Imperial College London
Department of Computing

Surface Modeling of Impasto Paintings

Leon Demetriou

June 2010

Supervisor: Dr. Simon Colton
2nd Marker: Dr. Iain Phillips

Abstract

Computer generated art has come a long way since the inception of graphics back in the 1960s. Non-photorealistic rendering techniques have managed to simulate the look and feel of natural media such as paints and canvases to a great extent. Despite this, it is still very difficult to get the *impasto* touch of real paintings. This occurs when paint is laid onto the canvas in thick strokes forming a 3D texture. Without such texture information digital art lacks the required elements for a completely realistic rendition. The aim of this project is to enhance an existing tool developed at Imperial College with a viewer that is capable of taking depth information into account to produce a three-dimensional model of the painting surface. Once we have built a surface model of the paint we are able to experiment with numerous lighting models. To further enhance the experience for users of the system, we have incorporated head tracking using ordinary web-cameras to simulate changing lighting conditions as a person alters their position relative to the canvas. Our proposed solution can be extended for use in digital galleries for displaying computer generated art.

Acknowledgments

Firstly I would like to thank my supervisor, Dr. Simon Colton for giving me the opportunity to work on such an interesting project together with all his help and constructive feedback along the way. I would also like to acknowledge my second supervisor, Dr Iain Phillips for his advice. Finally, I want to thank my friends and family for their continuous support over the years during my studies at Imperial College.

Contents

1 Introduction

1.1 Motivation

Simulation of art materials and artistic styles is a very important part of NPR[1]. This is an area of computer graphics that focuses on using a range of expressive and creative ideas for use in digital art [23, pages 3-10]. Photorealistic computer graphics, on the other hand, tend to be involved with algorithms that aim to reproduce reality as far as possible. In many cases they attempt to reproduce physical laws to match the exact output of the camera.

During ancient times, the Egyptians used to portray people on papyrus or linen, in a manner that deviated from reality. Given this freedom, they were able to convey more powerful messages since their drawings contained more information for study. Even now, medical and technical diagrams enhance certain aspects of an image in order to highlight features that are more important within that context. NPR techniques have a number of goals ranging from artificial intelligence to enhancing legibility.

We are interested in taking digital art produced from a range of rendering systems and adding further texture information to enhance its appearance and level of realism. Although we aim to produce surface models of the canvas that are a close match to an actual painting, we should not restrict ourselves to this domain alone. Once a desired surface model has been constructed, we will illuminate the surface of the painting to simulate the impasto look of thick paint. Most systems that perform this task do so in a static manner, so that the image will be generated from a fixed viewpoint and then exported to a particular file format. Rather than following an inactive methodology, we aim to allow the user to interact with the canvas in a 3D environment so that the lighting conditions change dynamically. We will also incorporate head tracking in our system so that the position of the viewer can be recorded and shading of the canvas altered accordingly. To give the reader some idea of what we are attempting to achieve, a sample of our work is shown in Figure 1.1. The images were produced using a painting generated by the *The Painting Fool* NPR rendering system (section 2.7.4).

[1] Non-Photorealistic Rendering

(a) Impasto look (b) Shiny impasto look

Figure 1.1: Impasto look for paintings

1.2 Contributions

Our project on surface modeling of impasto paintings explores a new method of viewing computer-generated art. The final product is a complete Java application, called TRePS[2], that enables the user to import paintings and build a detailed surface model of the paint on the canvas. This model can then be passed to a viewer that uses the extra 3D information to enhance the realism of the image. The following include the main components of our developed application:

- **Surface Modeler:** This is the core part of the TRePS system. Using the surface modeler the user has the capability of specifying different ways of building up the height of the painting. We use an image filter pipeline to achieve this by taking each brush stroke laid on the canvas and composing the overall thickness in a step by step manner. Once this has completed, we can save the result either as a height or surface normal texture map. These textures encode 3D information, that was not previously available, inside the RGB colour channels of an image.
- **Shading Models:** Using the textures generated from the surface modeler we are able to shade the canvas using various standard graphics lighting models. We have implemented both Gouraud and Phong illumination (section 2.2) by using the Java 3D and GLSL APIs. A depiction of the realism of shading can be seen in Figure 1.1 above.

[2]Thick Realistic Paint Surfaces

- **Motion Tracking:** Tracking viewer coordinates is important since we want to simulate altering the illumination of the canvas based on their position. We integrate the head tracking module with the Phong shader, by using OpenCV as the backbone for this task. Doing so enables us to sit a viewer in front of a monitor with a web camera installed, giving them the experience of being in an actual art gallery.

1.3 Report Structure

In this section we give a brief overview of the content available in the chapters to follow.

- **Background:** Chapter 2 of the report contains the background research necessary to understand the techniques we will use to develop our application. We provide descriptions of a number of surface modeling methods, a range of shading models as well as the Viola-Jones object tracking framework which we use for our own motion tracking module. In this chapter, we examine some NPR rendering systems that cover some of the areas of interest related to our project. *The Painting Fool* is also covered here since we improve the level of realism of the paintings it produces with the aid of TRePS.

- **Design Considerations:** This chapter focuses on our design and programming considerations which we make use of during implementation. Within this chapter, the reader will get an understanding of why we have chosen certain techniques and how these impact the development of our system. An overall system architecture is presented so that the main parts of TRePS can be comprehended. We describe in depth how brush strokes images are converted into the overall height field for the canvas and how this is then used for illumination purposes.

- **Implementation:** Chapter 4 describes in detail the three core parts of the project and how these were created using our design criteria from the previous chapter. We also present the development of graphical user-interface of TRePS and its integration with the backend of the system.

- **Experimental Design:** This section of the report examines what types of experiments were carried out to evaluate TRePS. Reasons for choosing each type of test are explained and where necessary we describe the conditions under which they have been conducted. We include both a quantitative and qualitative analysis of the TRePS system. Getting people's opinions on the level of realism achieved, when embossing the paint surface, is of particular importance and a number of experiments have been set up so we can obtain relevant feedback for our results.

- **Results and Analysis:** Chapter 6 gives the results of the experiments described in the previous chapter. Using the feedback we have gathered, we analyse these and document what they mean in the context of the system we have developed.
- **Conclusion & Future Work:** The final chapter goes over a summary of what we have managed to achieve in this project. We highlight various limitations identified along the way and list some pieces of future work that can be done to extend upon our system.

2 Background

In this chapter, we examine the key ideas taken from graphics and computer vision that are necessary to understand the system we will develop. We begin by studying surface modeling techniques and how these can be extended to three dimensions. These surfaces are then texture mapped to increase their level of realism with methods such as bump mapping. We then discuss a variety of local and global illumination models before moving to a description of the Viola-Jones object detection framework. Finally, we examine 3D software APIs as well as key NPR rendering systems.

2.1 Surface Modeling

Within 3D graphics generated scenes there are numerous objects of differing shape and size. The majority of these are approximated using flat polygons (such as triangles). This occurs because simple polygons can be rendered very efficiently via the standard graphics pipeline (section 2.6.3) while performing operations such as shading and texture mapping. In this section we look at how we can encode such graphical surfaces.

2.1.1 2.5D Data Structures

Before moving to actual 3D geometrical structures there are 2.5D data structures that can make two-dimensional images look as if they were specified in three dimensions. Such data structures are known as G-buffers (graphics buffers). Below we present a few such buffers and how they can increase the realism of a 2D image by storing 3D information [23, pages 183-185]. Values in 2.5D data structures are stored on a per-pixel basis making all points in an image available within an interactive environment.

- **ID-buffer:** Different parts of the image (or surface in our case) can be assigned different identifiers. These can be used to distinguish various entities for selection and display by referring to information within this buffer.
- **z-buffer:** This stores depth information (z-coordinates) of the object representing its distance from the view plane. This information can be used to remove surfaces

that are occluded depending on their distance from the viewer and their relative positioning to other objects. They can also provide structural information regarding object contours. An example of a z-buffer is a height map that we will make use of in our system. This is a grayscale image that stores depth information in the RGB colour channels for each pixel.

- **n-buffer:** The n-buffer stores information regarding the surface normal of a particular point on the object. This is especially useful for shading the object using techniques such as bump mapping. A normal map is a type of n-buffer that we also make use of in our system. This is an image that stores surface normals in the RGB colour channels for each pixel.
- **material-buffer:** Within this buffer we encode information regarding the different types of materials found inside the image. Material properties can also be encoded within this structure.
- **shadow-buffer:** For each light source within the scene we can generate shadow masks so that we can use different lighting conditions for the image.

One way to represent surfaces is through the use of height fields [1, page 263]. A function for obtaining the height can be specified in the form $y = f(x, z)$ as shown in Figure 2.1. For instance, they are useful for obtaining altitude levels on the earth's surface using latitude and longitude.

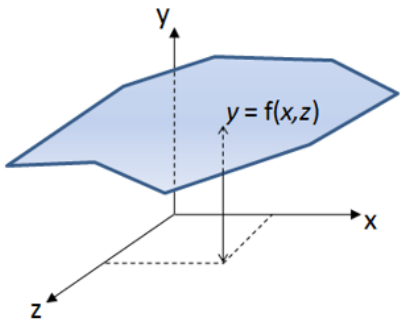


Figure 2.1: Height field

2.1.2 Polygonal Models

We now move onto methods where 3D geometrical information regarding the scene is used for specifying objects. The most common type of model used in computer graphics is that of a polygonal mesh [1, pages 262-264]. Such a mesh is composed of vertices,

edges and polygons connected to form the surface of the object. Each edge is shared by a maximum of two polygons and is formed by the connection of two vertices. A vertex is shared by a minimum of two edges whereas polygons are a closed connection of edges. A triangular mesh forming a dolphin is shown in Figure 2.2. As we can see from the mesh, such models are only an approximation of the object we are attempting to represent.

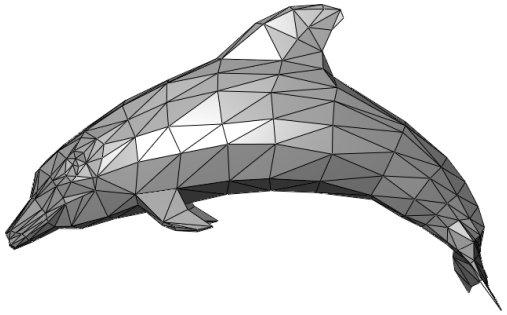


Figure 2.2: Triangular Polygon Mesh

2.1.3 Free-Form Surfaces

Specifying surfaces in terms of mathematical equations can generate more accurate geometry [23, pages 569-624]. Vertices within a polygonal model are directly connected together by straight lines (known as edges). In free-form surfaces, however, interpolation is used to calculate which points lie on the surface. Tensor product spline surfaces are a widely used method of describing free-form surfaces. A parametric curve of degree n, is shown in equation 2.1 where $\mathbf{c}_i$ defines the control vertices on the curve and u^i gives us the base function of the polynomial. The tensor product parametric surface of two such curves can be seen in equation 2.2. Figure 2.3 shows a single patch that is used to construct a much larger surface. Each patch has four control points at each corner which change the shape of the patch surface as they are displaced. When vertices are moved, neighbouring patches are also affected so that when all of them are joined together a continuous surface is constructed.

$$P(\mu) = \sum_{i=0}^{n} \mathbf{c}_i \mu^i \qquad 0 \leq \mu \leq 1 \tag{2.1}$$

$$P(\mu, v) = \sum_{i=0}^{n} \sum_{j=0}^{m} \mathbf{c}_{ij} \mu^i v^j \qquad 0 \leq u, v \leq 1 \tag{2.2}$$

Such free-form surfaces are then converted into a polygon mesh as we have described in the previous section. This allows the graphics hardware to render these efficiently through the standard pipeline. Different tessellation methods exist for polygonising the surface (usually in an adaptive manner) so that regions of higher curvature are represented with a greater number of polygons.

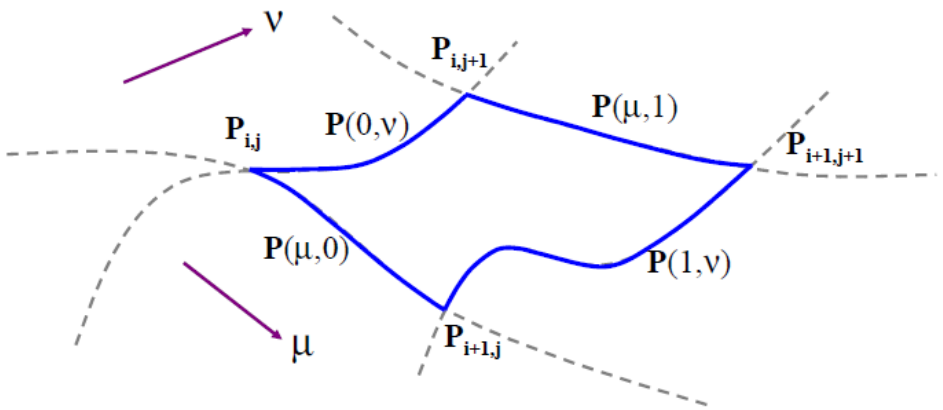


Figure 2.3: Parametric spline patch

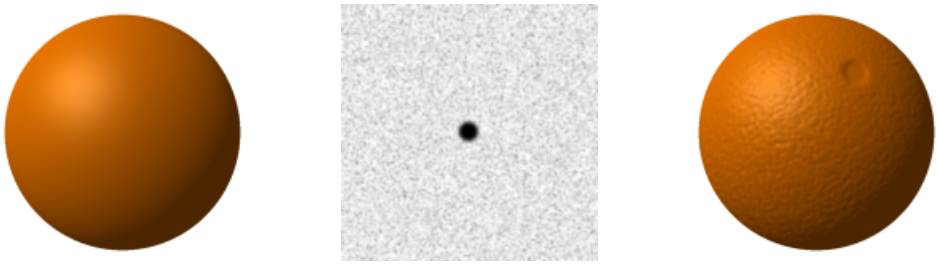
2.1.4 Texture Mapping

If we tried to render the geometrical models above we would see a wireframe representing the edges of connected vertices or a result similar to the mesh in Figure 2.2. To avoid increasing the number of polygons in the mesh we can use a technique known as texture mapping [1, pages 400-408] to improve the model. Textures are patterns that can be regular or complex in nature, which can be mapped onto a polygon mesh increasing the level of realism without adding extra polygons. Below we will focus on texture mapping methods that are relevant for increasing the level of surface detail.

2.1.4.1 Bump Mapping

If we were to model an orange using a polygon mesh and applied a shading model such as smooth shading, we would still find that it lacked realism. The reason for this is that a real orange has a bumpy surface whereas in our graphical model we have missed this level of detail. In order to achieve such effects, a method known as bump mapping can be used. Bump mapping was proposed by Blinn in 1978 [5] as methods at the time were unable to represent rough surfaces. This is a technique that can increase the complexity

of an image without increasing the geometrical complexity. The normal at any point on the surface defines the orientation of the surface at that particular point. As a result, by perturbing the normal at different points we can achieve small variations on the surface so that it appears more complex when we shade it. An example of bump mapping an orange to increase its complexity is shown is Figure 2.4. The amount of perturbation of the surface normal can be looked up in a 2D texture called a bump or height map. The partial derivatives of the texture indicate how to alter the surface normal as if the surface point was deformed by the height function.



(a) Spherical Orange (b) Bump Map (c) Bump mapped sphere

Figure 2.4: Bump Mapping

2.1.4.2 Displacement Mapping

Bump mapping is an illusion making the surface seem wrinkly and rough by perturbation of the surface normal. On the other hand, displacement mapping [1, page 487] actually modifies the geometry by displacing the surface points using a similar height function to bump mapping. This allows effects such as self-shadowing that are not possible with bump mapping. This technique, however, is more computationally expensive as it involves increasing geometrical complexity and generating new polygons for the mesh via tessellation algorithms.

2.1.4.3 Multi-texturing

The use of multi-texturing [1, page 418] is a powerful method that increases the realism of rendered scenes. An example can be seen in Figure 2.5. Most graphics cards can handle numerous textures at once using either a fixed or programmable pipeline. To achieve techniques such as bump mapping, we need to add a colour texture and normal texture to the polygon. Once this has occurred the shape can be illuminated to make it appear rough. If a normal texture was not used then as light rays strike the polygon

they would all reflect in the same direction, giving a flat appearance. Using an altered surface normal for each pixel changes the direction of the reflected ray, giving the illusion of 3D paint. More textures can be specified such as specular maps that define the level of shininess at each point on the canvas. We do not show these in Figure 2.5 to reduce clutter for the reader. Each texture is manipulated differently by the graphics card by specifying its type to the chosen graphics programming language. For instance each texel[1] contains either a colour value (colour map), normal vector (normal map) or shininess value (specular map). This is supported by both Java 3D and OpenGL. In a number of cases, textures need to be a power of 2 dimension in both width and height, since older graphics cards do not support NPOT[2] textures.

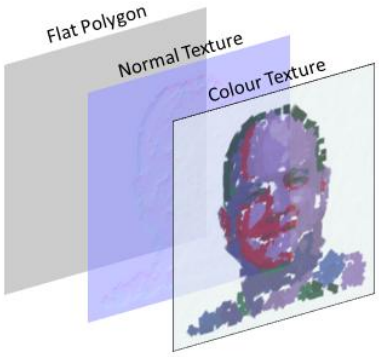


Figure 2.5: Multi-texturing

2.2 Shading Models

In this section we introduce some of the most popular shading models used in computer graphics. Such illumination techniques add further realism to the underlying three-dimensional geometrical model. These lighting effects work on flat polygons since this minimizes the computational effort required for shading.

2.2.1 Phong Shading

Rather than using precise physical models for light, we use a simpler approximation that is more computationally efficient. One such local illumination model was proposed by Phong [18] whereby the intensity of light calculated at any point on a surface is

[1] Texture Pixel
[2] Non-Power of Two

the result of a number of different light components. These light components are the ambient, diffuse and specular parts which are described below in further detail. Figure 2.6 shows the four vectors used to calculate the intensity of light at a particular point on the surface. The surface normal is represented by vector n whereas v represents the vector of the viewer with respect to that particular point. Vector l represents the direction to a light source and vector r is the path a perfectly reflected ray from l would take. We shall assume for the purposes of the equations below that these vectors have been normalized (vector size sums up to 1).

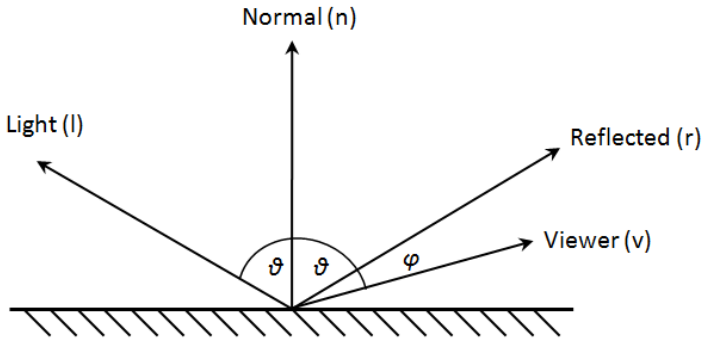


Figure 2.6: Vectors used in Phong shading

- **Ambient Light:** This component simulates light that is constant at every point in the scene. This does not take into consideration the position of the light source or the viewer and avoids the complexity of global illumination methods such as ray tracing and radiosity. Ambient light can be referred to as I_a while an ambient reflection coefficient k_a can be used to model how much of the ambient light is reflected.

- **Diffuse Reflection:** Surfaces magnified at the microscopic level are not entirely flat. Perfectly diffuse reflectors scatter incoming light equally in all directions so that they appear the same regardless of where they are viewed from. Rays of light striking such rough surfaces can be modeled mathematically using Lambert's law which states that the intensity of radiated light in any direction is proportional to the cosine of the angle between the light source (l) and the surface normal (n). From Figure 2.6 we can calculate $\cos\theta$ from the dot product of the normal and light vectors $(l \cdot n)$. Adding a diffuse reflection coefficient k_d as in the ambient case, we obtain the diffuse component as $k_d(l \cdot n)I_i$. The problem with the derived equation is that it will become negative given angles outside the range $0°$ to $90°$. In order to avoid this in practice, we use $\max((l \cdot n), 0)$ instead of $(l \cdot n)$.

- **Specular Reflection:** The last part of the equation involves highlights reflected off shiny materials. Such specular reflections are a different colour in comparison to ambient and diffuse lighting. In contrast to a diffuse reflector a specular surface is considered to be smooth, such as in the case of a mirror. The amount of light that a viewer sees depends on their positioning and that of the reflected ray. The cosine of angle ϕ can be calculated from the dot product of vector r and v. The Phong specular equation becomes $k_s I_i (r \cdot v)^q$. The specular reflection coefficient is represented by k_s whereas q defines the shininess coefficient. As q is increased, specular highlights are concentrated in a narrower areas. According to [1, page 297], as q approaches infinity we have mirror like surfaces; values ranging between 100 to 500 model metallic surfaces and values below 100 represent objects with broad highlights. In order to avoid negative values, as with the diffuse case, we use the $\max((r \cdot v)^s, 0)$ in the equation.

Adding the ambient, diffuse and specular components of light together, as described above, we obtain the entire Phong equation 2.3.

$$I_{phong} = k_a I_a + k_d I_i (l \cdot n) + k_s I_i (r \cdot v)^q \tag{2.3}$$

With the Phong shading model, normals are interpolated across each polygon. Using bi-linear interpolation we can obtain the value of normals at each point on the polygonal mesh. These can then be used to make an independent shading calculation for the Phong shading model as we have the required vectors as illustrated in Figure 2.6.

