Innovations – 2008

Joe Abreu – e9550158

Painterly shader – Cartoon Painterly Renderer

Abstract

The aim of this project is to create a 3D shader that is able to render a computer generated scene. More specifically, a scene which is visually similar to the backdrop created in the 1967 film Jungle Book. I will use a nurbs surface as the base that the painterly shader will act upon. The surface itself will play an important role when trying to produce a similar size and shape to the Jungle Book backdrop. The shader alone will attempt to recreate the exact characteristics of a hand painted backdrop. Overall I wish to compose a shader that will have well defined but simple parameters, to allow the user to produce a complex painted scene.

TABLE OF CONTENTS

CHAPTER I

INTRODUCTION

Example Cartoon Backdrops –

Characteristics to recreate

Backdrops are subtle, hand painted backgrounds and therefore the images created are 2d. The perception of depth, light, tone, colour and texture is all produced by the artists brush. There are a few issues that have to be considered when moving from a 2d to a 3d medium and these are as follows:

Perspective – Depth has to be created by the artist in a 2d medium, however with a 3d package perspective and depth are handled automatically. The only consideration is the conversion of a 2d image into a 3d modelled scene.

Colour and Texture – This will be the main focus for my shader itself. Reproducing similar hues and colour palettes will be the user's job to ensure the correctness with each scene. The texture or surface of the objects should be automated mostly by the shader.

Light and Tone – In the original backdrops the artist created the illusion of light and shadows through colour. These are handled in the 3d package so a further amendment must be made to connect the effects of lights on the shader, so it compliments with the original style of the painting.

No sharp defined edges – When painting a background, due to the nature of paint and to particular styles of painting, an image is built up through many layers of strokes. This leads to an image without any sharp or defined edges and the illusion of these edges can only be achieved through smaller finer brush strokes.

Image Detail – A painted image generally is of less detail than perhaps a photograph would be. The level of detail of a painting is normally determined by the size of the strokes. A painting can have varying areas of detail; one area may use fine small strokes meaning that there is a high level of detail, whereas areas with larger brushstrokes tend to be areas of low detail.

Artefacts – Imperfections within paintings are what give many images its character; furthermore recreating some artefacts should aid the recreation process. Too many artefacts may hinder the overall effect, and there must be coherence between each frame of an animation so artefacts don't look like random noise.

CHAPTER II - RESEARCH

Background Work

During the early years of computer graphics development there was a want to strive for photo-realistic images. A great deal of research was done into the way light bounces off of objects, how different materials react and the ways that realistic objects looked. Since the advancement of photorealistic renderings we have come to the point where it becomes near impossible to distinguish a realistic photo from a computer generated image; a new sub-genre of computer generated images was formed, non-photorealistic rendering (NPR). Renders that strived against realistic images to ones that we're freer, more artistic.

Direct / Indirect Rendering

Direct Rendering is where everything is handled within the 3d renderer. So all calculations are done, and the final image is the composite painterly rendering. For indirect Renderings, the 3d package renders out control images, which another normally 2d software uses the control images to calculate how the image should be 'painted'. Due to what outcomes I wish to achieve from the project I will utilize a direct method, by creating a RenderMan shader. This will aim to put a lot of the work upon the shader and not the user to control the render.

Painterly Rendering for Animation by Barbara J. Meier – In 1992 Meier developed a technique for painterly rendering. It describes that particles populate a surface, via a particle placer program. Control Images are rendered out from the geometry to describe such things as zdepth, colour, orientation. Using these control images and the particle data, a painterly renderer rendered a brush stroke texture at each of the particles of relevant size colour and orientation. The resulting image is many layered brush textures that resemble a hand painted canvas.

Deep Canvas – Deep Canvas was a technique developed firstly for the 1999 film Tarzan. In which, an artists paints an image of a particular frame of animation and the system recorded the stroke colour length, pressure, position etc. The computer will then repaint the image onto the 3d surfaces. Repeating this process until all of the 3d objects have been textured.

Spatio-temporal coherence - Temporal Coherence is important for animations, and should be maintained on a frame by frame basis.

'Frame by frame coherence is achieved when the media moves with the object. Without tracking, objects appear to be moving behind the media, this is known as the "shower door" problem.' [KAUSHIK PAL]

Entropy - Perfect coherence is often not wanted for NPR, because it is the slight imperfections that give hand crafted media their characteristics. So, recreating these faults will aid in the reconstruction of a painterly shader. Controlling these blemishes will be of importance, because if there are too many imperfections that occur randomly on a frame by frame basis, it may just look like distracting noise.

