Project Report

Geometry-based shading for shape depiction enhancement

An Implemetation

Real-Time Graphics Programming Project

Author:
Francesco Brischetto

Student Number:
958022

Submitted to:
Dr. Davide Gadia

7th March, 2022

Table of Contents

Executive Summary

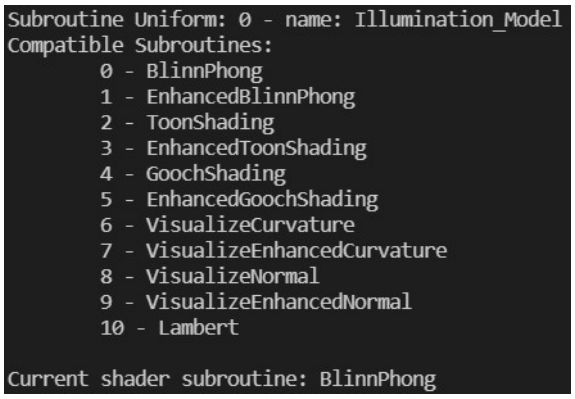
This project report describes the realization and implementation decisions of the shading technique proposed in [1].
The paper's goal is to describe an approach for **enhancing shape depiction** of 3D objects on **Non-Photorealistic Rendering (NPR) shading models** using *local geometry.*
The approach described provides **time-efficiency**, to make it usable in *interactive application*, without any constraints on the choice of material or illumination model.
The proposed method is inspired by **Normal Enhancement** and **Radiance Scaling** Shading approach and it tries to combine the advantages of both techniques while, also, relaxing their constraints.
The overall goal of my project is to implement some of the newest techniques in NPR as well as coding some of the most famouse ones.

1 Introduction

The advent of Computer Graphics has **not replaced artists**. Many non-photorealistic rendering techniques have focused on depicting shape through shading that *mimic hand-drawn illustration* but NPR models still **lack in expressive power**.
The proposed technique should *correlate* the enhancement functionalities to **surface feature variations**. To achieve this result, the paper [1] starts from analyzing two existing techniques that either perturbates the surface normal as in **Normal enhancement** or alterates reflected radiance based on local surface information as in **Radiance scaling**.
However, both of those methods have some limitations:
Normal enhancement operators are *restricted* on specific types of material and illumination models, and they are **not able to enhance** some important geometry features such as **concavities** and **convexities**. In contrary, Radiance scaling overcomes those limitations but (expecially in NPR shading) tends to mask subtle shading variations and hence **reduce effectiveness** of the overall technique.
The proposed techinque's goal is to *reformulate* NPR shading models with respect to **geometry surface features** combining the advantages of Normal enhancement and Radiance scaling while also **relaxing their constraints**.
The enhancement is achieved in two ways. First, by *modifying the surface normal* using a simple high-frequency enhancement operation. Second, by correlating reflected lighting intensity to **surface curvature** using a new scaling function.
In this way, we can achieve enhacement in diffierent extent, allowing users to produce more desirable enhacement results.

1.1 Project Overview

In this project I implemented all the shading models used in [1] :
Blinn-Phong, **Cartoon** and **Gooch** as well as their **enhanced version**, following the reformulation of the *reflectance radiance equation* described in paragraphs 6.1, 6.2 and 6.3 of [1].
Contextually, I realized also all the needed **helping operators**, such as *normal smoothing*, *sharpening* and *curvature* calculation. Lastly, I realized other *subroutines* to better visualize the differences between normal and enhanced normal, as well as curvature and enhanced curvature (that uses enhanced normal).
A list of all the usable subroutine in the project can be seen in **Figure 1**.



```
Subroutine Uniform: 0 - name: Illumination_Model
Compatible Subroutines:
        0 - BlinnPhong
        1 - EnhancedBlinnPhong
        2 - ToonShading
        3 - EnhancedToonShading
        4 - GoochShading
        5 - EnhancedGoochShading
        6 - VisualizeCurvature
        7 - VisualizeEnhancedCurvature
        8 - VisualizeNormal
        9 - VisualizeEnhancedNormal
        10 - Lambert

Current shader subroutine: BlinnPhong
```

Figure 1: All subroutines available in the project

2 Geometry-based shading Approach

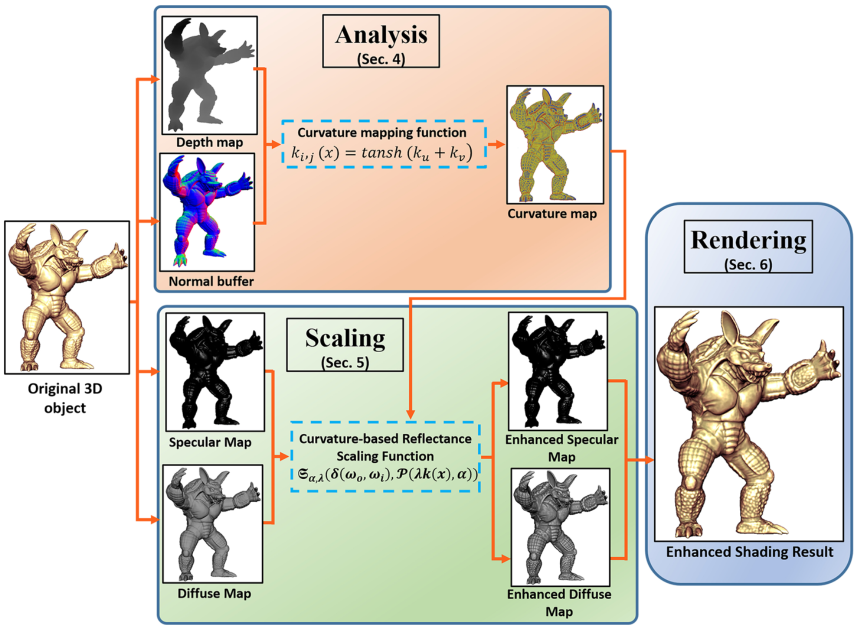
The key of this approach is *incorporating surface curvature* information into shape depiction enhancement technique. This is done with multiple steps:

1. **3D Shape Descriptor:** 3D Object surface shape is analyzed using the shape descriptor. **Curvature computation** extracts salient surface features. The contribution can be controlled using some parameters.

2. **Geometry-based Shading:** Surface normals are *smoothed* or *sharpened* to attenuate or exaggerate the surface depiction; Then, the **reflected lightning intensity** is scaled using also surface curvature previously calculated.

3. **Rendering:** This shading technique is applied to various **non-photorealistic rendering styles**, reformulating the reflected radiance equation in a way that takes the enhanced shading into account.

In **Figure 2** the rendering pipeline is graphically illustrated.



Analysis
(Sec. 4)
Depth map
Normal buffer
Curvature mapping function
$k_{i,j}(x) = tansh(k_u + k_v)$
Curvature map
Original 3D object
Scaling
(Sec. 5)
Specular Map
Diffuse Map
Curvature-based Reflectance Scaling Function
$\mathfrak{S}_{\alpha,\lambda}(\delta(\omega_o,\omega_i),\mathcal{P}(\lambda k(x),\alpha))$
Enhanced Specular Map
Enhanced Diffuse Map
Rendering
(Sec. 6)
Enhanced Shading Result

Figure 2: The rendering pipeline of the paper's approach. It combines estimating the surface curvature, scaling the reflected lighting and rendering the NPR result

3 Design and Implementation

In this chapter, I will cover all the implementation choices that I made during the realization of the project. Some of the operations described in the paper [1] where not clearly specified so I was forced to take some implementation decisions and use the paper as a guideline. Moreover, some operations where too general and customizable so I tried to simplify it to make the project simpler.
Code-wise, My project started from the code version **lecture03a** that we saw during lectures. I also used 3 standard models in Computer Graphics for testing, downloaded from **Stanford** website[1]: *Bunny*, *Armadillo* and *Dragon*.

3.1 Normal Enhancement Operators

The paper [1] describes two types of normal variations: *smoothing* and *sharpening*. Those two kind of enhancement operators works on the high-frequency components of the surface normal. Smoothing reduces those components, while Sharpening increase them.

3.1.1 Normal Smoothing Operator Implementation

To calculate smoothed surface normal I started from this idea[2]. The implementation of normal smoothing operator was done in model_v1 file, while *processing the Assimp mesh* in order to obtain an *OpenGL mesh*. I created a **new position in the vertex attribute**, that stores smoothed normal.
In this way, data can be loaded in vertex shader by simply add the line in **Code 1**.
The smoothing normal computation is done for each vertex. I **initialized** it to vector zero. Then, I **computed face normal** for each face of the mesh and I *added* it to each vertex of the face. Finally, for each vertex, I **normalized the result** of this sum.
By doing so, I implicitely consider the *kernel* of convolution as the *smallest one*, considering as **neighbourhood** only the faces that share the same vertex. I also implicitely defined the σ parameter, found in Equation 5 of 4.2.1 chapter of [1] that *weights* this operation as *1*. This greatly **simplifies** computation and code while maintaning the same *intent* as the original paper. The added lines of code are showed in **Code 2**.

```
layout (location = 2) in vec3 sm_normal;
```

Code 1: Smoothed normal loaded in vertex shader

```
for(GLuint i = 0; i < mesh->mNumVertices; i++)
{
  Vertex vertex;
  ...
  vertex.Sm_Normal = glm::vec3();
  ...
}
// For each face, I calculate the face normal and I add to the vertices' normal used
  by the face. Finally, I normalize the normal to obtain the smoothed surface normal.
for ( int i = 0; i < indices.size(); i += 3 )
{
  // I Compute the face normal using triangleNormal method, that computes normal
  starting from triangle points
  glm::vec3 faceNormal = glm::triangleNormal(
  vertices[indices[i]].Position,
  vertices[indices[i+1]].Position,
  vertices[indices[i+2]].Position);

```

[1] http://graphics.stanford.edu/data/3Dscanrep/
[2] https://www.reddit.com/r/opengl/comments/6976lc/smoothing_function_for_normals/

```



    // I add face normal to each of the 3 vertex normal of the face
    for ( int j = 0; j < 3; j++ )
    {
      vertices[indices[i+j]].Sm_Normal += faceNormal;
    }
  }
  for (auto &v : vertices)
  {
    // Normalizing the vectors accumulating face normals to obtain the smoothed surface
     normal
    // NOTE: Sigma parameter, found in Equation 5 of 4.2.1 chapter of the reference
    paper ( to control the quantity of convolution kernel used ) is implicitely 1.
    v.Sm_Normal = glm::normalize(v.Sm_Normal);
  }
```