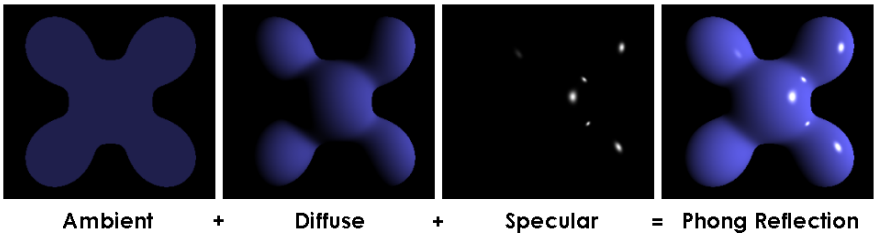


Figure 2.7: Phong light components

2.2.1.1 Blinn-Phong Shading

In Phong shading one must recalculate the angle between the reflected ray and the viewer (angle ϕ in Figure 2.6). This becomes computationally expensive when this has to be performed for each point on the surface.

James Blinn [4] proposed an approximation to the original Phong model that uses the halfway vector between the viewer and the light source (Figure 2.8). Angle ψ formed

between the halfway vector and the surface normal is half the size of angle ϕ. We can replace $(r \cdot v)$ in the original equation with $(n \cdot h)$ to avoid the continuous recalculation of the reflected ray. With a smaller angle in the Blinn-Phong case, we need to alter the shininess coefficient to achieve similar specular highlights between the two shading models so that $(r \cdot v)^q$ is closer to $(n \cdot h)^{q'}$. The reason why the Blinn approximation is faster to compute in comparison to the original Phong model is due to the fact that if we consider the viewer and light source to be an infinity then the the halfway vector is calculated only once and reused in the scene. With the original shading model proposed by Phong, the reflected ray has to be recalculated for each point on the surface as it depends on the surface curvature.

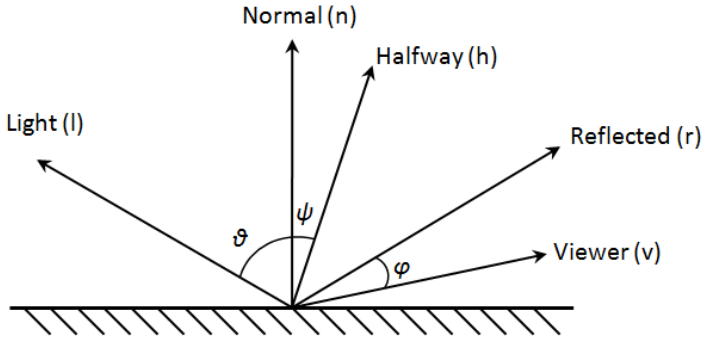


Figure 2.8: Vectors used in Blinn-Phong shading

2.2.2 Gouraud Shading

Another shading model that is less computationally expensive in comparison to the Phong model was proposed by Henri Gouraud [12] in 1971. To achieve smooth shading, the surface normal at each vertex is calculated by averaging normals of the polygons that meet at interior vertices of the polygonal mesh (Figure 2.9). This is different to what is performed in the Phong shading model where interpolation is performed across the entire polygon rather than at each vertex. Although Gouraud shading is faster than Phong shading, we cannot model specular highlights accurately as surface normals do not exist at each particular point.

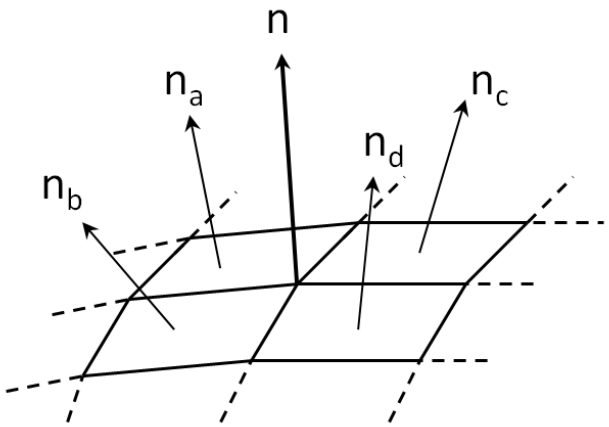


Figure 2.9: Gouraud normal interpolation

2.2.3 Flat Shading

Flat illumination shades each polygon in a scene based on the angle formed between the surface normal and the light source. This causes the entire polygon to be shaded with a constant colour. Although much faster than Gouraud and Phong lighting models, it gives polygons a faceted look and is unsuitable for obtaining a smooth shade across an object (Figure 2.10). The reason for this is that the human visual system is extremely sensitive to small changes in light intensity and we see stripes called *Mach bands* along the polygon edges [1, page 306].

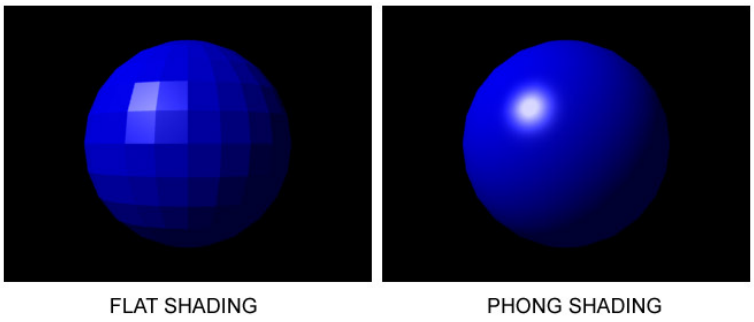


Figure 2.10: Flat shading versus Phong shading

2.2.4 Cook-Torrance Shading

In certain cases the Phong and Gouraud shading techniques produce results that appear artificial. To improve upon these models, a more detailed shading technique was proposed by Robert Cook and Kenneth Torrance [9] aimed for rough surfaces of metallic and plastic objects. While the Phong shading model calculates the amount of specular reflection using simple mathematical formulae, the Cook-Torrance method is based on physics by assuming that any surface is a collection of planar microfacets (microscopic facets) that act as perfectly mirror-like reflectors.

$$R_s = \frac{Geometric \times Roughness \times Fresnel}{(Normal \cdot View) \times (Normal \cdot Light)} \tag{2.4}$$

Equation 2.4 illustrates how to calculate the specular term R_s. The denominator is a cross product of the dot product of the normal and view vectors and the dot product of the normal and light vectors. The main part of the equation consists of the three components in the numerator which we will describe in more detail below.

- **Geometric Term:** The geometric attenuation factor is used to model the way light interacts with each microfacet of the surface. Two important effects of light interaction are masking and shadowing which involve computation of incoming and outgoing energy for the surface being shaded. Masking involves the blocking of a portion of the outgoing beam such as the reflected ray by another microfacet. Shadowing on the other hand involves the blockage of an incoming ray for instance directly from the light source. The geometric configurations can be used to analyse how much light actually escapes from the surface.

$$Geometric = min\{1, \frac{2(n \cdot h)(n \cdot v)}{v \cdot h}, \frac{2(n \cdot h)(n \cdot l)}{v \cdot h}\} \tag{2.5}$$

$$Geometric = min\{1, G_{masking}, G_{shadowing}\} \tag{2.6}$$

- **Roughness Term:** This is a statistical model of the microfacet orientations in the same direction as the half-way vector. The half-way vector h, as described in the Phong section, is the vector half-way between the view and light direction vectors. Only microfacets facing the same direction as the half-way vector will therefore contribute to the reflected energy. The roughness term (equation 2.7) is based on the complex Beckmann distribution that covers a wide range of materials of varying roughness with considerable accuracy. In the equation below a represents the angle between the surface normal and half-way vector while m is the root

mean square slope of the microfacets. The issue with this distribution is that it is computational expensive to evaluate and as a result not suitable for intensive usage. To avoid such problems, simpler Gaussian approximations can be used that work well in a most cases.

$$Roughness = \frac{e^{-(\frac{\tan a}{m})^2}}{m^2 \times \cos^4 a} \tag{2.7}$$

- **Fresnel Term:** This term forms a complete analysis of the reflection process by considering light as an electromagnetic wave. Using this effect, the reflected specular light is not a constant colour but view dependent. The behaviour of the Fresnel term is determined by the index-of-refraction of the material and also explains why many materials appear as mirror like surfaces when light strikes them at a grazing angle. The complete Fresnel equations are quite complicated and also expensive to compute. An approximation was proposed by Christopher Schlick [20] that is a non-linear interpolation between the refraction at normal incidence and the total reflection at grazing angles. Equation 2.8 demonstrates this approximation, where θ is the incident angle ($half \cdot view$) and F_0 represents the Fresnel reflectance when $\theta = 0$ (normal incidence).

$$Fresnel = F_0 + (1 - F_0)(1 - \cos\theta)^5 \tag{2.8}$$

2.3 Global Illumination Models

The shading models described previously are all local models in the sense that each polygon is shaded independently. This allows them to fit well into the graphics pipeline and be carried out in real time. The disadvantage of these methods is that they cannot correctly simulate phenomena such as reflections, shadows and refraction. For that extra realism, global lighting models are required. We will briefly describe two popular global illumination techniques known as ray tracing and radiosity. Further information regarding these global illumination methods can be found in [1, pages 625-658].

2.3.1 Ray Tracing

This model is suitable for producing photorealistic scenes especially if they contain highly reflective materials. A technique known as ray casting was first introduced in 1968 by Arthur Appel [2]. The basic idea behind ray casting is to fire rays from the

viewer for each pixel and find the closest object blocking its path. Once this has been found, the light sources and material properties can be used in conjunction with local illumination models described previously to calculate the of that particular pixel. This was further improved by Turner Whitted in 1980 [25] using ray tracing that could simulate specular reflection and refractive transmission as well. The approach was similar to ray casting but rather than stopping once a ray intersects an object, this is traced further by generating secondary rays of reflection, refraction and shadow. Rays are traced for each pixel of the camera for reasons of efficiency to avoid tracing rays that might eventually not reach the viewer. This process can continue recursively any number of times further adding realism to the scene. Although ray tracing produces some stunning scenes (Figure 2.11[3]), it is computationally expensive and not well suited for real-time applications.

Figure 2.11: Ray tracing

[3]This image was created by Gilles Tran with POV-Ray 3.6 using Radiosity. The glasses, ashtray and pitcher were modeled with Rhino and the dice with Cinema 4D.

2.3.2 Radiosity

Another global illumination technique is that of radiosity, which is based upon the concept that all energy in a scene is conserved. A scene is broken up into small flat polygons, otherwise known as patches, that are assumed to be perfectly diffuse. For each pair of these patches a form factor is calculated which represents a coefficient of how well these patches can receive energy from one another. This will depend on their distance, orientation and whether other patches block light from being transferred between the two patches in question. We therefore consider each patch to be a light source with both an emissive and reflective component. The emissive part is considered to be a constant across the patch and will apply to light sources whereas the reflective component sums up all the radiosity received from all other patches in the scene. The amount of accumulated radiosity reflected off a patch depends on its reflectivity (ρ_i). The radiosity of a particular patch (b_i) can be summarized in equation 2.9.

$$b_i = e_i + \rho_i \sum_{j=0}^{n} f_{ij} b_j \tag{2.9}$$

Once the form factors between patches have been calculated, we have a set of linear equations for the radiosity of each patch. These equations are solved iteratively so that each iteration produces intermediate radiosity values for a patch. This simulates how light bounces off the various surfaces to illuminate the scene in a number of steps. Eventually these will converge to the final radiosity values. Figure 2.12 (created by Hugo Elias) illustrates how a global illumination method such as radiosity increases the level of realism in comparison to local shading.

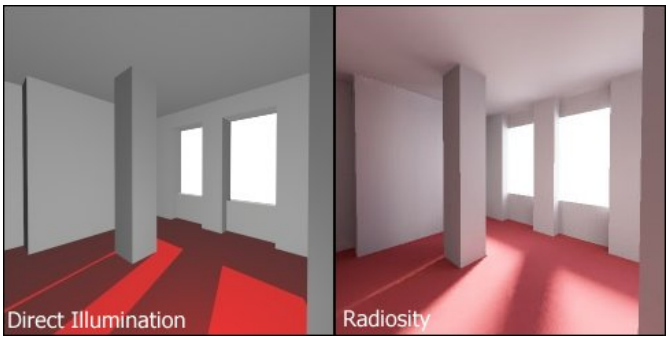


Figure 2.12: Local illumination versus radiosity

2.4 Viola-Jones Algorithm

Most lighting models described in the previous two sections take into account the positioning of the viewer to calculate specular highlights. Once a surface model of the painting has been constructed, we will be able to track coordinates of the observer and input these to modify the canvas illumination. An important framework designed for motion tracking is the Viola-Jones object detection algorithm. We will give a brief overview of this framework in this section as it forms an integral part of the system we have designed.

This is a framework proposed in 2001 by Paul Viola and Michael Jones [24] that is capable of achieving real-time detection rates. Although it can be trained to work on general object detection its primary usage is face detection. The framework consists of three main components namely feature types, learning algorithm and the cascade architecture.

- **Feature Types:** The main principle behind the Viola-Jones algorithm is to scan a sub-window capable of detecting faces across an input image. To avoid the time consuming approach of having to rescale the input image to different sizes and run the fixed size detector through these images, an adaptive face detector was devised. This is the so called integral image detector which rescales itself (rather than the input image) and is run many times through the image. Although one may assume that the same processing time is required this is not the case as the Viola-Jones detector is scale invariant that performs the same number of calculations regardless of the size of the input image.

  We will now describe how the scale invariant detector (integral image) is constructed. This is done by making each pixel in the input image equal to the entire sum of all pixels above and to the left of the selected pixel. Using this approach, Viola and Jones have demonstrated that the sum of pixels inside rectangles of any size can be calculated in constant time. The face detector used in the Viola-Jones algorithm uses features consisting of two or more rectangles as shown in Figure 2.13. Each feature is given by a single value and is calculated by subtracting the sum of the white rectangles from the sum of the grey rectangles. Viola and Jones have identified that a face detector of resolution 24×24 gives satisfactory results. Given all possible feature combinations, 160,000 can be generated in the 576 pixels contained in the base size detector. Features can be thought of as the computer's method of perceiving an input image for face detection.

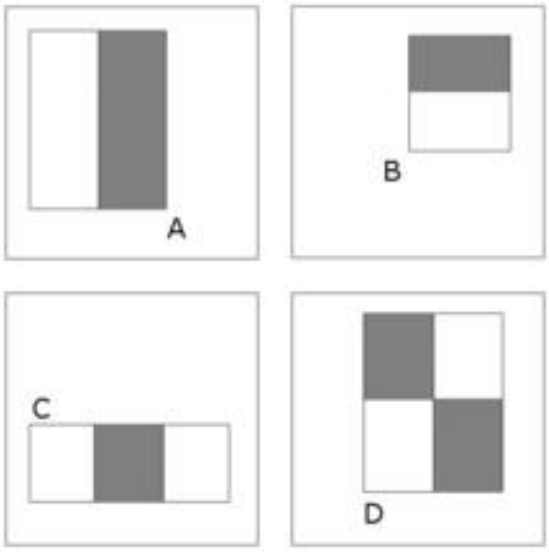


Figure 2.13: Viola Jones Feature Types

- **Learning Algorithm:** The next goal of the Viola-Jones method is to construct a mesh of features that are capable of detecting faces. To achieve this task, Viola-Jones use a modified version of the *AdaBoost* [11] machine learning boosting algorithm. This is capable of producing a strong classifier via a weighted combination of weak classifiers. The training procedure involves evaluating each feature on the samples to find the best performing feature, which is selected based on the weighted error it produces.

- **Cascade Architecture:** Classifiers generated by the *AdaBoost* learning algorithm are still not fast enough enough for real-time object detection. To speed up this process classifiers are arranged in a cascade in order of their complexity. Each successive classifier is therefore only trained with the samples that pass through the previous classifiers in the cascade (Figure 2.14). If during processing a classifier rejects the sub-window, no further computation is performed and the search moves onto the next sub-window. This means that rather than finding faces directly, the Viola-Jones algorithm discards non-faces. The reasoning behind this is due to the fact that it is easier to reject a non-face that detect a positive face. As a result, as a sub-window passes through each stage of the cascade (being classified as a maybe-face) its probability of actually being a face increases.

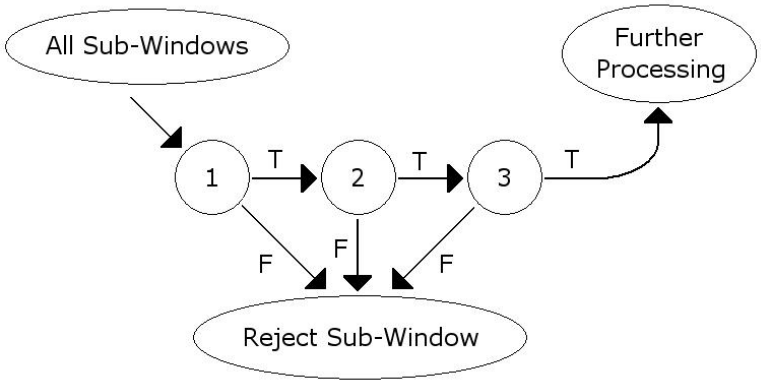


Figure 2.14: Cascade Architecture

2.5 Edge Detection

Edge detection is used to find object contours within images which is performed by identifying the rate of change of intensity at each pixel. Sharp intensity changes form strong evidence for the presence of an edge. Such a process is important to our system since we will use it to identify paint ridges in each brush stroke so that we can emboss the height map. A list of common edge detectors [10, pages 131-157] along with their respective kernels in the x and y directions are shown below. These are multiplied with the respective pixel intensity to calculate the edge gradient (equation 2.10).

$$G = \sqrt{G_x^2 + G_y^2} \tag{2.10}$$

Roberts Operator

$$G_x = \begin{bmatrix} +1 & 0 \\ 0 & -1 \end{bmatrix}, \qquad G_y = \begin{bmatrix} 0 & +1 \\ -1 & 0 \end{bmatrix} \tag{2.11}$$

Prewitt Operator

$$G_x = \begin{bmatrix} -1 & 0 & +1 \\ -1 & 0 & +1 \\ -1 & 0 & +1 \end{bmatrix}, \qquad G_y = \begin{bmatrix} +1 & +1 & +1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix} \tag{2.12}$$

Sobel Operator

$$G_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix}, \qquad G_y = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \tag{2.13}$$

Difference of Gaussians

From neurophysiological experiments carried out by David Marr and Ellen Hildreth [16], it was concluded that object boundaries were the most important cues linking the intensity of an image with its interpretation. The Marr-Hildreth edge detector uses the second derivative of the image to find a zero crossing position which is more accurate than searching for an extremum as in the case of using the first derivative. Equation 2.14 is the *Laplacian of Gaussian* function and G is shown in equation 2.15. The intensity function of the pixel is calculated by $f(x, y)$ whereas ∇^2 represents the Laplacian difference operator.

$$LoG = \nabla^2\{G(x, y, \sigma) \times f(x, y)\} \tag{2.14}$$

$$G(x, y, \sigma) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{\frac{-(x^2+y^2)}{2\sigma^2}} \tag{2.15}$$

The benefits of the Marr-Hildreth edge detector is that larger areas are considered for each pixel in contrast to classical edge detectors and the influence of the point decreases with the σ of the Gaussian. Changing the value of σ can suppress less important edges which is useful when attempting to define what value is most appropriate for identifying stroke ridges. The human retina was also shown to perform similar operations to the $\nabla^2 G$ operator which can be broken down into row and column filters to obtain increases in computation speed. Rather than using a direct implementation of the Marr-Hildreth edge detector we can use the difference of Gaussian masks with different σ values. This in effect means blurring the input image twice with different radii and then subtracting the one blurred image from the other. The resulting image will contain the edges of the original input image. By modifying the radii for the Gaussian blurs we can eliminate certain less important edges.

2.6 Software APIs

2.6.1 Java 3D

Java 3D [21] is a high level scene graph API that uses graphics function calls from lower level APIs such as OpenGL. A programmer constructs a scene graph containing graphical objects, lights, sounds and behavioural interactions. These scene graphs are structured as trees containing the objects to be rendered. Each node in the tree can have multiple children but only a single parent. This data structure is then passed to the Java 3D engine to be rendered. An application or applet written in Java 3D constructs a number of scene graphs which are then inserted into a virtual universe. Java 3D also supports a wide range of input devices such as head trackers and advanced viewing features allowing exploration of the graphical scene. This makes it an easy to use interface for virtual reality applications without the need to write a lot of lower level code to achieve the required functionality. In addition to the above, the Java 3D API uses object-oriented concepts for graphics programming.

2.6.2 OpenGL

OpenGL (Open Graphics Library) is a cross-platform API used for defining two and three-dimensional graphics [1]. The original authors were Silicon Graphics Ltd. (SGI) in 1992 whereas now its evolution is in the hands of Khronos Group, a non-profit organisation for open-source API development. It is used for a wide number of applications such as virtual reality, CAD, CAM, flight simulation and medical imaging.

OpenGL is a low-level API so that the programmer needs to be familiar with the graphics pipeline. This API serves to hide the complexities of interfacing with 3D accelerators as well as their differing capabilities. As a result, if some functions are not supported by the hardware, OpenGL attempts to perform them in software instead. It offers a range of primitives such as points, lines and polygons which pass through the pipeline and are converted to pixels. The API also exposes a range of standard graphics operations including geometrical transformations, texture mapping, lighting, anti-aliasing and blending. OpenGL also forms the base for many high level scene graph APIs such as Java 3D described previously. Due to its popularity, there are a number of bindings for different programming languages such as Java (JOGL).

2.6.3 GLSL

GLSL (OpenGL Shading Language) is a high level shading language that follows a syntax similar to the C programming language. This language gives the programmer direct control over the graphics pipeline (Figure 2.15[4]) without the need to write low-level code such as assembly. There are two separate processors that handle shaders differently. One is the vertex processor that runs the vertex shader which contains data for the vertices such as their position, colour and surface normals. Operations on the vertices can include transforming their position, texture coordinate generation, colour computation and lighting per vertex. Vertices are processed one at a time as they pass through the pipeline. The second programmable unit is the fragment processor that handles operations of computing colours and normals on a per pixel level in addition to texture application and fog computation. For more information regarding GLSL the reader can refer to the "Orange Book" [19].