CHAPTER III – Writing Shaders

How to Approach writing shaders:

'*Simple* shaders comprise at least 90% of the shaders that get written.' [Writing Renderman Shaders, SIGGRAPH 92, Course 21]

The majority of shaders are very simple shaders that comprise of simple techniques. Complex shaders are usually composed of a set of simple techniques. 'Writing Renderman Shaders' describes a very good system for creating complex shaders, they use the term "Divide and Conquer". The method details that writing a shader should consist of 4 phases:

- Pattern Generation – Creating a repeating or stochastic pattern that resembles a single characteristic of a shader.
- Layers – Many interesting patterns cannot be described by a single function, but are composed of subtle patterns on top of larger patterns.
- Illumination Model – Many shaders can use a simple illumination model like standard ones describes in plastic.sl, but for some more complex shaders that may use a patterned illumination model or for surfaces that might not reflect like a plastic surface will.
- Compositing – Some shaders have layers of totally different characteristics, illumination calculations upon each layer may be totally different. Each layer has to be composed together using the standard compositing equations.

To start writing a shader there are a few things that should be achieved first:

- Make sure you have good geometry
- Know what look you are going for
- Start with a simple shader that is closest to the look you are going for
- Work on aspects of the shader one layer at a time. Mimicking the way a real object achieves its characteristics.

Some important things to look out for when creating your own shader:

- Slow Rendering Times – This means something in your code may be too complex, resulting in an inefficient and unusable shader.
- Aliased Images – Aliasing will create artefacts in your image and will affect image coherence.
- Wasted Effort – Don't spend a lot of time on something that will only marginally affect the end result.

The first step to building a shader is to take the real world example of the effect you are going for and break it down into its characteristics. For example; a football has colour, bumps, dirt and possible decals. Each of these characteristics should be worked on one at a time until they match the real world model then composite each of the characteristics together.

Simple Shaders

Learning simple shaders help to develop the techniques involved to create more complex shaders. The following will be some techniques that I researched in the hope to aid the final composition of my painterly shader. The first and simplest shader is the constant shader.

```
surface constant ( )
{
        Oi = Os;
        Ci = Oi * Cs;
}
```

Figure 3.1 :

The surface uses four 'Standard Geometric Primitive Variables' [Renderman Specification 3.2], that describe how the surface of an object will look.

Oi and Ci are the variables that define the final surface opacity and final surface colour, respectively. Os and Cs, define the surface opacity and surface colour which has been defined in the RIB file that uses the shader. The output of this shader will be a constant color that was defined by the RIB file. See figure: 3.2.

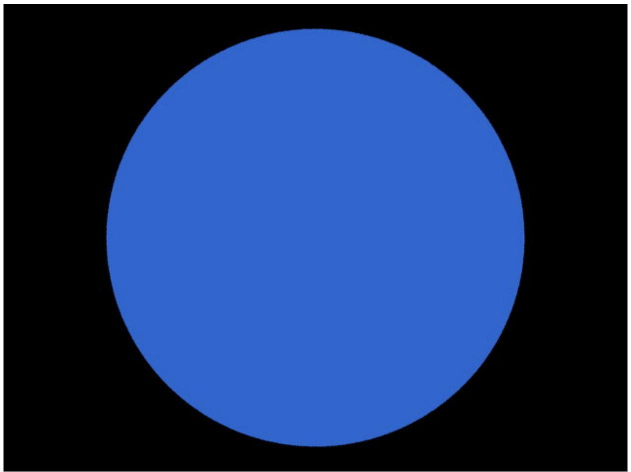


Figure 3.2 : constant shader

Creating a repeating pattern is an important technique used in many shaders to get something to appear many times over a surface.

```
surface tiles ( )
{
	/* Initialize Variables */
	color Ct;
	float grid_size = 5; /* Number of tiles */

	float sTile=floor(s*grid_size); /* vertical tile coordinate  */
	float tTile=floor(t*grid_size); /* horizontal tile coordinate */

	/* Texturing */
	Ct = color cellnoise(sTile,tTile); /* returns a random color per tile */

	/* Shading */
	Oi=Os;
	Ci = Oi * Ct;
}
```

Figure 3.3 : tiles.sl – create a tiled pattern

Splitting the shader into clear sections is a good way to keep a structure of a shader without getting lost. There are 3 main areas involved here:

- Initialize Variables – Create any variables, keeping them all in one place is a good way to understand what each variable does and what it contains.
- Texturing – Everything within this section is used to control the colour and opacity of a surface point.
- Shading – Takes everything computated from the shader and applies them to the output variables, generally colour and opacity for surface shaders.

To generate the tiled pattern, taking the texture coordinates s and t[1] they are multipled by how many tiles we want along each edge and round down the number to a whole. So that for every point it is inside a certain tile i.e s=0.1, t=0.55, grid_size=5;

sTile = floor(0.1 * 5) = floor(0.5) = 0.0;
tTile = floor(0.55 * 5) = floor(2.75) = 2.0;

So, we know that texture_coordinate(0.1,0.55) is in tile_coordinate(0,2). We can then create a colour for each tile by using a pseudorandom function cellnoise()[2]. Finally we put the randomly generated colour into our output variable Ci.

