Modelling and rendering
dynamic wrinkles in a virtual face

av
Mikael Nordenberg

Handledare: Jonas Beskow

Examensarbete i talkommunikation

Stockholm
april 2003

Examinator: ...........................................
(Godkänt den ......................... (signature)

Institutionen för tal, musik och hörsel
Kungliga Tekniska Högskolan
100 44 Stockholm
Sammanfattning

Den här rapporten beskriver ett sätt att skapa dynamiska rynkor i realtid i ett datoranimerat ansikte. Syftet är att utöka funktionaliteten i ett existerande datoransikte för att öka realismen och göra det möjligt att skapa nya ansiktsuttryck.


Rynk-intensiteten används sedan för att grafiskt rita rynkor i ansiktet. Visualiserings-algoritmen använder hårdvaruacceleration genom en teknik som heter ”bump mapping” för att klara realtidskravet.

Den implementerade prototypen visar att det är fullt möjligt att visualisera rynkor i realtid med dagens grafikkort. ”bump mapping”-metoden har en del begränsningar, speciellt med avseende på ljussättningen, men i relativt enkel miljö, som den här applikationen, är de fullt acceptabla.

Rapporten avslutas med en enkel utvärdering av systemet.
Master Degree Project in Speech Communication

Modelling and rendering dynamic wrinkles in a virtual face

Mikael Nordenberg

Approved
Year, month, day

Examiner
Björn Granström

Supervisor
Jonas Beskow

Abstract
This thesis suggests a way to add real time dynamic wrinkles to a computer-generated face. The purpose is to extend an existing virtual face to make it more realistic and expressive.

The work consists mainly of two parts. The first is to calculate the compression of the skin when the face is deformed. This is achieved through geometric calculations on the triangles the face is composed of. The compression is then converted to wrinkle intensity using a translation function. The wrinkle intensity is then used to graphically render wrinkles on the face. The rendering algorithm uses hardware-accelerated graphics to meet the demands of real time animation. The algorithm uses a technique called bump mapping to visualize the wrinkles.

The implemented prototype shows that real time dynamic wrinkles can be implemented using existing hardware. The bump mapping algorithm has a few limitations, especially regarding the lighting model, but these limitations are hardly noticed in this fairly simple application.

The thesis ends with a brief evaluation of the application.
# Contents

Contents ........................................................................................................................................................................... 1  
1. Introduction ........................................................................................................................................................................ 2  
   1.1. Task description ......................................................................................................................................................... 2  
   1.2. Previous work ............................................................................................................................................................... 2  
   1.3. Background ................................................................................................................................................................. 2  
   1.4. Construction of the face ............................................................................................................................................... 3  
   1.5. Extending functionality .................................................................................................................................................... 3  
   1.6. Introduction to 3D-graphics ....................................................................................................................................... 3  
2. Wrinkle model ................................................................................................................................................................. 5  
3. Artificial skin compression algorithm ............................................................................................................................... 6  
   3.1. Triangle compression ...................................................................................................................................................... 6  
   3.2. Vertex compression ....................................................................................................................................................... 8  
   3.3. Square length motivation .............................................................................................................................................. 11  
4. Overview of wrinkle rendering .......................................................................................................................................... 12  
   4.1. Method 1 - Alpha shading ........................................................................................................................................... 12  
   4.2. Method 2 - Bump map blending .............................................................................................................................. 14  
   4.3. Method 3 - Interpolated bump mapping .................................................................................................................... 15  
5. Graphics tools ................................................................................................................................................................. 18  
   5.1. OpenGL ...................................................................................................................................................................... 18  
   5.2. GeForce3 ................................................................................................................................................................. 18  
6. Implementation of interpolated bump mapping .................................................................................................................. 21  
   6.1. Reference spaces ......................................................................................................................................................... 21  
   6.2. Lighting model ......................................................................................................................................................... 21  
   6.3. Bump mapping on the GeForce3 ............................................................................................................................ 24  
7. Perceptual evaluation ......................................................................................................................................................... 29  
8. Results and discussion ...................................................................................................................................................... 30  
   8.1. Problems ................................................................................................................................................................. 30  
   8.2. Future work .............................................................................................................................................................. 30  
9. References ................................................................................................................................................................. 32  
10. Appendix ..................................................................................................................................................................... 33  
   10.1. Vertex program ...................................................................................................................................................... 33  
   10.2. Texture shaders configuration ............................................................................................................................. 34  
   10.3. Register combiner configuration ........................................................................................................................ 35
1. Introduction

1.1. Task description
The assignment was to create wrinkles in a computer rendered virtual face. The application rendering a virtual face without wrinkles already existed. The main requirement was that the virtual face should be animated in real time. Bump mapping was suggested as a rendering aid to help achieve that requirement.

1.2. Previous work
Some efforts have been done in the field of real time dynamic wrinkles. Two of them, which have been studied more in detail, are described here:

Pascal Volino and Nadia Magnenat-Thalmann have presented an algorithm [1] using triangle deformation to calculate wrinkles. They measure the ratio between deformed and undeformed triangles edges. This ratio is then used to calculate wrinkle amplitude using a translation function. The algorithm presented in this thesis eliminates some drawbacks that were found in their algorithm, especially the triangle compression calculations. Their wrinkle rendering is built using mesh subdivision, which should be slower than bump mapping, although they do mention bump mapping as an alternative rendering technique.

C. Pelachaud et al. has presented a talking head capable of showing wrinkles [2]. Their approach is based on feature points affecting a radial area of skin. They focus more on single large wrinkles, like the ones occurring when smiling (nasolabial furrow). Compared to the approach presented in this paper, theirs is very limited regarding wrinkle shape, because of the radial pattern, and is therefore not suitable when modelling arbitrary wrinkles.

1.3. Background
Computer graphics hardware is developing at a fast rate. The performance is doubled in less than 18 months. Besides performance improvements, new functionality is constantly added, to make 3D-graphics look more realistic. Some of these newly introduced functions are necessary to create wrinkles using bump mapping.

Creating a realistic real time animated face does not only require the latest technology, but also a lot of fine tuning of parameters, textures etc, which makes it a complex task. Also, in many applications where virtual faces occur, typically in games, speed is usually the most important factor, and time consuming calculations required for advanced 3D-graphics is not highly prioritised.

If virtual faces become more realistic they may be usable in more areas of applications, and people might feel more comfortable interacting with something more resembling a human.