Code 2: Smoothed normal computations added in processMesh method of model_v1 file

3.1.2 Normal Sharpening Operator Implementation

Normal Sharpening operator is implemented in **fragment shader** and uses the smoothed normal previously described and passed to the *vertex shader*.
In vertex shader, I apply *normal Matrix transformation* before passing them to the fragment shader as showed in **Code 3** .
In fragment shader, I calculate the **mask** by subtracting the original normal vector with its smoothed version. Then, I add the mask multiplied by a **scaling factor** λ to the normal vector, with a typical process called **unsharp masking** as described in Equation 6 of 4.2.2 chapter of [1]. Finally, I normalize the result.
The **Code 4** is added to the fragment shader to implement this process.

```
vSMNormal = normalize( normalMatrix * sm_normal );
```

Code 3: Transformation applied to smoothed normal in vertex shader

```
// Computing the mask for Unsharp Masking
vec3 mask = vNormal - vSMNormal;
// calculating enhanced Normal using the Unsharp Masking technique. This is defined,
  in the reference paper, in equation 6 of chapter 4.2.2
vec3 eNormal = vNormal + lambda * mask;
// normalization of the per-fragment enhanced normal
vec3 N_I = normalize(eNormal);
```

Code 4: Calculation of normal sharpening operator in fragment shader

A Comparison between normal vector and enhanced normal vector (through sharpening) is showed in **Figure 3** that shows two subroutines created in the project.

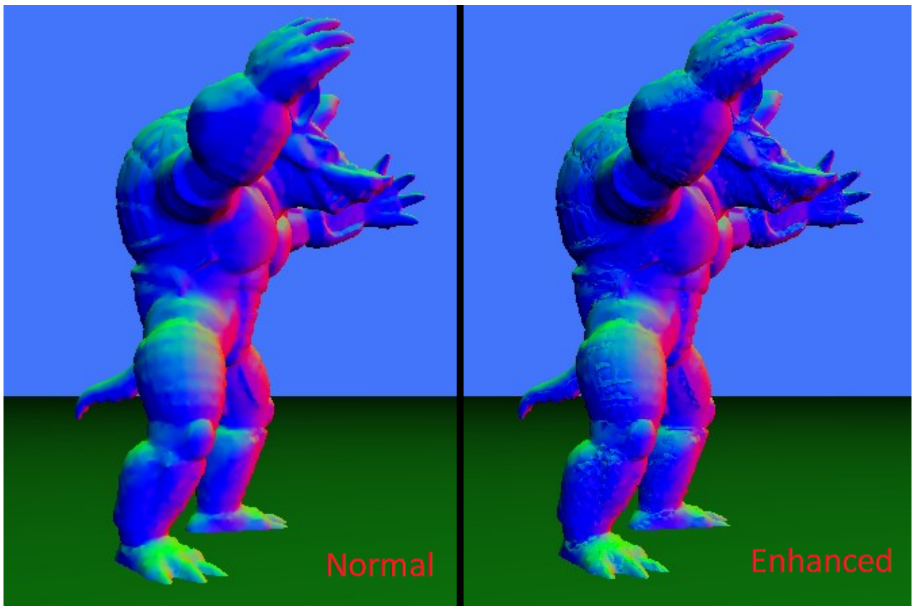


Figure 3: Comparison between normal and enhanced normal vector (through sharpening)

3.2 Curvature Analysis

Curvature analysis is another key point of the technique described in [1].
However, curvature calculation was not so clear and straightforward in the paper because they introduced the **second fundamental tensor** and the computation of **Hessian of depth field** by differentiating the gradient with a *Sobel filter*.
The approach implemented in my project, instead, is simpler. It starts from here[3] and considers **screen-space normals of neighbouring** fragments as well as **depth of the current fragment** to compute a curvature value.

3.2.1 Screen Space Surface Curvature Implementation

Surface curvature is computed in screen space, using OpenGL *partial derivatives functions* of the normal **dFdx**, **dFdy** and the *depth* of the current fragment. However, we cannot use **z-values** directly, but we need to *linearize* them as it is specified in openGL documentation[4].
Because of projection properties, a non-linear depth equation is used and it is proportional to 1/z. For this reason, we have good precision when z is small, so the object is *close to the camera*, and much less precision when it is far away.
As a result, we need to **transform** non-linear depth values of fragments back to its linear form in order to use them in *surface curvature* calculation. To achieve this, we need to *revert* the process of projection, re-transforming the values to **normalized device coordinates (NDC)** and applying the inverse equation, using far and near planes.
Surface Curvature and Linearized Depth functions are visible in **Code 5**.

```

float LinearizeDepth(float depth)
{
  float z = depth * 2.0 - 1.0; // back to NDC
  return (2.0 * near * far) / (far + near - z * (far - near));
}


```

[3]https://madebyevan.com/shaders/curvature/
[4]https://learnopengl.com/Advanced-OpenGL/Depth-testing

```
float curvature(vec3 N_I)
{
  // We compute curvature exploiting partial derivatives of the Enhanced Surface
  Normal
  vec3 dx = dFdx(N_I);
  vec3 dy = dFdy(N_I);
  float depth = LinearizeDepth(gl_FragCoord.z);
  float curvature_value = (cross(N_I - dx, N_I + dx).y - cross(N_I - dy, N_I + dy).x)
   * 4.0 / depth;
  return clamp(curvature_value, -1, 1);
}
```

Code 5: Curvature and LinearizeDepth functions in fragment shader

A Comparison between curvature computation using normal vector or enhanced normal vector (through sharpening) is showed in **Figure** 4 that shows two subroutines created in the project.