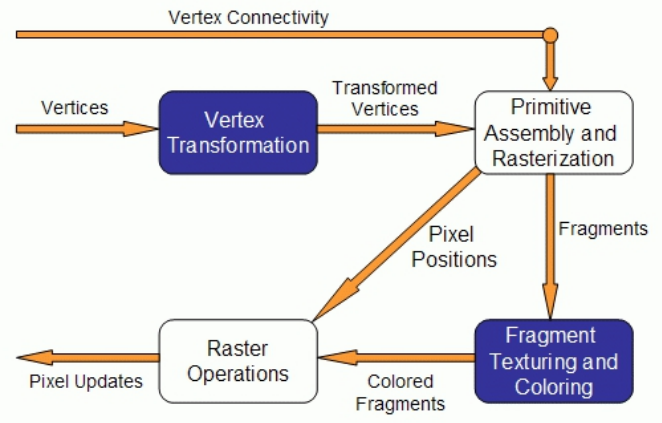


Figure 2.15: Graphics Pipeline

2.6.4 OpenCV

OpenCV [14] (developed by Intel) is a library of programming functions for real time computer vision. It is primarily written in C but there are Java wrappers available making it particularly suitable for our needs. It offers face detection based on the Viola-Jones object tracking framework along with other features such as gesture recognition, stereopsis and HCI. The library has four major search modes of operation when running face detection and these are listed as follows:

[4]http://www.lighthouse3d.com/opengl/glsl/images/pipeline.gif

- **Scale Image:** This option will downscale the image it is searching rather than "expand" the feature coordinates in the classifier cascade (section 2.4). This mode of operation cannot be used in conjunction with any of the other three.
- **Do Canny Pruning:** If it is set, then an edge detector is used based on the Canny operator to reject certain image regions containing edges that are out of range for the searched object. Based on the threshold values, these can decrease the amount of computation required and increase the detection accuracy.
- **Find Biggest Object:** When this mode of operation is selected then the search function attempts to locate the largest object within the image.
- **Do Rough Search:** This mode of operation should only be used when the Find Biggest Object flag has been set and the minimum number of faces to detect is greater than 0. If we have enabled this mode then the function does not search for possible objects of a smaller size once it has detected the face. Using a rough search strategy speeds up detection considerably and used especially in cases where processing needs to be done in real-time. The downside of using such a mode of operation is that accuracy is reduced as no further search is performed to detect further faces if the algorithm has detected at least one.

2.7 NPR Systems

2.7.1 ArtRage

ArtRage is an interactive painting package developed by Ambient Design Ltd that allows users to experiment with different artistic media to produce digital paintings. It simulates mediums such as acrylic and oil paints, spray paints, pencils, crayons as well as other tools ranging from tracing to smearing. *ArtRage* was designed to be used with a tablet PC although its intuitive interface allows mouse interaction as well. Figure 2.16 illustrates the software environment available to the user. The bottom left of the window contains numerous available mediums for drawing onto the canvas. Some excellent features of the software include the impasto feel of its paints together with its dynamic colour mixing model. More information regarding this tool can be found on the *ArtRage* website along with a free download of the software with a subset of working features (`www.artrage.com`).

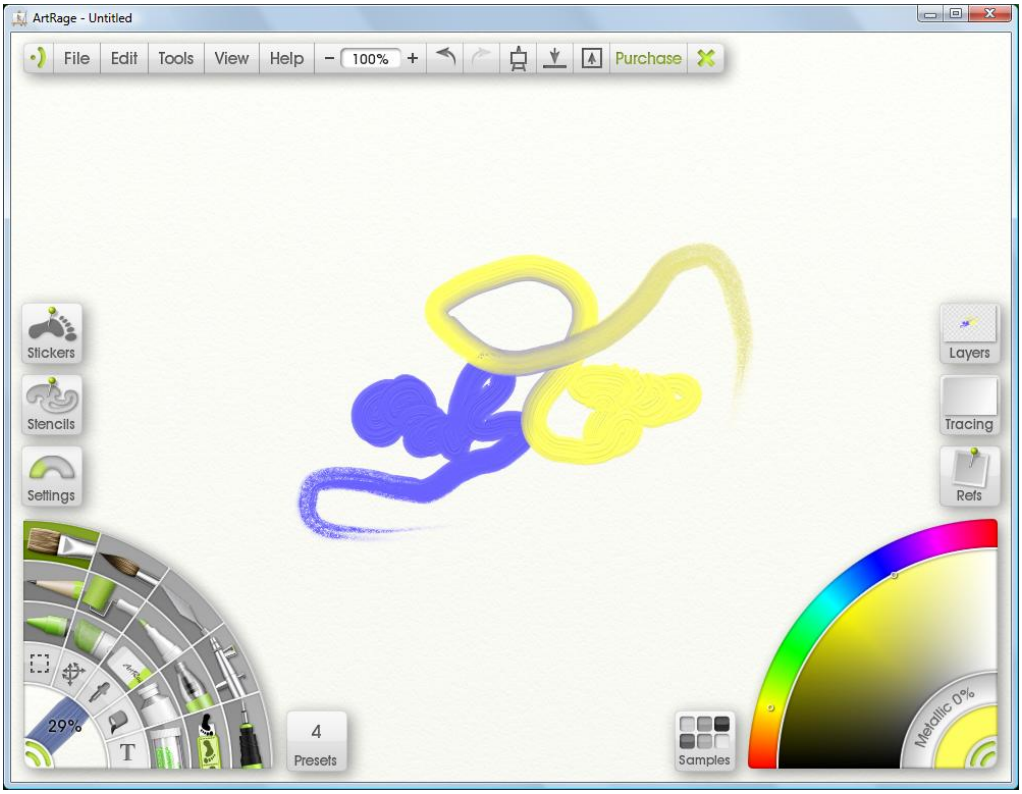
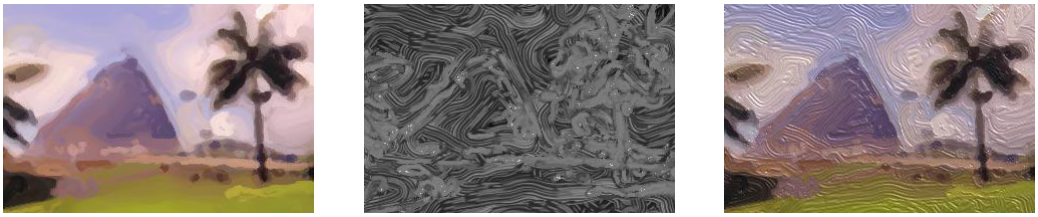


Figure 2.16: ArtRage 3 Environment

2.7.2 Fast Paint Texture

This is a method proposed by Aaron Hertzmann [13] for adding realistic paint effects to NPR images. While other systems that simulate the flow and buildup of paint on the canvas [3, 6] can produce realistic renditions, the computations are quite expensive due to the nature of fluid dynamics involved. In *Fast Paint Texture* the system takes a list of ordered brush strokes and composes an image. Each of those strokes are also assigned a height map and an opacity map. A number of these textures can be assigned to a stroke if desired and selected at random to produce more realistic and accidental painting effects. These texture maps are used to produce a height field for the entire painting using composition with ordinary alpha blending. Once this stage has completed, the painting is rendered with the use of bump mapping for extracting the surface normals and then lighting conditions computed with the Phong illumination model. The entire process for processing an entire painting with tens of thousands of strokes takes only a few seconds with modern graphics hardware. An example of the results of this method is demonstrated in Figure 2.17.



(a) Original painting (b) Height field (c) Painting with lighting

Figure 2.17: Fast Paint Texture

2.7.3 IMPaSTo

Baxter et al. [3] have produced a painting model that can be used in interactive painting systems with a range of artistic media. Creating an accurate model to simulate the entire physical behaviour of paint is challenging. To avoid such problems and to achieve real-time simulation, simplified physics along with a number of heuristics needed to be used in this system. Within this model the 3D paint surface is represented by 2.5D data structures of multiple height fields. This allows for one active wet layer and any number of dry layers of paint. The algorithm used when simulating the paint spreading across the canvas achieves realistic stroke texture. It conserves the overall paint volume and pigment mass while it interacts with the brush and painting surface. To further increase the level of realism a colour mixing and compositing engine for paint blending and layering has been implemented which is based on the Kubelka-Munk optical model. This avoids the limitations of the linear, additive RGBA colour space. Real-time performance is achieved due to a graphics hardware implementation using programmable fragment shaders. This realistic model has been incorporated into a prototype painting system called *IMPaSTo* providing users with a three-dimensional brush (tablet or haptic armature) to experiment with various painting styles.

2.7.4 The Painting Fool

This is a computer program developed at Imperial College London by Dr. Simon Colton. *The Painting Fool* is a non-photorealistic rendering system that takes a digital image and produces an artistic rendering of the image in a two-stage process [8]. In the first stage, it segments the image based on certain parameters to produce a list of segmentations. The second part of the process renders these segmentations using various simulated artistic media such as acrylic paints, oil pastels, pencils etc. *The Painting Fool* renders multiple painting layers by rendering each segment in turn. To achieve such renderings multiple curves are produced which are composed of numerous complex brush strokes.

We will now describe the painting format stored by *The Painting Fool* as it will form the basis of our system. *The Painting Fool* exports a folder which includes three important pieces of information. The first part includes an image displaying the painting that was created. A painting example by this rendering system can be viewed in Figure 2.18. The second item found within the folder is an XML file representing the brush strokes that were used to construct the image. *The Painting Fool* also exports a PNG image that contains a list of the images of the brush strokes that were laid onto the canvas. Using the XML representation we can extract the bounding box of each stroke in the respective PNG image and place these onto the painting. By doing so we are able to have access to the entire construction process of building a canvas.

Figure 2.18: Impasto Portrait by *The Painting Fool*

2.7.5 Wet & Sticky

We present another painting model that was first introduced in Tunde Cockshott's PhD thesis in 1991 [7]. An extension of *Wet & Sticky* [6] was to model textured and shiny paint using bump mapping and standard illumination techniques. As in the case of *IMPaSTo* [3], the role of this model is not an accurate simulation of reality even though it is based on real world elements. The aim of *Wet & Sticky* was to produce computer generated paintings with marks containing the same level of detail and complexity as real paintings while allowing for accidental effects at the same time. This model is based on the ideas of cellular automata where cells are arranged in the form of a two-dimensional grid. *Wet & Sticky* is composed of three parts, namely the *Paint Particles*, *Intelligent Canvas* and the *Painting Engine*. The *Paint Particles* represent the paint which contains attributes such as their colour and liquid content whereas the cells in the *Intelligent Canvas* hold information regarding their orientation and absorbency. In a nutshell, the canvas can be thought of an ice-cube tray where as paint fills up one cell it overflows into the neighboring cells (Figure 2.19). The *Painting Engine* controls how the state of the canvas changes which can be achieved either while an artist is working on it or through a self-evolution mechanism simulating effects such as paint ageing, gravity and liquid diffusion. Since the cells of the canvas contain paint volume information, as in a reservoir, these can be interpreted as a height map for the painting. This height map is then used as a texture for the perturbation of surface normals in a technique known as bump mapping. The surface canvas is then illuminated using the Phong shading model.

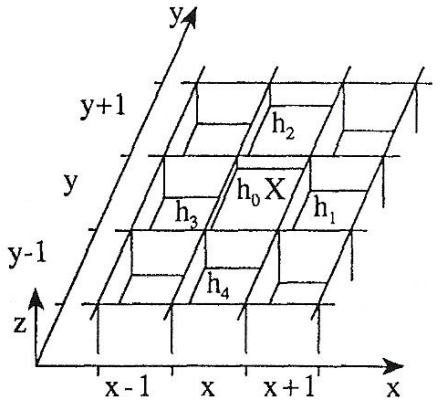


Figure 2.19: Cell Reservoir Model

2.8 Summary

We have covered a range of useful techniques that will prove useful during the design and implementation phases of our system. We have primarily focused on surface models and how these can be improved with the aid of texture mapping. A number of lighting models were described; the most important ones in our case being those involving local shading as these can be performed in real-time. To develop our motion tracking module we make use of the Viola-Jones object detection framework and therefore have a section describing its operation. To form a complete background research, we also examine some NPR rendering systems as well as relevant libraries and programming languages. *The Painting Fool* will form the basis of the system we will develop, as it generates the brush strokes that we will be emboss to form a textured surface.

3 Design Considerations

The design of TRePS (Thick Realistic Paint Surfaces) can be broken down into four main components. As we described in the introduction our main aim is to enhance two-dimensional images with 3D information to increase their level of realism. The first part will consist of designing a method to efficiently capture the surface roughness of the canvas. Once we have managed to formulate such a module we can then use this to experiment with different shading models to give the texture of the canvas a convincing look. Given an enhanced painting, we will then design a module that will enable tracking a person's motion to achieve dynamic lighting effects in real-time. Finally, we go over some design decisions involving the GUI developed for TRePS. In the following sections, we highlight the key design decisions for each module and present an overall system architecture.

3.1 Realistic Paint Texture

Within this section we describe the design of the core module for surface modeling of the paint canvas. Initially, we start of with our input which may either be a simple image or a specialised format exported by *The Painting Fool*, known as a stroke file (section 2.7.4). This file type can then be used to construct a detailed texture containing the height information of the paint.

3.1.1 Internal Representation

The reason we want to support multiple formats is to allow the backend of our system to be used in different scenarios. The obvious case is importing an ordinary image in formats such as JPEG and carrying out the necessary steps to obtain a surface model. Using plain images will pose a problem in most cases as it is difficult to infer the surface model by simply observing the colour of the paint. This is due to the fact that large depth changes will be difficult to detect without knowledge of how paint was laid on the surface. To tackle this issue we will need to use certain heuristics that are described in more detail in the coming sections.

The second format TRePS will support is that of *The Painting Fool* (Chapter 2, section 2.7.4). Figure 3.1[1] illustrates the design for storing strokes extracted from the exported XML file. Using a custom file reader that we will develop, we can retrieve each stroke image from its respective PNG file and store this directly as a BufferedImage in the *Stroke* abstract class. *SimpleStroke* objects store only a subset of the stroke data and are only used to define how to retrieve the actual brush image whereas *SmoothStroke* stores all the data including the intermediate points allowing to re-create the stroke if necessary. The *Painting* class is composed of the brush strokes extracted during the input phase and these will be stored in a LinkedList as they will be accessed in a serial manner. This class also stores the width and height of the canvas itself in pixels which is retrieved from the *painting* element in the XML file. A *Painting* class can be considered an extension to the Java *BufferedImage* class enabling us to work directly with paintings containing brush strokes. This particular class has the necessary methods (`addStroke(Stroke s)`, `getStroke()`) needed to interact with the strokes, while abstracting away its internal structure.

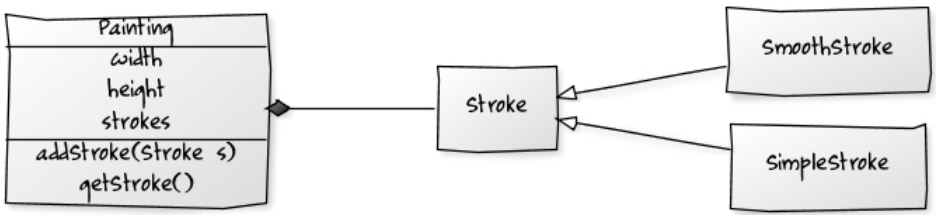


Figure 3.1: Internal Representation of Strokes

Although, we are working with the format supported by *The Painting Fool,* TRePS has been designed in such a way to be able to work with any NPR system that gives access to the strokes used to build the painting. By having each brush stroke used to construct a portrait we are able to process these in turn by generating their individual height field and then using this to add to the overall height map of the canvas. We have chosen to design TRePS so that all stroke images are buffered into memory before any processing begins to take place. Although this in effect demands a large amount of system memory we are able to manipulate the strokes much faster and can offer a quick undo stack. Based on sample files, with 512MB of allocated heap space we are able to store on average 50,000 strokes assuming each one has an approximate size of 10KB.

[1]UML diagrams were created using an online tool called yUML. More information can be found on their website at http://yuml.me

3.1.2 Stroke Processing

We will now describe how we take the brush stroke images and produce a height field to model how thick paint rises above the surface. The approach we follow is similar to that proposed by Aaron Hertzmann [13] but differs in certain key areas. Rather than creating pre-defined height and opacity maps and then using these as textures directly on the strokes, we will use edge detection to identify the ridges created on the brush strokes. The key reasons for using a form of edge detection to produce a height map for a stroke is as follows:

- Executing edge detection filters on the stroke image produces far better results than simply assigning a random texture map. This is due to the fact that we are given the stroke image prior to the calculation of height information and therefore have no control over where to emboss particular areas. The accuracy of edge detection will enable us to identify paint ridges so we can alter the height accordingly based on the colour information in the stroke itself. Since the region surrounding the stroke is transparent due to the PNG alpha channel, edge detection should work well and each stroke will obtain its own distinct signature rather than having one height map for all of them.
- Other solutions for height map construction attempt to reproduce physical laws as performed by *IMPaSTo* [3] and *Wet & Sticky* [6]. The issues faced by these applications is that despite the fact that they produce extremely realistic results, the real-time paint simulation is computationally expensive. To tackle this problem, they attempt to utilize graphics hardware and make use of parallelism. In our case, to avoid the processing overhead we make extensive use of image filters for edge detection. This is less costly to run and does not require any special hardware.
- Rather than running a complete fluid simulation, as done in some rendering systems, we use simpler edge detection filters (Chapter 2, section 2.5). By doing so we avoid having any constraints on the type of surfaces we create even though these may not appear realistic. A key component of this project is to also explore areas of creativity in digital art.

3.1.2.1 Ridge Identification

Edge detectors work well for finding object contours where there are steep intensity changes. However, the results produced are rather binary in nature since white denotes the appearance of an edge and black denotes its absence. We need a smoother method of moving from ridge to ridge to capture the entire impasto effect of the stroke.

One way to avoid the problem of binary edge detection is to pass the output of the edge detector to a blur filter that will spread out the height and produce a smoother effect. Although this is acceptable, a more accurate way of identifying paint ridges is to use both the grayscale texture of the original stroke and the result of the edge detector. Since we know the identifier of the stroke, we can calculate its ordering relative to other strokes placed on the canvas. As a result, as we process each stroke we raise the maximum value allowed for paint. Since we are using height maps this value will be between 0 and 255. According to the stroke ID, we can assign a particular threshold for each stroke within that range and use a mixing function that will take some portion of the height based on the colour and the rest based on the edge detection results. A relative weight can be applied to the two values in order to change the importance of grayscale colour or edges based on the types of strokes being processed.

We now describe how the paint buildup process works by applying the methodology above. Since the ID of the stroke defines its absolute ordering we can use a mathematical function to place it within the range of 0-1. This weight is then multiplied by 255 to scale in the range of a grayscale height value. We can also cap the height range within certain minimum and maximum values for height if we expect the paint to start from above the zero level of the canvas but never exceed the maximum possible height. Figure 3.2 illustrates how the paint buildup works using a variety of mathematical functions. The most common is the linear one as we do not bias strokes based on their ordering but in certain scenarios we may need to do so. For example quadratic and cubic functions give more importance to the height of strokes placed on the canvas at the end which usually includes fine detail. Logarithmic functions for instance build the height of canvas relatively fast using only a subset of the initial strokes and less height is then added for strokes arriving later on.

As we can see from the graph in Figure 3.2, strokes at the bottom start from a relatively low height and increased gradually as we move on to consequent brush strokes. Attempting to perform the opposite process would create a height map that would appear to push paint into the canvas. This is because as strokes (with lower height) are composed on top of other strokes (with a greater height field) the overall thickness of the paint will decrease. This is due to the fact that alpha blending is enabled. Rather than supporting this process directly, a colour invert image filter can be used that will reverse the grayscale values of the actual height map generated using the standard procedure described in the previous paragraph.

Despite the fact that we do not simulate physical processes as done in other methods, but rather use a more mathematical approach for each stroke, the height map of a stroke can be constructed relatively accurately. In addition it is relatively fast to compute the height of the canvas containing thousands of strokes.