---

[1] s and t range from 0 to 1 and specify the position on a texture the current sample point is at.

[2] cellnoise() returns a value which is a pseudorandom function of its arguments. Its return value is uniformly distributed between 0 and 1

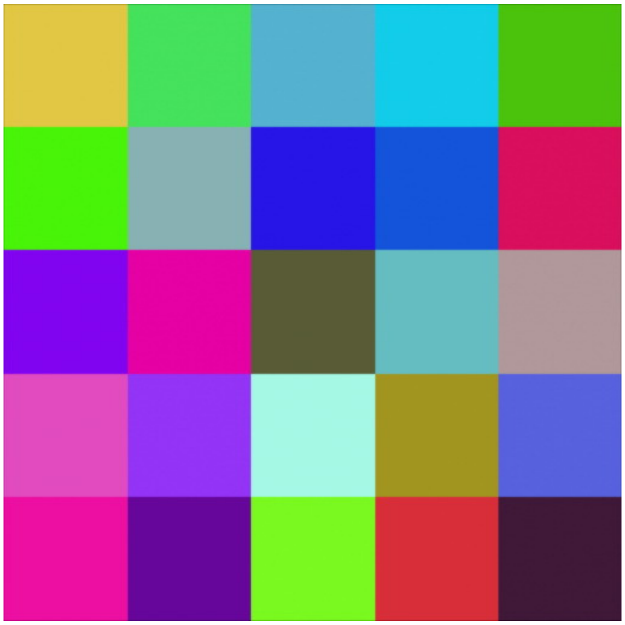


Figure 3.4: tiles shader

Once different patterns have been made, to produce the final shader the layers have to be composed together.

```
surface layers ( )
{
/* Initialize Useful Variables */
        color Ct;
        color red = color "rgb" (1,0,0);
        color green = color "rgb" (0,1,0);
        float dist; float inLine;

/* Texturing */
        /* Layer 0 */
        Ct = Cs;                /* Background Color */

        /* Layer 1 */
        dist=abs(s-0.5);        /* distance from vertical center */
        inLine = 1-step(0.1,dist);
        Ct = mix(Ct, red, inLine);

        /* Layer 2 */
        dist=abs(t-0.5);        /* distance from horizontal center */
        inLine = 1-step(0.1,dist);
        Ct = mix(Ct, green, inLine);
/* Shading */
        Oi=Os;
        Ci = Oi * Ct;
}
```

Figure 3.5 : layers.sl – layering patterns

Just like splitting each section into group it is a good idea to sub-split those sections into logical areas, in this case; separating each layer. In Figure 3.5 Layer 0 describes the background colour by first setting the color variable Ct to Cs. Layer 1 describes colouring a vertical line in the center of the texture. Determining whether the given sample point is close enough to the center determines how the compositing equation works. Using the mix()[3] function it chooses whether the background colour should be replaced with the new layer colour, indicating that it is within the line or left as it is, denoting that it is outside the line. This same process is repeated for every layer until the final chosen colour is plugged into Ci.

---

[3] Mix(color1,color2,value): blends between the given input colours by using the input value.

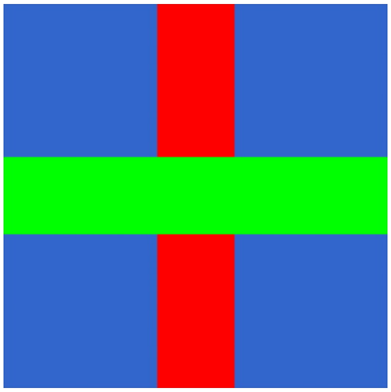


Figure 3.6: layers shader

A really important feature of creating shaders is being able to add stochastic adjustments to calculations. This can be achieved by using the function noise()[4]. Utilising noise can be a really powerful feature to creating realistic, more natural shaders.

```
surface brownian ( )
{
        /* Initialize Useful Variables */
        color Ct;
        float i; float mag=0; float frequency=1;

        /* Texturing */
        for(i=0;i<layers;i+=1) {
                mag += (float noise(P * frequency) - 0.5) * 2 / frequency;
                frequency *= 2.1;
        }
        Ct = mag+0.5;

        /* Shading */
        Oi=Os;
        Ci = Oi * Ct;
}
```

Figure 3.7 : brownian.sl – layered noise shader

Primarily the use of noise may not be to produce an image much like figure 3.8, but to use the values return by the function to control other aspects of a shader, for example size of displacement in a simple brownian displacement shader, generates a surface that is very similar to many varieties of fruit. Noise effectively is used to dirty up a shader to avoid it looking too pristine, too perfect and has a little more realism.