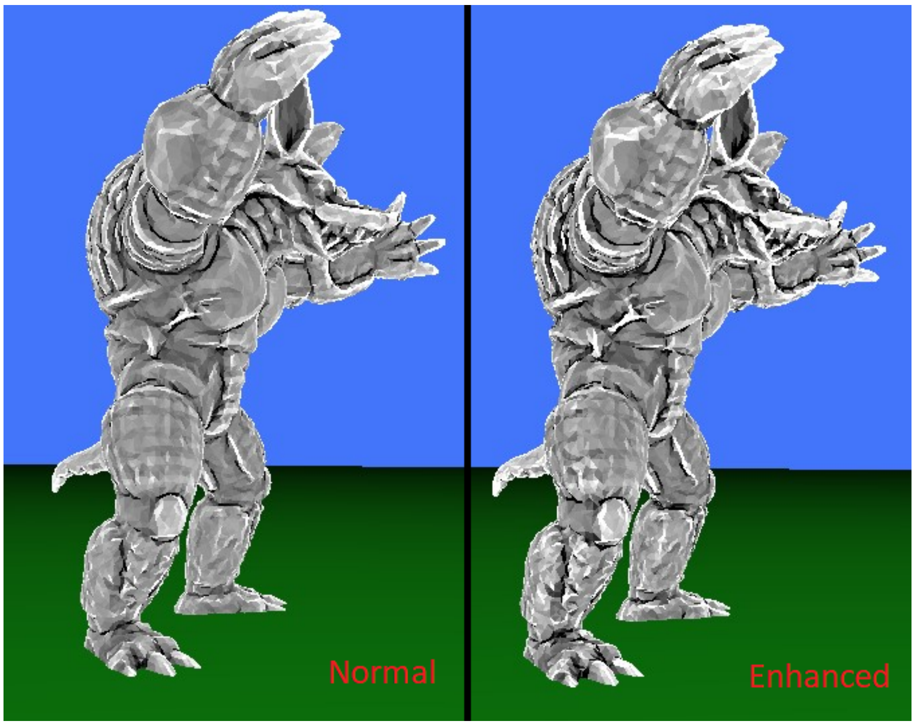


Figure 4: Comparison between curvature computation using normal vector or enhanced normal vector (through sharpening)

3.3 Non-Photorealistic Rendering shading styles

As previously said, the goal of [1] is to render 3D objects *without any constraint* on the choice of material or illumination, while taking into consideration the way of Geometry-based Shading technique that *enhances the lighting* at each surface point.
To achieve this, the authors of [1] demonstrated their approach with three different **non-photorealistic shading styles**: *Blinn-Phong Shading*, *Cartoon shading* and *Gooch* Shading. For each of those styles, they **adapted** their technique by choosing properly an **intensity mapping** function δ to be used in the reflected radiance equation.

3.3.1 Enhanced Blinn-Phong Shading implementation

In the context of non-photorealistic rendering, it is common to make use of *simple shading models* such as Blinn-Phong shading model.
Geometry-based shading technique can alter surface shading to enhance surface fine-scale **geometric details** in a non-photorealistic manner. This process is performed by incorporating enhanced surface normal and surface curvature measure into Blinn-Phong.
As **intensity mapping** function [1] choosed $\delta_j = \rho_j$, where j $\in$ { a, d, s } iterates over the components of Blinn-Phong's shading models: ambient, diffuse and specular.
With this approach, using a single light, the lightning equation becomes:

$$L_r(x, \omega_o) = \sum_j \rho_j(\omega_o, l)\mathcal{G}_j(x, \omega_o, l)L_j(l) \tag{1}$$

In this equation, l is the direction of the light source at point x, L_j corresponds to the light intensity of each component, $\rho_a = 1$, $\rho_d(l) = (n' \cdot l)$, $\rho_s(x, l) = (n' \cdot r)^f$;
$n'(x)$ is the **enhanced surface normal**, r is the reflection direction, f is the shininess parameter and $\mathcal{G}_j(x, \omega_o, l)$ corresponds to the **Curvature-based Reflectance Scaling Function**.
In particular, $\mathcal{G}_j$ is computed as:

$$\mathcal{G}_j(\delta, P) = \frac{\delta}{e^P(1-\delta)+\delta} \quad \text{where} \quad P_{\lambda,\alpha}(k) = pow(\lambda|k|, \alpha) \tag{2}$$