This thesis presents three ways of creating dynamic wrinkles to a real time animated virtual face. The most advanced method is described in more detail, and can be considered the main approach.
The face is an already existing 3D-model [3], developed at CTT, the Centre for Speech Technology, at KTH.

1.4. Construction of the face

To understand how the wrinkles in this application are computed, a basic understanding of the construction of the face is needed.

The virtual face has a number of points called vertices describing the face geometry. Each vertex has a position in 3D-space. The vertices are connected to each other by triangles. The triangles represent the surface of the face and may have attributes such as colour, shininess etc. When the face is animated, vertices are moved, and the triangles connected to those vertices are deformed.

A number of parameters control the position of the vertices. The parameters apply a geometric transform (rotation, translation or scale) to a varying degree to each of the vertices, based on vertex weights. One parameter can for example be jaw opening. If the jaw should open, all vertices affected by the jaw movements are moved, since they are the only ones with weights other than zero. The rules for these transformations are defined, along with vertices information, in a file describing the virtual face.

1.5. Extending functionality

Since this application is based on an already existing virtual face, the intention is to introduce minimal extra data to accomplish the wrinkles. The face already is animation programmed, and it would be preferred not to interfere with those parts. This requires that the wrinkle algorithm use the face’s building blocks to analyse its current state and mathematically compute if and how much it should be wrinkled. It is also preferred that the sections of the face unaffected by wrinkles stay intact. This means extending the application to handle the new wrinkled surfaces only when the model specifies so and the hardware platform permits it.

1.6. Introduction to 3D-graphics

This section will briefly explain a few common terms and techniques used in 3D-graphics.

Polygons

3D-objects (models) are built of polygons. The objects surface is composed of polygons. The face used in this work is built exclusively of triangles. 3D-graphics often use polygons and not just triangles, but triangles have the big advantage of always being flat in the 3D-space. A quadrangle can easily be made to form a curved surface, and may therefore be impossible to render correctly. Any polygon can be converted into one or more triangles, so this is not a limiting factor.

Vertices

Vertices describe the position of polygon corners. When the position of the vertex changes, the triangle changes accordingly. Many triangles can share the same vertex to eliminate redundancy.
Normals
Each vertex may also be associated to a normal direction. The normal is used when calculating how light should be reflected off the surface.

Textures
Texturing produces more realistic looking surfaces. Texturing allows a surface to have an image painted onto it. This way the colour detail of a surface increases without increasing the number of polygons. The image painted to the surface is called a texture.

Bump mapping
One problem with 3D-graphics is that an extremely high polygon count is required to create realistic 3D-objects. In real life not many surfaces are completely flat. Texturing works in some cases, but when the lighting changes in a scene with too few polygons one often notice the flattened look. Bump mapping [4] is a way to visually simulate small displacements on a surface that is almost flat.

The idea is to apply a texture map, containing normals instead of colours. These normals are used when calculating the light reflected off the surface. This requires more calculations than ordinary rendering, which makes bump mapping a computationally intensive operation.

Reference spaces
Vectors and vertices define directions or points in 3D-space using a reference coordinate system, or reference space. Three directions forming an orthogonal base and the origin position define this reference space. Every vector is defined using a reference coordinate system, and is said to be defined in the associated reference space. Sometimes a vector needs to be transformed between reference spaces to be comparable to other vectors.

Do not confuse reference space with the mathematical term vector space, which means the complete space reachable by using any linear combination of the base vectors.

The use of different reference spaces is practical, since 3D-objects should stay independent of each other. If two objects were to be rendered side by side, a rotation to one of them should not affect the other.

Animation programming
When programming animated 3D-graphics one need to change the state of the scene between every frame. Some things must be recalculated to reflect the change in time since last frame. This consumes computing time from the processor and need to be minimized. I will describe some calculations as “per frame calculations”. These are necessary to perform each frame update in the animation. Other calculations are described as “pre-computation calculations”, and are only necessary once, when the application loads the model from a file.
2. Wrinkle model

One can think of the skin as a flexible and deformable material, but not compressible. This means that the skin can be stretched, but not compressed, without affecting the surface shape, as shown in Figure 1. Compressing the skin more than it can absorb without deformation of the surface produces wrinkles.

![Figure 1. Simplified skin model.](image)
3. Artificial skin compression algorithm

The first task in my approach is to calculate skin compression. The compression calculations algorithm had to meet some requirements for the application to work. They are:

- Use the already existing face models to calculate the skin compressions.
  This requirement implies keeping required modification to the existing face models to a minimum. The existing models have a lot of work put into them already, and it would be a daunting task to redo them.

- Take maximum advantage of the data in the existing model to achieve highest possible resolution.
  This requirement is important, because it is preferred to get a smooth look of the wrinkles, which requires dense compression data.

- Has to be fast enough for real-time animation.
  To get a smooth animation, the algorithm calculating skin compression should not load the CPU too much. The rendering of 3D-graphics alone is quite CPU-intensive, which leaves even less for other calculations. Pre-computation of some data lightens the CPU load during run-time.

3.1. Triangle compression

To calculate compression one need to look at area elements. The smallest area elements in the computer model of the face are triangles. This means that if the highest possible resolution shall be achieved, the compression algorithm should base its calculations on triangles.

The algorithms can be divided into pre-computation calculations and per frame calculations.

Pre-computation calculations

When a face model is loaded, the algorithm makes some initial calculations to improve the speed of calculations later done per frame.

The model stores information about the direction of the wrinkles at each vertex. The wrinkles can only assume one direction at each vertex. Wrinkle direction is defined as the direction perpendicular to the extension of the wrinkles, see Figure 2.

![Wrinkle direction](image)

*Figure 2. Wrinkle direction definition.*
Wrinkle direction is important, because compression in the direction perpendicular to the wrinkle direction should not make wrinkles visible, but compression in the direction of the wrinkle direction should.