---

[4] noise() returns a value which is a pseudo random function of its arguments;

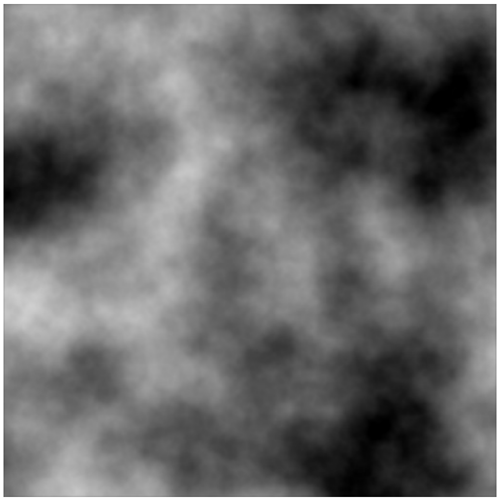


Figure 3.8: brownian.sl - 2d noise shader

To create shader that can be driven by user input, one of the strongest ways to accomplish this is with a texture map. Allowing a user to connect an image to the shader. The connected image can be used in a variety of ways; one of the obvious is to drive the colour of the shader, as shown in figure 3.10.

```
surface texmap ( string mapname = "";)
{
        /* Initialize Useful Variables */
        color Ct;

        /* Texturing */

        if( mapname != "" )
                { Ct = color texture(mapname);}
        else
                { Ct = Cs; }

        /* Shading */
        Oi = Os;
        Ci = Oi * Ct;
}
```

Figure 3.9 : texmap.sl – connect a texture map to surface color

Figure 3.9 shows a shader argument ‘string mapname’ which can be defined by a calling RIB file. This is used in the code to determine if a calling RIB file has set a texture to look up. If it has, it will lookup the colour in the file at the same texture location of the current sample point (s and t). Otherwise, it will just use the input colour.

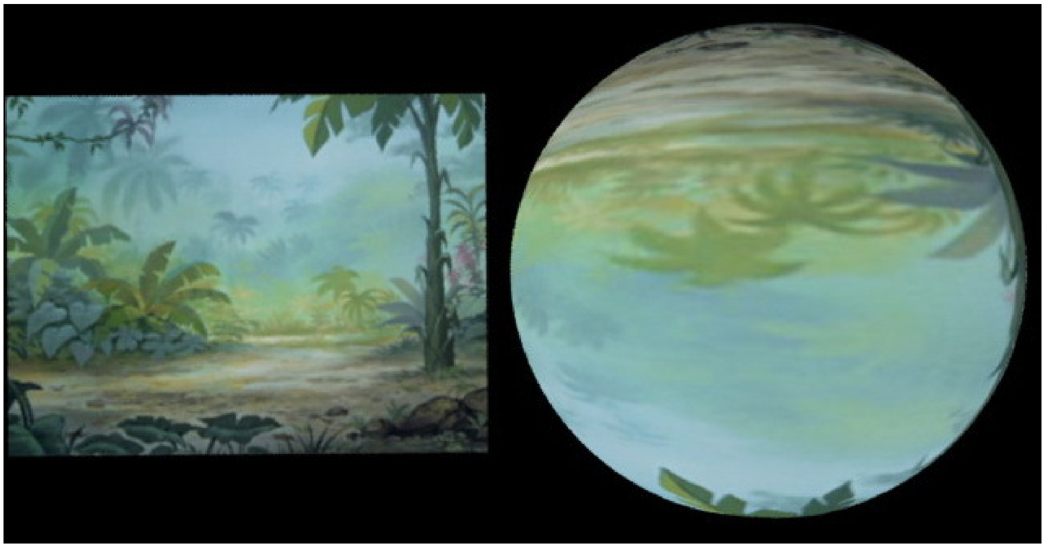


Figure 3.10: texmap.sl – user defined texture map

For a painterly renderer, there needs to be some way to create the look of different brush strokes. A renderman function is available that lets the shader find how far a point is from an arbitrary line of 2 points, which can be used to determine a position that a stroke has been made. The function is called ptlined()[5].

```
surface arbitrary_line ()
{
        /* Initialize Useful Variables */
        color Ct;
        color blue=color "rgb" (0,0,1);
        float edge_threshold = 0.025; /* edge threshold to interpolate between */
        point start=point (0.1,0.3,0);
        point end=point (0.7,0.7,0);
        point pos=point(s,t,0);

        /* Texturing */
        Ct = Cs;
        float dist = ptlined(start,end,pos); /* distance from center of line */
        float inLine = 1 - smoothstep(0.1-edge_threshold,0.1+edge_threshold,dist);
        Ct = mix(Ct,blue,inLine);

        /* Shading */
        Oi=Os;
        Ci = Oi * Ct;
}
```

Figure 3.11 : arbitrary_line.sl – draw an arbitrary line on a texture

---

[5] float **ptlined** (point Q, P1, P2)
Returns the minimum perpendicular distance between the point Q and the line segment that passes from the point P0 to the point P1 (not the infinite line which passes through P0 and P1).