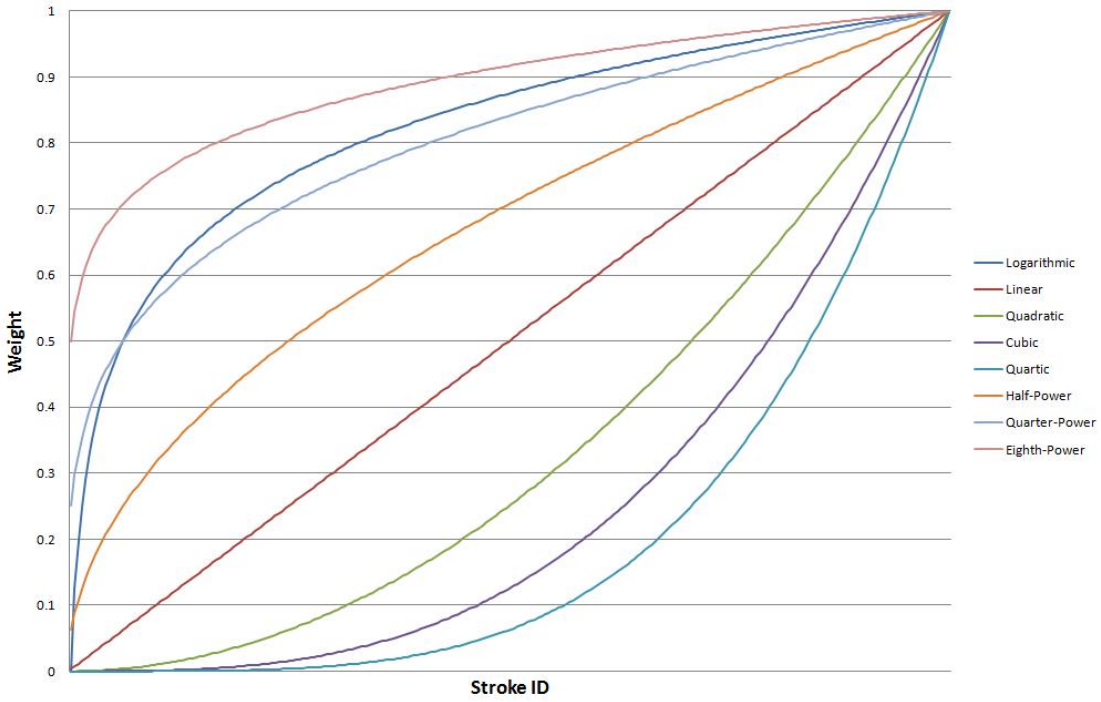


Figure 3.2: Paint Buildup

3.2 Shading Models

Given that we can now represent the surface of a painting using either a height or normal map, we can move onto the design of shading the canvas so that it appears realistic. We have chosen to use local shading models rather than global illumination, and bump mapping over displacement mapping. The key reasons for these choices are highlighted below:

- Local shading models such as Phong shading can produce extremely realistic results especially when a detailed texture map of the surface exists. We have already produced such a detailed texture and therefore can use the benefits of such models.
- Global illumination models such as ray tracing will not improve the realism of shading dramatically since we do not have a great amount of depth in the painting but only an *impasto* effect that raises oils and acrylics off the surface by a few millimetres. Such methods are also not fast enough for real-time performance and therefore not suitable for our needs.
- We use bump mapping rather displacement mapping due to its computational

efficiency. To represent the complexity of the painting surface using an accurate polygonal model would require tens of thousands of polygons which will become difficult to render unless expensive graphics hardware is used. Bump mapping can provide the same level of realism as we have the height value at each pixel along with the surface normal. The only effects not possible with bump mapping are occlusions and self-shadowing which are often not particularly visible in impasto paintings.

3.2.1 Illumination

To enable our solution to work on a range of graphics hardware, we have decided to build two shading models. One will be based on smooth shading and the other will be the Phong illumination model. The UML diagram of the shader design is shown in Figure 3.3. The abstract *Shader* class is the viewer that all other shaders must extend. This allows for further illumination models to be added in the future making the shader architecture extensible. It contains both a colour and normal texture image and contains the abstract method `createView()` implemented in both the Phong and Gouraud shaders.

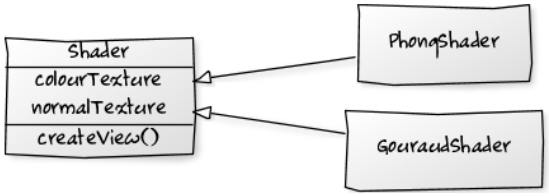


Figure 3.3: Shaders

3.2.1.1 Smooth Shading

Smooth shading (Gouraud) relies on the support of the graphics card to perform the dot product of vectors efficiently in hardware. The *GouraudShader* class will implement this functionality by performing the dot product of the surface normal (obtained from the normal map) with the light vector. This will produce a grayscale value that will then be combined with the colour from the colour texture to give the shaded value on the canvas. This is performed for all pixels on the painting surface to give the illusion of perturbations. The distance of the light and its position are variable and we can use the keyboard and mouse as input devices to change these in an intuitive manner. Moving the mouse across the screen can dynamically change the lighting position.

3.2.1.2 Phong Shading

We will now explain the design of a more realistic lighting model in comparison to Smooth shading described previously. To achieve this, we use OpenGL's high level shading language called GLSL (Chapter 2, section 2.6.3). Most fixed graphics pipelines use the Blinn-Phong shading model as an approximation to the Phong model. Since we have the capabilities to modify the fixed functionality of the graphics pipeline (Figure 2.15), we will use the Phong model in its original form. Using the detailed normal texture containing surface normals per texel, we can implement per pixel lighting rather than per vertex illumination. This is because we only have four vertices in our case (one polygon with four corners). As a result, the calculations for performing Phong illumination per pixel on the canvas will be handled by the fragment shader. We obtain both the normalized vectors for the viewer and light rays and then begin the colour calculation for each pixel. The surface normals are found from the normal map and we perform the usual mathematical calculations (Chapter 2, section 2.2.1). The ambient, diffuse and specular values are then multiplied by the actual texture colour to give the final output colour to be written to the polygon.

3.3 Motion Tracking

In this section we describe the design of the motion tracking module using Java and OpenCV (Chapter 2, 2.6.4). Motion tracking can be integrated into both shading models we have described above. In the sections below we show how mouse and camera input devices can be used for the task of motion tracking.

3.3.1 Mouse Coordinates Tracking

In the Smooth shading model we will make use of tracking the mouse coordinates on the monitor. As the canvas will be shown in full screen mode the (x, y) coordinates of the cursor specify the positioning of the light. Only a single light source is used so this avoids the complexity of tracking multiple objects. The user will be allowed to shift the mouse and shading on the canvas will be changed in accordance to the new position of the light. Since the mouse cannot provide z coordinates we enable the use of the keyboard to move the light source either closer or further away from the painting. Java offers numerous listener classes for capturing both mouse and keyboard events.

3.3.2 Head Tracking

Rather than tracking mouse coordinates alone we add another level of interaction for the user which involves head tracking with the aid of a camera. The Phong shading model adapts its shading based on the positioning of the viewer.

To be able to update both the lighting conditions on the canvas as well as monitoring the positioning of the viewer we need to run head tracking as a separate thread. The head tracking module will make use of a single camera and tracking will be performed in a window of fixed size. Since the viewing experience can only be made active for one user, we disable the detection of multiple faces and use only the largest object within the capture window. To obtain the coordinates of the viewer position, we use certain simple heuristics to detect the eye position. When we retrieve a sub-image representing a face we find the midpoint of the interpupilary distance by assuming it will be half way along the x-axis and one third of the way down from the y-axis. These coordinates are then transformed to a global level so that they map onto the canvas. As the viewer shifts position the head tracker picks up these changes and changes the positioning of the virtual camera within the rendering window whereas the light source remains static. The Phong shader will request viewer coordinates continuously from the head tracking module and any changes will be made visible on the three-dimensional canvas. The architecture for this is shown in Figure 3.4.

Figure 3.4: Motion Tracking Architecture

In order for head tracking to be performed accurately we need to perform some minor camera calibration. Firstly, the camera needs to be set up so that its centre position matches the centre position of the viewer when sitting in front of the painting. Any slight movement of the viewer should then be captured by the camera and these coordinates are sent directly to the shader. To allow complete calibration we have designed a dialog that gives the user capabilities to change all camera and algorithm options used for tracking. The calibration process should occur prior to viewing and in future releases of TRePS we may incorporate an auto-calibration feature. These are highlighted in the list below as follows:

- **Window Size:** This represents the width and height of the image that we capture from the camera. These will include resolutions of 640×480, 320×240 and 160×120. Larger window sizes require more processing time due to higher resolution.

- **Tracking Size:** The size of the integral image (Chapter 2, section 2.4) which is sweeped across the input image. This represents the area in which to perform head tracking and the minimum allowed is 20×20.
- **Classifier Type:** The classifier type denotes which XML Haar Cascade classifier (Chapter 2, section 2.4) to use. Depending on the type specified different features are searched for in the tracking window. We are most interested in profile and frontal faces as we aim to identify the positioning of a person's head.
- **Tracking Mode:** OpenCV offers a range of tracking modes including Canny Pruning, Image Scaling and Rough Search (Chapter 2, section 2.6.4). The most suitable mode for our case is Find Biggest Object as secondary faces should not be tracked and need to be eliminated from the search quickly.
- **Threshold:** This value ranges from 0 to 255 to filter out pixel values that are below the threshold set.
- **Scale Factor:** The scale factor ranges from 0 to 1 and takes the rectangle in which a face was detected and scales it down to a smaller area for identifying the positioning of the eyes.
- **Eye Positioning:** Once we have a scaled window in which to search for eye co-ordinates, we set up the heuristics described previously for calculating the inter-pupilary distance. Values of 0.5 for the x-axis and 0.3 for the y-axis will identify the centre of the eyes for a monocular view. Adjustments to these values can switch windows to both left and right eyes.
- **Painting Rotation:** The amount of rotation to shift the painting in both the x and y directions as the viewer changes position. This simulates the changing appearance of an object as we view it from different angles.

3.4 User Interface

To enable designers to create and illuminate paintings, we have designed a front-end user interface for TRePS. The sketch of the interface is shown Figure 3.5. This will developed in Java Swing. It contains four main components that will describe in brief below.

- **Toolbar/Menu:** This contains all application controls such as creating new paintings, adding filters, performing camera calibration and specifying the lighting model to use for illumination.

- **Filter Pane:** This includes both the stroke and texture filter panes. These are tree like structures representing the pipeline of filters that were used to create the surface model of the painting. Stroke filters are always run prior to the texture filters as these relate to the entire image. Support for adding, removing and changing the position of the filter within the chain is also supported within these filter panels. We separate the stroke and texture filters into separate trees so that the designer can easily distinguish their usage.
- **Image Pane:** To view the results of the height and normal maps generated we store the updated image within this panel. The texture is updated whenever a filter chain is executed so that it provides feedback for the designer.
- **Status Bar:** This panel displays the global application state by providing information such as painting load time, filter execution time and useful error messages.

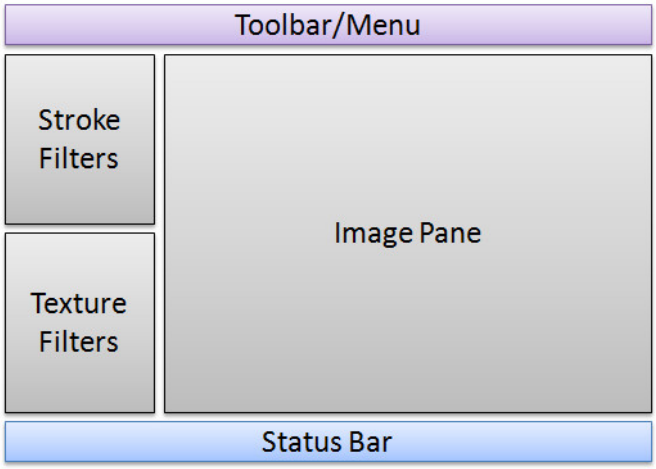


Figure 3.5: User interface design

3.5 Programming Considerations

In this section we go over some programming considerations to examine why we have chosen certain languages and libraries.

- **Java:** We have chosen Java as the core development language to build TRePS for a number of reasons. Firstly, it is open-source and cross-platform compatible which is a key requirement for the system we are creating. Using an object-oriented approach we are able to decouple the main components of the system,

keeping TRePS as modular as possible. Java Swing also gives us the necessary components to build the front-end GUI without requiring any external libraries.

- **Java 2D:** Offered as part of the standard Java development kit, Java 2D has the required capabilities of working directly with BufferedImages and image filters. It has compositing features such as alpha blending and will be used extensively in processing brush stroke images to create the height field for the canvas.
- **Java 3D:** This 3D API (Chapter 2, section 2.6.1) integrates well with the Java language and will be used for creating the three-dimensional canvas and illumination models. Java 3D will be used to form the core of the renderer.
- **GLSL:** OpenGL's shading language is used in conjunction with Java 3D to create one of the illumination models. GLSL can be integrated directly with Java 3D and allows us to separate our shader design from the application itself. Using GLSL also gives us a large improvement in performance since we rely on the hardware to perform the rendering rather than using some form of software emulation.
- **OpenCV:** Java wrappers exist for OpenCV and as a result we use it to build our head tracking module. We have selected this API as it is simple to use and robust in terms of face detection which is very important if we are to achieve real-time motion tracking.

3.6 Summary

In this chapter we have covered the key modules that form our system. We started off giving an in-depth description of how we model the surface of the painting canvas using edge detection along with certain other image filters. This enables us to build detailed texture maps of both height and surface normals. Using these texture maps our shaders (Gouraud and Phong) can perform an accurate illumination of the canvas surface. We have given an in-depth description of how head tracking works in parallel to the illumination of the canvas. The position of the viewer is identified and this is sent to the shader in order to perform the relevant updates on the canvas. Finally we have highlighted our key programming decisions of choosing Java, OpenGL and OpenCV for development.

4 Implementation

In this chapter we give a description regarding the entire implementation of TRePS. We provide information regarding the GUI and how this interacts with the other modules of the system. The front-end was implemented as a standalone application but since this is decoupled from the back-end it can easily be integrated into another application such as *The Painting Fool*. We give the user an in-depth perspective of how the surface modeler engine works and how its output is used in the illumination models built with Java 3D and GLSL. Finally, we describe the method of head tracking and how this is utilized by the shader to dynamically change the shading of the canvas.

4.1 System Structure

Using Java as the core development language we were able to create the entire system using a layered architecture. This can be seen from Figure 4.1[1] in which packages higher up use only classes from packages below them as we abstract away lower level details. Since we used a layered software engineering methodology we start our examination of the system from the bottom up.

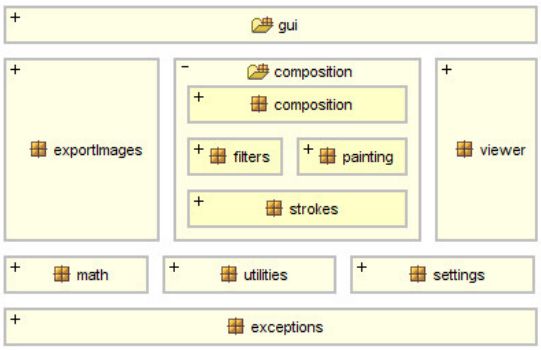


Figure 4.1: Package Structure

[1]The package diagram was created using HeadWay's Structure101 tool. More information can be found at http://www.headwaysoftware.com/products/structure101/index.php

exceptions: At the lowest level of the system we have the exception handling classes. All exception classes extend an abstract TRePS exception and include error handling for using invalid file formats, incorrect program parameters in addition to numerous others specific to surface modeling.

math: Classes in this package include various mathematical utilities required either for 3D graphics such as vectors or image filter functions. The code found in this package is not in any way coupled with other parts of the system so can be reused in other applications if necessary.

utilities: These classes relate to some other utilities required by the system. It includes some colour converters required during image manipulation in addition to some string based utilities.

settings: We store global application settings in this package including various camera calibration parameters to be used for head tracking. These follow the Singleton design pattern in order to have only one instance of such a class in the entire system.

composition.strokes: Stroke related classes are located within this sub-package. This involves storing the internal model of strokes extracted from the XML file exported by *The Painting Fool*. Here we follow a direct implementation of the design described in the Chapter 3, section 3.1.

composition.filters: The image filter framework classes are found in this package. We won't give a lot of detail in this section as provide more depth later on in the chapter but it is important to note that all filters are implemented so that that they are easily extensible and serializable.

composition.painting: This package includes classes needed to represent a painting that can be manipulated by our system. Paintings are an extension of Java's *BufferedImage* class with extra information such as brush strokes.

composition.composition: To create the model of the canvas enhanced with height and surface normals we require to use classes from the filters, painting, and strokes packages. Composition classes use their lower level building blocks to use image filters with the aid of alpha channel transparency to build the entire texture of the painting using each brush stroke.

exportImages: A package that encapsulates numerous classes we have created, using a Factory pattern, to easily export images to common file formats such as PNG, JPEG and GIF.

viewer: Within this package we find all the shader classes for illuminating the canvas

including the Gouraud and Phong models. We have also placed the head tracking module here as it communicates directly with one of the shaders.

gui: All Swing and graphical user interface classes are found in this package. All these components come together to form the GUI for our system which is described in more detail in the next section.

4.2 Front End GUI

A screenshot of TRePS is shown in Figure 4.2 using the latest *Nimbus* interface available from Java 6 Update 10. The look-and-feel of the application can be changed at runtime using the *Interface* options located in the menu bar. During startup the application defaults to the standard appearance detected from the Java Runtime Engine. As seen from the diagram below the implementation has been a close match to our proposed design with the main difference being the use of tabs. We will now move to a more detailed description of these elements.

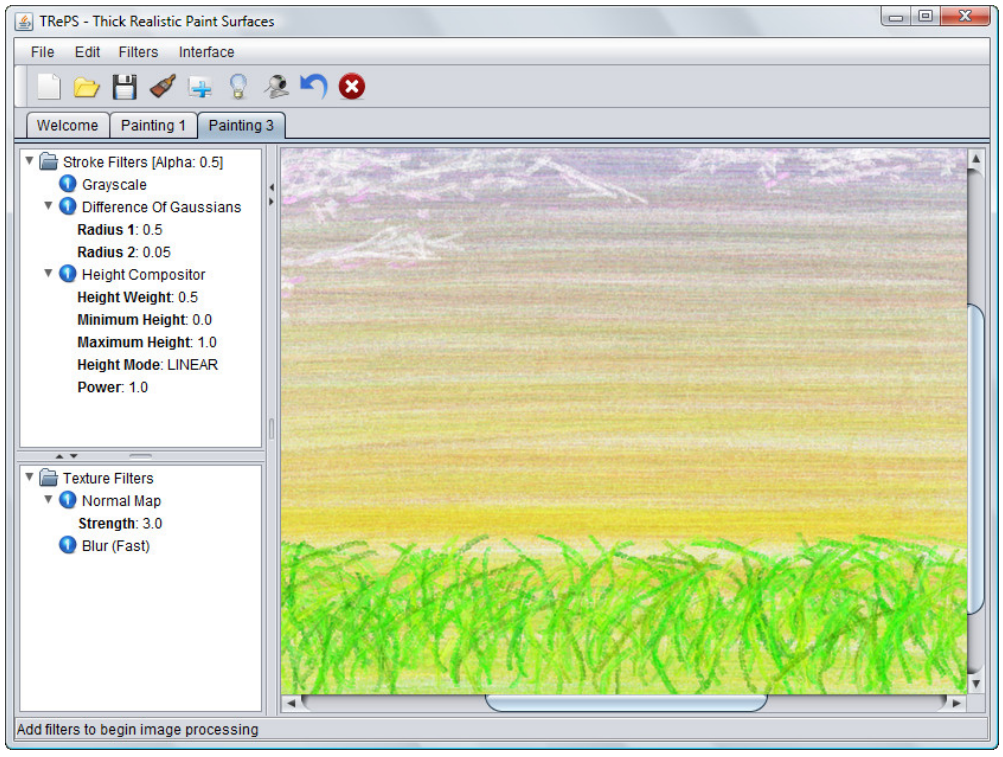


Figure 4.2: GUI Implementation

4.2.1 Filter Panels

Figure 4.3 illustrates the usage of filter panels for visualizing the image filter chain execution. We have divided them into brush stroke filters and texture filters as explained in the design chapter. Stroke filters have the additional option for setting the level of alpha transparency so that different brush stroke blending can be achieved. Expanding a filter displays all its attributes which are dynamically loaded from the back-end class. The two panels shown both make use of Java's *JTree* component for organizing items in a tree like fashion. A custom tree cell renderer is used in order to display the filters within the panes.

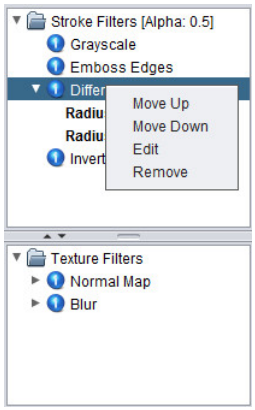


Figure 4.3: Filter Panels

4.2.2 Multi-Threading

The main reason we have adapted our design slightly is to provide the user with the flexibility to work with multiple paintings at the same time. This was achieved using Java's *JTabbedPane* that enables the user to switch from one tab to another. We have set a limit of a maximum of 10 such tabs since the memory requirements of holding thousands of strokes in RAM for each painting becomes an issue. To enable multiple paintings to be open at once the user needs to start the application with a larger heap size than the default value allocated by the JVM (e.g 512MB).

Since each tab in the application allows the designer to run chains of image filters, we do not want to freeze program execution until these complete. Some filter chains may take more than a minute to complete depending on the image filters, stroke count and final image size. To avoid these problems we make use of a Java *SwingWorker* event dispatching thread. During initialization the *SwingWorker* object takes two generic

parameters one being the final result and the second being the value that should be returned at regular intervals (such as progress information). Dispatching a long running thread is achieved with the two methods available from the *SwingWorker* class. The first is the `doInBackground()` method, where we execute our long running process for executing image filters on each stroke. The second is the `done()` method where the thread returns the result of the processing it has performed.

4.2.3 Additional Features

While the main aim is to use our image filter framework to construct surface models, the system is flexible enough to enable the user to import any texture file that represents either a height or normal map. The reason this is useful is due to the fact that an actual painting can be used if a realistic representation of its surface exists. Filter chains can also be exported to XML format and loaded at future times to avoid the need to repeat the process of setting up the filters. A one level undo stack is also available to allow the user to reset the painting to its original texture. This is implemented by having each stroke object (Chapter 3, section 3.1.1) store two separate images for each brush stroke. One is the modified version of the stroke image and the other is the original image obtained during the import stage. If an undo is required we simply reset the modified version to its original stroke.

4.3 Paint Surface Modeling

The core module of our system is the painting surface modeler that takes either an ordinary image or a stroke file and outputs its surface texture so that it can be illuminated using our shading models. We will describe how this is implemented using a number of image filters and how the height maps of individual strokes are then composed to create the overall height and surface map of the canvas.

4.3.1 Image Filter Framework

In this section we describe the implementation[2] of a framework for handling various image filters to process paint strokes and finally compose the height field of the canvas. We can split the filters into two running modes. One mode will run the filters on each

[2]Many thanks to JH Labs for explanations and implementation tips on creating image filters in Java. We have also used the Java Image Editor they have created for testing the suitability of certain filters. For more information please visit http://www.jhlabs.com/ip/filters/index.html

stroke separately (known as stroke filters) and the second will run the filters on the entire image (known as texture filters). Their distinction can be seen in Figure 4.3.