First of all a wrinkle direction for the triangle is calculated using a mean of the tree wrinkle directions at the vertices associated to the triangle. This is the wrinkle direction referenced hereafter. Each triangle size is measured by projecting a line in the wrinkle direction into the triangle. The line intersects one of the vertices to assume the maximum length and thereby maximum resolution as shown in Figure 3.

\[ \text{Figure 3. Size calculation of a triangle in the wrinkle direction.} \]

The square length $L^2$ is calculated and stored for later usage. Also the intersection points $P_1$ and $P_2$ are stored. $P_1$ is always one of the vertices defined in the model, in Figure 3 this is $V_1$. $P_2$ on the other hand is, in the general case, a new point on the line intersecting the other two vertices in the triangle. The position of $P_2$ is calculated as a linear combination of these, in Figure 3 as $P_2 = aV_2 + (1-a)V_3$. Only the factor $a$ needs to be stored to recalculate $P_2$.

The measured size, $L^2$, is considered the “relaxed size” of the triangle. $L^2$ is later used as reference value to determine if, and how much the triangle has been compressed. The square distance is used because a square root operation is computationally costly. Using the square length affects the result very little, as explained later.

**Per frame calculations**

With each frame redraw, each triangle needs to be examined to determine how much it is compressed in the wrinkle direction. This is done by recalculate $P_1$ and $P_2$ using the transformed vertices in the model and the factor $a$ in the linear combination. Figure 4 shows an example of a deformed triangle.
The square distance $l^2$ between the two points is calculated. The resulting length is then divided by the pre-calculated relaxed square length $L^2$. This gives the triangle compression factor $\alpha$ as:

$$\alpha = \frac{l^2}{L^2}$$

*Equation 1. Calculating compression factor*

The compression factor $\alpha$ is measured for each triangle in the surface.

### 3.2. Vertex compression

To give the face a smooth appearance, compression needs to be continuous across the surface. We need a way to interpolate the compression across the triangles constructing the surface. One way is to translate the compression factor from the triangles to the vertices. This makes sure that the compression factor is continuous on the corners of the triangles. The graphics hardware will automatically make it continuous on the edges and inside the triangles, as explained in the implementation chapter.

All triangles surrounding a vertex contribute to the vertex compression. Consider, for example, the case represented by Figure 5 (a).

One simple way would be to calculate a mean of the triangles compression factor, i.e.:

$$\alpha_v = \frac{\alpha_1 + \alpha_2 + \alpha_3 + \alpha_4}{4}$$

*Equation 2. A simple way of calculating vertex compression factor*

But this does not give satisfactory results, because the shape of the triangles should affect how much they contribute. A more detailed tessellation$^1$ of the surface, as shown in

---

$^1$ Division of a surface.
Figure 5 (b), should not affect the vertex compression but using a simple mean clearly would.

\[
\begin{align*}
\alpha & = \frac{1}{\sum w_i} \sum w_i \alpha_i \\
\text{and} \\
w_{tot} & = \sum w_i 
\end{align*}
\]

*Equation 3. Calculating vertex compression factor using a weighted mean*

where \( N \) is the total number of triangles surrounding the vertex and \( w_i \) is the weight factor multiplied to each triangle’s compression factor \( \alpha_i \). The weight should make the vertex compression factor independent of surface tessellation. Two methods were considered:

- \( w_i = \) Triangle area
- \( w_i = \) Triangle angle towards vertex

Both methods were tested, and it is difficult to directly see which one is the best. Intuitively the angle variant seems more correct, and that is why it was chosen for this application.

To make generation of the wrinkles more configurable, wrinkle intensity was introduced. Wrinkle intensity is a floating-point number describing the amplitude of the wrinkles, and is calculated as shown in Equation 4.

This function is a modified version of the amplitude function described in [1]. The modification makes the function more configurable.
\[ I_v = \frac{2}{1+(\lambda \alpha_v)} - 1 \]

*Equation 4. Wrinkle intensity*

Where:

<table>
<thead>
<tr>
<th>$I_v$</th>
<th>wrinkle intensity for vertex</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\alpha_v$</td>
<td>compression factor for vertex</td>
</tr>
<tr>
<td>$\lambda$</td>
<td>offset factor (described below)</td>
</tr>
<tr>
<td>$\beta$</td>
<td>appearance factor (described below)</td>
</tr>
</tbody>
</table>

The surface constants $\lambda$ and $\beta$ are specified as surface parameters in the model, and can be used to model different types of skin.

If $\lambda < 1$ the surface is wrinkled, even when it is relaxed. If $\lambda$ assumes 1, the surface will wrinkle at slightest compression. Setting $\lambda > 1$ requires some compression before wrinkles become visible. Figure 6 (a) shows how three different $\lambda$ affect the translation function.

$\beta$ represent how fast wrinkles will be visible when the surface is compressed. Figure 6 (b) shows the effect of three different $\beta$. A higher value of $\beta$ gives a steeper derivate at compression factor = 1, making wrinkles appear easier.

The two parameters are experimentally fine tuned until a satisfactory result is achieved.

The wrinkle intensity is a property of the vertices in the model just as position, colour or texture coordinates. Some of these properties are recalculated during animation (wrinkle intensity, position), and some are static (texture coordinate etc.).

*Figure 6. How compression factor translates to wrinkle intensity for different $\lambda$ and $\beta$.*
3.3. Square length motivation

When calculating the size of triangles, the distance between two points is calculated. The square distance is an intermediate result of this, and if the square root operation could be eliminated, the performance would benefit.

If a model uses fairly small triangles and transformation of the vertices is not too extreme, the compression factor will be similar on all triangles surrounding a vertex. If the compression factors are similar, the use of square length in the compression factor almost gives a true square vertex compression factor, i.e.:

\[
\alpha_v = \frac{1}{w_{\text{ref}}} \sum_{i=1}^{N} w_i a_i = \left( \frac{1}{w_{\text{ref}}} \sum_{i=1}^{N} w_i \sqrt{a_i} \right)^2
\]

**Equation 5. Square length approximation**

When the wrinkle intensity function is applied this fact can be utilized. A square root operation can be done “for free” by adjusting \( \beta \) and \( \lambda \) properly. This is the reason for not performing the square root operation on the triangle compression factor.
4. Overview of wrinkle rendering

There are a number of ways that wrinkles can be rendered to a surface. One of the requirements in this thesis was that the rendering should be fast enough to allow real-time animation of the face; therefore this study focused on methods that, at least partly, could be hardware accelerated.

Three approaches were evaluated and are presented below with an overview of their main characteristics. All are using predefined wrinkle maps. The alternative would have been to calculate where wrinkles should appear, but the wrinkle map keeps complexity down and at the same time makes the wrinkles appearance, amplitude and position easily configurable. Wrinkle maps are also “compatible” with graphics hardware, since they can be stored in texture memory.

