Bachelor's thesis

Information Technology

Digital Media

2016

Antti Tujula

LIGHTING AND NORMAL MAPPING IN COMPUTER GRAPHICS

– Implementing normal mapping in HactEngine

TURUN AMMATTIKORKEAKOULU
TURKU UNIVERSITY OF APPLIED SCIENCES

BACHELOR'S THESIS | ABSTRACT

TURKU UNIVERSITY OF APPLIED SCIENCES

Information Technology | Digital Media

2016 | 62

Instructor: Principal Lecturer Mika Luimula, Adj.Prof.

Antti Tujula

LIGHTING AND NORMAL MAPPING IN COMPUTER GRAPHICS

This Bachelor's thesis examines the use of surface detail mapping and dynamic lighting in 3D graphics and explains the relation between these techniques. Surface detail mapping refers to techniques that use textures to modify the appearance of models in 3D graphics. Some of these techniques can be used to modify the actual shape of a model and some of them set up new normal vectors, which are required for calculating light reflections. By combining proper lighting and surface detail mapping, a 3D model can achieve a nearly realistic appearance.

This thesis covers the basic principles of 3D modeling and introduces the most common vector and matrix mathematics used in 3D graphics. It also introduces the functionality of a basic lighting model called the Blinn-Phong shading model and explains the relation between lighting and surface detail mapping. In addition, it will also introduces the math behind normal mapping that is one of the most common surface detail mapping in 3D graphics.

This thesis is a part of the HactEngine project, which is an open source multi-platform game engine developed by Indium Games, a Finnish game development company. A Game engine is a tool that will speed up the process of developing games and applications. This thesis introduces the implementation of normal mapping in this game engine. HactEngine received Tekes funding in 2015 and it will be released as open source when it is ready.

KEYWORDS:

Normal mapping, lighting, surface detail mapping, shader, Blinn-Phong shading model

OPINNÄYTETYÖ (AMK) | TIIVISTELMÄ

TURUN AMMATTIKORKEAKOULU

Tietotekniikka | Mediatekniikka

2016 | 62

Ohjaaja: yliopettaja Mika Luimula, dos.

Antti Tujula

VALAISTUS JA NORMAALIKARTAT TIETOKONEGRAFIIKASSA

Tämän opinnäytetyön tarkoituksena oli tutkia pintatekstuurien ja dynaamisen valaistuksen käyttöä 3D-grafiikassa, sekä selvittää näiden tekniikoiden yhteyttä toisiinsa. Pintatekstuurit ovat tekstuureja, jolla saadaan luotua yksityiskohtia 3D-mallien pintaan, joko muokkaamalla mallin pinnan verteksejä tai asettamalla mallin pinnoille useita valaistuksen laskemiseen käytettäviä normaalivektoreita. Pintatekstuureilla saadaan luotua illuusioita, jotka yhdessä valaistuksen kanssa saavat mallin näyttämään tarvittaessa hyvinkin realistiselta.

Työ aloitettiin tutkimalla teoriaa 3D-mallinnuksesta ja tähän liittyvästä matematiikasta. Tämän jälkeen työssä tutkittiin 3D-grafiikassa yleisesti käytössä olevaa valaistustekniikkaa, Blinn-Phong-valaistusmallia sekä selvitettiin erilaisten valaistusmallien yhteyttä pintatekstuurien toiminnassa. Työssä keskityttiin normaalikarttojen teknilliseen toteutukseen, jossa selvitetään tämän pintateksturointimenetelmän toiminta matemaattisesti.

Työn käytännön osuudessa ohjelmoitiin normaalikartoille tuki HactEngine-pelimoottorille. Pelimoottori on pelinkehitystä nopeuttava työkalu. HactEngine on Indium Games -yrityksen kehittämä alustariippumaton pelimoottori, jolle myönnettiin Tekes-rahoitus vuonna 2015. Moottori julkaistaan avoimena lähdekoodina sen valmistuttua, jonka jälkeen moottoria voidaan vapaasti käyttää pelien tai sovellusten kehittämiseen.

ASIASANAT:

normaalikartat, valaistus, pintateksturointi, sävytinohjelmointi, Blinn-Phong-sävytysmalli

CONTENT

1 INTRODUCTION.....9

2 VECTORS AND MATRICES.....10

2.1 Vectors.....10

2.2 Unit vectors.....11

2.3 Vector dot product.....12

2.4 Vector cross product.....12

2.5 Matrices.....13

2.6 Identity matrix.....13

2.7 Scaling matrix.....14

2.8 Translation matrix.....14

2.9 Rotation matrix.....15

2.10 Matrix determinant.....16

2.11 Matrix inverse.....16

2.12 Matrix transpose.....17

3 3D MODELING.....18

3.1 Vertices.....18

3.2 Edges.....19

3.3 Faces.....19

3.4 Indexing.....20

3.5 Right-handed and left-handed coordinate systems.....20

3.6 3D spaces.....23

3.7 Object space.....23

3.8 World space.....23

3.9 Camera space.....24

3.10 Screen space.....24

4 SHADERS AND RENDERING.....26

4.1 Rendering.....26

4.2 Shaders.....26

4.3 OpenGL Shading Language (GLSL)....29
4.4 Uniforms....29
4.5 Attributes....30
4.6 Vertex shader....30
4.7 Fragment shader....31

5 SHADING MODELS....32
5.1 Light intensities....32
5.2 Sunlight and RGB color....32
5.3 Absorption & reflection of color....34
5.4 Phong reflection model....35
5.5 Surface normals....36
5.6 Ambient component....37
5.7 Diffuse component....37
5.8 Specular component....39
5.9 Attenuation and final color....42

6 TEXTURE MAPPING....45
6.1 Normal mapping....46
6.2 Object space normal mapping....48
6.3 Tangent space normal mapping....49

7 NORMAL MAPPING IN HACTENGINE....55
7.1 HactEngine introduction....55
7.2 Entity and Properties....55
7.3 Materials....57
7.4 Mesh....57
7.5 Asset manager....58
7.6 C++ implementation....59
7.7 GLSL implementation....59

8 CONCLUSION....61

Appendix 1. Normal mapping vertex shader (normal.vert)
Appendix 2. Normal mapping fragment shader (normal.frag)

PICTURES

Picture 1: 3D model of a dolphin 18
Picture 2: 2D coordinate 20
Picture 3: Right and left handed coordinate systems 21
Picture 4: Model transformation 22
Picture 5: Space matrix calculations 25
Picture 6: Color combinations 33
Picture 7: Color red reflection 34
Picture 8: Color cyan reflection 35
Picture 9: Phong reflection model 36
Picture 10: Cross product 36
Picture 11: Angle of incidence 38
Picture 12: Blinn-Phong reflection vectors 41
Picture 13: Cube with diffuse mapping (left) and cube with diffuse and normal mapping (right) 47
Picture 14: Object space normal map 48
Picture 15: Tangent space normal map 49
Picture 16: TBN vectors 50
Picture 17: Tangent space UV map 51

TABLES

Table 1: Supported platforms of different graphical frameworks 28
Table 2: Comparison between different surface detail mappings *46*

EQUATIONS

Equation 1: Vector formation 10
Equation 2: Vector magnitude 11
Equation 3: Unit vector 11
Equation 4: Vector dot product 12
Equation 5: Vector Cross product 12
Equation 6: Identity matrix 13
Equation 7: Scaling matrix 14
Equation 8: Translation matrix 14
Equation 9: Axis rotation matrices 15
Equation 10: Arbitrary rotation matrix 15
Equation 11 Determinant 16
Equation 12: Matrix inverse 16
Equation 13: Matrix transpose 17
Equation 14: Red surface reflect 34
Equation 15: Cyan surface reflect 35
Equation 16: Ambient component 37
Equation 17: Diffuse component 39
Equation 18: Specular component 41
Equation 19: Blinn-Phong reflection 42
Equation 20: Final color 43
Equation 21: Attention 43
Equation 22: Normal map texture conversion 49
Equation 23: Formula for calculating tangent and bitangent 52
Equation 24: Tangent space to world space matrix 53
Equation 25: World space to tangent space matrix 54

LIST OF ABBREVIATIONS (OR) SYMBOLS

Aspect ratio	Describes the proportional relationship between image width and height.
C++	A high-performance, cross-platform object-oriented high-level programming language.
FBX	FBX (Filmbox) is proprietary 3D file format.
GLM	Open source header only C++ mathematical library for the OpenGL Shading Language (GLSL)
Lua	Powerful, efficient, lightweight, embeddable scripting language.
Material in 3D graphics	A combination of attributes which describes how a surface of given material should look like.
Quaternions	A quaternion is a four-element vector that can be used to encode any rotation in a 3D coordinate system. A quaternion is composed of one real element and three complex elements.
SDL	(Simple DirectMedia Layer) is a Cross-platform low-level development library for game development.
SWIG	Software development tool that connects programs written in C or C++ with a variety of high-level programming languages.
Tekes	Tekes is a organization that provides innovation funding for companies, research organisations, and public sector service providers.