The whole code is very similar to the patterns made by Figure 3.5 [layer.sl], but it uses a function ptlined() to determine the distance and not the center of the texture. The resulting image Figure 3.12 shows a red line placed at a set location, this is the simplest idea behind creating different kinds of brush strokes. Adding to this shader could be done to affect the brush width along the line and creating some stochastic effects to make it appear more realistic. There is another important function involved here and that is smoothstep()[6]. It is used to interpolate between colours. This is very important for a problem known as aliasing. When objects are viewed far away, fine detail of the shader becomes too small to be properly represented on screen as a result the renderer may choose an incorrect pixel color. Anti-aliasing solutions are not a simple thing to manage and in some situations the perfect anti-alias solution is inefficient for a shader.

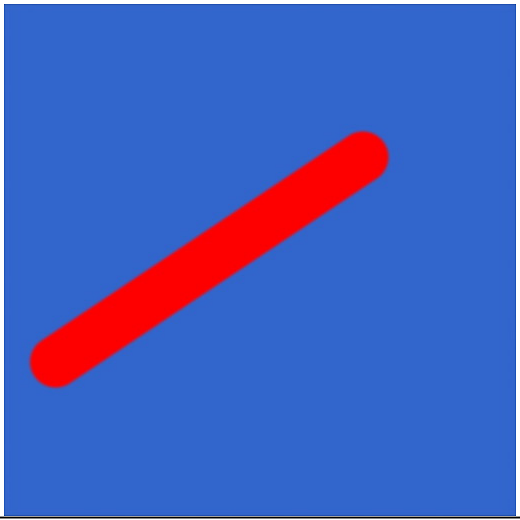


Figure 3.12: arbitrary_line.sl – an arbitrary line

---

[6] smoothstep(min, max, value) - returns 0 if value < min, 1 if value > max, and performs a smooth Hermite interpolation between 0 and 1 in the interval min to max.

CHAPTER IV – Painterly Shader

Basic Approach

I have determined different characteristics that my shader should have:

- A painting has varying layers of paint
- Lower layers have less detail
- Lower layers use fewer colors
- Higher layers use smaller finer brush strokes
- Each brush stroke uses a single tone of colour
- Brush strokes do not have sharp edges
- Strokes can be opaque or stippled with opacity in places.

My shader should include a function parameter to the texture map that the user can input. This map will control the base paint colour of the brush strokes. The user will create a map that will colour an object, it can be as detailed or sparse as required, other parameters of the shader will be able to fine tune how the texture map finally affects the look.

Light colour and shadows will be taken into consideration to determine how the brush stroke colour will be lightened or darkened when painted on the surface.

Code Snippets

Shader Parameters:

- float **Ka** – Ambient Contribution
- float **Kd** – Diffuse Contribution
- float **Ks** – Specular Contribution
- float **roughness** – roughness of the specular function
- float **impressionist** – if not 0.0 will use complementary colours for shadows
- float **grid_size** – the last size of the brush stroke grid
- float **stroke_length** – how long each stroke is
- float **start_width** – the starting width of the brush stroke
- float **end_width** – the ending width of the brush stroke
- float **stipple_frequency** – how much noise applied to a strokes line width
- string **base_color_map** – string to the user's control colour map
- color **underpaint** – colour of the background below the paint strokes
- float **color_variation** – how much colour variation occurs from the control colour map

```
float find_t(point A; point B; point pos)
{
        vector AB = B - A;                              // line to compare
        vector AP = pos - A;                    // vector from start to pos
        float angle = AB . AP;                          // angle between vectors

        vector APonAB = AP * cos(angle);    // vector of AP projected onto AB;

        float T = length(APonAB) / length(AB);

        return T;
}
```

Figure 4.1: find_t – finds the ratio of a point projected onto an arbitrary line AB

Figure 4.1 shows the code for finding the value of T that any point pos is along the line AB. This is function is required so that I will be able to linearly interpolate between the starting stroke width and end stroke width. I was able to use simple trigonometry to work t by finding then length of vector AP when projected onto vector AB, and finding the ratio between these two vectors.

```
color complementary_color( color inColor )
{

        /* extract color space into individual elements */
        float r = comp( inColor, 0 );
        float g = comp( inColor, 1 );
        float b = comp( inColor, 2 );

        /* calculate values  */
        float largest  = r > g ? (r > b ? r : b ) : (g > b ? g : b );
        float smallest = r < g ? (r < b ? r : b ) : (g < b ? g : b );
        float total = largest + smallest;

        /* find complementatry hue */
        r = total - r;
        g = total - g;
        b = total - b;

        /* return the complementary color */
   return color "rgb" ( r, g, b );
}
```

Figure 4.2: complementary_color – *returns the complemntary color*

One of my functions I made was complementary_color(). Its initial purpose was to reflect an impressionistic painting look by using the complementary colour as the shadow colour of an object, which would hopefully create an image that resembled more of a painted look. Observing the effect of how colour values change when changing the hue led me to devise how to work out a colours complementary colour.

```
float snoise(float x)
{
        return (2*noise((x)) - 1);
}
```

Figure 4.3: snoise – modifies the renderman noise function to the range -1< noise < 1.

```
float scellnoise(float x; float y)
{
        return (2 * cellnoise((x),(y)) - 1);
}
```

Figure 4.4: scellnoise – modifies the renderman cellnoise function to the range -1< noise < 1.