The wrinkle map

A wrinkle map is a matrix describing where the wrinkles shall appear. The wrinkle map can for example be a rectangular matrix of surface normal directions. The wrinkle map shall preferably have high enough resolution not to be a major contributor to unwanted visual artefacts. If the wrinkle map has too low resolution too many pixels on the screen will share the same wrinkle data, resulting in a blocky appearance. The resolution of the wrinkle map is limited by the memory capacity used for textures by the graphics hardware.

Each vertex in the model has a predefined texture coordinate describing what position the vertex has in the wrinkle map. This way the wrinkle map is correctly aligned to the model, placing the wrinkles in the right position.

Wrinkle interpolation

To make the skin look smooth, the wrinkles need to be interpolated across the surface. This turned out to be a big problem in most rendering approaches because the graphics hardware is not well suited to draw wrinkles.

The surface is rendered as triangles, and the wrinkle intensity is interpolated across each triangle. The interpolated wrinkle intensity is hereafter called \( I \), assuming a value in the range \([0,1]\), in all calculations.

4.1. Method 1 - Alpha shading

Alpha shading is the simplest, and least realistic technique described here, but also has the widest support among graphics hardware. Only a few relatively simple hardware operations need to be supported.

The wrinkle map in this case is a rectangular matrix consisting of a scalar value per matrix element, describing the maximum amount of shadow the area gets when wrinkled.
The rendering occurs in two passes. The first pass renders the face without wrinkles. This results in lit, smooth skin well integrated with the surrounding surfaces. The second pass use the wrinkle map to darken each pixel according to wrinkle intensity using the following formula:

\[ C_{\text{out}} = C_{\text{in}} \cdot (1 - I \cdot W) \]

*Equation 6. Alpha blending*

<table>
<thead>
<tr>
<th>$C_{\text{out}}$</th>
<th>resulting pixel colour</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C_{\text{in}}$</td>
<td>pixel colour after first rendering pass</td>
</tr>
<tr>
<td>$I$</td>
<td>wrinkle intensity at this pixel</td>
</tr>
<tr>
<td>$W$</td>
<td>shadowing value at this pixel retrieved from the wrinkle map</td>
</tr>
</tbody>
</table>

The biggest drawback of this method is that it just darkens the area between wrinkles. No light calculations are made. The method is not as bad as it might seem because, in reality, this is often the way wrinkles look on skin; small darkened stripes.

Since it does not take into consideration the position of the lights and eye, directing a light-source at the wrinkled area would leave the wrinkles unaffected.

The wrinkle map can be arranged to simulate light from a given direction by shadowing pixels in a special way, as shown in Figure 8.
When wrinkled, the surface will have a majority of the darkening on the lower side of each wrinkle, creating the effect of light coming from above.

The limited realism and wide hardware support for this method of rendering wrinkles makes it ideal as a secondary, fallback method, if a more advanced and better looking method fails. See Figure 12 for an example.

4.2. Method 2 – Bump map blending

Bump map blending is the second best method presented here. It combines a bump mapped surface with a flat surface.

The wrinkle map describes a normal per element just like standard bump mapping. The idea is to render both surfaces using the current lighting, and then combine the two using colour blending. Equation 7 shows the formula is used to calculate the output pixel colour.

\[
C_{\text{out}} = I \cdot C_{\text{bump}} + (1 - I) \cdot C_{\text{flat}}
\]

Equation 7. Bump map blending

| \( C_{\text{out}} \) | resulting pixel colour |
| \( I \) | wrinkle intensity at this pixel |
| \( C_{\text{bump}} \) | colour from bump mapped surface pixel |
| \( C_{\text{flat}} \) | colour from the normal, flat surface |
Figure 9 shows how blending could affect a surface.

This is an experimental method, not derived from any physical calculations of surface wrinkling. It is, however, correct in the two extremes, no wrinkle intensity at all and maximum wrinkle intensity, resulting in a flat surface or a bump mapped one. The interpolation between these two cases is of course a very rough approximation.

The advantage of this method is that it is fairly simple and that the wrinkles are affected by the current lighting and eye position.

The problem is that the wrinkles amplitude does not change correctly according to the wrinkle intensity value. The wrinkles just fade in and out. See Figure 12 for an example of a rendered face using this method.

### 4.3. Method 3 - Interpolated bump mapping

Interpolated bump mapping is the most sophisticated of the three methods. This method is the main approach of this thesis.

Interpolated bump mapping actually adjusts the height of the wrinkles according to the wrinkle intensity, before rendering them, as demonstrated in Figure 10.
Studying non-uniform scaling [5] indicated that the normal vectors should be affected by the inverse scaling matrix that are applied to position vectors. This means that if we have a wrinkle intensity of 0.5 and want to half the wrinkle height, the normals x and y components shall be halved (see Figure 11).

Figure 10. Adjusted wrinkles.

Figure 11. Normal adjustment due to wrinkle intensity.
The strategy is to let the wrinkle intensity adjust the normal and thereby adjust the wrinkle height. If the wrinkle intensity=1, the normal will be left unadjusted at maximum wrinkle height, but if the wrinkle intensity is less than one, the normal will be adjusted towards the z-axis. The wrinkles are then rendered using bump mapping.

The steps involved in realizing this method are described in “Implementation of interpolated bump mapping”. See Figure 12 for an example of a face rendered using this method.

*Figure 12. Different rendering techniques, from left to right: Alpha shading, Bump map blending, Interpolated bump mapping.*
5. Graphics tools

OpenGL 1.3 (Open Graphics Library) was used to program the visual effects. The existing face is rendered using OpenGL, so there was no other alternative. OpenGL, as yet, does not support the functions needed to program interpolated bump mapping, but there are additional extensions available that do. The problem with extensions is that they are not part of the standard, and therefore often applies to one manufacturer’s graphics cards only. After examining the market of 3D-acceleration, the choice fell on the GeForce3 (GF3) graphics card manufactured by NVIDIA because of programming flexibility and low cost.

5.1. OpenGL

OpenGL is an open standard developed by SGI. The purpose was to create a uniform interface for programming graphics that could be hardware accelerated. It is the dominating API (Application Programming Interface) for 3D-graphics in multi platform applications.