1 INTRODUCTION

Computer calculation and rendering power have evolved enormously in the past 20 years to a point where computers have become irreplaceable tools for many industries. Computer graphics are now a widely used tool in a product's design and prototyping. Graphics processing units (GPUs) are constantly evolving and in the near future they can be powerful enough to produce realistic looking real-time pictures with technologies like ray tracing [1]. Until that time comes, there are some other optimized methods that can be used to simulate the properties of light in real-time computer graphics.

Blinn–Phong is a widely used shading model that was the default shading model in OpenGL until version 3.1, where the Fixed Function Pipeline was removed [2]. Blinn–Phong shading uses model planes and their normals for calculating light reflections. These calculations can be expensive for the GPU, if the model has large amounts of planes and vertices. This is why surface detail mapping has become a widely used optimization technique [3]. One of the surface detail mapping methods is normal mapping, which can be used to reduce the amount of vertex points and planes of the 3D model with minor losses if any in rendering detail [3].

The focus of this thesis is to explain the use of normal mapping in real-time computer graphics and to explain the theory and calculations behind it. This thesis a part of the HactEngine project, which is an open source game engine being developed by the Indium Games. The main goal was to learn the most efficient way to implement normal mapping and add support for it into the HactEngine game engine. All the source code from this thesis is open source and can be used freely based on its licensing. The code is programmed with C++ and GLSL (OpenGL Shading Language). The reader should to have some basic experience about vector and matrix calculations before reading this thesis.

2 VECTORS AND MATRICES

This part of the thesis will cover some vector and matrix math that are widely used in 3D graphics. This information is necessary in order to understand the theory behind lighting and surface detail mapping. All calculations in this thesis are performed in a right-handed coordinate system. The difference between left and right handed systems will be presented later in this thesis in section “3.5 Right-handed and left-handed coordinate systems”.

2.1 Vectors

A vector is a geometric object that has a magnitude (or length) and a direction [4]. In 3D graphics, each vector consist of 3 components: x, y and z. A vector can be formed between two points by subtracting the end point position from the start point position. All vectors have a head (represented as the arrow end) and a tail (the non-arrow end). The head is the location where the vector ends and represents the direction where the vector is pointing. The vector's tail is the location where the vector starts [4]. In a right handed coordinate system vectors are column vectors [4].

If P1 and P2 are points in a 3D space:

$$P1=\begin{bmatrix} x_1 \\ y_1 \\ z_1 \end{bmatrix} \quad P2=\begin{bmatrix} x_2 \\ y_2 \\ z_2 \end{bmatrix}$$

$$\overrightarrow{P1P2} = P2-P1 = \begin{bmatrix} x_2 \\ y_2 \\ z_2 \end{bmatrix} - \begin{bmatrix} x_1 \\ y_1 \\ z_1 \end{bmatrix} = \begin{bmatrix} x_2-x_1 \\ y_2-y_1 \\ z_2-z_1 \end{bmatrix}$$

Equation 1: Vector formation

The magnitude of a vector can be calculated with the equation:

$$\vec{u} = \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

$$|\vec{u}| = \sqrt{(x^2 + y^2 + z^2)}$$

Equation 2: Vector magnitude

2.2 Unit vectors

Unit vectors are vectors that have a magnitude (length) of 1. The normalized vector or versor $\hat{u}$ of a non-zero vector u is the unit vector in the direction of u. Unit vectors are used for calculations that only require the direction of a certain vector [6].

The normalized vector $\hat{u}$ of any non-zero vector u can be calculated by:

1. first calculate the length of vector u, then,

2. divide each of the components (x, y and z) of vector u by its length.

$$\vec{u} = \begin{bmatrix} 2 \\ 1 \\ 2 \end{bmatrix} \qquad |\vec{u}| = \sqrt{(2^2 + 1^2 + 2^2)} = 3$$

$$\hat{u} = \frac{\vec{u}}{|\vec{u}|} = \frac{1}{|\vec{u}|}\vec{u} \qquad \hat{u} = \begin{bmatrix} \frac{2}{3} \\ \frac{1}{3} \\ \frac{2}{3} \end{bmatrix} \quad |\hat{u}| = 1$$

Equation 3: Unit vector

2.3 Vector dot product

The dot product is one of the most important operations in 3D graphics [6]. It is used for many tasks, such as projecting a vector along another, and for finding the magnitude of a vector. It can also be used to measure an angle between two vectors [3]. The dot product works consistently in any number of dimensions. When the operation is used for calculating an angle between 2 vectors, it can be simplified by using unit vectors. This way, the dot product results in the cosine of the angle between these 2 vectors and their magnitudes can be ignored in the calculations [6].

$$\vec{v}\cdot\vec{c}=|\vec{v}||\vec{c}|\cos(\vec{v},\vec{c}) \qquad \hat{v}=\frac{\vec{v}}{|\vec{v}|} \quad \hat{c}=\frac{\vec{c}}{|\vec{c}|} \quad |\hat{v}|=1 \quad |\hat{c}|=1$$

$$\hat{v}\cdot\hat{c}=|\hat{v}||\hat{c}|\cos(\hat{v},\hat{c}) \;=\; \hat{v}\cdot\hat{c}=1*1\cos(\hat{v},\hat{c}) \;=\; \cos(\hat{v},\hat{c})$$

Equation 4: Vector dot product

2.4 Vector cross product

The cross product of 2 vectors results in a vector perpendicular to the two vectors. This means that the cross product operation can be used to calculate vectors that point either straight up or straight down from the surface that is formed by the 2 vectors. A cross product with 2 unit vectors does not necessarily produce a unit vector. A unit vector is only produced if the cross product vectors are in a 90 degree angle with each other. The cross product can be used to calculate the normal vector for a plane in 3D models [6]. The cross product needs a 3D space to work.

$$\vec{a}\times\vec{b}=|\vec{a}||\vec{b}|\sin(\vec{a},\vec{b})$$

Equation 5: Vector cross product

2.5 Matrices

Matrices are arrays in mathematics that can be arranged into rows and columns. Matrices can be used to transform vectors and so to move, scale and rotate 3D models [4]. Multiplying a model matrix a with a proper translation matrix will move the positions of each of the model's points (vertices) and by this way move the vectors (edges) and surfaces (faces) that form the model. All example matrices in this thesis are for a right handed coordinate system, so they will not work in left handed coordinate systems. More info about coordinate system handedness can be read in section “3.5 Right-handed and left-handed coordinate systems”.