Figures 4.3 and 4.4 are simple function that were used to modify the range that the renderman functions noise() and cellnoise() return. The function on their own return a value that is normally distributed around 0 to 1, therefore it has an average of 0.5. But modifying these values to the range -1.0 to +1.0 meant I could use these as stochastic modifiers that would normally distribute around the same value.

```
float cell_s = s * i;
float cell_t = t * i;

float ss = mod(cell_s, 1);
float tt = mod(cell_t, 1);
```

Figure 4.5: cell coordinates and cell texture coordinates

cell_s and cell_t describe the index of which cell the current sample point is residing in.

ss and tt are the adjusted s and t parameters for the new cell. This mean that the texture surface ranges from 0 to 1 using s,t and each cell within this texture surface ranges from 0 to 1 using ss and tt.

```
point start = point (0.5, (0.5 - stroke_length/2), 0);
vector up = vector(0.0,stroke_length,0.0);
point end = start + up;

start = rotate(start, cellnoise(cell_s,cell_t)*i * 2 * PI , point( 0.5,0.5,-1 ), point( 0.5,0.5,1 ));
end = rotate(end,  cellnoise(cell_s,cell_t)*i * 2 * PI , point( 0.5,0.5,-1 ), point( 0.5,0.5,1 ));
point pos=point (ss,tt,0);
```

Figure 4.6: cell index and cell coordinates

To begin our brush stroke we have to place it within one of the cells. Placing the start of our arbitrary brush stroke in the center of the tile means that we can rotate it without any fear of it clipping outside of the cell. We rotate the stroke by a stochastic amount that I dependant upon the cell index. This means that on a frame by frame basis the stroke directions should look the same on the surface, no matter in which direction the camera is look at it. This help keeps temporal coherence within an animation.

```
ratio_t = find_t( start, end, pos );
line_width = start_width * ratio_t + ( 1 - ratio_t ) * end_width;
line_width += (snoise( ratio_t * stipple_frequency )) * (line_width/4);
```

Figure 4.7: adjusting line width

One thing to notice about painting a picture is that, stroke widths vary as an artist places strokes upon his canvas. It is never uniformly distributed, no matter how careful; the bumpiness of the canvas or pressure applied from the artist or even the amount of paint left on the brush will have an affect on stroke width. Figure 4.7 tries to recreate this effect by adding some random noise along the width of each stroke.

The noise of stroke width is dependant upon the shader parameters and the distance from the start of the stroke.

```
color stroke_color;
if( base_color_map != "" ) {
stroke_color = color texture( base_color_map, (floor(s*i)+0.5)/i  , (floor((1-
t)*i)+0.5)/i  );
} else {
stroke_color = Cs;
}

stroke_color += scellnoise(i*s,i*t)* color_variation;
```

Figure 4.8: picking stroke colour

The next step is to choose the colour of the brush stroke. If the user has supplied a base_color_map then it will lookup the colour that is in the center of the current cell index. Otherwise, it will use the colour as defined in the RIB file. Finally some variation to the brush colour is applied to dirty up the shader a little as so it isn't so perfect. By default the colour variation is set to 0.0, so it will use the exact lookup colour unless explicitly specified.

```
color shadow_intensity = (Ka*ambient()+Kd*diffuse(Nf)) + Ks*specular(Nf,V,roughness);
float intensity = (comp(shadow_intensity,0) + comp(shadow_intensity,1) +
comp(shadow_intensity,2)) / 3.0;

color complementary = complementary_color( stroke_color );

if( impressionist != 0.0 ) {
        stroke_color = intensity*stroke_color + (1 - intensity)*complementary;
} else {
        stroke_color = stroke_color*shadow_intensity;
}
```