OpenGL works as an intermediate layer between the application and the graphics hardware. The idea is to make the graphics functions independent of the hardware used. One of the original ideas was that the programmer should not even know if a hardware accelerated graphics card was installed, so all basic functionality could be software emulated. This idea is becoming more and more obsolete, since many advanced functions can only be done on dedicated hardware to run at an acceptable speed. In addition to that, many new functions, published by the extension mechanism, are supported only by one vendor’s graphics cards. To keep the standard usable, the OpenGL architecture review board (ARB) is repeatedly extending the standard to include more of the widely accepted new functionality.

OpenGL works with vertices describing corners of polygons. If a triangle is to be drawn, three vertices are input to the API and a triangle is drawn. OpenGL works as a state machine, meaning it stores the current state and applies this state to all vertices input.

All vertices are affected by a number of operations, or stages, before the pixels are coloured on the screen. These stages constitute the rendering pipeline. The rendering pipeline takes, for example, care of transforming vertices so that they get the correct position on the screen.

OpenGL stores transformation matrices, colours, normal etc in its state. For an in depth view of OpenGL, please study the OpenGL Programming guide [6].

5.2. GeForce3

GeForce3 is a fairly advanced graphics card. It is reasonable to assume that all desktop computers in a near future will have an equally powerful graphics card. GeForce3 is the first graphics card that enables programming of the graphics card’s GPU (Graphics Processing Unit). This is a major leap forward for graphics programming.

Instead of using the regular OpenGL functionality, three stages in the rendering pipeline can be reprogrammed by using extensions on the GeForce3, increasing the flexibility of the hardware.
Figure 13 shows the three programmable stages in the GeForce3. The vertex program stage is only applicable to vertices. This means that it is called once per vertex, which is far less than the other two stages, and it can therefore handle much heavier calculations without noticeable reduction of the performance. The two per-pixel stages on the other hand are extremely time critical.

**Vertex programs**

The first, and only stage associated with vertices, is the transform and lighting stage, or T&L. How T&L shall work are clearly defined by OpenGL, but NVIDIA lets the programmer override the standard implementation to implement a tailor-made T&L. These programs are called vertex programs [7] and are applied to each vertex input to the OpenGL rendering pipeline.

Typically the vertex program transforms the position of the vertex from a 3D-objects local reference space to a global reference space called clip-space. After the transform the vertex is lit using a lighting model. A number of special effects like skinning\(^1\) and fog\(^2\) can be applied at this stage.

Vertex programs pass information to per-pixel operations using special registers. These registers can for example be primary colour or texture coordinate, but can be used for any kind of information needed in the per-pixel operations. The registers contain four values; typically colour information (RGBA), texture coordinate (STRQ) or a vector (XYZW).

**Texture shaders**

The second programmable stage is the texture shaders [8]. The texture shaders perform per-pixel operations, meaning that they are called once for every pixel output by the

---

\(^1\) Transformation of vertices often used to create animation.

\(^2\) Blending a colour with the vertices usually depending on depth, creating a fog-like effect.
hardware. Texture shaders extend the functionality of the texture units. GF3 has four texture units. Normally each texture unit is programmed to fetch a colour from a texture and deliver it to the pixel blending stage (register combiners), as shown in Figure 14.

![Figure 14. Standard texture unit.](image)

But GF3 introduces a way of making the texture units dependant of each other. Each texture unit can be programmed to one of a number of different predefined functions. These functions take as input the result of the previous texture unit and the texture coordinates of the current unit. The input data are combined in some way defined by the function and produce the texture coordinates from which the colour is fetched for this texture unit. Figure 15 shows how the operation scheme of a texture unit in GeForce3.

![Figure 15. Overview of a texture unit in GeForce3.](image)

Many of these functions use vectors instead of colours, and therefore need to translate the colour information (RGBA) to a vector (XYZW). Each predefined texture shader function in the GF3 uses its own translation from RGBA to XYZW to assure the data is used most appropriately. See NVIDIA’s documentation for a detailed description of the texture shaders’ functionality.

**Register combiners**

The third programmable stage is the register combiners [9]. The purpose of the register combiners is to mix all fragments from textures, colours, fog etc. to produce one output colour to assign to the pixel. The register combiners on the GF3 consist of eight general combiners, and one final combiner. Each of the general combiners can be set to one of five predefined functions. The output of the first unit can be used as input in the next one, if needed. The final combiner performs special operations often used in colour blending.
6. Implementation of interpolated bump mapping

This chapter will present a detailed view of the implementation of interpolated bump mapping using GF3. First there is some information regarding reference spaces to help understand the transformations needed in the vertex program. Then the used lighting model is explained. Finally the GeForce3 implementation of bump mapping is described in detail.

6.1. Reference spaces

Different vectors are defined in different reference spaces. Vertices in objects are for example defined in their local object space, whereas light direction vectors are defined in eye space. This application works mostly with the three orthogonal reference spaces described below.

Texture space

Texture space (tangent space, surface-local space) is a space local to each point on an objects surface. It is spun of three orthogonal base vectors, where the x- and y-axis are directed along the tangent of the surface, and the z-axis is the normal direction. The three vectors form a left-hand system. Bump map normals are defined in texture space. It is important that the texture space of a bump mapped object is defined so that the bump map of that object is correctly aligned along the surface.

Object space

Object space is a space local to a 3D-object. Each object has its own object space. Vertices and normals describing the shape of an object are defined in object space. The transformation matrix that transforms a vector between object space and eye space in OpenGL is called the modelview matrix.

Eye space

Eye space is the space with the eye in the origin, z-axis towards the eye, x-axis to the right and y-axis pointing up. This is the global space in this application, and is used for calculating light vectors.

6.2. Lighting model

Bump mapping uses local surface normals to calculate surface lighting at high detail. Some approximations are necessary to accomplish this in real time rendering, as discussed in the results chapter.

The lighting model typically uses surface and light colours, normal direction, light direction vectors and possibly light distance to calculate the pixel colour. There are many lighting models available, each having different advantages and disadvantages.

The default lighting model used by OpenGL is Gouraud shading. It is a fairly simple technique that linearly interpolates colour across polygons. The colours at each polygon’s corners are usually calculated using the corners normal direction, light direction, material

---

1 A left had coordinate system is defined by \( x \times y = -z \).
coefficients etc. Gouraud shading is not suitable for bump mapping, since it cannot produce the colour variation required by bump mapping to simulate small surface displacements. Instead most bump mapping implementations uses Blinn’s lighting model [10], which is a variant of Phong’s reflection model [11]. It is popular because it looks fairly realistic and is relatively simple to implement. Blinn’s model consists of three parts: ambient, diffuse and specular, as visualized in Figure 16.