2.6 Identity matrix

Identity matrices, or unit matrices, can be used to reset and to initialize matrices. Multiplying a vector with an identity matrix results in exactly the same vector [5].

$$I = \begin{bmatrix} 1000 \\ 0100 \\ 0010 \\ 0001 \end{bmatrix}$$

Equation 6: Identity matrix

2.7 Scaling matrix

Scaling matrices can be used to change the magnitude of each of the vector components and so to scale 3D models. Vectors can be scaled by multiplying them with a scaling matrix. The example matrix in equation 7 shows the location of each component for separate scaling along different axes. This way the model can be scaled separately along each axis [5].

$$S = \begin{bmatrix} x & 0 & 0 & 0 \\ 0 & y & 0 & 0 \\ 0 & 0 & z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Equation 7: Scaling matrix

2.8 Translation matrix

Translation matrices can be used to move vectors in 3D space and so to move 3D model positions. Each component is presented separately so that the vectors can be moved along each axis [5].

$$S = \begin{bmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Equation 8: Translation matrix

2.9 Rotation matrix

A vector can be rotated by multiplying it with a rotation matrix. There are four different types of rotation matrices, one for each axis (x, y and z) and one for rotating around an arbitrary axis. In the below equations, alpha is the rotation in radians [5]. These matrices can be used for rotating vectors and so to rotate 3D models.

$$Rot_x(\alpha)=\begin{bmatrix}1 & 0 & 0 & 0\\ 0 & \cos(\alpha) & -\sin(\alpha) & 0\\ 0 & \sin(\alpha) & \cos(\alpha) & 0\\ 0 & 0 & 0 & 1\end{bmatrix} Rot_y(\alpha)=\begin{bmatrix}\cos(\alpha) & 0 & \sin(\alpha) & 0\\ 0 & 1 & 0 & 0\\ -\sin(\alpha) & 0 & \cos(\alpha) & 0\\ 0 & 0 & 0 & 1\end{bmatrix} Rot_z(\alpha)=\begin{bmatrix}\cos(\alpha) & -\sin(\alpha) & 0 & 0\\ \sin(\alpha) & \cos(\alpha) & 0 & 0\\ 0 & 0 & 1 & 0\\ 0 & 0 & 0 & 1\end{bmatrix}$$

Equation 9: Axis rotation matrices

A matrix that can be used for rotating a vector around an arbitrary axis is a bit more complicated.

The rotation matrix looks like this:

$$where:\ c_\alpha=\cos(\alpha)\ \ s_\alpha=\sin(\alpha)$$

$$Rot_u(\alpha)=\begin{bmatrix}x^2(1-c_\alpha)+c_\alpha & xy(1-c_\alpha)-zs_\alpha & xz(1-c_\alpha)+ys_\alpha & 0\\ xy(1-c_\alpha)+zs_\alpha & y^2(1-c_\alpha)+c_\alpha & yz(1-c_\alpha)-xs_\alpha & 0\\ xz(1-c_\alpha)-ys_\alpha & yz(1-c_\alpha)+xs_\alpha & z^2(1-c_\alpha)+c_\alpha & 0\\ 0 & 0 & 0 & 1\end{bmatrix}$$

Equation 10: Arbitrary rotation matrix

The vector components (x, y, z) which represent the rotation axis must be normalized into unit vectors. In 3D graphics, these matrices are usually formed with external mathematical libraries like GLM [8]. Quaternions can also be used to calculate and to form rotation matrices, but they are not straightly related to normal mapping, so they won't be presented in this thesis.

2.10 Matrix determinant

The determinants are useful values that can be computed from the elements of square matrices. The determinant of a matrix D is denoted *det(D), det D, or |D|.*

2x2 matrix determinant can be calculated:

$$D_2 = \begin{vmatrix} a_{11} a_{12} \\ a_{21} a_{22} \end{vmatrix} = a_{11} a_{22} - a_{21} a_{12}$$

3x3 matrix determinant can be calculated:

$$D_3 = \begin{vmatrix} a_{11} a_{12} a_{13} \\ a_{21} a_{22} a_{23} \\ a_{31} a_{32} a_{33} \end{vmatrix} = a_{11} \begin{vmatrix} a_{22} a_{23} \\ a_{32} a_{33} \end{vmatrix} - a_{12} \begin{vmatrix} a_{21} a_{23} \\ a_{31} a_{33} \end{vmatrix} + a_{13} \begin{vmatrix} a_{21} a_{22} \\ a_{31} a_{32} \end{vmatrix}$$

$$= a_{11} a_{22} a_{33} - a_{11} a_{32} a_{23} - a_{12} a_{21} a_{33} + a_{12} a_{31} a_{23} + a_{13} a_{21} a_{32} - a_{13} a_{31} a_{22}$$

Equation 11: Determinant

2.11 Matrix inverse

If a square matrix *A* is multiplied with another matrix *B* and their multiplication results in an identity matrix, then *B* is called the inverse matrix of *A* and can be written as A^{-1}. With matrix calculations, there is no concept of division. However, with an inverse matrix the same kind of effect can be achieved. A matrix has an inverse matrix if:

$$B = A^{-1}$$

$$A B = B A = I$$

Sometimes matrix has no inversed matrix:

First the positions of a and d is swapped, minus signs are put in front of b and c, and everything is devided by the determinant (ad-bc)

$$A^{-1} = \begin{bmatrix} a\, b \\ c\, d \end{bmatrix}^{-1} = \frac{1}{det(A)} \begin{bmatrix} d - b \\ -c\, a \end{bmatrix} = \frac{1}{ad - bc} \begin{bmatrix} d - b \\ -c\, a \end{bmatrix}$$

Equation 12: Matrix inverse

2.12 Matrix transpose

A matrix transpose can be used to swap the order of matrix rows and columns. The same operation can be used for vectors by turning row vectors into column vectors [5].

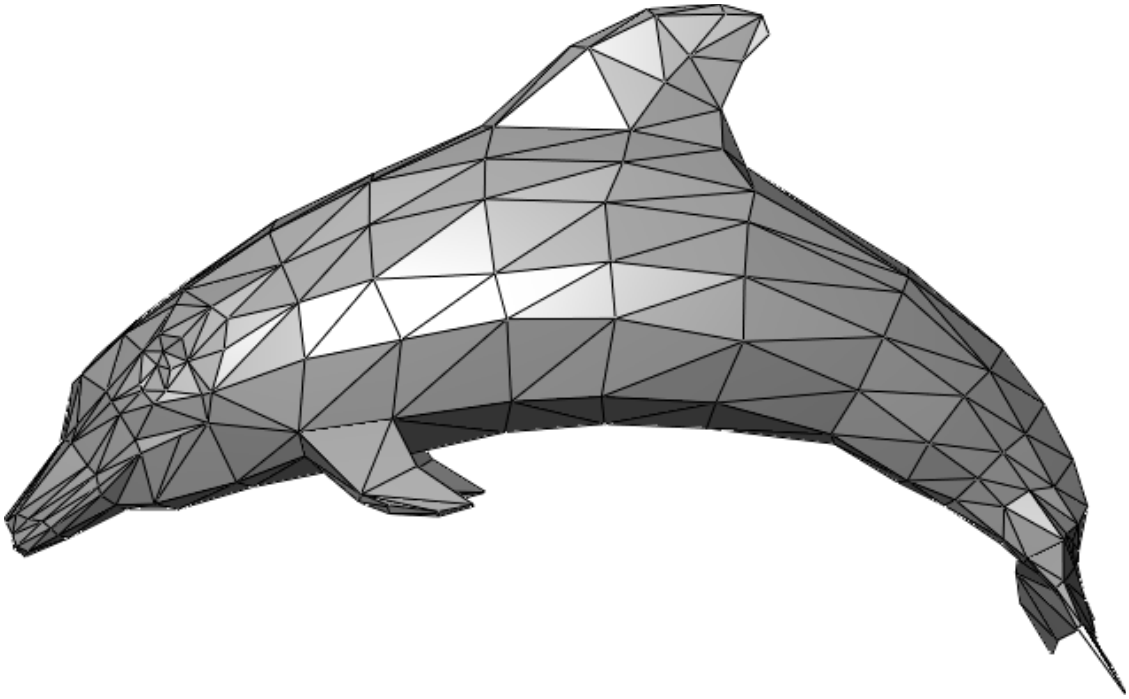
$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}^T = \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}^T = \begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix}$$

Equation 13: Matrix transpose

3 3D MODELING

To understand the basics of surface detail mapping one should understand the basics of 3D modeling and 3D graphics. 3D modeling is a process of developing a mathematical representation of three-dimensional surface objects with computers [1]. They have become an irreplaceable tool for many industries and have replaced the old traditional design methods. These models can also be found in various amounts in different media, like movie special effects, animations, commercials and video games [1].



Picture 1: 3D model of a dolphin [16]

3.1 Vertices

3D models are formed from mathematical points in 3D space, called vertices. Each vertex consists of three components called x, y and z, which represent the location of a point in 3D space [4].