Figure 4.4 shows a UML diagram of the image filters we will use for height map construction. An abstract *ImageFilter* class is used which all other filters extend. The `runFilter` method can be used to run the filter either on a stroke or an entire image. We now give a brief description of the major filters in use and why they aid with the height map construction of the painting canvas.

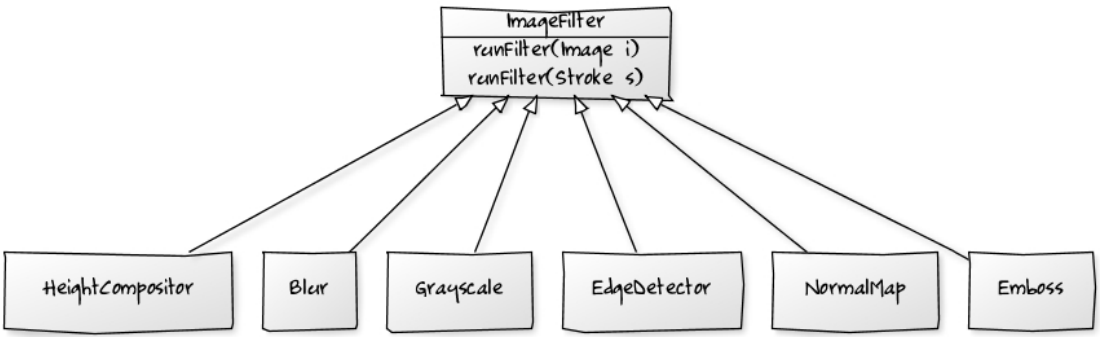


Figure 4.4: Image Filters

- **Grayscale:** A simple image filter for taking the average of the three colour channels (RGB). This is needed since we first need to convert either a brush stroke or an entire image to an initial height map that contains single dimensional height data for each pixel.

- **Blur:** This is another simple filter that calculates the average height based on its neighbouring pixels. Blurring can either be performed by weighting each neighbour differently or taking the average RGB value of the neighbours. The surrounding pixels to be taken into account is be a parameter to the filter and their number is calculated by equation 4.1 where n represents the neighbourhood level.

$$Pixels = (2n + 1)^2 - 1 \qquad n > 0 \tag{4.1}$$

- **Emboss:** This will act as a bump map filter by taking into account the brightness at each pixel in the image and calculating a surface normal and shading the pixel based on its relative position to a light source. This filter is useful in cases where paint ridges in a stroke or an image are not clearly defined and embossing will aid to produce better results when passing the image to the edge detector.

- **EdgeDetector:** The EdgeDetector filter is an implementation a number of classical edge detection operators as described in Chapter 2, section 2.5. These can

then be used to identify rates of change of intensity at each pixel in the paint stroke for ridge identification. This can be achieved by multiplying the intensity of the pixel by the correct 3×3 kernel for the x and y directions [10, pages 131-157].

- **HeightCompositor:** The height compositor filter is an implementation of the ridge identification method (Chapter 3, section 3.1.2) that combines both the results of the edge detection output and the grayscale image of the stroke. It is a purely stroke based filter and cannot be used on plain images.

- **NormalMap:** This is an image based filter that will run on the final height texture after all strokes have been processed. A normal map is a texture where each pixel stores the value of the surface normal at that particular point. The (x, y, z) coordinates of the normalized vector are stored in the RGB colour channels of the image. The surface normal at a particular pixel is found by taking the height difference from the top, bottom, left and right pixels and forming vectors from these values. The cross product of each two pairs of vectors (top×right, right×bottom, bottom×left, left×top) is calculated to find four normals which are then averaged. This represents the surface normal which is then saved in the normal map after it has been normalized. Algorithm 4.1 demonstrates how this is done in practice.

Algorithm 4.1 Surface Normal Calculation

```
for (int x=0; x < width; x++) {
   for(int y=0; y < height; y++) {
      int pixelHeight = getHeight(x,y)
      int zTop = pixelHeight - getHeight(x,y+1)
      int zBottom = pixelHeight - getHeight(x,y-1)
      int zLeft = pixelHeight - getHeight(x-1,y)
      int zRight = pixelHeight - getHeight(x+1,y)
      Vector top = new Vector(0,1,zTop)
      Vector bottom = new Vector(0,-1,zBottom)
      Vector left = new Vector(-1,0,zLeft)
      Vector right = new Vector(1,0,zRight)
      Vector normal1 = crossProduct(top,right)
      Vector normal2 = crossProduct(right,bottom)
      Vector normal3 = crossProduct(bottom, left)
      Vector normal4 = crossProduct(left,top)
      Vector surfaceNormal = (normal1 + normal2 + normal3 + normal4)/4
      surfaceNormal.normalize()
      normalMap.setRGB(x, y, surfaceNormal)
   }
}
```

4.3.1.1 Filter Serialization

In order to enable filter chains to be imported and exported from the tool we need to serialize them. Although a number of implementation options exist, such as direct object storage, we chose to use eXtensible Markup Language (XML) that is also be human readable. To achieve this task we use the *SimpleXML*[3] library for Java with its powerful serialization framework. The reason we chose this library over others is that it requires little configuration on our part. The way *SimpleXML* serializes our filter classes is by making use of the annotations; a feature available in Java 6. This is simply syntactic metadata that the library uses to identify what fields exist in the class and how these should be represented in text format.

4.3.2 Height Composition

The final stage of the surface model construction is to compose each stroke to increase the height of the canvas. The way this is done is by using a *Compositor* class that processes each image stroke. We start off with a black image (representing zero height) and set the alpha channel to some transparency value less than 1 to blend together the individual height maps for each stroke. Depending on the alpha value set the overall height will be composed differently. Using high transparency ($alpha < 0.5$) will use a number of strokes for the final height composition whereas using a more opaque blending mode tends to ignore strokes because of the back-to-front rendering mode.

To run an image filter pipeline we make use of a *FilterChain* class we developed, which extends the abstract *ImageFilter* class shown in Figure 4.4. Its purpose is to act as a placeholder for adding image filters so that they are executed in some pre-defined order. This enables us to keep the processing stage modular as each filter is implemented independently. A number of pipelines can then be constructed using this method. For instance one filter chain can be used directly for brush strokes and the other chain for the entire image. This is the internal structure behind the filter panels seen in Figure 4.3. The surface model can now be represented as either a grayscale height map which can be used in displacement mapping or a normal map used for bump mapping.

[3]Simple is a high performance XML serialization and configuration framework for Java. More information can be found at http://simple.sourceforge.net

4.4 Lighting Models

Within this section we provide the implementation of both lighting models. These were programmed using mostly Java 3D with the use of some language specific code from the OpenGL shading language (GLSL) to achieve GPU acceleration. As explained in section 3.2.1 there are two different shaders extending an abstract viewer class. The general viewer is responsible for taking the BufferedImages created by the surface modeling module (colour and normal textures) and loading these into a *Texture* object that Java 3D can use. This class also creates the geometry which is then used by the concrete shader classes to alter its flat appearance.

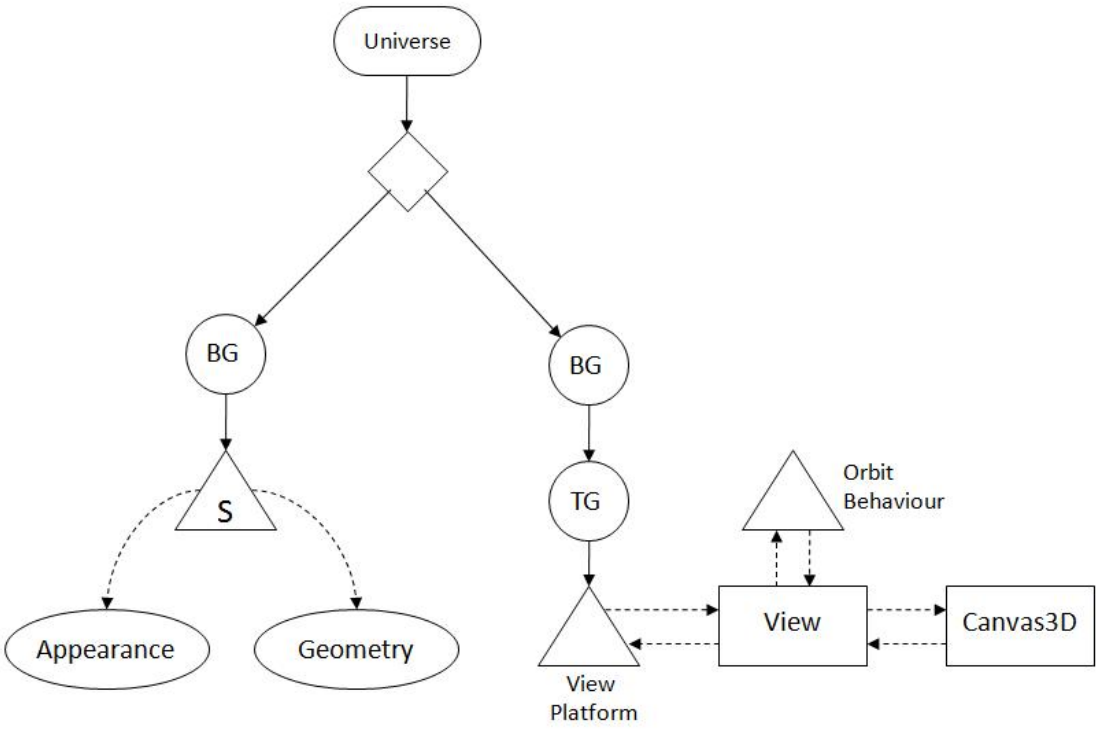


Figure 4.5: Java 3D Scene Graph

Figure 4.5 above shows the Java 3D scene graph structure that is set up to be rendered within a 3D canvas. The scene graph uses the same symbols for definition as described in the Java 3D API [22]. Within this tree-like structure, there exist a number of nodes (Java 3D classes). Between nodes two types of relationships are possible. One is the *parent-child* relationship (solid arrows) and the other is a reference (dotted arrows). References can be used to connect certain classes in other parts of the scene graph such

as appearance and geometry. The *parent-child* relationship on the other hand is more strict and only allows each node to have at most one parent.

We now begin describing the scene graph starting from the root and moving downwards. The highest element is the *Universe*. All other nodes should be connected to this object and can be considered a virtual world in which all other Java 3D objects interact. Below the root is a diamond like structure known as a *Locale.* This acts as a reference point within the universe to determine the location of other virtual objects. It can be considered a landmark within the virtual world.

Moving to the left of the *Locale* we find a group node known as a *BranchGroup*. This represents a content branch group as it encapsulates the objects found within the virtual universe. Below that is a leaf node S (Shape3D class) that will form the painting canvas to be rendered. This references two different node components of *Appearance* and *Geometry.* The *Appearance* node defines how a polygon is to be rendered by setting up relevant textures and the *Geometry* node is a creation of the polygon itself. For our geometry, we only have four vertices which are connected together to form large rectangle representing the painting canvas.

The right hand side of the tree is the entire view structure of the system. This *Branch-Group* node stores a *TransformGroup* object. The transform group class can change the positioning, orientation and scaling of its children via a *Transform3D* object. A number of different components interact with each other via reference pointers so that any changes to the shape in the left hand side of the tree are correctly viewed on the three dimensional canvas. We have added an *OrbitBehavior* node to the view so that a user can interact with the canvas by rotating and translating using their mouse. It also supports zooming with the aid of the mouse scroller. To avoid the possibility of the canvas disappearing from the viewable region, we have added the capability of the canvas to reset itself to its original position. This is done by replacing the current transform with the inital transform.

4.4.1 Smooth Shader

The first illumination model we have implemented is smooth shading based on the Gouraud model. The constructor for this class takes both a colour image and a surface normal image. The only component the shader alters in the scene graph is the *Appearance* node (Figure 4.5) as all other items do not require any modifications.

We begin by defining a light map which is simply an image of the same dimensions as the colour and normal maps. The entire light map is a single colour that defines the directional position of the light rays. These rays are encoded directly in the RGB

channels of the image at each pixel. If one was to view the light map in isolation then it would appear as a single polygon changing colours as the light source changes position. Using Java 3D, we are able to perform the dot product of the light ray vector with the surface normal for each pixel. This is due to the fact that each pixel in the light map texture contains the light ray vector and each pixel in the normal map texture contains the surface normal vector.

The issue faced during implementation of the Gouraud shader was how to convert the mouse coordinates representing the light position to a normalized scale representing the light vector. The window displaying the painting could have a variable height and width during runtime as the user can rescale the canvas. Coordinates tracked in this window need to be transformed to values in the range of 0 to 255 for x and y directional vectors. This is demonstrated in Figure 4.6 where window coordinates of size 1600×1200 are remapped to a new window of size 256×256 using linear interpolation for each axis. For instance global coordinates of $(1200, 350)$ map to light vector coordinates as follows:

$(\frac{1200}{1600} \times 256, \frac{350}{1200} \times 256) = (192, 75)$

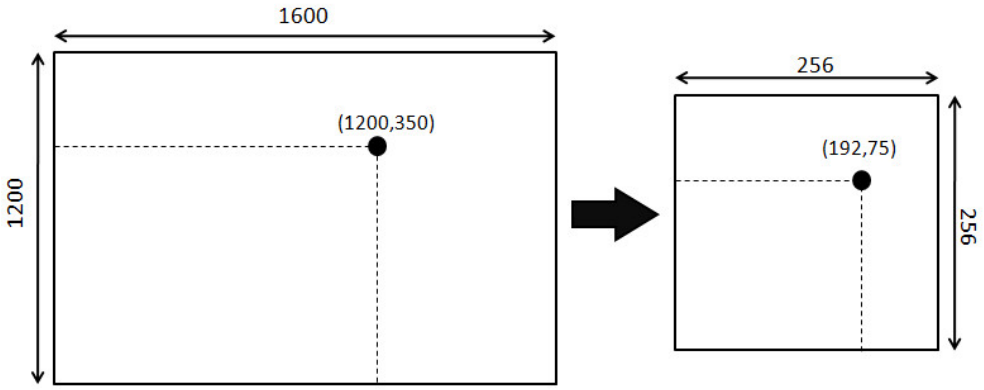


Figure 4.6: Coordinate Scaling

Once the light vector coordinates have been calculated, they are used as the red and green colour channels of the light map. The blue channel is obtained not from mouse coordinates but rather from the value specified at the keyboard. Once the (x, y, z) values are transformed to rgb we are able to update the light map texture. This is done by using the `repaint()` method available in Java 2D that sets the background colour of the image.

As explained previously, the graphics card performs the dot product of the light map with the surface normal map. This produces a new grayscale texture that represents the diffuse shading by the scattering of the light rays. This diffuse image is then combined

with the actual painting RGB value from the colour texture (Chapter 2, section 2.1.4.3) to give the illusion of a rough surface. In order for these to be changed at runtime, we need to set the capabilities in Java 3D to allow reading and writing of the textures when compiling the scene graph.

4.4.2 Phong Shader

Just as in the case of the Smooth shader, the Phong shader we have implemented only modifies the *Appearance* node in the scene graph. This shader is slightly more complex since it programs the graphics card (altering its fixed pipeline) and executes the lighting model directly on the GPU. The Phong illumination model is not written in Java code but a programming language similar to C that has some extra extensions built in (Chapter 2, section 2.6.3). The way this is loaded into Java 3D is shown in Figure 4.7 where both vertex and fragment programs are loaded directly as String objects. These are then wrapped within a special Shader class provided by Java 3D.

```
String vertexProgram =
  StringIO.readFully(getClass().getResource("/shaders/phong.vert"));
String fragmentProgram =
  StringIO.readFully(getClass().getResource("/shaders/phong.frag"));
Shader[] shaders = new Shader[2];
shaders[0] = new SourceCodeShader(Shader.SHADING_LANGUAGE_GLSL,
                Shader.SHADER_TYPE_VERTEX,
                vertexProgram);
shaders[1] = new SourceCodeShader(Shader.SHADING_LANGUAGE_GLSL,
                 Shader.SHADER_TYPE_FRAGMENT,
                fragmentProgram);
```

Figure 4.7: Loading Shader in Java 3D

In Figure 4.8 we display the code for the Phong fragment shader. The vertex shader is much simpler and only does the necessary setup of viewer and light source vectors. In the code we have a number of different variables that require some explanation. Prior to the main body of the function, we have the light source and eye vectors passed from the vertex shader. This is the use of the `varying` keyword in the program so that the fragment shader can receive the vectors from the vertex unit. The colour and normal textures are specified as `uniform` values since we only use the shader to read texel positions and do not modify these in any way. Within the main function we perform some calculations to specify the bounding sphere of the light source. We then sample the necessary pixel values from the textures using `texture2D` available in GLSL. Ambient,

diffuse and shininess values are accessed from Java 3D using the `gl_*.*` primitives. As a result each pixel value's colour passes through this pipeline and its new colour is output in `gl_FragColor`.

```
varying vec3 lightVec;
varying vec3 eyeVec;
varying vec2 texCoord;
uniform sampler2D colorMap;
uniform sampler2D normalMap;
const float invRadius = 0.01f;

void main (void) {
  float distSqr = dot(lightVec, lightVec);
  float att = clamp(1.0 - invRadius * sqrt(distSqr), 0.0, 1.0);
  vec3 lightVector = lightVec * inversesqrt(distSqr);
  vec3 eyeVector = normalize(eyeVec);
  vec4 baseCol = texture2D(colorMap, texCoord);
  vec3 bump = normalize( texture2D(normalMap, texCoord).rgb );
  vec4 vAmbient = gl_LightSource[0].ambient * gl_FrontMaterial.ambient;
  float diff = max( dot(lVec, bump), 0.0 );
  vec4 vDiffuse = gl_LightSource[0].diffuse * gl_FrontMaterial.diffuse;
  vDiffuse = vDiffuse * diff;
  float specular = clamp(dot(reflect(-lVec, bump), vVec), 0.0, 1.0);
  specular = pow( specular, gl_FrontMaterial.shininess );
  vec4 vSpecular = gl_LightSource[0].specular;
  vSpecular = gl_FrontMaterial.specular * specular;
  gl_FragColor = ( vAmbient*baseCol + vDiffuse*baseCol + vSpecular) * att;
}
```

Figure 4.8: Phong Fragment Shader

4.5 Viewer Motion Tracking

In this section we describe the implementation of the head tracking module using OpenCV and its integration with the Phong shader. OpenCV uses the Viola-Jones object tracking framework in order to detect faces within an image (Chapter 2, section 2.4). Although it supports tracking multiple faces we disable this feature and only search for the largest face within the window as the viewing experience on one monitor can only be performed for one user.

The implementation of motion tracking is found within the *HeadTracker* class located in the viewer package (Figure 4.1). Prior to the initial implementation we had performed some initial camera testing and found that some webcams perform slightly differently under different conditions and certain calibration parameters need to be set. As a re-

sult, we created a camera calibrator that can be accessed from our system giving the user all the necessary options to modify both camera and tracking algorithm parameters (Chapter 3, section 3.3.2). This is displayed in Figure 4.9. The class containing these parameters is called *CameraSettings* and is found within the settings package. *CameraSettings* uses a Singleton pattern since we want to restrict the instantiation of the class to just one object that can be accessed anywhere throughout the application.

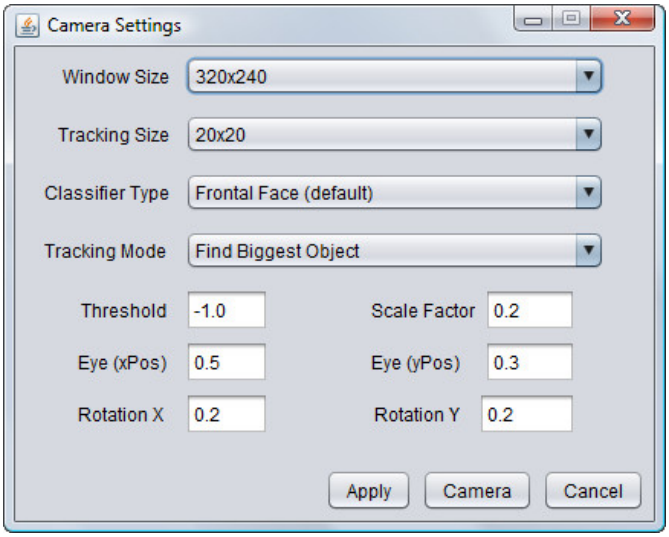


Figure 4.9: Camera Calibration Window

We now move to the description of the algorithm itself and how this is used to detect where the viewer is with respect to the canvas. A small part of the main tracking loop is shown in Figure 4.10. In order for this to work the user needs to have already installed OpenCV on their system as it uses native libraries to interface with the connected camera. Initially, we start off by creating an *OpenCV* object that handles the camera on our behalf. We set its capture size and cascade classifier and then it begins requesting camera images. Having a while loop that continuously gets images from the camera enables us to receive live feedback on the positioning of the viewer. We need to flip the images horizontally so that we manage mirror-like interaction. We store this image in a *MemoryImageSource* class so that it can be displayed in the visual tracking window providing the user visual feedback of their position in relation to the canvas. The actual face detection is performed by the `cv.detect()` method call that uses the Viola-Jones algorithm to search for the face in the image. This gives us an array of rectangles in `squares` and we access the first face image using `squares[0]`.