![Diagram of lighting model](image)

**Figure 16. Different reflections used by Blinn’s lighting model.**

Ambient light is the light surrounding all things. It is of equal intensity and colour everywhere. Reflected light colour depends on material ambient factor and ambient light colour.

Diffuse light is the light reflected from a surface equally in every direction. Its colour depends on the material diffuse factor, the colour of the light-source and the angle at which the light hits the surface.

Specular light is the light reflected from a surface in the mirror direction of the light. Its colour depends on material specular factor and the colour of the light-source.

When calculating ambient light, the eye position is irrelevant, since the surface reflects evenly in every direction. Diffuse light reaches the eye if the surface is visible to the eye and the light-source. This is a fairly simple test, and besides that the eye position is not required for the light calculations. When calculating the specular term, on the other hand, the eye vector needs to be considered. The difference between the eye direction vector and the mirror direction vector affect the amount of light received by the eye.

Blinn’s lighting model uses a help vector, called the half-angle vector to aid the lighting calculations. The half-angle vector is defined as follows:
\[ H = \frac{E + L}{|E + L|} \]

*Equation 8. Half-angle vector definition*

Where:

<table>
<thead>
<tr>
<th>( E )</th>
<th>Normalized vector pointing towards eye</th>
</tr>
</thead>
<tbody>
<tr>
<td>( L )</td>
<td>Normalized vector pointing towards light-source</td>
</tr>
</tbody>
</table>

Figure 17 shows the geometrical relation between \( H, E \) and \( L \).

*Figure 17. Geometrical relation of half-angle vector*

Summing ambient, diffuse and specular light results in:

\[ C = C_a m_a + C_d m_d (N \cdot L) + C_s m_s (N \cdot H)^n \]

*Equation 9. Blinn’s lighting model*

Where:

<table>
<thead>
<tr>
<th>( C )</th>
<th>Fragment output colour</th>
</tr>
</thead>
<tbody>
<tr>
<td>( C_a )</td>
<td>Ambient light colour</td>
</tr>
<tr>
<td>( m_a )</td>
<td>Material ambient light factor</td>
</tr>
<tr>
<td>( C_d )</td>
<td>Diffuse light colour</td>
</tr>
<tr>
<td>( m_d )</td>
<td>Material diffuse light factor</td>
</tr>
<tr>
<td>( C_s )</td>
<td>Specular light colour</td>
</tr>
<tr>
<td>( m_s )</td>
<td>Material specular light factor</td>
</tr>
<tr>
<td>( N )</td>
<td>Normalized surface normal</td>
</tr>
</tbody>
</table>
The highlight exponent describes the surface roughness. If \( n \) assumes a large value \((\geq 10)\), the surface will look very shiny, but if a smaller value is used \((<10)\), the surface looks rougher. A perfect mirror has \( n = \infty \).

### 6.3. Bump mapping on the GeForce3

Presented here is an explanation of how all three stages are programmed to achieve interpolated bump mapping.

**Vertex program implementation**

In order to compare light and normal vectors they need to be transformed to the same reference space. I chose to transform light vectors to texture space using the vertex program. This eliminates the need of transforming the normal vectors defined by the bump maps, since they are defined in texture space already. The vertex program then passes the light vectors to the per pixel operations for the lighting computations.

The transformed light vector automatically gets linearly interpolated between the vertices by the hardware. This may cause the light vector to be un-normalized, which results in lost highlights. See the results and discussion chapter for more information regarding approximations.

The vertex program has many tasks. Besides the proprietary operations, enabling bump mapping etc, the program also performs some basic operations normally taken care of by OpenGL. This is necessary since the vertex program completely replaces OpenGL’s default program. The vertex program carries out the following tasks:

- Transformation of the vertex’s coordinates from object space to clip space\(^1\) using OpenGL’s modelview and projection matrix.\(^*\)
- Calculation of base vectors for texture space using vertex normal and one tangent vector defined by application.
- Transformation of light vector to texture space using texture space base vectors.
- Calculation of half-angle vector, and transform it to texture space.
- Passing on all texture coordinates.\(^*\)

The two tasks that are marked with a * represent tasks normally done by OpenGL. The three other tasks are specific for this application.

See appendix for a listing of the vertex program.

---

\(^{1}\) The reference space used for clipping and viewing.
**Texture shaders implementation**

The texture shaders’ task is to adjust the normal vectors supplied by the wrinkle map according to the current wrinkle intensity. The adjusted normal must be of unit length to function as predicted in the light calculation.

Unfortunately it is not possible to calculate the normal adjustment and then renormalize it, keeping the direction unaffected, using the texture shaders on the GF3. There is, however another way of renormalizing the vector, but it also affects the direction slightly. Three texture units are required to achieve the normal adjustment.

**Texture unit 1**

The first unit fetches the normal from the wrinkle map and passes it on to the following texture units.

The wrinkle map describes a normal per element, but the normals are not of unit length. In order to describe a normal direction one need two values, \( x \) and \( y \) are used for that. The \( z \)-value of the normal is set to 1. The reason for this is that the constant \( z \)-value is used to offset the resulting texture coordinates, as explained below.

Figure 18 shows an example of wrinkle map coordinates, and their interpreted direction.

![Figure 18. (a) Three normals stored in wrinkle map, and (b) the directions they represent.](image)

**Texture unit 2 & 3**

The second and third units are configured as dot products, as shown in Figure 19.

![Figure 19. Texture unit configuration.](image)
The dot products make it possible to deform the normal and create new texture coordinates. Setting \((S_2, T_2, R_2) = \frac{1}{2}(I, 0, 1)\) and \((S_3, T_3, R_3) = \frac{1}{2}(0, I, 1)\) results in:

\[
(U_x, U_y) = \frac{1}{2}(IX + 1, IY + 1)
\]

**Equation 10. Texture coordinates modified by wrinkle intensity**

Keeping \(z = 1\) in the wrinkle map allows offsetting the resulting texture coordinate \((U_x, U_y)\) from the range \([-0.5, 0.5]\) to the range \([0,1]\), which is better suited for texture addressing.