3.2 Edges

In 3D models, 2 vertex points can be thought to form a mathematical vector. The connection between these 2 vertex points form a single edge of the model. This is why these vectors are called edges. Like vectors, edges have a direction and a magnitude. In 3D graphics, models can be rotated, scaled, and translated with matrices. Each of these matrices can be used separately or they can be combined to form a 4x4 sized transformation matrix. This way, model transformations can be performed with a single operation. To be able to perform calculations between 4x4 matrices and vectors, a four-dimensional vector is required. This is why many 3D graphics calculations are done with 4-dimensional vectors that consist of the components x, y, z and w [3]. Here, w can hold special information about the vector. The components x, y, and z are often divided with the w component in many calculations, in order to turn the 4D vector into a 3D vector.

3.3 Faces

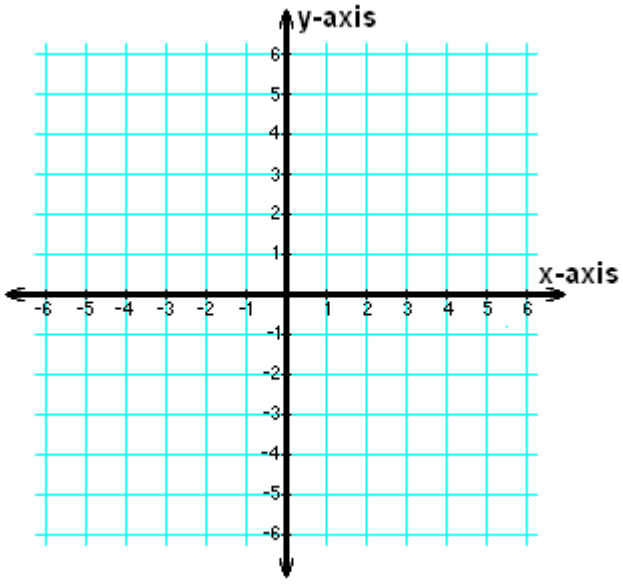
Three edges can be combined together to form a triangle, which is the simplest surface in 3D space. These surfaces are called faces. A model gets formed when multiple faces share the same edges and vertices. This can be seen in Picture 1. Faces can have more than 3 edges. A group of multiple faces is called a mesh.

3.4 Indexing

Indexing vertex points is a memory optimization method used in 3D graphics. 3D models are formed by finding faces that have the same vertex points. Without any optimization, each of the faces have a separate list of their own vertex points, even if these points are identical. This means that the same vertex point location can be loaded to the memory multiple times. By indexing vertices, each of these points gets loaded into memory only once and their location is indexed to reduce memory consumption [7].

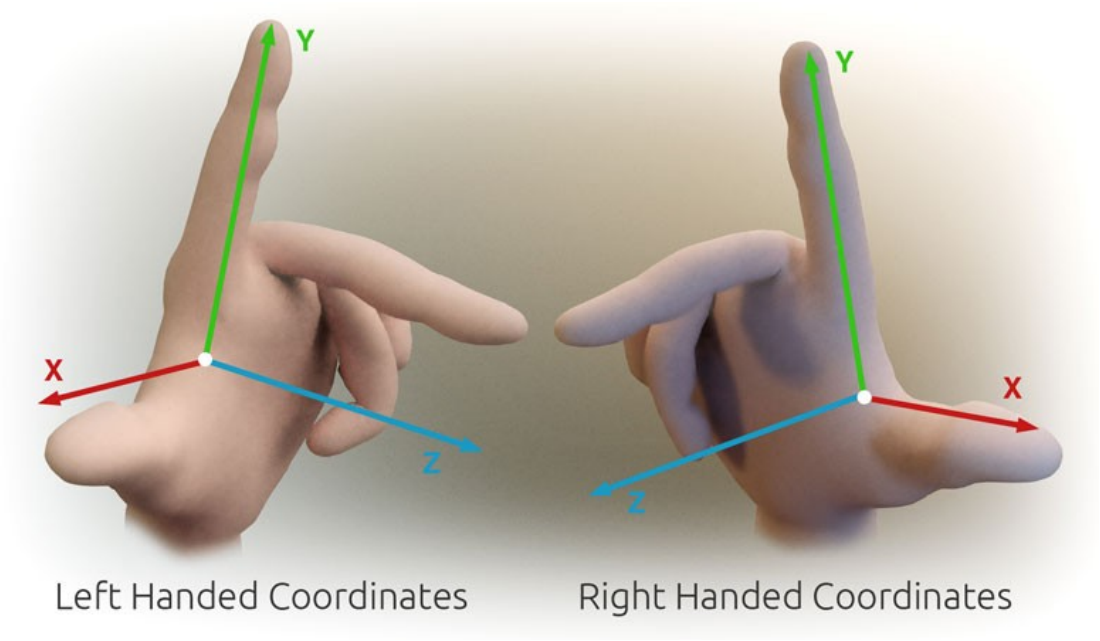
3.5 Right-handed and left-handed coordinate systems

2D spaces have a coordinate system that has 2 axes: x and y. Here, y usually increases towards the up direction and x usually increases towards the right direction. 3D space is not much different. It has 3 axes instead of 2. The third axis is named z and it represents depth in that coordinate system. However, this z axis can increase into 2 different possible directions [4].



Picture 2: 2D coordinate [17]

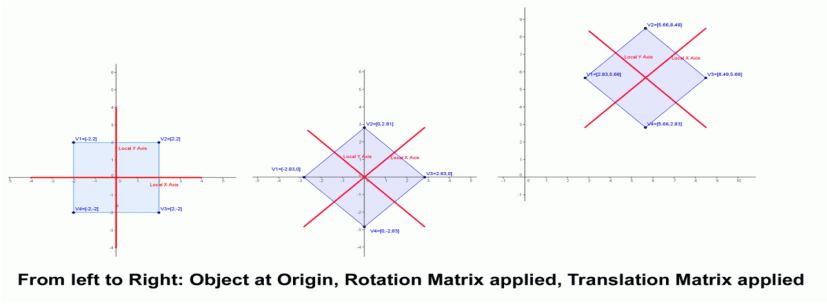
There are 2 different types of 3D coordinate systems, which are left and right handed coordinate systems. The systems get their names from hand positions that help in remembering these systems. A right hand system can be demonstrated by placing the right hand into a position where the thumb points to the right, the index finger points up and the middle finger points straight toward the eyes. The left hand works so that the thumb points again to the right and the index finger points up, but this time the middle finger points away from the eyes [4]. The hand positions are shown in picture 3.



Picture 3: Right and left handed coordinate systems [18]

The difference between these systems is that a right handed system uses column vectors and matrices where as a left handed system uses row vectors and matrices. This causes a difference in the order how the model scaling, rotation and translation calculations need to be done. It is crucial to calculate these in the correct order in both of these systems. In a right handed system, the calculations are done right-to-left, and the model scale and rotation get calculated before translation. A left handed system uses row vectors and matrices and the calculations are done from left-to-right. Transformation order is

the same but matrices are multiplied in reversed order. Picture 4 demonstrates model transformation.



Picture 4: Model transformation [19]

OpenGL is an open source graphical framework which is used by many rendering programs. OpenGL works on multiple platforms and is the only framework that supports almost every platform on the market. In order to simplify this thesis, all calculations are presented in a right-handed coordinate system. By default, OpenGL uses a right-handed coordinate system where the z axis increases toward the screen. Almost every brand of modeling software (e.g. Blender, Maya and 3ds Max) uses a right-handed coordinate system. There are also many programs and frameworks that use a left-handed system by default. One of these frameworks is Microsoft DirectX [7].

Coordinate axes can be modified in these systems. Most modeling software use the z axis as the up axis. This does not make a huge difference, because every 3D model can be converted to any of these systems [4].

3.6 3D spaces

There are multiple coordinate spaces involved in 3D graphics and each of these has their own origin. These systems are [5]:

- object space
- world space
- camera space
- screen space

3.7 Object space

Object space is the local space for each 3D model object. This space is needed, so that every single object can be rotated, scaled, and translated freely. Every object space has its own transformation matrix that keeps the information about the rotations, translations and scales [5].

3.8 World space

Whenever a 3D modeler wants to keep multiple separate 3D objects in the same scene, and to be able to scale, rotate or translate them individually, another object space is required. These spaces are separated with a hierarchy where rotating the parent object will also rotate its child objects. But rotating the child element does not affect the parent object's rotation [5].