In this equation, $P_{\lambda,\alpha}$ is the **curvature mapping** function; its magnitute and strength are controlled using two parameters (λ, α); δ corresponds to the previously declared **intesity mapping** function.
Code-wise, I started from Blinn-Phong implementation, already present in **lecture03a** repository, and modified it following **Equations 1** and **2** as we can see in **Code 6**, introducing all the previously listed functions.

```



// Curvature-Based Reflectance Scaling Function
float Lr(float curvature_value, float delta)
{
  // We apply the curvature mapping function that uses lambda and alpha parameters to
   apply non-linear mapping
  float P = pow (lambda * abs(curvature_value), alpha);
  // Uses as intensity mapping function the second parameter, delta
  // This function maps intensity mapping and curvature mapping functions in the
  reflectance radiance equation
  // This aims to correlate the reflected lightning intensity to surface curvature
  float G = delta / ( exp(P) * ( 1 - delta ) + delta );
  return G;
}
```

```

// a subroutine for the Enhanced Blinn-Phong model using Shape Depiction Enhancement based on local Geometry
subroutine(ill_model)
vec3 EnhancedBlinnPhong()
{
  // Computing the mask for Unsharp Masking
  vec3 mask = vNormal - vSMNormal;
  // calculating enhanced Normal using the Unsharp Masking technique. This is defined, in the reference paper, in equation 6 of chapter 4.2.2
  vec3 eNormal = vNormal + lambda * mask;
  // normalization of the per-fragment enhanced normal
  vec3 N_I = normalize(eNormal);
  // calculating curvature value using enhanced normal
  float curvature_value = curvature(N_I);
  // Implementing equation 12 of chapter 6.1 of the reference paper. I calculate the Curvature-Based Reflectance Scaling factor for each of the Blinn-Phong components. NOTE: Reference paper use the costant 1 as rho_a component for ambient
  float rhoA = 1;
  float G_a = Lr(curvature_value, rhoA);
  // ambient component can be calculated at this point
  vec3 color = Ka * G_a * ambientColor;
  // normalization of the per-fragment light incidence direction
  vec3 L = normalize(lightDir.xyz);
  // Lambert coefficient
  float rhoD = max(dot(L,N_I), 0.0);
  // if the lambert coefficient is positive, then I can calculate the specular component
  if(rhoD > 0.0)
  {
    // This is the Curvature-Based Reflectance Scaling factor for the diffuse component. NOTE: Reference paper use the lambertian coefficient as rho_d for diffuse
    float G_d = Lr(curvature_value, rhoD);
    // the view vector has been calculated in the vertex shader, already negated to have direction from the mesh to the camera
    vec3 V = normalize( vViewPosition );
    // in the Blinn-Phong model we do not use the reflection vector, but the half vector
    vec3 H = normalize(L + V);
    // we use H to calculate the specular component
    float specAngle = max(dot(H, N_I), 0.0);
    // shininess application to the specular component
    float rhoS = pow(specAngle, shininess);
    // This is the Curvature-Based Reflectance Scaling factor for the specular component. NOTE: Reference paper use the lambertian coefficient as rho_s for specular
    float G_s = Lr(curvature_value, rhoS);
    // We add diffusive and specular components to the final color using our Curvature-Based factors
    color += vec3( Kd * G_d * diffuseColor +
    Ks * G_s * specularColor);
  }
  return color;
}
```

Code 6: Enhanced Blinn-Phong subroutine and Curvature-Based Reflectance scaling function implemented in fragment shader

A Comparison between standard Blinn-Phong and Enhanced Blinn-Phong using Geometry-based shading is showed in **Figure 5** that use two subroutines created in the project.

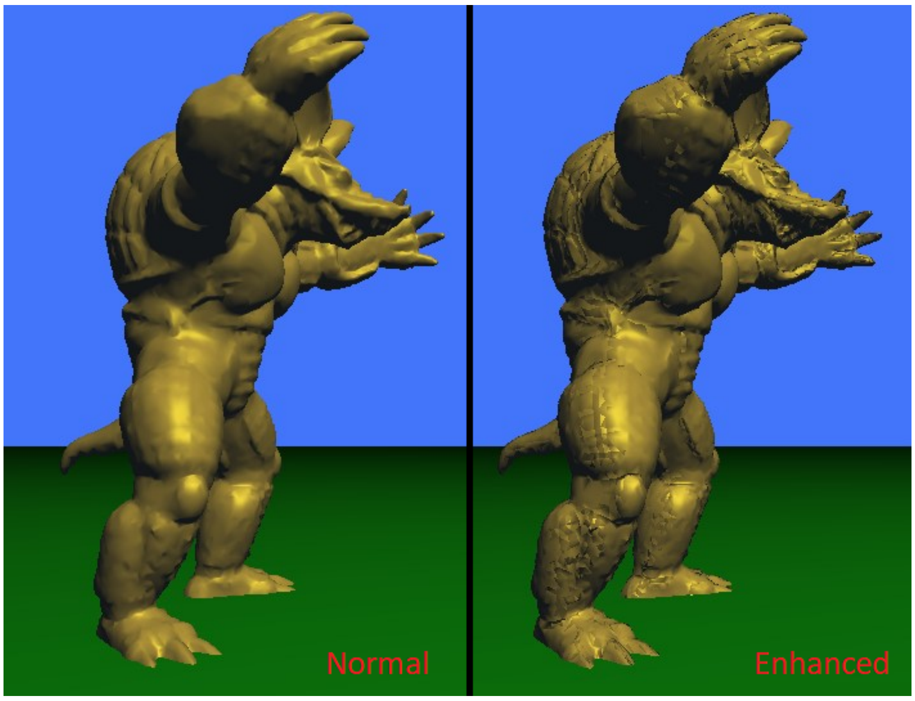


Figure 5: Comparison between standard Blinn-Phong and Enhanced Blinn-Phong using Geometry-based shading