Figure 4.8: determine shadow colour

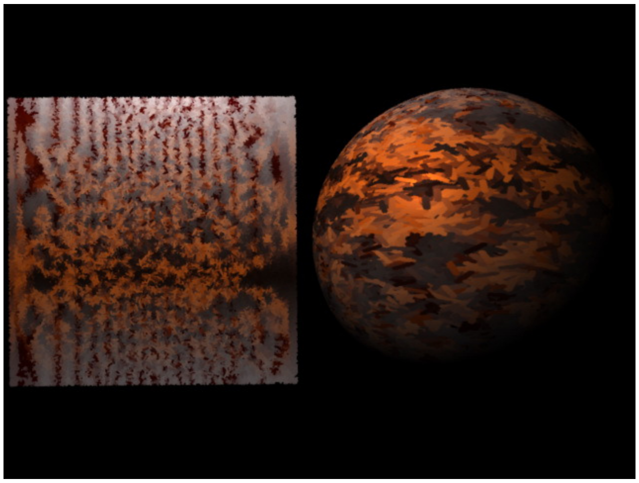
Once the colour of the stroke has been determined, this is altered by calculating how much light falls upon the surface. The equation is a simple plastic illumination model, which gets multiplied by the stroke colour to determine the final stroke colour. If the user has input an impressionist parameter then it will take the intensity of the shadow colour and multiply the current stroke colour by its complementary colour producing an impressionistic painting.

```
dist = ptlined(start,end,pos);

float inside_stroke=1-smoothstep(line_width/2-edge_threshold,line_width/2+edge_threshold,dist);

Ct = mix(Ct,stroke_color,inside_stroke);
```

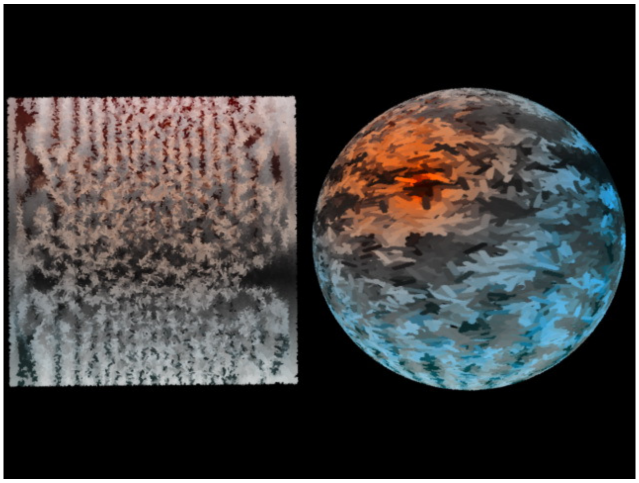
Figure 4.9: check current sample to see if it is or isn’t inside the brush stroke.

The last thing that has to be calculated for the current sample point is whether or not it lies inside or outside the current brush stroke. This is achieved by finding the distance of the sample point from the centre of the brush stroke line. This distance is compared to the line width of the stroke (plus an edge threshold for anti aliasing purposes), to see if it is further away in which case the equation returns 0.0, or 1.0 if it lies within the stroke. This value is used to mix the current brush colour with what has already been previous painted onto the canvas or rendered image.

RESULTS



The Rendered images here show how the painterly shader looks on a flat surface and a sphere primitive. The brush strokes can be seen up close, but from a distance the stroke seem to merge into a regular picture, which is a normal effect form some types of painting. Stroke colour is affect by the light and produces a nice shadow look to the render.



This render utilizes the impressionist function to attempt a more painterly shadow. I am not convinced that this type of shadow shows up enough that it is actually a shadow and not just a change in stroke colour.

My Shader Code – painterly_shader.sl:

```
float snoise(float x)
{
return (2*noise((x)) - 1);
}

float scellnoise(float x; float y)
{
return (2 * cellnoise((x),(y)) - 1);
}

float find_t(point A; point B; point pos)
{
vector AB = B - A;
vector AP = pos - A;
float angle = AB . AP;

vector APonAB = AP * cos(angle);

float T = length(APonAB) / length(AB);

return T;
}

color complementary_color( color inColor )
{
float r = comp( inColor, 0 );
float g = comp( inColor, 1 );
float b = comp( inColor, 2 );

float largest  = r > g ? (r > b ? r : b ) : (g > b ? g : b );
float smallest = r < g ? (r < b ? r : b ) : (g < b ? g : b );
float total = largest + smallest;

r = total - r;
g = total - g;
b = total - b;

return color "rgb" ( r, g, b );
}

surface layered_strokes (
        float Ka = 1;
        float Kd = 0.5;
        float Ks = 0.5;
        float roughness = 0.1;
        float impressionist = 0.0;
        float grid_size = 50;
        float stroke_length = 0.6;
```

```
    float start_width = 0.2;
    float end_width = 0.2;
    float stipple_frequency = 1;
    string base_color_map = "";
    color underpaint = 1;
    float color_variation = 0.0;)
{
normal Nf=faceforward(normalize(N),I);
vector V = normalize(-I);
color Ct = underpaint;
float edge_threshold = 0.025;
float i;

for(i=1;i<=grid_size;i+=1.0) {

    float cell_s=s*i;
    float cell_t=t*i;

    float ss = mod(cell_s,1);
    float tt = mod(cell_t,1);

    point start=point (0.5, (0.5 - stroke_length/2), 0);
    vector up = vector(0.0,stroke_length,0.0);
    point end = start + up;

    start = rotate(start, cellnoise(cell_s,cell_t)*i * 2 * PI , point( 0.5,0.5,-1 ), point(
0.5,0.5,1 ));
    end = rotate(end,  cellnoise(cell_s,cell_t)*i * 2 * PI , point( 0.5,0.5,-1 ), point(
0.5,0.5,1 ));
    point pos=point (ss,tt,0);

    float line_width;

    float ratio_t = find_t( start, end, pos );
    line_width = start_width * ratio_t + ( 1 - ratio_t ) * end_width;
    line_width += (snoise( ratio_t * stipple_frequency )) * (line_width/4);


    color stroke_color;
    if( base_color_map != "" ) {
        stroke_color = color texture( base_color_map, (floor(s*i)+0.5)/i  ,
(floor((1-t)*i)+0.5)/i  );
    } else {
        stroke_color = color cellnoise(i*s,i*t);
    }

    stroke_color += scellnoise(i*s,i*t)* color_variation;

    color shadow_intensity = (Ka*ambient()+Kd*diffuse(Nf)) +
Ks*specular(Nf,V,roughness);
```

```
    float intensity = (comp(shadow_intensity,0) + comp(shadow_intensity,1) +
comp(shadow_intensity,2)) / 3.0;
    color complementary = complementary_color( stroke_color );

    if( impressionist != 0.0 ) {
        stroke_color = intensity*stroke_color + (1 - intensity)*complementary;
    } else {
        stroke_color = stroke_color*shadow_intensity;
    }

    float dist = ptlined(start,end,pos);
    float inside_stroke = 1 - smoothstep(line_width/2-
edge_threshold,line_width/2+edge_threshold,dist);

    Ct = mix(Ct,stroke_color,inside_stroke);

}
Oi = Os;
Ci = Oi * Ct;
}
```