The texture coordinates \((U_x, U_y)\) address a special normalization texture that transform \((U_x, U_y)\) to \((N_x, N_y, N_z)\) as follows:

\[
(N_x, N_y, N_z) = \left(2U_z - 1, 2U_y - 1, \sqrt{1 - (2U_x - 1)^2 - (2U_y - 1)^2}\right) = \left(IX, IY, \sqrt{1 - (IX)^2 - (IY)^2}\right)
\]

**Equation 11. Creating a new deformed normal vector**

Figure 20 shows the transformation of the wrinkle map normal to the adjusted normal output by the texture shaders. The surfaces in grey show the vector’s possible positions before and after the transformation.

![Figure 20. Wrinkle map normal transformation using texture shaders.](image)

The adjusted and normalized normal \((N_x, N_y, N_z)\) is passed on to the register combiners for light calculations.
**Renormalizing error**

Using the described technique for normalizing always results in vectors of unity length but may change the vectors direction. The following error analysis determines the factors affecting the error, and specifies limits to keep it within acceptable levels.

Renormalizing the vector this way does not change the direction too much if certain requirements are met. The difference in angle between this normalization and a correctly normalized vector, for all different normal directions and wrinkle intensities, can be seen in Figure 21.

![Figure 21. Angle error when normalizing using only z.](image)

Note that large error occurs when the normal angle is large (i.e. far from the z-axis). An acceptable angle error is about 10 degrees, since this does not affect the visual appearance too much. The wrinkle intensity will assume all values between 0 and 1; it is therefore necessary to limit the angle of the normals described by the wrinkle map.

Limiting the normals’ angle to about 53 degrees, which means keeping $\sqrt{x^2 + y^2} \leq 0.6$ in the wrinkle map, will keep the normalizing error within 10 degrees.

Note that it is possible to use larger angles in the wrinkle map, but one must then be aware that under certain circumstances the renormalizing procedure may cause visible artefacts to the face.
Register Combiners implementation

The register combinators are responsible for colouring the pixel appropriately, using a lighting equation. I’ve selected a special variant of Blinn’s lighting equation (see Equation 9) because it is easy to implement using the register combinators, shown in Equation 12. The difference is that the exponent is selected to a value of four to keep complexity down. The exponent has to assume an integer value since it is calculated using the register combinators. The other difference is introduction of a fading factor, \( f \), defined by Equation 13. The fading factor exists for two reasons. First it eliminates lighting the backside of a surface. It also reduces popping of highlights when the surface just becomes visible. It is a simple trick to make bump mapped surfaces look better.

\[
C = C_a m_a + f \cdot \left( C_d m_d (N \cdot L) + C_s m_s (N \cdot H)^4 \right)
\]

*Equation 12. Implemented variant of Blinn’s lighting equation*

| \( C \) | Fragment output colour |
| \( C_a \) | Ambient light colour |
| \( m_a \) | Material ambient light factor |
| \( C_d \) | Diffuse light colour |
| \( m_d \) | Material diffuse light factor |
| \( C_s \) | Specular light colour |
| \( m_s \) | Material specular light factor |
| \( N \) | Adjusted normal supplied by texture shaders |
| \( L \) | Normalized light direction vector in texture space |
| \( H \) | Half-angle vector in texture space |
| \( f \) | Fading factor |

The fading factor is calculated as:

\[
f = \min \left( 8 \cdot \max \left( L_z, 0 \right), 1 \right)
\]

*Equation 13. Definition of the fading factor*

| \( L_z \) | The z component of the light direction vector in texture space |
7. Perceptual evaluation

A brief evaluation to investigate how people experienced dynamic wrinkles was performed. The test consisted of two video clips showing a talking face. The only difference between the two clips was that one of them used the face with wrinkles and the other without. Moderate wrinkles were applied to the forehead, and were caused by the face raising its eyebrows during periods of the animation.

The clips were shown to ten people. Five with the wrinkles first and then the non-wrinkled face, and five in the reverse order. None had any idea what to look for. Two questions were asked:

1. Did you see any difference between the two faces?

Here are the results:

<table>
<thead>
<tr>
<th>Test group</th>
<th>Yes</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wrinkles first</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Wrinkles last</td>
<td>1</td>
<td>4</td>
</tr>
</tbody>
</table>

2. Which face did you prefer?

All were asked this question. The ones who didn’t see the difference were first informed of the wrinkles and then shown the clips again.

<table>
<thead>
<tr>
<th>Preferred wrinkled face</th>
<th>Preferred non-wrinkled face</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>2*</td>
</tr>
</tbody>
</table>

* One person thought the face looked sceptical with wrinkles and one person thought the face might show wrong expressions using wrinkles.
8. Results and discussion

The implemented solution shows that it is possible to create dynamic wrinkles in real time, although some approximations are needed to use hardware acceleration when rendering bump mapped surfaces.

It’s difficult to draw any conclusions from the results of the evaluation. The animated wrinkled face only had wrinkles in the forehead, which could explain why the wrinkles seemed difficult to notice. This work is not focused on trying to make the face better at expressing itself, but to create the wrinkle technique. To really evaluate the contribution of wrinkles to the communicative situation is left for another study.

The result of question #2 may indicate that the wrinkles look well integrated and fairly realistic, but it is a very brief evaluation, and a more robust and thorough evaluation is desirable.

8.1. Problems

Many of the problems regarding the implementation have to do with mathematical approximation. Since most calculations are extremely time critical, a sacrifice in precision is often necessary. The following approximations are degrading the visible quality.

- Interpolated light vectors are not renormalized. This results in lost highlights across a surface. Refining the register combiners program could approximate a renormalization and improve quality. This effect is acceptable because other surfaces, not affected by wrinkles are also Gouraud-shaded.
- Register combiners operate with 9-bit resolution. This makes it unwise to use a higher exponent than four in the lighting model, as the gradient of colour across surfaces will become visible.
- Register combiners make it impossible to use a floating-point number as exponent in the lighting model, instead an integer value must be used. One can use almost any integer value by multiplying different power of twos.
- Using multiple overlapping wrinkle maps will create independent wrinkles blended together. In reality the wrinkles should affect each other, combining the normals in some way.

Also, to achieve a fairly realistic wrinkled face a reasonably advanced 3D-accelerated graphics card need to be present. At the present time this prevents the application’s ability to run on most computers, but it is likely that most future standard graphics hardware will have support for bump mapping.