3.3.2 Enhanced Cartoon/Cel Shading implementation

Cartoon shading consists of **quantizing** the amount of diffuse shading and **mapping** each discrete value to a different color. In order to apply Geometry-based shading technique, the choice of reflectance *mapping function* should be consistent with this finding. To this end, [1] used the following formula:

$$\delta_d = \frac{\lfloor (0.5 + (\mathcal{Q}_L \times pow(\rho_d, r))) \rfloor}{\mathcal{Q}_L} \tag{3}$$

Where δ_d corresponds to **reflected diffuse intensity** that we use in cartoon shading style, r is a *power level* and it is used to enhance the intensity of color, $\mathcal{Q}_L$ is the *quantization level* parameter; δ_a and δ_s are chosen in the same way as Blinn-Phong shading model.
Unfortunately, [1] doesn't describe how to **combine** the three components (ambient, diffuse and specular). For this reason, I choosed to introduce in the final equation two weights, for ambient and diffuse components, using the full δ_s component.
The **merging equivalence** used to sum the three components togheter is:

$$Intensity = weight_a \delta_a + weight_d \delta_d + \delta_s \tag{4}$$

Code-wise, I started from implementing standard Cartoon Shading, following the paper [2]. The implementation is shown in **Code 7**. Then, I modify it following **Equation 3** and **4** leading to the **Code 8** to obtain the final result.

```
// a subroutine for the Cartoon/Cel Shading model
subroutine(ill_model)
vec3 ToonShading(){
  // normalization of the per-fragment light incidence direction
  vec3 L = normalize(lightDir.xyz);
  // normalization of the per-fragment normal
  vec3 N = normalize(vNormal);
  // Intensity parameter used in standard toon/cel shading
  float intensity = dot(L,N);
  // Color choice based on intensity parameter
  if (intensity > 0.95)       return shinestColor;
  else if (intensity > 0.5)   return shinyColor;
  else if (intensity > 0.25)  return darkColor;
  else                        return gloomyColor;
}
```

Code 7: Standard Cartoon shading subroutine implemented in fragment shader

```
// a subroutine for the Enhanced Cartoon/Cel Shading model using Shape Depiction
  Enhancement based on local Geometry
subroutine(ill_model)
vec3 EnhancedToonShading(){
  // normalization of the per-fragment light incidence direction
  vec3 L = normalize(lightDir.xyz);
  // Computing the mask for Unsharp Masking
  vec3 mask = vNormal - vSMNormal;
  // calculating enhanced Normal using the Unsharp Masking technique. This is defined
  , in the reference paper, in equation 6 of chapter 4.2.2
  vec3 eNormal = vNormal + lambda * mask;
  // normalization of the per-fragment enhanced normal
  vec3 N_I = normalize(eNormal);
  // NOTE: Reference paper use the costant 1 as rho_a component for ambient
  float rhoA = 1;
  // Intensity parameter used in standard toon/cel shading, but using our enhanced
  normal, for the ambient compinent
  // Equation 13 of the Chapter 6.2 of the reference paper applied only to diffuse
  and ambient components
  float deltaA = floor(0.5 + (Ql * pow(rhoA, r))) / Ql;
  // NOTE: Reference paper use the lambertian coefficient as rho_d for diffuse
  float rhoD = max(dot(L,N_I), 0.0);
  // Intensity parameter used in standard toon/cel shading, but using our enhanced
  normal, for the diffuse component
  // Equation 13 of the Chapter 6.2 of the reference paper applied only to diffuse
  and ambient components
  float deltaD = floor(0.5 + (Ql * pow(rhoD, r))) / Ql;
  // Calculation of specular component, specified as in the reference paper, using
  the same as Phong model
  vec3 R = normalize(reflect(-L, N_I));
  vec3 V = normalize( vViewPosition );
  float specAngle = max(dot(R, V), 0.0);
  // shininess application to the specular component. NOTE: Reference paper use the
  lambertian coefficient as rho_s for specular
  float deltaS = pow(specAngle, shininess);
  // Composition of the final intensity to apply, then, the color choice. NOTE: In
  the paper is not specified how the three components are composed. This is my
  solution that considers only Ambient and Diffuse components, while maintaning full
  specular component
  float intensity = myWeightA * deltaA + myWeightD * deltaD + deltaS;
  // Color choice based on intensity parameter
  if (intensity > 0.95)       return shinestColor;
  else if (intensity > 0.5)   return shinyColor;
  else if (intensity > 0.25)  return darkColor;
  else                        return gloomyColor;
}
```

Code 8: Enhanced Cartoon shading subroutine implemented in fragment shader

A Comparison between standard Cartoon shading and Enhanced Cartoon using Geometry-based shading is showed in **Figure 6** that use two subroutines created in the project.