Let's assume a modeler has made a model of a pool table. The model consist of a parent object that is a table, which has 15 ball child objects. Each of these balls has their separate object space where the origin is set to the center of the ball. The objects are rotated around their own object space origin, and if an object is moved away from their origin point, they start to orbit around it. In real

life, the balls are rotating around their center point. To be able to move these balls correctly, they need to be moved into another object space. in this case to the table's object space. Objects can be converted to new spaces by multiplying their transformation matrix with the new space's transformation matrix which transforms the object's vertices into that object's space. After this, the balls are moved to their new positions in the different space [5].

The modeler can also rotate the table and still keep the balls in their correct positions. This can be done by rotating the whole model in the pool table's object space with a proper rotation matrix. There can be multiple object spaces in each of the graphical scenes but the topmost one is usually called the world space [5].

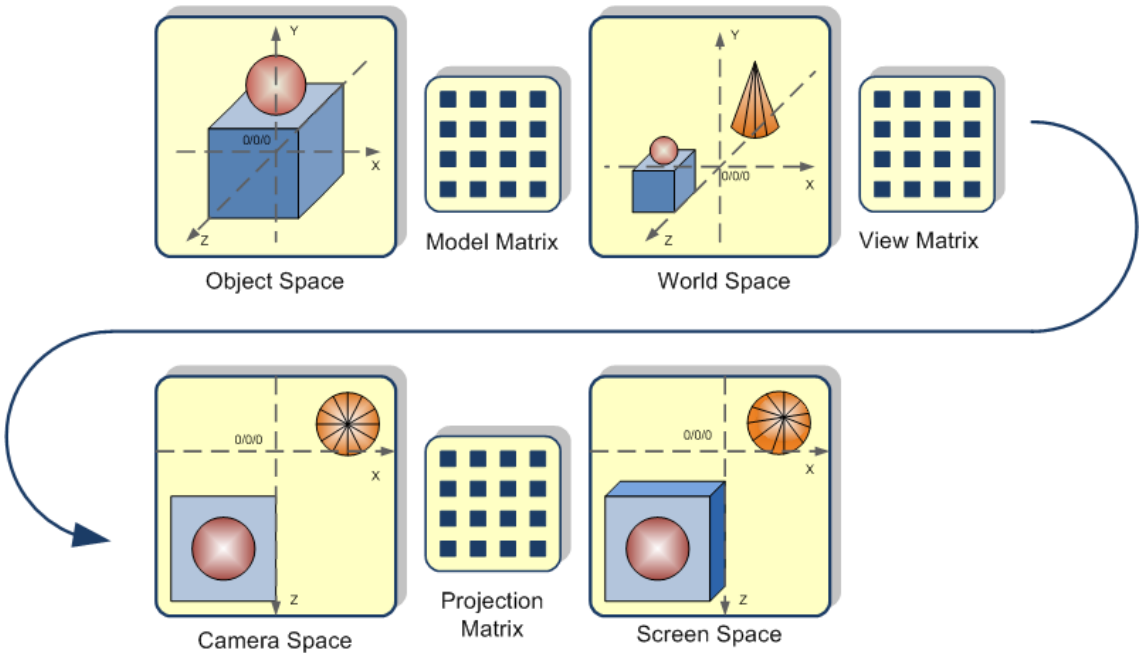
3.9 Camera space

The next space is called the camera space, which is needed in order to move the viewpoint into the world. There are no real cameras in 3D graphics. The user can move the camera or the whole 3D world to get the same end result. Multiplying vertices which are in world space with the camera's view matrix will transform the vertices into the camera’s space. After this, the scene looks like the user is looking at it through the camera [5].

3.10 Screen space

The last space is a projection space where the scene is projected onto the screen. The coordinate space transformation changes the 3D coordinates onto the 2D screen. The screen space transformation will also define the projection with which the models are viewed. These projections are usually either perspective or orthographic projections [5]. Also, the aspect ratio must be taken into account in these calculations, so that the scene won't be distorted when the user is changing the screen size from widescreen to the old style non-widescreen view. This transformation is required as long the display machine

uses 2D displaying [5]. In Picture 5, the changes between spaces can be seen. After this multiplication the image can be rendered onto a screen.



Picture 5: Space matrix calculations [20]

4 SHADERS AND RENDERING

4.1 Rendering

Rendering is the process of generating an image of a 2D or 3D model onto a screen. In the early ages of computing, the CPU (central processing unit) also called a processor, was responsible for calculating every stage of the rendering process. The CPU is optimized for calculating complex equations but it isn't fast when the request is to handle a huge stack of data and performing the same operation multiple times [14]. Every pixel on the screen can be calculated in slots and to make this fast a new device was required [10]. The GPU (Graphics Processing Unit) was born.

There are 2 major types of rendering called offline and real-time rendering. Offline rendering, or pre-rendering, has been used in animations and realistic images that take a long time to render before showing on the screen. Because images get rendered offline, this system can use techniques that require heavy calculations. One of these techniques is ray-tracing where the model gets hit by millions of separate light-rays and every light reflection is calculated separately. This technique is still much too heavy for real-time rendering but the quality of the rendered scene can look almost like in real life [1].

Real-time rendering is widely used in programs like games that require a high frame rate and cheaper rendering calculations. GPUs are constantly evolving, which has made it possible to create better looking graphics each year. Both of these methods can be used with rendering paths written in high-level programming languages. These programs are called shaders.

4.2 Shaders

One of the biggest changes in 3D graphics has been the growth of the popularity of graphical shader programming [10]. Before shaders, graphics

programming was really limited and calculations were done with certain simple formulas [10]. Then came the idea to create high-level programming languages that provided more freedom to the programmers. Shaders are small programs that can be used for modifying the geometries of 3D models and for calculating and rendering a color of a single pixel onto the screen.

One of the first shading languages was called RenderMan, and was developed by Pixar in the early 1980s. It was used in animation movies like Toy Story and Bug's Life [10]. RenderMan was an offline rendering language, but it showed that the way of the future was real-time rendering shaders.

Currently, the most popular shading languages and frameworks are DirectX, and OpenGL, which have slightly different shading languages and support different platforms [12]. This thesis focuses on OpenGL, which is a cross-platform framework that supports platforms like Linux, Windows and OS X. There is also framework called OpenGL ES, which is meant for mobile platforms.

These systems are still really popular but next generation frameworks are also coming to the market. In September of 2013, AMD announced a new, low-level graphics framework called Mantle, designed to be an alternative to OpenGL and Direct3D. The idea of Mantle was to allow direct access to AMD hardware with minimum driver overhead. Most importantly, it paved the way for parallel programming in shaders, increasing the performance available to graphics programming. This was something that OpenGL and DirectX didn't really offer at the time. After some time passed, Microsoft announced that it was developing similar support for its new DirectX 12. In 2014, Apple followed suit and announced their own graphics API called Metal [12]. This same year the OpenGL holding company, Khronos Group Inc, announced their next generation API, called Vulkan. Vulkan is currently the only cross-platform next generation API on the market. Since the other next generation frameworks support only limited platforms, it will surely make Vulkan a tempting platform for future developers. Currently, there is still a limited amount of devices that support

these next generation frameworks. Table 1 shows platform support across different frameworks.

Table 1 Supported platforms of different graphical frameworks

Framework	Windows	Linux	OS X	Android	iOS	Other
DirectX <= v.11	✓	-	-	-	-	Xbox, Windows Phone
OpenGL	✓	✓	✓	-	-	-
OpenGL ES	✓	✓	✓	✓	✓	PlayStation 3
Next Generation:						
Vulkan	Since: OpenGL 4.X	Since: OpenGL 4.X	Since: OpenGL 4.X	Since: OpenGL ES 3.1	Since: OpenGL ES 3.1	-
Metal	-	-	Since: OS X El Capitan version 10.11	-	Since Apple A7, iOS 8	-
Mantle	-	-	-	-	-	Only AMD GPUs
DirectX 12	Windows 10	-	-	-	-	Xbox One, Windows Phone

4.3 OpenGL Shading Language (GLSL)

GLSL is the base programming language in the OpenGL framework. It is a C-style language, and covers most of the features one would expect with such a language [13]. It is simple but powerful. Support for GLSL first came with OpenGL version 2.0 and the old Fixed Function Pipeline was first deprecated in version 3.0 and finally removed in version 3.1. OpenGL provides 5 different types of shader stages that can be used for different purposes. Some of them are only available in newer versions of OpenGL [13]. Each stage has a set of inputs and outputs, which are passed from a prior stage to subsequent stages [13]. Shaders can pass values from one to another, and some shader programs get run more often than others. The most common shader stages are vertex and fragment shaders. The popular surface detail mapping system, normal mapping, requires the use of both of these shaders. This thesis will present these 2 most common shader stages.