Using the rectangle detected from OpenCV representing the face of the user we pass this image to calculate the interpupilary centroid. Retrieving the parameters from the

camera calibrator for the positioning of the eyes we are able to narrow our search inside the larger image. This can be seen from Figure 4.11 where the large red square represents a viewer's face and the smaller green rectangle the central eye for monocular vision. The actual coordinates recorded are those at the centre of the green square. This is done using Java 2D which gives us a number of drawing primitives enabling us to write directly on the image before it is exported to the head tracking window.

```
cv = new OpenCV();
cv.capture( width, height );
cv.cascade(classifierPath);
while( cv!=null && capturing ) {
  cv.read();
  cv.flip( OpenCV.FLIP_HORIZONTAL );
  MemoryImageSource mis;
  mis = new MemoryImageSource( width, height, pixels, 0, width );
  img = createImage( mis );
  squares = cv.detect( 1.2f, 2, OpenCV.TRACKING_MODE, size, size );
  repaint();
}
```

Figure 4.10: Head Tracking

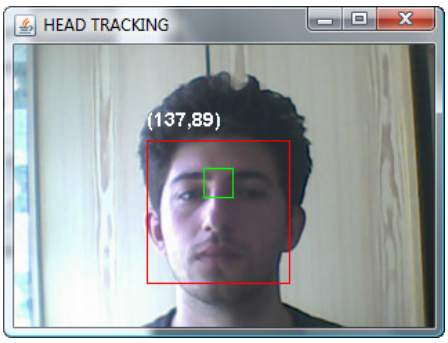


Figure 4.11: Visual Camera Feedback

We now describe how coordinates generated by the head tracking module are used to modify the illumination of the canvas. The head tracking class is initialized by the Phong shader only if the tracking device requested was that of the camera. Once this has occurred the shader begins requesting x and y coordinates for the viewer positioning from the *HeadTracker*. We do not track the z coordinates since the variance in the size of the window cannot be used to accurately provide this. Instead we perform coordinate scaling as demonstrated in Figure 4.6 and shift the canvas slightly simulating the effect

of a moving viewer. The amount of rotation can be specified as a parameter for either axis up to a maximum of 90°. By performing this slight rotation the canvas illumination is automatically updated by the shader. It must be noted that coordinates are used every time a frame can be generated by the graphics card and not every time they have been updated by the tracking module.

4.6 Summary

In this chapter we have given the reader the necessary details needed so that they can make changes to TRePS if required. The package structure diagram provides the layering of the code we have written and a small description of each package is documented. The front end user interface is given a brief overview as well as its interaction with backend modules. The rest of the chapter described the surface modeling engine which uses an image filter framework written in Java 2D, the illumination models implemented using both Java 3D and GLSL and finally the head tracking system built with the aid of OpenCV.

5 Experimental Design

In this chapter, we describe the experimental setup used to evaluate the entire system. We begin with a description of the different hardware resources used and then move on to various the performance testing carried out. In addition we go over the usability of the GUI. We study its stability to uncover any faults and possible fixes for the future releases. We also carry out an evaluation of the accuracy and robustness of our head tracking module within a range of operating environments. Finally, we highlight the testing carried out to obtain relevant feedback on the entire aesthetics of the paintings produced and the setups users most prefer.

5.1 Setup

The evaluation we have carried out has utilized a range of hardware platforms that we will list below. Since we want to identify the minimum resources required by our system we have used three different types of hardware specification. Section 5.1.1 shows these three different setups. The next two sections provide a list of both the webcams and monitors used in the evaluation process. These will be referred by their code letter.

5.1.1 Computer Hardware

Setup A - Low

- **System Type:** Laptop
- **Operating System:** Microsoft Windows Vista Business
- **Processor:** Intel Core Duo 1.66GHz
- **RAM:** 2GB
- **Graphics Card:** Intel Graphics Media Accelerator 950 (224MB shared memory)

Setup B - Medium

- **System Type:** Laptop
- **Operating System:** Microsoft Windows 7 Home Premium
- **Processor:** Intel Core 2 Duo 2.00GHz
- **RAM:** 2GB
- **Graphics Card:** Nvidia GeForce 8400M GS (128MB dedicated memory)

Setup C - High

- **System Type:** Desktop
- **Operating System:** Microsoft Windows 7 Business
- **Processor:** Intel Core 2 Quad 2.50GHz
- **RAM:** 4GB
- **Graphics Card:** Nvidia GeForce GT220 (1GB dedicated memory)

5.1.2 Web Cameras

Code	Model	Price	Min Distance	Sensor	Face Tracking
A	LifeCam VX-5000	£25	50 cm	CMOS	✓
B	PC-Line SF050916658	£4	65 cm	CMOS	✗
C	PC-Line P03GWKT09	£7	50 cm	CMOS	✓
D	Logitech C250	£20	45 cm	CMOS	✓

5.1.3 Monitors

Code	Model	Price	Size (inches)	Resolution	Output
A	Dell 242 Wide	£200	24	1920×1080	DVI/HDMI
B	HP w2007v	£150	20	1680×1050	VGA
C	ACER P195HQLb	£100	18.5	1366×768	VGA
D	ACER Aspire 5600	-	15.4	1280×800	-
E	Dell XPS M1330	-	13.3	1280×800	-

5.2 Performance Testing

To assess TRePS completely we first carried out performance tests to measure its speed and responsiveness under a variety of conditions. These measurements are carried out on all hardware platforms shown in section 5.1.1 and are listed as follows:

- **Filter Tests:** Since there are different methods available of composing the painting surface using image filters we need to measure the overall execution time of each filter chain. Times measured will include those of individual filters in addition to times running either stroke or texture based image filters. These measurements were taken using JProfiler[1] and the filter chains tested are shown below.
  - **Standard:**
    - `Strokes [Grayscale > EdgeDetector > PaintBuildup]`
    - `Texture [NormalMap]`
  - **Simple (No Strokes):**
    - `Texture [Grayscale > NormalMap]`
  - **Fast (No Edge Detection):**
    - `Strokes [Grayscale]`
    - `Texture [NormalMap]`
  - **Complex:**
    - `Strokes [Grayscale > DifferenceOfGaussians > PaintBuildup]`
    - `Texture [NormalMap > Blur]`
- **Frames per Second:** Since we are rendering two different lighting models we need to measure the frames per second (FPS) generated by the graphics card so we can ensure this is at an acceptable level for the viewing experience. FPS measurements were made using FRAPS[2].

[1] JProfiler is an award winning profiler written in Java that is used to find performance bottlenecks, identify memory leaks and resolve threading issues. It can be found at http://www.ej-technologies.com/products/jprofiler/overview.html

[2] FRAPS is a graphics benchmarking tool that measures FPS and can calculate statistics based on the running application. More information can be found at http://www.fraps.com

5.3 User Testing

In this section, we describe the process by which we evaluated the front end GUI (Figure 4.2) developed to create surface models for paintings. This is an important part of the experimental process as we want to obtain feedback on how users found the interface and what parts require improvements for future releases. To carry out end-user testing, we selected ten users with some basic knowledge in graphics and computer vision as the GUI is mainly for designers. We gave them a brief introduction to the system and asked them to construct three different surface models and render these using both the Gouraud and Phong shader. As they perform their task we would monitor them to observe what aspects they had particular problems with and the time needed to complete these. In order to avoid biasing the results each user performed their evaluation on the high performance hardware (system A) with monitor B. The data gathered was both from questionnaires and our observations made while studying the users performing their evaluation of the system. These are described in more detail in the next two sub-sections.

5.3.1 Questionnaire

Once users had completed the usability testing phase, we gave them a questionnaire to fill in. This contained a variety of questions based on their experience of using the application and are shown below in Table 5.1. The questions asked were phrased in such a manner to avoid any biasing and were structured so that a Likert[3] scale could be used.

	Question
1	The system is easy to use
2	I could not create surface models I wanted
3	The system was not fast enough
4	It was easy to manipulate a range of image types
5	The toolbar was useful and gave me complete control
6	I could revert changes to filters after these were executed
7	The system was able to handle multiple paintings at once
8	Filters were easy to visualize in the tree-like panels
9	I received constant feedback of what I was doing
10	The image filters available were not good enough to produce correct models

Table 5.1: End-User Questionnaire

[3] http://en.wikipedia.org/wiki/Likert_scale

5.3.2 Nielsen's Heuristics

The second part of this evaluation phase consisted of checking how well the system faired against Nielsen's ten usability heuristics [17]. These were published in 1994 and are probably the most common heuristics for user interface design. The results can be found in the next chapter.

5.4 Robustness & Stability Testing

Another important aspect of the evaluation process was to measure the robustness and stability of the system. This was done via monkey and stress testing. The reason this type of evaluation is vital, is to find the limits of the software and identify any areas of instability.

5.4.1 Monkey Testing

Monkey testing is a form of random testing where the system is subjected to arbitrary input. The observed results can be used to identify any areas of weakness that was not discovered during the usability testing phase. These tests are listed below as follows:

1. Import a stroke XML file that is not valid
2. Import a stroke XML without any brush strokes
3. Load an external surface normal texture of incorrect width and height
4. Load a surface normal texture that does not contain vectors for each pixel but random values for the RGB colour channels
5. Run stroke filters with complete transparency ($alpha = 0$)
6. Attempt to run illumination models without any surface model constructed
7. Add random image filters to both stroke and texture panels and execute the surface model construction
8. Attempt to run head tracking without a camera connected to the system
9. Export an empty filter chain and then attempt to re-load it back into the application
10. Attempt to import an invalid XML filter chain

5.4.2 Stress Testing

The tests described in the previous section are known as "dumb monkeys" as they attempt to make the system fail by using random input. "Smart monkeys" on the other hand are used to obtain valuable information while the system is put under excess load. Stress tests performed in this area are as follows:

1. Import extremely large texture maps into the program and attempt to run the illumination models. Ideally these should be greater than 3000 pixels in both width and height.
2. Load a stroke XML file with a very large number of brush strokes. Average working sizes were approximately around 5000 strokes so to put the system under load we would use paintings in the tens of thousands of brush strokes.
3. Execute an image filter chain containing more than 20 filters
4. Run the surface construction filters for more than ten paintings by having these execute in parallel.
5. Render more than 4 illumination models at the the same time.

5.4.3 Motion Tracking

The final stage of the robustness and stability evaluation involves testing our head tracking system. All experiments that will we list below were performed on all four cameras (section 5.1.2). These tests will enable to us to find the accuracy of the head tracker, which cameras perform best and under which conditions. The experiments carried out are as follows:

- **Speed and Accuracy:** Measure the speed and accuracy of detection for a known list of positions (Table 5.2) within the tracking window of the camera. These predefined positions will range in distance from the camera and angle from its centre of projection. The angle is measured from the centre of projection of the camera in any direction so that an angle of 0° means the viewer's head is directly aligned with camera lens. These experiments were carried out in a room with ample ambient light and with the camera facing a white wall without any texture. Detection time is automatically generated by our system since it can measure the difference from the time when the camera attempted to start detecting faces and the time when a face rectangle was returned. To measure the accuracy of the detection we run the test to detect a face at a particular position a number of times and count how many of those times it was actually detected.

	Distance (cm)	Angle (degrees)
Points 1	$60 \leq d < 80$	$0° \leq \alpha < 10°$
Points 2	$60 \leq d < 80$	$10° \leq \alpha < 25°$
Points 3	$60 \leq d < 80$	$25° \leq \alpha < 60°$
Points 4	$80 \leq d < 120$	$0° \leq \alpha < 10°$
Points 5	$120 \leq d < 200$	$0° \leq \alpha < 10°$

Table 5.2: Position List

- **Multiple Faces:** Run the detection algorithm with more than one face present in the tracking window of the camera. Although the object detection framework should detect these in our case we only want to find the largest face detected.
- **Variable Backgrounds:** Run the head tracking system to detect faces in a range of background environments. It should be easier to detect faces in a constant background environment rather than highly textured backgrounds as the Viola-Jones algorithm needs to perform extra computational processing using the cascade classifier. The accuracy for each environment is recorded so that we can recommend a suitable setup for users wishing to use the system.
- **Variable Lighting:** We also want to measure the robustness of head tracking under different illumination conditions. To perform this test we have setup the camera detection system in a room where the ambient lighting can be regulated via a variable switch. This will enable us to measure any changes in head tracking performance given both dark and bright conditions.

5.5 Aesthetics Evaluation

The last part of the evaluation phase involved testing the realism and aesthetics of the 3D paintings we have generated. This will involve another round of user testing but this time we will collect feedback based not on the user interface but the canvas renderings. The aesthetics evaluation was conducted using the same 10 users that took part in end-user testing.

In order for us to collect relevant feedback we structured the viewing phase for users into three distinct rounds. These were set up using hardware system A (section 5.1.1) and a combination of monitors A,B and C. For head tracking purposes we used 3 cameras of type C. The painting used for display on each monitor differed in the way

the surface model was constructed or the type illumination model used. The image, however, displayed was the same to avoid users biasing their decision based on the best painting rather than the actual 3D rendering.

The first round of viewing consisted of setting up three different painting renderings on each of the three monitors. In this round, head tracking was disabled and only mouse and keyboard interaction was available with the painting. The first monitor displayed a simple polygon with a colour painting textured mapped onto it so that it would appear flat. The second consisted of the same image used on the first monitor but this time rendered using the Gouraud illumination model (Chapter 1, Figure 1.1(a)). The third monitor had a rendering using the Phong illumination model (Chapter 1, Figure 1.1(b)).

The second round of viewing involved setting up the three monitors so that only the Phong illumination model was used. Again as before, no head tracking was enabled in this round. The only method of interacting with the canvas was via the keyboard and mouse. On the first monitor we used the edge detection and paint buildup filters on each stroke to obtain an accurate surface model for the canvas. This was illuminated using ordinary white light and a highly specular value to make the paint appear shiny. The second monitor used the same illumination conditions as for the first painting but contained just a single image without any stroke information. The last monitor used a more bumpy surface by increasing the value to the normal map filter, less shininess and an orange colour for the light source. The last round for viewing consisted of the same illumination conditions for each monitor as in round 2 but the difference being that we had also enabled head tracking. This would give us an indication whether the viewing experience was enhanced with the use of a camera. A quick overview of the aesthetics evaluation can be seen in Table 5.3 below.

	Monitor 1	**Monitor 2**	**Monitor 3**	**Tracking**
Round 1	Simple flat canvas	Gouraud Shading	Phong Shading	✗
Round 2	Phong Shading A	Phong Shading B	Phong Shading C	✗
Round 3	Phong Shading A	Phong Shading B	Phong Shading C	✓

Table 5.3: Monitor setup for aesthetics evaluation

Once we had shown each of the renderings to our 10 users we gave them a questionnaire to fill in so that we could analyse the results and find out which surface and illumination conditions were most pleasing to viewers. This is shown in Figure 5.1 on the next page.

1. Which painting was the most visually appealing in round 1?
   a) Monitor 1
   b) Monitor 2
   c) Monitor 3
2. Which painting was the most visually appealing in round 2?
   a) Monitor 1
   b) Monitor 2
   c) Monitor 3
3. Did head tracking add to your viewing experience of the canvas?
   a) Yes
   b) No
4. Which painting in round 2 had the better illumination conditions?
   a) Monitor 1
   b) Monitor 2
   c) Monitor 3
5. Which surface model was preferable for the paint texture in round 2?
   a) Monitor 1
   b) Monitor 2
   c) Monitor 3
6. Were the methods of interaction with the painting canvas acceptable?
   a) Yes
   b) No
7. Place a ranking on a scale from 0 (worst) to 10 (best) for each painting you have just seen in rounds 1 and 2.

Figure 5.1: Aesthetics Questionnaire

5.6 Platform Compatibility

Despite the fact that we developed TRePS using Java, we still need to check that it can be used across a range of platforms without any issues. This is due to the fact that it utilizes OpenGL's shading language (GLSL) and also because it uses OpenCV specific code. Firstly, we install OpenCV and Java 3D on the host computer which is a prerequisite to run TRePS. Once this has occurred, we start up the system and load a particular painting from *The Painting Fool.* The surface model is then generated using our image filter chain which is then rendered using both illumination models we have implemented. For the Phong shading model, we also check that head tracking is working as expected. This procedure was carried out on Windows, Linux and Macintosh platforms. The specific operating systems we have conducted our experiments on include *Microsoft Windows 7 Business, Mac OS X Snow Leopard* and *Ubuntu 10.04 LTS.* The results of these tests can be found in section 5.6.

5.7 Summary

In this chapter we have examined a range of techniques used to evaluate our system for improving the visual realism of digital paintings. The overall hardware setup concerning computers, monitors and webcams was provided in section 5.1. Using this equipment we carried out a performance evaluation of the system and then moved on to the end-user evaluation of the GUI. We measured the stability of our application by using monkey and stress testing and also studied the accuracy of the head tracking module. To complete the evaluation, we asked a number of users to provide their feedback on the visual appearance of the painting renderings by asking them to view the canvas in a series of structured rounds. Further details regarding aesthetics evaluation can be found in section 5.5.

6 Results and Analysis

This chapter examines the quantitative and qualitative data gathered from the evaluation experiments carried out in Chapter 5. These are presented in the forms of graphs and charts, where possible, so that they can be viewed with greater ease for the reader. At the end of each evaluation section, we analyse the results to discover their importance and how they can be used to improve the system in the future.

6.1 Performance Testing

6.1.1 Filter Performance

We run the image filter chains on a painting with approximately 5000 brush strokes and a painting with a single image and measure their execution times. We also break the chain into its respective image filter components. These tests are carried out on all hardware systems (A,B,C) found in section 5.1.1. The image filter chains that were tested can be found in Chapter 5, section 5.2.

	Standard	**Simple**	**Fast**	**Complex**
Grayscale	0.92	-	0.90	0.93
EdgeDetector	1.43	-	-	-
PaintBuildup	0.78	-	-	0.80
DifferenceOfGaussians	-	-	-	5.83
STROKE TOTAL	3.13	-	0.90	7.56
Grayscale	-	0.38	-	-
NormalMap	0.49	0.43	0.46	0.48
Blur	-	-	-	1.24
TEXTURE TOTAL	0.49	0.81	0.46	1.72
TOTAL	**3.62**	**0.81**	**1.36**	**9.28**

Table 6.1: Filter Chain execution times (seconds)

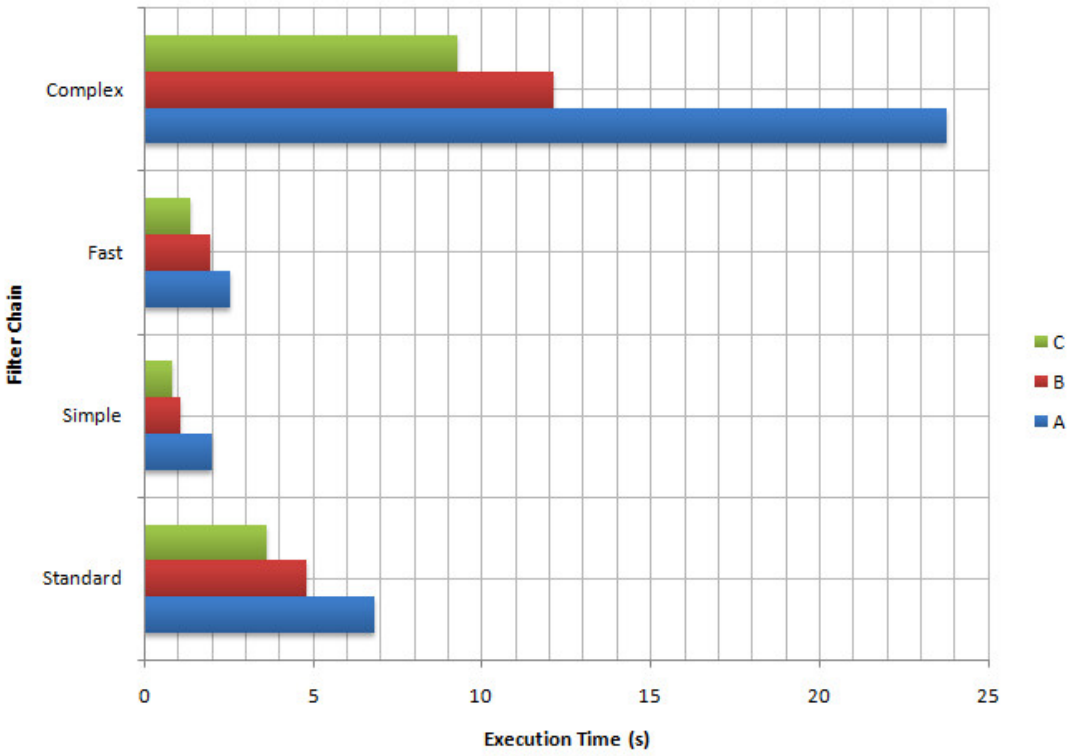


Figure 6.1: Filter Chain execution times per system

Table 6.1 shows the execution time of the image filter chains. As we expected, the *Simple* chain runs the fastest since it does not take into account brush strokes and is executed only once on the entire image. Overall, running the stroke chains takes considerably longer to complete than the texture chains since for the first type the image filters have to be executed thousands of times per stroke and then blended using alpha compositing. The most computationally expensive chain to execute was the *Complex* chain due to the inclusion of the *DifferenceOfGaussian* filter (Chapter 2, section 2.5). This filter blurs the image twice each time with a difference radius and then takes the difference between the images to identify edges. This has to be performed for each stroke in the painting and takes considerably more processing time than any other image filter.

Figure 6.1 displays a graph of each of the four filter chains running on each of the hardware systems (section 5.1.1). System C performs the fastest for all chains due to its faster processor and greater amount of RAM. The *Complex* chain takes the greatest amount of time compared to all other types even when running on system A. Execution time may take a number of minutes if the stroke count moves into the tens of thousands.

