Once the eye shader was applied to the geometry in Maya, and animated rotating the lines in the iris appeared to alias. In order to counteract this “popping” I employed the filtered versions of functions such as smoothstep and abs in the calculations of the iris pattern. Most of this is solved at render time if a very low shading rate is used (eg, 0.05) and a very high pixel sample (eg 32 by 32). This solves most of the aliasing but it increases render times, so I for the mpeg I have included I have a shading rate of 0.5 and a pixel sample of 3 by 3, so there are some noticeable artifacts.

8. Conclusion & Further Work

It takes about a minute to render a frame of a sphere with this eye shader attached. There is huge scope for development of different areas of this project. I am continuing to work on the procedurally calculated pupil size, based on the light intensity on the surface, which is almost finished. Currently it only works if there is only 1 light in the scene but I am hoping to get it to work for any number of lights.
The tear duct is another area that could be developed further. I have included the shader for the surface of the duct itself, but it would be an interesting extension to have the duct as separate geometry, but have a method in the eye shader that is incorporated into the vein function. This method would calculate how far the pupil is from the tear duct, and generate the amount of veins accordingly – i.e. when the right eye is looking to the right, the pupil is further away from the duct and so there would be more distinct veins on the surface at the area of the duct and if the eye was looking towards the duct there would be less.

I would also like to experiment more with TCL scripting so I could translate the MEL script to change the intensity of the lights to apply to Renderman.

The eye is a detailed object to model, shade and animate. Having a shader that the user can tweak according to their characters requirements is a useful tool, along with scripts to procedurally add in keyframes of animation to automate things like blinking.

9. References