Some of the graphic calculations are still made on the CPU side of the program. OpenGL provides a way to pass these values into the shader program. There are 2 different types of inputs that get passed through one stage to another [13]. These are uniforms and attributes.

4.4 Uniforms

A uniform is a global GLSL variable, which is sent from the CPU side of the program to the GPU. They are called uniforms, because their values are “uniform”, i.e. they do not change between shader stages [13]. Variables that are static during the whole run of a single shader program can be passed around as uniform variables. The most common uniform variables are textures, which cannot be used as attributes, the model matrix, camera matrix, projection matrix and the light positions.

4.5 Attributes

Attributes are user-defined input values which, unlike uniforms, can change between shader stages. Attributes are used for passing data like UV-coordinates, vertices and their normal vectors. After OpenGL version 3.0, came support for vertex array objects (VAOs) and vertex buffer objects (VBOs), which made it easier and faster to pass attributes into shaders. A VAO is an OpenGL object, which stores one or more VBO objects to supply vertex data. It also informs which VBO objects are currently in use and attached to which shader variable. A VBO is an object which is used as the source for vertex array data, like a list of float values [13]. There was also a third input called varying, but its name was changed to in and out qualifiers. Variables marked with the in qualifier can be used to pass data, which is created inside one shader stage, to another [13].

4.6 Vertex shader

A vertex shader is a programmable shader stage that handles the processing of individual vertices [13]. The shader stage gets fed with vertex data, where it calculates its transformation into the post-projection space. The shader is also used for calculating and sending vertex data into the fragment shader. The vertex shader will be executed roughly once for every vertex in the rendered model.

4.7 Fragment shader

The fragment shader is a shader stage that will process fragment data from the vertex rasterization. A fragment has a screen space position (x, y), a depth value (z), and all the interpolated data from previous stages. Each sample of the pixels covered by a primitive generates a fragment [13]. This shader can be used for calculating a fragment color that gets hit by a light source. The most common fragment shader outputs are the end color of a single fragment.

5 SHADING MODELS

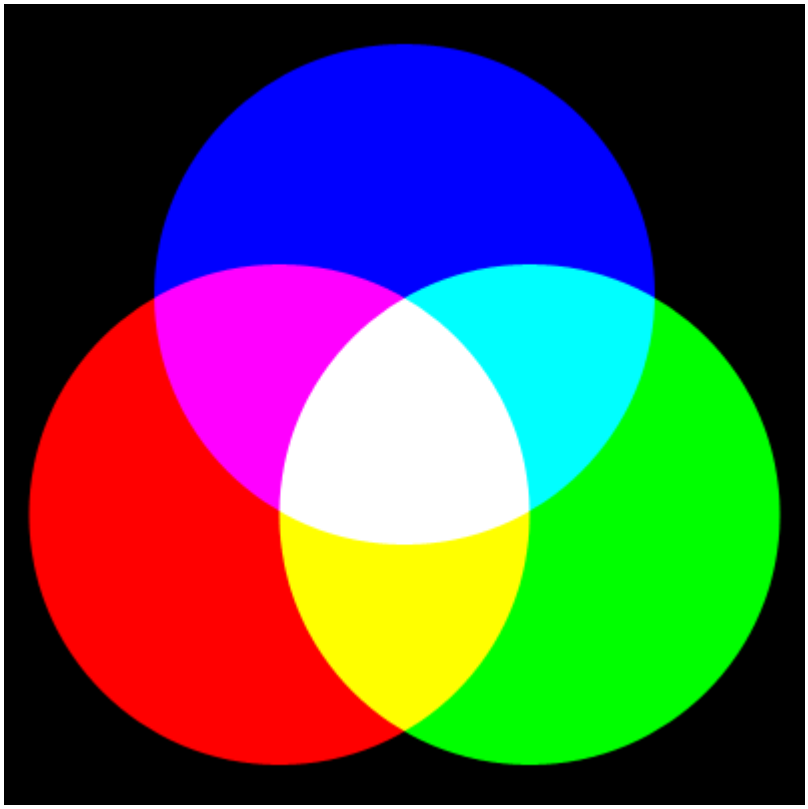
Before one can really understand the use of surface detail mapping they need to know something about lighting in 3D graphics [3]. There are many kinds of different lighting systems in 3D graphics where some of them give more realistic lighting than others [1]. More realistic techniques require much calculation power from the computer, so some of these realistic techniques can't be used in real-time rendering [1]. This thesis introduces a basic shading system that is called the Phong shading model. However, surface detail mapping works with any of these systems.

5.1 Light intensities

Most of these lighting systems are loosely based on the behavior of light in real life [3]. The full implementation of light's behavior is still much too heavy to be calculated in real-time. This is why there are reflection systems that try to imitate light's behaviors, but are much faster to calculate and are therefore more suitable for real time rendering [1].

5.2 Sunlight and RGB color

White light (sunlight) contains all the colors that humans can see [3]. This can be demonstrated by channeling light through a prism, which results in a rainbow. The same thing can be seen in nature, when sunlight is shining through drops of water. White light can also be constructed by taking red, green, and blue light and pointing them on top of each other in a dark room, like is shown in a Picture 6. It can be seen that the color in the center will be white.

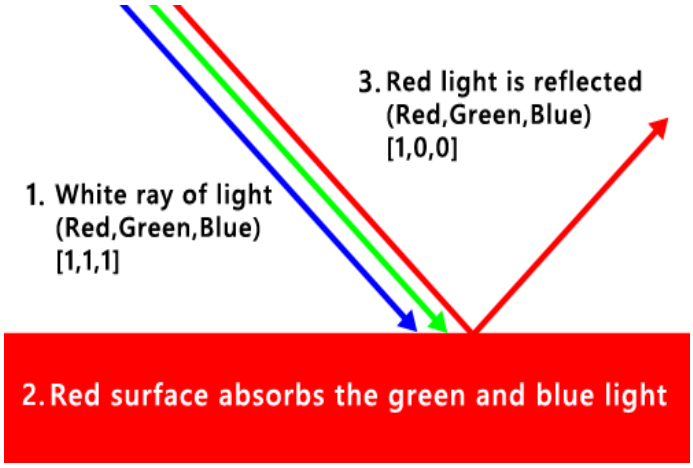


Picture 6: Color combinations [21]

More variations can be created by changing the color of one of the spotlights. This same phenomenon is used in computer graphics in an RGB color system. The center color changes when one of the light components changes brightness. In the RGB color space, each of these color values are represented as a value between 0 and 255 [3]. “RGB” stands for “red”, “green” and “blue”, each of which point to the value of the specific color component. White color can be formed when each of these components are 255. A bright red color can be formed by setting the red value to 255 and the other values to 0 and so on. In computer graphics, the color values can also be represented as values between 0 and 1, so they can be used in vector multiplication. The color can be transformed to the 0 to 1 system by dividing the color component values by 255 [3]. RGB system has a 3 separate components, so it can be stored in a 3D vector and use in vector and matrix calculations. The RGB system can also be expanded to the RGBA system, which has an extra component presenting the alpha channel (transparency).

5.3 Absorption & reflection of color

There are lights that have different colors, but there are also surfaces that have a certain color. What happens when a certain colored light will hit a certain colored surface? If there is a red car that gets pointed at by a white light, the red surface drains the blue and green components from the light and reflects a red color out from the surface [3]. This can be seen in Picture 7.