6.1.2 Rendering Performance

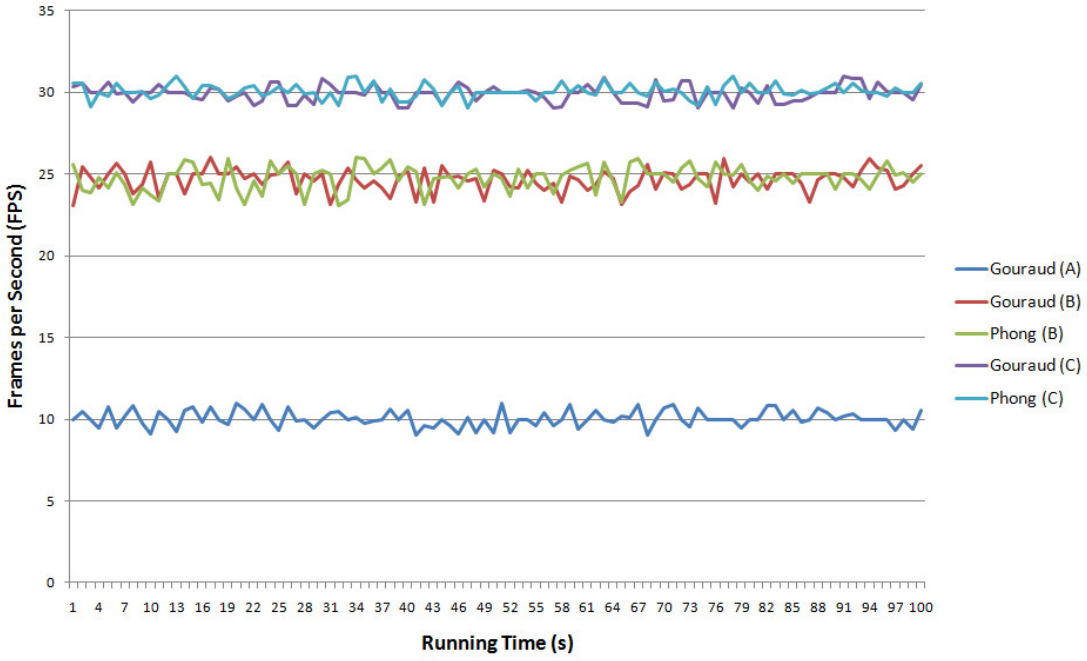


Figure 6.2: Frames per Second (FPS)

Figure 6.2 shows the frames per second (fps) rate generated by FRAPS for the Smooth and Phong illumination models we have implemented, running on each of our hardware systems. System A of the lowest performance could not render the Phong illumination model since its onboard graphics card was not programmable. For the Smooth illumination model, it could manage an average fps rate of 10 through a running time of 100 seconds. For system B both illumination models rendered at an average of 25fps whereas on the highest performing system we managed to get 30fps for both Smooth and Phong shading. Frames per second is not the same as monitor refresh rate. FPS measures how many different still images the graphics card is able to render every second. This will depend on the GPU[1] and the complexity of the scene. The refresh rate is the number of times a monitor redraws an image on the screen so that it does not flicker which depends on which frame it has received from the graphics card. Despite the fact that system A renders only at 10fps, the 3D roughness of the canvas still appears realistic as we do not have a great amount of scene motion.

[1]Graphics Processing Unit

6.2 User Testing

During the end-user testing phase we observed the users as they attempted to construct a surface model and render this using TRePS. Our key observations of some of the difficulties they encountered are as follows:

- Some users were confused by the *New* texture button on the toolbar as they were attempting to use the *Open* button to import a new painting. In a future release, we need to make this more explicit so that users do not waste time searching for how to start working on a painting.
- Most users did not use the keyboard accelerators available in the application. Instead they preferred to achieve their tasks using the buttons in the toolbar. Although this was fine for more novice users, it would speed up their usage of the system if we had made the availability of keyboard commands more visible.
- When users were manipulating image filters within the stroke and texture panels many attempted to change their order by dragging them around. This feature is currently not supported by TRePS but will be added to a future release of the system. Right-clicking the filters to expose the available options to the users also did not seem clear in certain cases as a few users attempted to find options in the menu and toolbar.
- Many users questioned why there did not exist a single button to streamline the entire process from surface generation to rendering. This was a good observation that we did not consider during design and will be incorporated within the system to aid first time users.

6.2.1 Questionnaire

Figure 6.3 illustrates the results of the questionnaire after the users had completed their evaluation of the system (Table 5.1). The system scored 79% overall but some of answers we received meant there was scope for improvement in various areas. For question 3, when asked if the system was fast enough, some users found it took long for some of the filter chains to complete their execution. This was true in some cases where it may have taken more than 30 seconds to finish composing a painting. Although we had a progress bar while filters where running, a better approach would be to show the painting being composed interactively as each brush stroke is added. To improve the performance of the system there is also scope for parallelism when processing the strokes and within the image filters themselves. Another question that received a relatively poor score was whether users were able to revert the changes they had made. Some users complained

that a one-level undo stack was not enough and that the application should support more. In our case, having a large undo stack would increase the memory requirement dramatically as all brush strokes are kept in RAM for fast filter execution. To support a greater undo stack we would have to change our storage scheme and read a stroke from the PNG file each time it was requested. This would make execution time increase but enable a larger undo stack. Users also found (according to question 10) that a greater range of image filters would be better for building surface models. As a result, we have thought of adding image filters that move towards more physical laws of paint simulation such as in the case of *Wet & Sticky* [6].

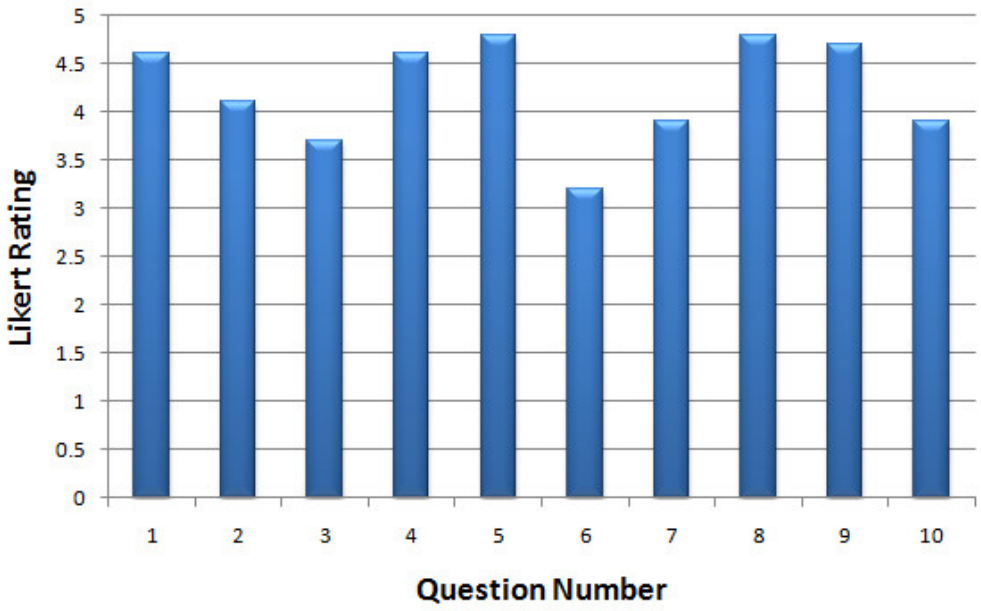


Figure 6.3: User Interface Questionnaire

6.2.2 Nielsen's Usability Heuristics

In this section, we evaluate the front-end of the system based on Nielsen's usability heuristics. The GUI can be seen in Chapter 4, Figure 4.2. The results are given below as follows:

- **Visibility of system status:** The user can view the status of the system in the status bar at the bottom of the application. Progress bars are also used whenever filters are running, as these may take a while to complete.
- **Match between system and real world:** We have created stroke and texture

panels as placeholders for image filters. Their ordering is important as it denotes how they will be executed. It also gives the user the indication which image filters are used for brush strokes and which are used for the entire canvas.

- **User control and freedom:** The application is simple enough that no complex dialogs are required. Exits are always clearly marked so that the user does not get stuck. We also support a one level undo stack so that the user can revert changes to the painting surface.
- **Consistency and standards:** We keep the terminology and icons within the application constant so that users do not need to worry if different words or actions mean different things.
- **Error prevention:** We prevent the user from making errors as much as possible, due to manual input, as we use sliders and number spinners to change the values used in filters.
- **Recognition rather than recall:** All actions the user can take are available from the central toolbox. A tooltip appears whenever the mouse hovers over a button so that users do not need to remember what each one does. This avoids the problem of information overload.
- **Flexibility and efficiency of use:** We have created a GUI that enables users to create surface models efficiently with the simple method of using image filters. In addition, we have provided keyboard accelerators that speed up the interaction with the system for more advanced users.
- **Aesthetics and minimalistic design:** The front-end was developed with the smallest possible number of components that still enables users to achieve their task. We avoid displaying unnecessary information and users have the capability to change the look and feel of the application based on their preferences.
- **Help users recognise, diagnose and recover from errors:** Whenever something goes wrong we provide error messages in a non-cryptic language. We explain what has happened and what steps are required to recover from the error. This can occur when a user attempts to import files of an incorrect format or when there is some error with filter execution.
- **Help and documentation:** A user guide is bundled with the application in PDF format which users can consult should they run into difficulties.

6.3 Robustness & Stability Testing

6.3.1 Monkey Testing

The table below illustrates the results of the monkey tests given in section 5.4.1.

	Results	Passed
1	The system detects that the stroke file was invalid and displays an error asking them to import a valid XML file	✓
2	A flat surface model is created as there are no brush strokes to emboss the painting which is as expected	✓
3	The incorrect texture is used as a surface model and the illumination model proceeds as normal. This is not correct and we have rectified this mistake by checking if the width and height of the imported texture match that of the painting	✗
4	The shading model is applied using the incorrect texture as a surface normal map. We could not avoid this, as there is no way of knowing that the RGB values at each pixel do not represent surface vectors of a strange surface	✗
5	A flat surface model is created since each stroke is transparent and no height buildup can occur which is again correct	✓
6	The illumination models run as usual but with flat surface	✓
7	A strange surface texture is constructed. No error is thrown as we give the user the freedom to experiment with all image filters	✓
8	A fatal error occurs when no camera is connected that shuts down our application. We could not manage to fix this error directly as the catch block for capturing the exception is never executed. This is because the Java wrappers for OpenCV shut down the JVM before we manage to handle the exception. There are certain external libraries that can check USB ports for connected devices but this will only work on certain platforms so we have avoided this fix. To work around this error occurring, we warn the user with a dialog to ensure a camera is connected before commencing head tracking	✗
9	An empty filter chain is loaded in the stroke and texture panels when the empty XML file is loaded which is as expected	✓
10	When an invalid filter chain is imported an error is thrown to let the user know that there is some problem with the file. We had to alter the error message slightly to make it more understandable for novice users	✓

6.3.2 Stress Testing

The table below illustrates the results of the stress tests given in section 5.4.2.

	Results
1	Large texture maps can be handled by the application and rendered correctly. The virtual camera zooms out slightly to make the entire canvas visible within the width and height range of the monitor. Texture size also depends on the graphics card and system C (section 5.1.1) can support sizes of more than 8000 pixels in both width and height.
2	The system can handle paintings with brush strokes in excess of 50,000 which is far greater than the average working size we have been using throughout the project. Unfortunately, when moving into the hundreds of thousands of strokes the application runs out of Java heap space. To fix this limitation we would need to read each file from disk every time it was requested increasing processing time considerably.
3	When a filter chain of more than 20 image filters is executed it often takes more than 3 minutes to complete the surface model of the painting. The increased execution time is expected as the chain is run for thousands of brush strokes. The application runs normally without any degradation in performance allowing users to work with more paintings in other tabs if required.
4	The system can handle parallel execution of surface models for multiple paintings up to the maximum memory heap set for the JVM. For a maximum of 1GB, TRePS is able to process 5 paintings with an average stroke count of 5,000. To obtain better performance it is recommended to run at most 3 surface model constructions in parallel as this leaves some memory for the canvas renderer.
5	Rendering multiple illumination models in parallel depends on the maximum available memory allocated to the Java heap and the particular graphics card connected to the system. System C can render 5 different paintings at the same time of an average brush stroke size of 5,000 and an image of 1200 pixels in width and height. Systems A and B begin to show some noticeable decrease in performance when running multiple lighting models. As a result, on such systems we recommend rendering only one canvas at a time.

6.3.3 Motion Tracking

We now describe the results of the tests carried out to measure the accuracy and robustness of our head tracking module. The actual tests are explained in more detail in section 6.3.3 and the cameras used are listed in section 5.1.2.

6.3.3.1 Speed and Accuracy

For the first motion tracking test we measure the speed and accuracy of detection for each of the five points bins at a set angle and distance range from the camera as shown in Table 5.2. Figures 6.4 and 6.5 display the results of our findings regarding speed and accuracy of camera detection. Looking at the graphs, the reader will notice that we have no data for camera A (Microsoft LifeCam VX-5000). Unfortunately, when we connected this to the system we received no data and it appears it is one of the models that is not compatible with the OpenCV libraries. It seems the camera model does not play a role in head tracking as none of the models B,C or D seemed to perform better overall. What we have noticed during experimentation is that accuracy decreases as the angle from the centre of projection of the camera increases. This is due to the fact that moving to wide angles shifts the face to the edge of the tracking window. In such cases, some of the viewer's face will no longer appear in the tracking window and as a result face detection no longer works which is the reason for the poor accuracy for point 3. Increasing distance from the camera does not seem to affect results and we have detected faces up to 2 metres away. We must note that moving relatively close to the camera can cause face detection to fail altogether and as a result have chosen our points to begin from a minimum distance relative to those specified in section 5.1.2.

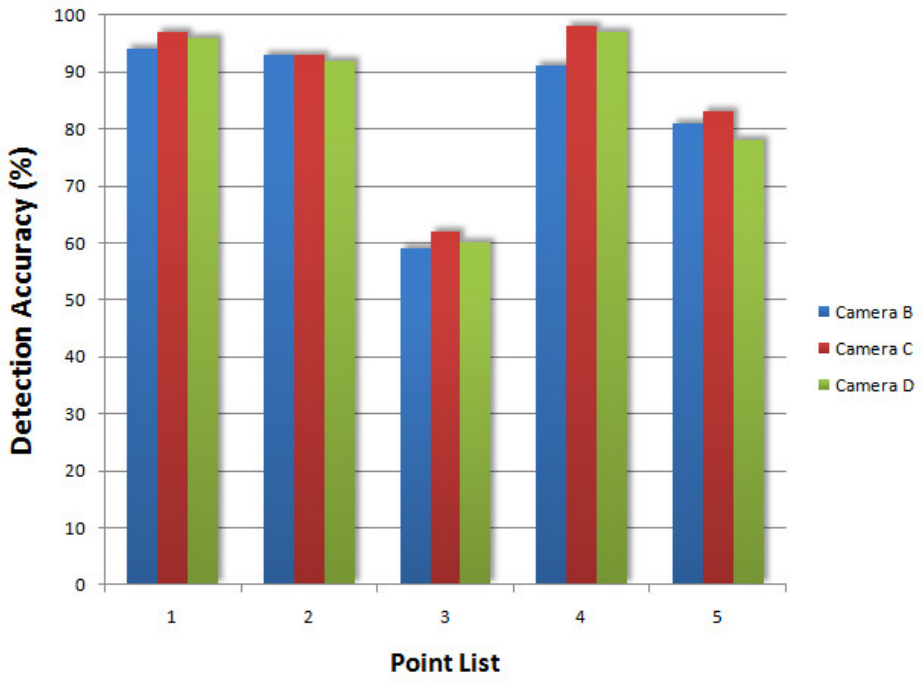


Figure 6.4: Camera Detection Accuracy

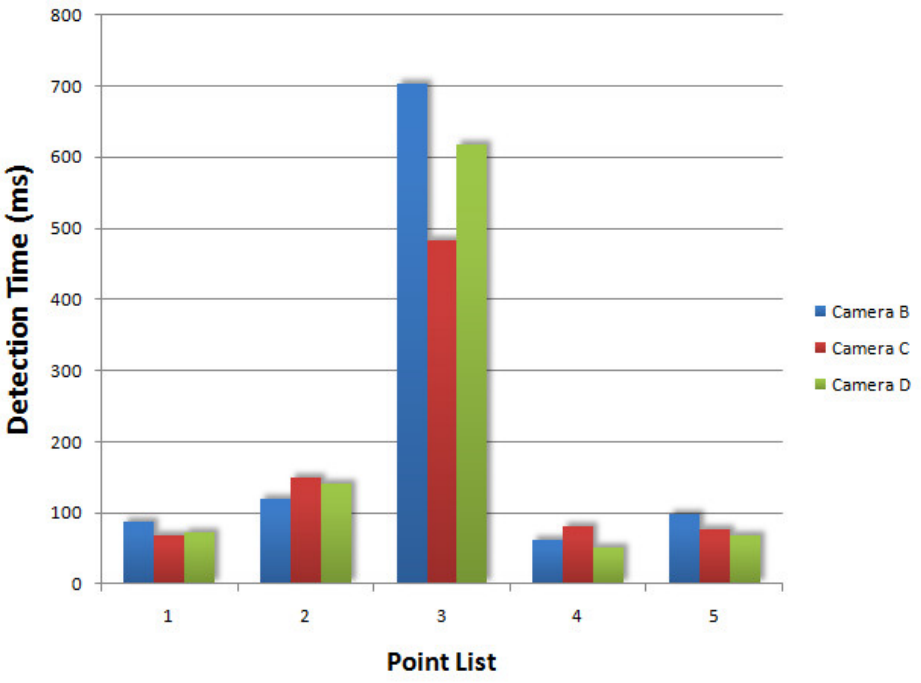


Figure 6.5: Camera Detection Time

6.3.3.2 Multiple Faces

We also tested the system with multiple faces appearing in the tracking window. As expected, only the largest face was detected in the window was used for head tracking. By shifting faces within the window we found that as soon as they changed order in terms of size the new largest face was detected and the viewer coordinates updated accordingly. When this event occurred there was jerkiness in the rendering window as the illumination conditions change to match those of the new viewer. We cannot do much to avoid this, although it is not a major issue since the viewing experience is only suitable for one user.

6.3.3.3 Variable Backgrounds

To test the robustness of the head tracking system we tested the viewer facing the camera had different backgrounds behind them. Changing the background colour did not make any difference to either the accuracy or detection time. On the other hand, using highly textured surfaces increased the detection time and decreased the accuracy

slightly, as in some cases incorrect objects in the background where identified as faces. Although the results regarding detection accuracy were not altered by much (-15% approximately), it is recommended to use the system with a simple background to the camera. This is because in cases where incorrect objects are detected, there is a sudden change in the illumination of the canvas as it updates lighting conditions, causing the viewing experience to be degraded.

6.3.3.4 Variable Lighting

By changing the lighting conditions in the room we were able to measure the detection accuracy of the system. What we noticed from these tests was that neither accuracy nor detection speeds were affected by using a range of brightness conditions. The only cases where face detection began to fail, was under extreme bright and dark conditions where the face blends in with the rest of the environment.

6.4 Aesthetics Evaluation

Based on the results of the aesthetics questionnaire in Figure 5.1, users seemed to appreciate the 3D realism that we had added via both Gouraud and Phong shading. This was the case as none had voted for the flat surface as being the best during the first round of viewing. Between the two shading models, most users opted for the more advanced Phong illumination model with 7 votes in comparison to 3 for the smooth shading method. The reason that some users preferred Gouraud shading to the Phong model was because the latter was too shiny, exaggerating the impasto effect. Based on the results of questions 2,4 and 5 there seems to be no precise rendering that users preferred. Some liked the thick surface model generated, as it increased the level of shadowing. Others enjoyed the orange light used in one of the monitors during the second viewing round as this illuminates paint ridges over a much wider area of the canvas which was not particularly visible when using simple white light. In question 3 of the questionnaire 8 users opted that head tracking improved their viewing experience whereas the others disagreed. This gives us a strong indication that when head tracking is enabled users do actually feel as if the are in an art gallery. Figure 6.6 shows the results of the votes when users were asked to rank each painting during each viewing round. In round 1 Phong shading is preferred to Gouraud and flat shading whereas in round 2 there is slightly more appreciation for the painting with the different coloured light (see Appendix, Figure 8.6).

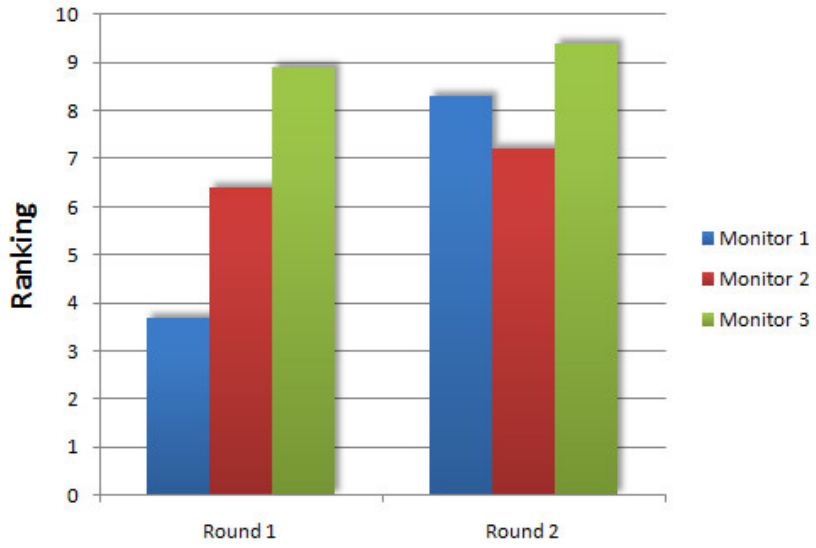


Figure 6.6: Painting Rankings

6.5 Platform Compatibility

Table 6.2 shows the results of running TRePS on each of the three operating systems as described in section 5.6. The development of the system was carried out on a Windows platform so all the modules that were tested ran smoothly as expected. On *Ubuntu* we were able to achieve the same results as on *Windows 7* without any particular problems. The only difference was that TRePS starts up with a different look-and-feel to the GUI which is as expected since it uses the JVM default. On the Mac OS platform we managed to construct a surface model and run the Gouraud shading model. Unfortunately, the Phong shading model did not work and as a result head tracking could not be loaded. The reason for this is that the Java 3D libraries for Mac are on an older version than other platforms as they access native libraries to perform hardware rendering. We hope that this will be fixed in future releases so that TRePS's full functionality can be used.