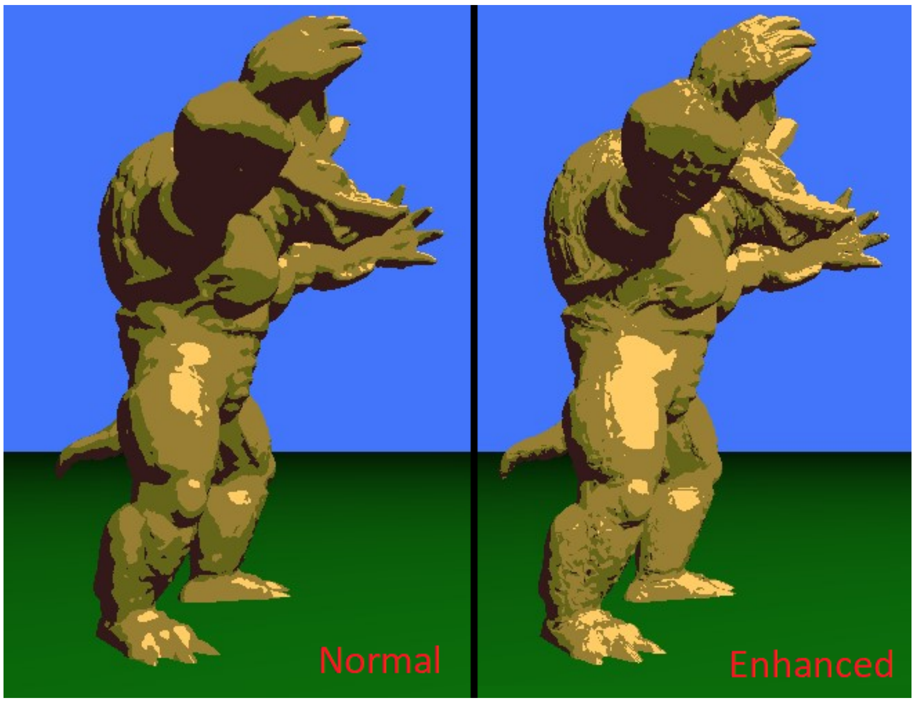


Figure 6: Comparison between standard Cartoon shading and Enhanced Cartoon using Geometry-based shading

3.3.3 Enhanced Gooch Shading implementation

Another place where a Geometry-based Shading technique is needed is when using **blending** between colors to convey the shape. The main idea in Gooch Shading is to create **cool-to-warm transition** for technical illustration.
To properly scale the *cool-to-warm shading*, [1] use Geometry-based shading technique with Gooch shading style. The choice of the intensity mapping function is inspired by Gooch approach as following:

$$\delta_j = \left(\frac{1+\rho_j}{2}\right) k_{cool} + \left(1 - \frac{1+\rho_j}{2}\right) k_{warm} \tag{5}$$

Where j ∈ {a, d, s} iterates over the ambient, diffuse, and specular component of Gooch shading model; ρ_j is computed based on Blinn-Phong shading model; kcool and kwarm correspond to **cold** and **warm colors** of Gooch shading model.
Unfortunately, [1] doesn't describe how to **combine** the three components (ambient, diffuse and specular). For this reason, I choosed to introduce in the final equation two weights, for ambient and diffuse components, using the full ρ_s component.
The **merging equivalence** used to sum the three components togheter is:

$$Intensity = weight_a \rho_a + weight_d \rho_d + \rho_s \tag{6}$$

Code-wise, I started from implementing standard Gooch Shading, following the paper [3]. The implementation is shown in **Code 9**. Then, I modify it following **Equation 5** and **6** leading to the **Code 10** to obtain the final result.

```
// a subroutine for the Gooch Shading model
subroutine(ill_model)
vec3 GoochShading(){
  // normalization of the per-fragment light incidence direction
  vec3 L = normalize(lightDir.xyz);
  // normalization of the per-fragment normal
  vec3 N = normalize(vNormal);
  // Lambert coefficient
  float lambertian = dot(L, N);
  // weight used in standard gooch shading
  float weight = ( lambertian + 1.0 ) * 0.5;
  // Diffuse component of standard gooch shading
  vec3 kCool = min(CoolColor + DiffuseCool * SurfaceColor, 1.0);
  vec3 kWarm = min(WarmColor + DiffuseWarm * SurfaceColor, 1.0);
  vec3 kFinal = mix(kCool, kWarm, weight);
  // Calculation of specular component
  vec3 R = normalize(reflect(-L, N));
  vec3 V = normalize( vViewPosition );
  float specAngle = max(dot(R, V), 0.0);
  // shininess application to the specular component
  float specular = pow(specAngle, shininess);
  // Final color composition of the standard gooch shading
  return vec3(kFinal + vec3(1) * specular);
}
```