CHAPTER V - Conclusion & Further Work

It is hard to be able to evaluate the success of the project as unlike Photorealistic Renderings, NPR has no absolute goal. Photorealistic Rendering can be compared to real life counterparts and checked to see how close they match, whereas a painterly renderer can hardly be compared to a painting and checked to see if each brush stroke compares exactly. It is the artistic style that is to be recreated and art is subjective.

The goal of this project was to recreate a shader that looked reminiscent of an old cartoon backdrop. I believe I have both successful in some areas but not successful in others. I have been able to produce a shader that does indeed produce a look as if it has been painted. It still may look too structured and not stochastic enough, and maybe still a bit too procedural, but the foundations of a good shader are set down.

One area that I was not successful with is that I didn't manage to produce a scene that would compare to a jungle book backdrop. Due to the complexity of such a scene, and the processing power/time required to render a single frame containing possibly 20 or 30 instances of my shader, I was not able to produce a single render of such a scene. This is due to the fact that overcoming limitations in my shader and the techniques that I employed, many layers of brush strokes (50 for very detailed images) were painted onto each texture resulting in an exponentially long rendering time. This is fine for simple scenes involving a single sphere, which was used during the developmental stage of shader generation.

My shader is not complete, and certainly not without flaws, but it does have some of the main principles within it for being able to create a painterly renderer. With extra work and care the shader will only but look more impressive. The following are possible direction of further work and improvement:

- **Stochastic Brush Placement** - A more complete stochastic brush placement – each brush stroke is confined to it own cell index, if it wants to go beyond this boundary it becomes clipped and it is noticeably so. This nature relieves a heavy amount of work the shader has to do but I'm certain that a more accurate but less efficient method could be found.

- **Displaced Edges –** Brush strokes are "painted" onto the 3d surfaces. This creates an effect much more like the deep canvas technique that was used for the 1999 film Tarzan. This means that the shader would not replicate a background too accurately because object edges are always sharper than would be expected from a hand-painted backdrop. But this is one of the limitations of a 3d shader. Meier's technique, which takes 3d data and generate 2d images which are then used to create the final 2d painterly effect, is an indirect method and has the advantage that every frame could look like they have been hand painted. This direct method always retains its 3d look and feel.

- **Z-Depth Stroke Size –** It would have been a nice idea to alter brush size based on if objects were in the foreground or background. Closer objects would require smaller brush strokes to obtain greater detail whereas objects that are further away, could use larger brush strokes. This idea was not considered in depth due to the fact temporal coherence between frames may have been affected.

- **Efficiency –** Further development could and should go into making the image as efficient as possible, one possible solution to this is to create each layer with overlapping brush strokes, as a limitation of the current method clips strokes extend beyond its “grid space”.

REFERENCES

GOOCH, B. & GOOCH, A., 2001. *Non-Photorealistic Rendering*.

GREEN, S. et. al., 1999. *Non-Photorealistic Rendering* (SIGGRAPH ’99 Course Notes.) New York: ACM Press.

IAN STEPHENSON, *Essential Renderman, 2nd edition.*

L. GRITZ & A. APODACA. *2000*, *Advanced Renderman*

T. STROTHOTTE & S. SCHLECTWEG, *2002, Non-Photorealistic Computer Graphics*

KAUSHIK PAL, 2004, A shader Based Approach to Painterly Rendering