Picture 7: Color red reflection

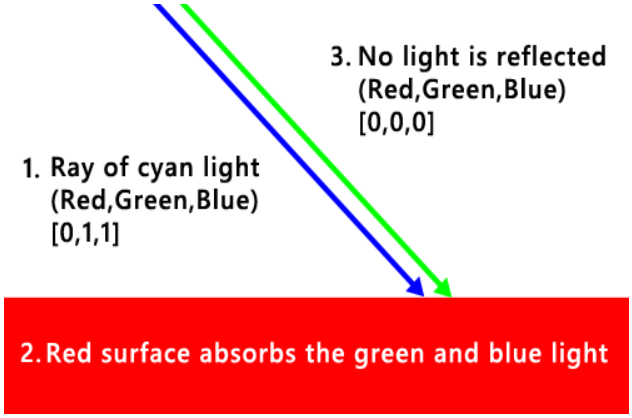
This is simple to understand and can also been demonstrated with the RGB color system by multiplying the surface color components with the light's color components. The multiplication is done by multiplying each of the components individually.

$$\text{white light} = \vec{wl} = [1,1,1]^T \quad \text{red surface} = \vec{rs} = [1,0,0]^T$$

$$reflected\ light = \vec{rl} = [\vec{wl}_x * \vec{rs}_x, \vec{wl}_y * \vec{rs}_y, \vec{wl}_z * \vec{rs}_z]^T = [1*1, 1*0, 1*0]^T = [1,0,0]^T$$

Equation 14: Red surface reflect

What happens when a red surface gets pointed at with cyan light with the color value RGB(0, 255, 255)? This is interesting, because this time the surface does not reflect any light out from the surface [3]. This can been seen in Picture 8.



Picture 8: Color cyan reflection

This may sound odd, but it's certainly true. This same phenomenon happens whenever the reflected surface can't reflect a certain kind of color. This can be demonstrated again by multiplying the light color with the surface color [3].

$$\text{cyan light} = \vec{cl} = [0,1,1]^T \quad \text{red surface} = \vec{rs} = [1,0,0]^T$$

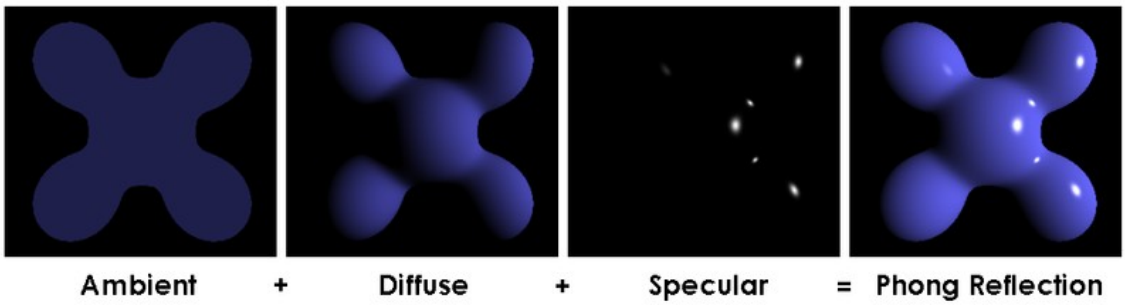
$$\textit{reflected light} = \vec{rl} = [\vec{cl}_x * \vec{rs}_x, \vec{cl}_y * \vec{rs}_y, \vec{cl}_z * \vec{rs}_z]^T = [0*1, 1*0, 1*0]^T = [0,0,0]^T$$

Equation 15: Cyan surface reflect

5.4 Phong reflection model

The Phong shading model is one of the simplest reflection systems in computer graphics. This system is also light-weight enough that it can be used in real-time rendering. It does not produce fully realistic looking lighting, but it's still good enough for most cases and is simple to understand. The system consist of 3 components, which are the ambient, diffuse and specular components. All of

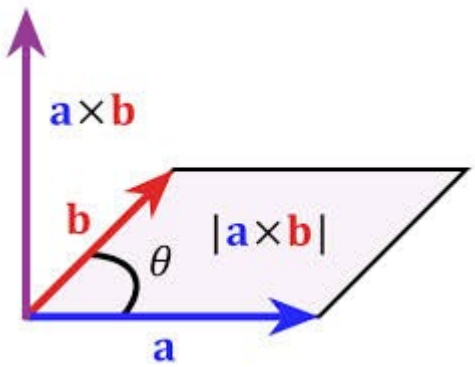
these components require surface normals in order to work [3]. The effect of each of component can be seen in Picture 9.



Picture 9: Phong reflection model [22]

5.5 Surface normals

Surface normals are unit vectors, which point up from the model's surface. Its job is to inform the direction in which surface is facing. A 3D model gets formed from faces, which share the same vertices and edges. Each of these faces has their own normal vectors which point straight up from the face. These faces can be oriented toward any direction in 3D space, and so a normal vector can point to any direction in 3D space. The easiest way to calculate surface normals is to calculate the cross product between 2 edge vectors from each face [3]. This is demonstrated in Picture 10.



Picture 10: Cross product [23]

This normal vector is used for many calculations related to computer graphics and is a really important component in 3D lighting. Normal mapping is a system where these surface normals are modified so that each fragment coordinate on a face has its own normal vector. More info about this technique can be read in the section about normal mapping.

5.6 Ambient component

The ambient component is the simplest component in the Phong shading model. It represents the base color of a model's surface. Without this component, unlighted surfaces would be colored black, which usually looks unnatural. The ambient component can be formed with a coefficient. By changing the value of the coefficient, the user can modify the color of the ambient component. The ambient color can be calculated by multiplying the surface color with the light's color and then multiplying the reflected color value by the ambient coefficient with vector scalar multiplication [3].

white light = $\vec{wl} = [1,1,1]^T$ red surface = $\vec{rs} = [1,0,0]^T$ ambient coefficient = $c = 0.1$

reflected light $= \vec{rl} = [\vec{wl}_x * \vec{rs}_x, \vec{wl}_y * \vec{rs}_y, \vec{wl}_z * \vec{rs}_z]^T = [1,0,0]^T$

ambient component = $\vec{ac} = \vec{rl} * c = [1,0,0]^T * 0.1 = [0.1,0,0]^T$

Equation 16: Ambient component

In the above example, the end result is a dim red color that the surface is set to, if it does not get hit by a light source.

5.7 Diffuse component

The diffuse component is the most important component in the Phong reflection model. It determines the main color of the surface when it gets hit by a light. It is

determined by the angle at which the rays of light hit the surface, called the angle of incidence (AoI) [3]. One can imagine a person holding a white piece of cardboard towards a flashlight in a dark room. The surface color of the piece of cardboard will change depending on the angle between the surface normal and the light source. The cardboard will be the brightest when it is facing straight toward the light and gets darker when the angle between the light and cardboard's surface normal increases. This is demonstrated in Picture 11. Surfaces that are not facing toward the light will be completely dark (apart from the ambient component).

Picture 11: Angle of incidence [24]

The diffuse component mimics this phenomenon. To calculate the diffuse component one must know which direction the cardboard is facing and the direction of the light source. Each model surface will have their own normal vector, which is pointing away from the surface and shows the direction toward which the cardboard is facing. This normal vector can be calculated by taking the cross product of 2 vectors formed by the surface's edges. The light source's direction can be calculated by creating a vector from the surface to the light's position. Now there are 2 different vectors which can be used in the calculations.

Both of these vectors should be normalized into unit vectors. This way, the vector lengths can be ignored in dot product calculations and the dot product directly results in the cosine of the angle between these 2 vectors. The actual

angle is not very important with diffuse calculations. The cosine of the angle will return the coefficient, which can be multiplied with the combined color of the surface and the light. If the light is facing straight toward the surface, the angle will be 0. The cosine of 0 is 1, so the surface color will be 100 % of the combined color from the light and the surface. Angles bigger than 90 degrees will return a negative cosine value, so these faces are facing away from the light. This means that the diffuse component coefficient has to be set to 0, in order to get the correct result, and therefore the surface does not get color from the diffuse component.