Code 9: Standard Gooch shading subroutine implemented in fragment shader

```
// a subroutine for the Gooch Shading model using Shape Depiction Enhancement based
  on local Geometry
subroutine(ill_model)
vec3 EnhancedGoochShading(){
  // normalization of the per-fragment light incidence direction
  vec3 L = normalize(lightDir.xyz);
  // Computing the mask for Unsharp Masking
  vec3 mask = vNormal - vSMNormal;
  // calculating enhanced Normal using the Unsharp Masking technique. This is defined
  , in the reference paper, in equation 6 of chapter 4.2.2
  vec3 eNormal = vNormal + lambda * mask;
  // normalization of the per-fragment enhanced normal
  vec3 N_I = normalize(eNormal);
  // NOTE: Reference paper use the costant 1 as rho_a component for ambient
  float rhoA = 1;
  // Equation 14 of the Chapter 6.3 of the reference paper applied only to diffuse
  and ambient components
  float deltaA = (1 + rhoA) * 0.5;
  // NOTE: Reference paper use the lambertian coefficient as rho_d for diffuse
  float rhoD = max(dot(L,N_I), 0.0);
  // Equation 14 of the Chapter 6.3 of the reference paper applied only to diffuse
  and ambient components
  float deltaD = (1 + rhoD) * 0.5;
  // Calculation of specular component, specified as in the reference paper, using
  the same as Phong model
  vec3 R = normalize(reflect(-L, N_I));
  vec3 V = normalize( vViewPosition );
  float specAngle = max(dot(R, V), 0.0);
  // shininess application to the specular component
  float rhoS = pow(specAngle, shininess);
  // Diffuse component of standard gooch shading but using our previously calculated
  delta as weight, for diffuse and ambient component
  vec3 kCool = min(CoolColor + DiffuseCool * SurfaceColor, 1.0);
  vec3 kWarm = min(WarmColor + DiffuseWarm * SurfaceColor, 1.0);
  vec3 kFinalA = mix(kCool,kWarm, deltaA);
  vec3 kFinalD = mix(kCool,kWarm, deltaD);
  // Composition of the final color. NOTE: In the paper is not specified how the
  three components are composed. This is my solution that considers only Ambient and
  Diffuse components, while maintaning full specular component
  return  myWeightA * kFinalA + myWeightD * kFinalD + vec3(1) * rhoS;
}
```

Code 10: Enhanced Gooch shading subroutine implemented in fragment shader

A Comparison between standard Gooch shading and Enhanced Gooch using Geometry-based shading is showed in **Figure 7** that use two subroutines created in the project.

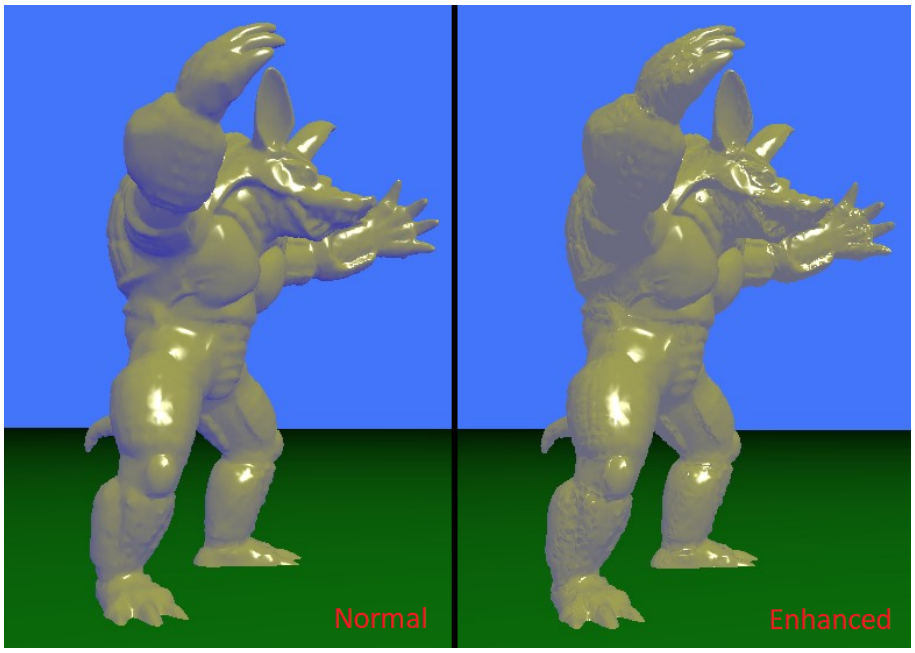


Figure 7: Comparison between standard Gooch shading and Enhanced Gooch using Geometry-based shading

4 Conclusions

This project helped me to understand in *deep* some of the simpler and most famouse **Non-Photorealistic Rendering (NPR)** shading techniques as well as implementing some new strategy to improve them.
It also helped me to understand better the overall **rendering pipeline** and *OpenGL* language. Even if I didn't implement everything introduced in [1], I tried to create a technique **closer to the original**, maintaining the same *intent* of the authors, using local geometry to enhance shape depition of the testing 3D models.
My implementation provides also a fairly high and stable frame-per-second (fps), making it possible to use the shader in *interactive application* without any constraints on the choice of material or illumination model.

References

[1] Riyad Al-Rousan, Mohd Shahrizal Sunar, Hoshang Kolivand (2017) *Geometry-based shading for shape depiction enhancement*

[2] Decaudin Philippe (1996) *Cartoon-looking rendering of 3d-scenes*

[3] Gooch Amy, Gooch Bruce, Shirley Peter, Cohen Elaine (1998) *A non-photorealistic lighting model for automatic technical illustration.*