	Surface Modeler	**Smooth Shading**	**Phong Shading**	**Head Tracking**
Windows	✓	✓	✓	✓
Linux	✓	✓	✓	✓
Mac OS	✓	✓	✗	✗

Table 6.2: Platform compatibility results

6.6 Summary

In this chapter we presented the results of the experiments we carried out described in chapter 5. The performance of the system was analysed by testing the filter execution time and FPS count on different hardware platforms. We undertook an end-user evaluation of the front-end GUI and identified some issues that need to be improved in future releases. A few errors were discovered by monkey testing and operating limits were found using stress testing. Section 6.3.3 shows the results of the robustness of our head tracking module under different operating conditions by using a range of camera setups. Finally section 6.4 illustrates the results of the viewing experience with a group of ten users. We found that the parameters for the rendered paintings depend more on their personal preferences than anything else.

7 Conclusion and Future Work

7.1 Conclusion

Within this report we have given a detailed outline of how digital art may evolve in the future. We have taken flat static images and turned them into dynamic pieces of digital art where the user can observe the impasto look of thick paint. The idea we have presented may form a new method of viewing art in galleries.

We have shown the strengths of bump mapping using surface normal maps to emboss the texture of the canvas. Canvas illumination using programmable shaders, gives the developer the opportunity to modify the fixed graphics pipeline. We have taken this approach and incorporated our own shading model based on Phong lighting to improve the realism of paintings. To avoid the problem of our system being usable only on programmable graphics cards we have also implemented a simpler shading model based on the Gouraud illumination.

Using the Gouraud and Phong models we were able to embed head tracking using cameras within the system to significantly improve the viewing experience. We have chosen the use of simple cameras in comparison to more complex head mounted devices as we want to avoid the need for the viewer to possess any special equipment. Based on our results from chapter 6, we have deduced that its accuracy is great enough for the purpose of motion tracking (assuming the correct conditions are in place).

When developing the system we used an object-oriented approach with Java as the main programming language. Using a layered software engineering design we able to break up the system into a number of distinct modules. These can easily be extended to improve the capabilities of our application, TRePS, such as adding more image filters for paint buildup or shading models for illumination.

The TRePS system we have built covers all major project aims as documented in Chapter 1, section 1.2. Based on the evaluation we have carried out, users have found that our methods greatly enhance the realism of impasto paintings. As a result, our system can act as a starting point for much larger research projects in this creative area and we hope that more work will continue on such an interesting idea.

7.2 Future Work

Below we list some pieces of future work that would greatly enhance the TRePS system. In addition, there are some areas of creativity that we have unfortunately not had the time to explore and these are also examined.

- **Multiple Cameras:** The head tracking code that we have written uses the Viola-Jones object detection framework to measure the motion of the viewer relative to the canvas. To achieve this we used only a single camera. Unfortunately, this reduces the tracking window scope so that viewing the canvas at large grazing angles is not possible. In order to make this feasible we would need to incorporate a number of cameras that cover a 180° hemisphere view around the painting. This will enable motion detection over a larger area that is not possible using a single camera.

- **Detecting Illumination Conditions:** A very interesting extension to this project is to avoid using virtual light sources within the canvas. Instead, we could take advantage of the connected camera(s) so that illumination conditions of the room are converted to light sources inside the virtual world. This can be used to automatically calibrate the system with an art galleries conditions. This could be achieved with some sort of localisation and mapping algorithm such as SLAM.

- **Additional Lighting Models:** Due to time constraints we have used only the Gouraud and Phong illumination models. An improvement would be to incorporate the Cook-Torrance model (section 2.2.4) that represents surfaces using microfacets and accounts for both wavelength and colour shifting. A rigorous examination of the bidirectional surface scattering reflection distribution function [15] (BSSRDF) of painting surfaces would also be useful in deciding whether more complex models add any further realism to impasto paint. Using programmable shaders would also give developers the freedom to experiment with lighting conditions that deviate from reality so that more creativity can be added.

- **Three-Dimensional Models:** To emboss the painting surface of the canvas we have used bump mapping that alters surface normals rather than actually displacing the geometry of vertices. By using bump mapping in TRePS we have managed to reduce the computational load in comparison to displacement mapping. Despite this fact, we have carried some initial experiments using Blender[1] for displacement mapping and have managed to produce some 3D sculptures from

[1] Blender is an open-source 3D content creation suite with a range of powerful capabilities written in C, C++ and Python. More information can be found on their website at http://www.blender.org

the painting canvas. This was produced using generated texture maps exported by the TRePS system.

- **Paint Colour Mixing:** The brush strokes we have been working with did not use any colour mixing when blending strokes together in a back-to-front manner. Rather than just improving the surface models that we generate, we can also attempt to increase realism by modifying the appearance of the colour texture. This will involve finding a technique to compose the colours of brush strokes as done in the *ArtRage* (section 2.7.1) and *IMPaSTo* (section 2.7.3) painting systems.
- **LiveView Painting:** Another interesting extension that can be added to the system is to build the surface texture of the canvas as the renderer draws the strokes. This will make the painting process extremely realistic. The viewer will be able to examine the fine shading of the painting as strokes are drawn on the canvas surface. Examples of *The Painting Fool* (section 2.7.4) demonstrating its artistic capabilities can be viewed on the software's webpage[2].
- **Optimisation Improvements:** As discovered from stress testing the application can only handle paintings of up to 50,000 strokes since these are buffered in memory for faster execution. To be able to process paintings with a far larger number of strokes we would need to change the internal storage model of the system so that these are requested from the hard disk whenever they need to be placed on the canvas. This approach would cater for far larger portraits but increase filter execution time.

7.3 Closing Remarks

We hope that our innovative idea of improving the realism of impasto paintings can be used as a starting point for future work in this area. Imagine being able to produce realistic surface models using either active laser or stereo vision. Such paintings could then be embossed by our system and then sent over the Internet to another part of the world for dynamic display. Such ideas are not too far away and may soon emerge as a new virtual viewing experience that people can take advantage of. Some of the paintings produced by *The Painting Fool* which have been embossed by our system can be seen in the Appendix.

[2] http://www.thepaintingfool.com/demo/index.html

Bibliography

[1] Edward Angel. *Interactive Computer Graphics: A Top-Down Approach Using OpenGL*. Addison-Wesley, 4th edition edition, 2006.

[2] Arthur Appel. Some techniques for shading machine renderings of solids. In *AFIPS '68 (Spring): Proceedings of the April 30–May 2, 1968, spring joint computer conference*, pages 37–45, New York, NY, USA, 1968. ACM.

[3] William Baxter, Jeremy Wendt, and Ming C. Lin. Impasto: a realistic, interactive model for paint. In *NPAR '04: Proceedings of the 3rd international symposium on Non-photorealistic animation and rendering*, pages 45–148, New York, NY, USA, 2004. ACM.

[4] James F. Blinn. Models of light reflection for computer synthesized pictures. *SIGGRAPH Comput. Graph.*, 11(2):192–198, 1977.

[5] James F. Blinn. Simulation of wrinkled surfaces. pages 286–292, 1978.

[6] Tunde Cockshott, John Patterson, and David England. Modelling the Texture of Paint. *Computer Graphics Forum*, 11(3):217–226, September 1992. EG92: Cambridge, UK., Editors: A. Kilgour and L. Kjelldahl.

[7] Tundee Cockshott. *Wet and Sticky: A Novel Model for Computer-Based Painting.* PhD thesis, The University of Glasgow, UK, 1991.

[8] Simon Colton, Michel F. Valstar, and Maja Pantic. Emotionally aware automated portrait painting. In *DIMEA '08: Proceedings of the 3rd international conference on Digital Interactive Media in Entertainment and Arts*, pages 304–311, New York, NY, USA, 2008. ACM.

[9] R. L. Cook and K. E. Torrance. A reflectance model for computer graphics. *ACM Trans. Graph.*, 1(1):7–24, 1982.

[10] E. R. Davies. *Machine Vision: Theory, Algorithms, Practicalities.* Morgan Kaufmann, 2005.

[11] Yoav Freund and Robert E. Schapire. A decision-theoretic generalization of on-line

learning and an application to boosting, 1997.

[12] Henri Gouraud. Continuous shading of curved surfaces. pages 87–93, 1998.

[13] Aaron Hertzmann. Fast paint texture. In *NPAR '02: Proceedings of the 2nd international symposium on Non-photorealistic animation and rendering*, pages 91–ff, New York, NY, USA, 2002. ACM.

[14] Atelier hypermédia at the École Supérieure d'Art d'Aix-en Provence. OpenCV Library. `http://ubaa.net/shared/processing/opencv/`, May 2010.

[15] Henrik Wann Jensen, Stephen R. Marschner, Marc Levoy, and Pat Hanrahan. A practical model for subsurface light transport. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 511–518, New York, NY, USA, 2001. ACM.

[16] D. Marr and E. Hildreth. Theory of edge detection. *Proceedings of the Royal Society of London. Series B, Biological Sciences*, 207(1167):187–217, 1980.

[17] Jakob Nielsen. Ten Usability Heuristics. `http://www.useit.com/papers/heuristic/heuristic_list.html`, 2010.

[18] Bui Tuong Phong. Illumination for computer generated pictures. *Commun. ACM*, 18(6):311–317, 1975.

[19] Randi J. Rost. *OpenGL Shading Language*. Addison-Wesley Publishers, 2006.

[20] Christophe Schlick. An inexpensive BRDF model for physically-based rendering. *Computer Graphics Forum*, 13:233–246, 1994.

[21] H. A. Sowizral and M. F. Deering. The Java 3D API and Virtual Reality. 19(3):12–15, May 1999.

[22] Henry Sowizral, Kevin Rushforth, and Michael Deering. *The Java 3D API Specification, Second Edition*. Addison-Wesley, 2000.

[23] Thomas Strothotte and Stefan Schlechtweg. *Non-Photorealistic Computer Graphics: Modeling, Rendering, and Animation*. Morgan Kaufmann Publishers, 2002.

[24] P. Viola and M. Jones. Robust real-time face detection. In *Proc. Eighth IEEE Int. Conf. Computer Vision ICCV 2001*, volume 2, page 747, 2001.

[25] Turner Whitted. An improved illumination model for shaded display. *Commun. ACM*, 23(6):343–349, 1980.

8 Appendix

Figure 8.1: Painting Colour Texture

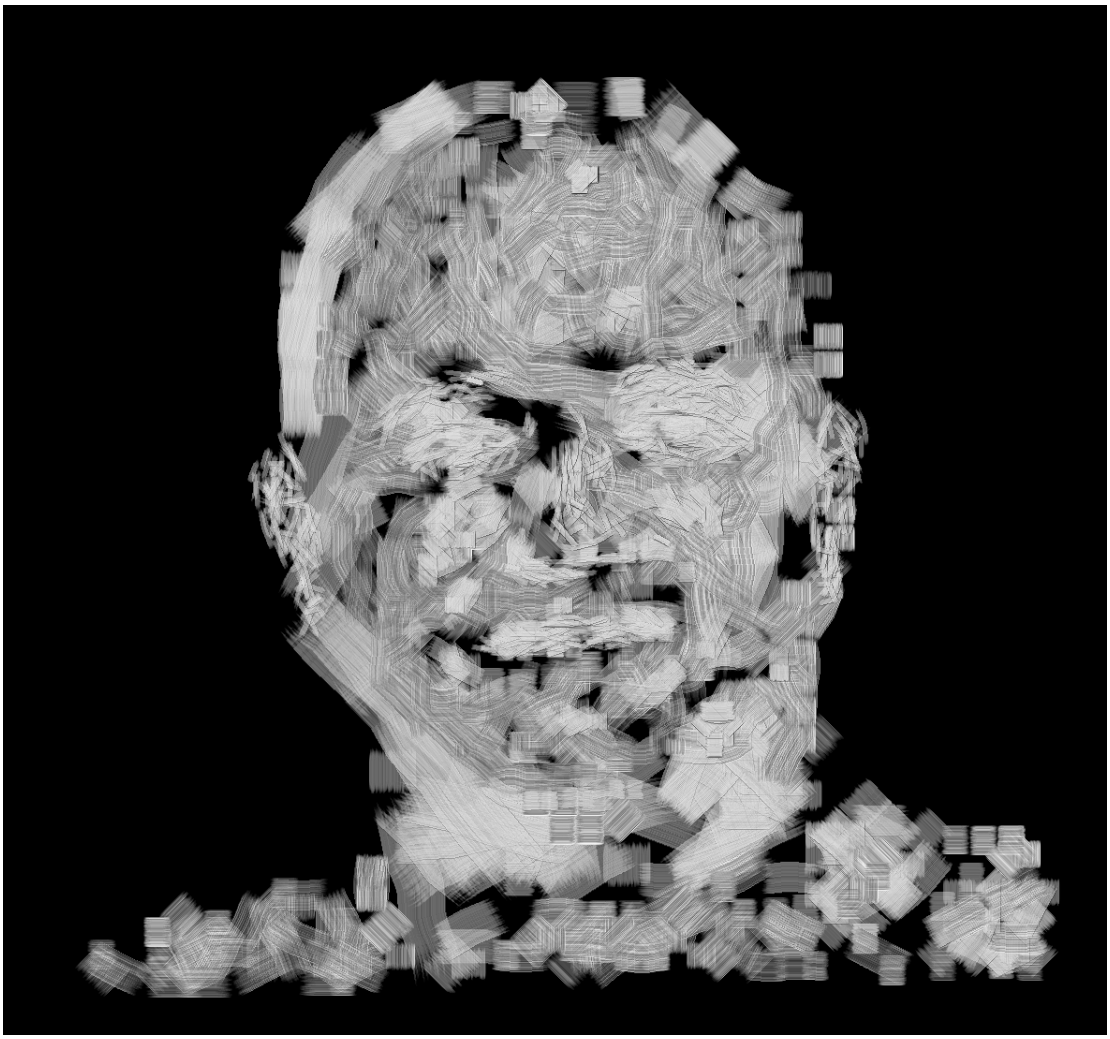


Figure 8.2: Painting Height Map

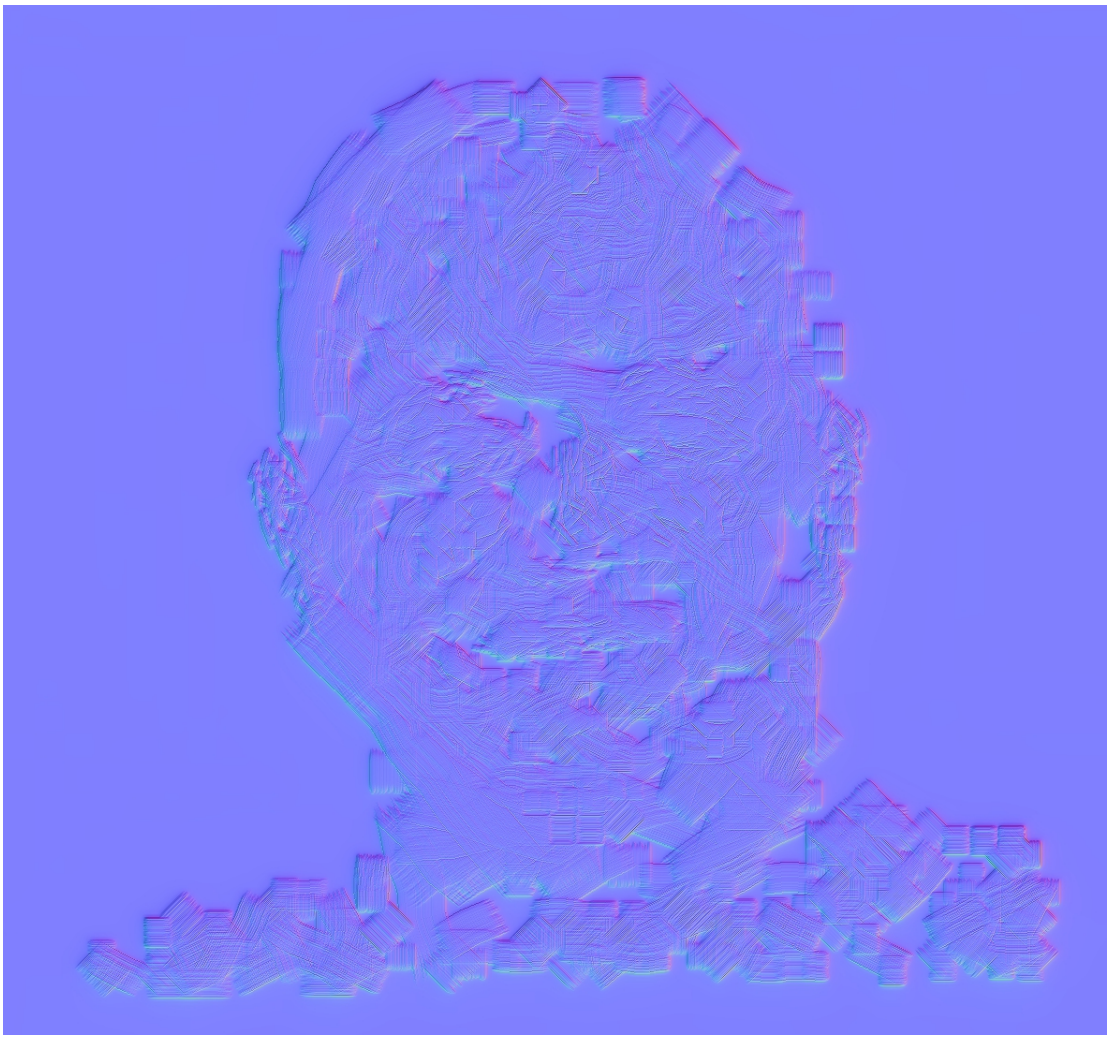


Figure 8.3: Painting Surface Normal Map

Figure 8.4: Gouraud Shaded Painting

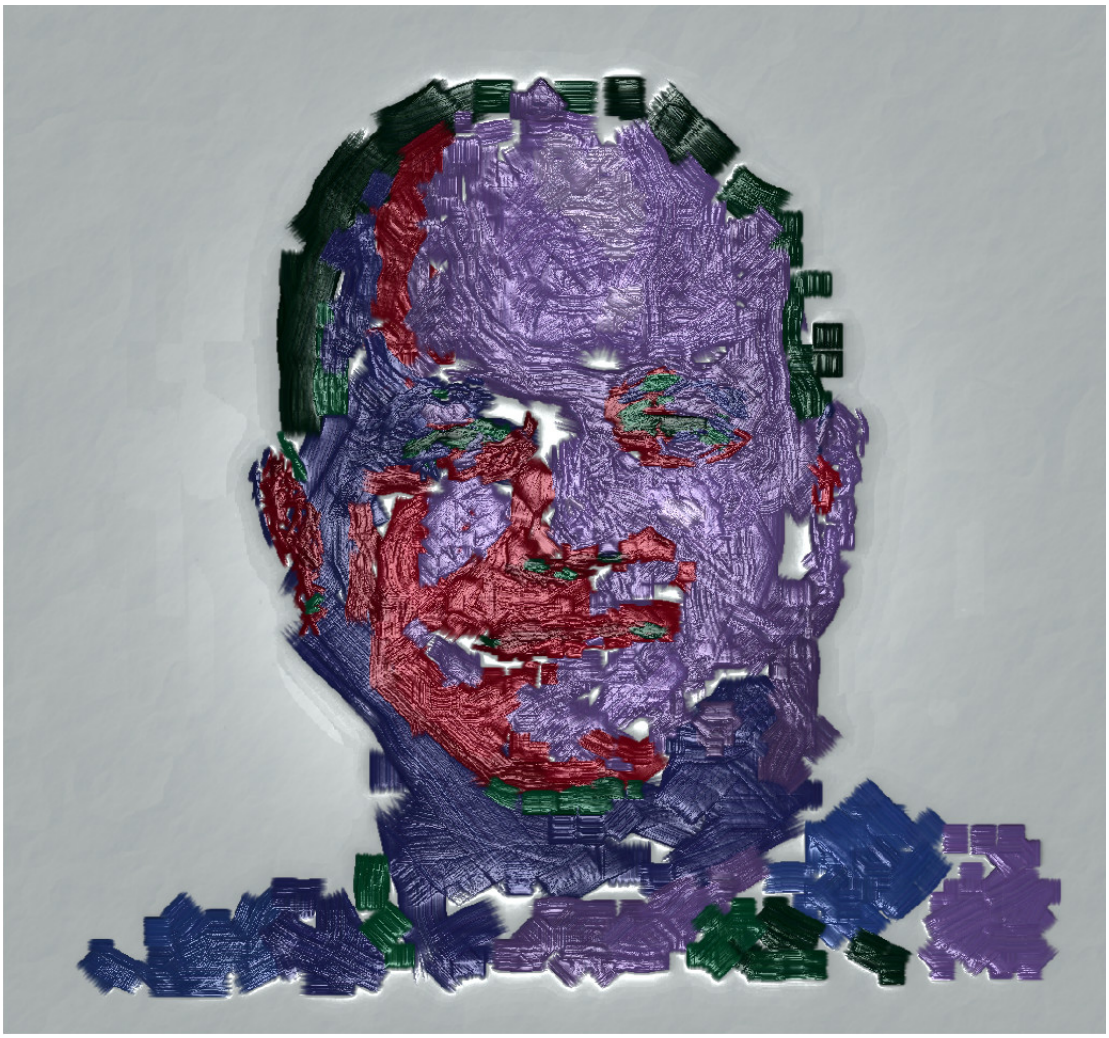


Figure 8.5: Phong Shaded Painting

Figure 8.6: Phong shaded painting using orange light and very thick paint