8.2. Future work

The focus in this thesis is on the technique of creating and rendering wrinkles. Now when that has been accomplished is must be utilized more to really see the effect of this work. A face has many places where wrinkling occur, and adding wrinkles to the computer face around the eyes and mouth for example (besides the forehead), might produce very nice results.
The wrinkle maps is another area that needs work. The one used by this work is very simple and far too regular to be realistic. More complex areas that should be wrinkled demands more complex wrinkle maps and 3D-model, which is one reason this work only wrinkled the forehead of the face.

Also the techniques used to create wrinkles can be developed further:

NVIDIA has released a programming API, Cg, which helps the programmer to utilize the functionality of new graphics hardware. Converting the application to Cg probably makes it easier to maintain and upgrade at a later time.

It may be possible to optimise the polygon model to increase the effectiveness of the wrinkles. As always the more polygons, the greater the realism.

NVIDIA lists a simple and limited, but still useful normalizing algorithm that can be implemented in the register combiners program. Normalizing light and eye vectors per pixel will produce better results.
9. References


10. Appendix

10.1. Vertex program

---
// Program input:
// v[OPOS], containing the vertex position in object space (x,y,z,w)
// v[NRML], containing the vertex normal direction in object space (x,y,z)
// v[1], containing the vertex tangent direction in object space (x,y,z)
// v[3], light direction in eye space (x,y,z)
// v[TEX0-TEX3], containing texture coordinates (s,t,r,q)
---

// Constant registers:
// c[0-3], multiplied modelview and projection matrix
// c[4-7], modelview matrix
// c[12-15], inverse modelview matrix
// c[16].x, the constant 0.5
// c[16].y, primary colour alpha component
---

// Program start
---
// Output transformed homogenous vertex position
DF4 o[OPOS].x, c[0], v[OPOS];
DF4 o[OPOS].y, c[1], v[OPOS];
DF4 o[OPOS].z, c[2], v[OPOS];
DF4 o[OPOS].w, c[3], v[OPOS];
---

// Create texture space
---
// Normalize and move normal into R1
DF3 R1.w, v[NRML], v[NRML];
RSQ R1.w, R1.w;
MUL R1, v[NRML], R1.w;
// Make ds orthogonal to normal and move to R0
DF3 R0.w, R1, v[1];
MAD R0, -R1, R0.w, v[1];
// Normalize ds
DF3 R0.w, R0, R0;
RSQ R0.w, R0.w;
MUL R0, R0, R0.w;
// Calc dt (R2) from ds (R0) cross normal (R1)
MUL R2, R0.zyxw, R1.zxyw;
MAD R2, -R1.zyxw, R0.zxyw, R2;
---

// Out: (in object space)
// R0 = ds, normalized
// R1 = normal, normalized
// R2 = dt, normalized
---

// Light vector to texture space
---
// Light dir to model space
DF3 R3.x, c[12], v[3];
DF3 R3.y, c[13], v[3];
DF3 R3.z, c[14], v[3];
// Normalize
DF3 R3.w, R3, R3;
RSQ R3.w, R3.w;
MUL R3, R3, R3.w;
// Light vec to texture space
DF3 R4.x, R0, R3;
DF3 R4.y, R2, R3;
DF3 R4.z, R1, R3;
// Adjust to lie in [0,1] and output to diffuse color
MAD o[COL0], R4, c[16].x, c[16].x;
// Out:
// R0 = texture space ds, normalized
// R1 = texture space normal, normalized
// R2 = texture space dt, normalized
// R3 = model space light dir, normalized
// R4 = texture space light dir, normalized

// Half-angle vector to texture space
// Calc coordinates in eye space
DP4 R5.x, c[4], v[OPOS];
DP4 R5.y, c[5], v[OPOS];
DP4 R5.z, c[6], v[OPOS];
DP4 R5.w, c[7], v[OPOS];
// Make non-homogenous and build vertex -> eye vector by negating (eye=[0,0,0])
RCP R5.w, R5.w;
MUL R5, R5, -R5.w;
// Normalize
DP3 R5.w, R5, R5;
RSQ R5.w, R5.w;
MUL R5, R5, R5.w;
// Calc half-eye vector = normalize(coord->eye + coord->light)
ADD R5, v[3], R5;
DP3 R5.w, R5, R5;
RSQ R5.w, R5.w;
MUL R5, R5, R5.w;
// Transform to model-space
DP3 R6.x, c[12], R5;
DP3 R6.y, c[13], R5;
DP3 R6.z, c[14], R5;
// Transform to texture space
DP3 R5.x, R0, R6;
DP3 R5.y, R2, R6;
DP3 R5.z, R1, R6;
// Adjust to [0,1] and output to col1
MAD o[COL1], R5, c[16].x, c[16].x;

// Move the alpha factor to col0.w
MOV o[COL0].w, c[16].y;
// Pass through the texture coords
MOV o[TEX0], v[TEX0];
MOV o[TEX1], v[TEX1];
MOV o[TEX2], v[TEX2];
MOV o[TEX3], v[TEX3];
// End of program
END

10.2. Texture shaders configuration

texture_2d();
dot_product_2d_1of2(expand(tex0));
dot_product_2d_2of2(expand(tex0));
pass_through();
10.3. Register combiner configuration

const0 = (0, 0, 0, 0); // Ambient light * Material ambient factor
{
    rgb{
        spare0 = expand(tex2) . expand(col0); // spare0 contains diffuse term
        spare1 = expand(tex2) . expand(col1); // spare1 contains specular term
    }
    alpha{
        discard = expand(col0.b); // Light z
        discard = expand(col0.b);
        spare0.a = sum(); // 2 * Light z
        scale_by_four(); // 8 * Light z
    }
}
{
    rgb{
        spare = spare1 * spare1; // specular^2
    }
}
{
    rgb{
        spare = spare1 * spare1; // specular^4
    }
}
{
    const0 = (1, 1, 1, 1); // Diffuse light color * Material diffuse factor
    const1 = (1, 1, 1, 1); // Specular light color * Material specular factor
    rgb{
        discard = unsigned(spare0) * const0; // Diffuse light
        discard = unsigned(spare1) * const1; // Specular light
        col0 = sum(); // Diffuse + specular
    }
}
out.rgb = col0 * unsigned(spare0.a) + const0; // (Diffuse + Specular)*shading + Ambient
out.a = col0.a; // The alpha factor