Cosine examples:
$\cos(0°)=1.0,\ \cos(45°)=0.707,\ \cos(-45°)=0.707,\ \cos(90°)=0.0,\ \cos(120°)=-0.5$

surface to light unit vector $=\hat{L}$
surface normal unit vector $=\hat{N}$

white light = $\vec{wl}=[1,1,1]^T$ red Surface = $\vec{rc}=[1,0,0]^T$

reflected light $=\vec{rl}=[\vec{wl}_x*\vec{rc}_x, \vec{wl}_y*\vec{rc}_y, \vec{wl}_z*\vec{rc}_z]^T$

Calculations without normalized unit vectors: $\frac{\vec{N}\cdot\vec{L}}{|\vec{L}||\vec{N}|}=\cos(\vec{N},\vec{L})$

Calculation with normalized unit vectors: $\frac{\hat{N}\cdot\hat{L}}{1*1}=\cos(\hat{N},\hat{L})\ =\ \hat{N}\cdot\hat{L}=\cos(\hat{N},\hat{L})$

Only the positive cosine angles counts negative values are set to zero:
diffuse component = $\vec{dc}\ =\ max(0.0, \hat{N}\cdot\hat{L})*\vec{rl}$

Equation 17: Diffuse component

5.8 Specular component

The specular component is used to calculate the shininess of a surface. Specular means a perfect mirror reflection, and gets its name from the real life phenomenon, where light gets reflected away from a shiny surface, before it can get mixed with the surface's colored layer [3]. Again, a car is a good

example of this phenomenon. When looking at a new red car in the sunlight, some parts of the car are seen as red, but some other parts seem to reflect a white light. At these parts, the light gets reflected away from the wax surface, before it hits the red paint layer, and the surface ends up looking white. A perfect specular surface is like a mirror, where light gets reflected in the exact same angle as it hits the surface. If light is hitting the surface at a 20 degree angle, it will be reflected away from the surface with exact same 20 degree angle [3]. If the angle between the person's eyes and the reflected light ray is small enough, the color the person sees is a bright reflected color (e.g. white).

As with the diffuse component, the surface normal vector is important with these calculations. Calculating the specular component requires vectors from the surface toward the light and from the surface toward the camera. These vectors are shown in Picture 12 as the *L* and *V* vectors [3]. Vector *R* is a perfect reflection vector, i.e. a mirrored version of the direction of the light. The main goal here is to calculate the angle between the reflected *R* vector and the camera *V* vector. If the angle is small, light gets reflected toward the camera. Like in real life, some objects are shinier than others [3]. This is why the angle gets powered by a constant value that represents the material’s shininess. Each surface also has its own specular color component. All of these vectors should be normalized into unit vectors to get the correct result.

surface to light unit vector $= \hat{L}$
surface to camera unit vector $= \hat{V}$
surface normal unit vector $= \hat{N}$
reflection unit vector $= \hat{R} = normalize(\hat{L} - 2 * \hat{N} \cdot \hat{L} * \hat{N})$

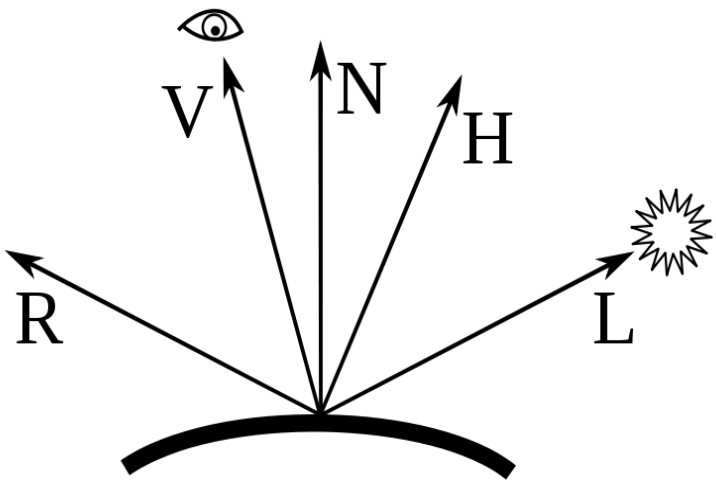
Again only the positive cosine angles count.The surface is on the wrong side of the model if cosine has a negative value and so it is set to zero to get the correct results.

constant material shininess = $M = 0.6$ surface specular color = $\vec{ssc} = [1,1,1]^T$
lightColor = $\vec{lc} = [1,1,1]^T$

$$reflected\ light = \vec{rl} = [\vec{ssc}_x * \vec{lc}_x, \vec{ssc}_y * \vec{lc}_y, \vec{ssc}_z * \vec{lc}_z]^T = [1,1,1]^T$$

$$specular\ coefficient = s = max(0.0, \hat{V} \cdot \hat{R})^M$$
specular component = $\vec{sc} = s * \vec{rl}$

Equation 18: Specular component



Picture 12: Blinn-Phong reflection vectors [25]

There is also an improved version of the Phong model, called the Blinn-Phong reflection model. In this reflection model, one calculates a halfway vector between the camera vector *V* and the light vector *L*. This is shown in Picture 12.

The half vector can be calculated with vector addition from *L* and *V*. Again, it is important to normalize the vectors into unit vectors [10]. This is a more efficient way, because the half vector saves one relatively expensive dot product calculation process, which is otherwise used to calculate the *R* vector [10]. In this method, the specular coefficient can be calculated from taking the dot product of *H* and *N,* instead of *R* and *V* [10].

$$\text{surface normal unit vector} = \hat{N}$$
$$\text{surface to camera unit vector} = \hat{V}$$
$$\text{surface to light unit vector} = \hat{L}$$

$$\text{half angle unit vector} = \hat{H} = normalize(\hat{V} + \hat{L})$$
$$\text{constant material shininess: } M = 0.6$$

$$\text{specular coefficient: s} = max(0.0, \hat{H} \cdot \hat{N})^M$$

Equation 19: Blinn-Phong reflection

5.9 Attenuation and final color

Now that there are 3 components which will form the end color of the surface, which gets hit by a light source. The color gets formed by adding the value of these components together. These components are vectors, so the final color can be achieved with vector addition. If the scene needs an alpha channel, the final color can be converted to a 4-dimensional vector by adding the alpha value as the last component. A value of 1 represents a fully opaque surface and values below this represent the percentage of the surface's transparency. Some texture formats automatically provide alpha channels. For these cases, an alpha value can be achieved from the texture's alpha channel. Also, a surface material can have an opacity value. The final opacity can be calculated by multiplying the material's opacity with the texture's alpha channel.

$\vec{ac}$ = ambient component
$\vec{dc}$ = diffuce component
$\vec{sc}$ = specular component

$$\text{Final color} = \vec{ac} + \vec{dc} + \vec{sc}$$

Equation 20: Final color

There is still one more thing that should be taken into account in this process. Light loses its intensity when the distance between the light and the surface increases [9]. This can be seen for example when a dark hall is lighted by a single candle's light. The light does not have enough intensity to light up the whole hall. This loss of brightness over distance is called attenuation, and it should be taken into account in the calculations to make lights behave like they do in real life. One way to reduce the light's intensity over distance is to simply use a linear equation, so that the light will lose its intensity linearly when the distance gets bigger [9]. In real life, however, light is much brighter at closer ranges. Here is one equation that can be used to calculate this effect [9].

k_c = constant
k_l = linear term
k_q = quadratic term
d = distance

$$F_{att} = \frac{1.0}{k_c + k_l * d + k_q * d^2}$$

Equation 21: Attention

Equation 20 shows a good solution to calculate attenuation. Here, k_c is a constant to avoid division by zero. In most cases, its value is 1.0, but the brightness of the light can be increased by lowering the constant value [9]. k_l is a linear term, which gets multiplied by the distance *d,* from the light to the surface [9]. The last value is k_q, which represents the quadratic term of the attenuation and gets multiplied by the distance squared [9]. This equation will result in a nicely behaving light attenuation. Good values for the equation depend on many aspects, but they are mostly floating point values that are

between 1 and 0 and the quadratic component has the smallest value. Some 3D model file formats (like .fbx) support and provide these values as part of their material data. This allows the modeler to determine the surface and light data inside the modeling software. Attenuation can be added by simply multiplying each of Phong components by the attenuation coefficient.

6 TEXTURE MAPPING

Real life surfaces are rarely completely flat and this is why realistic looking 3D surfaces must also have roughness, holes and bumps on them [9]. One way to achieve this is to model these details straight into the 3D models. However, this takes a lot of time and generates a lot of vertices and faces into the models. The graphics card (GPU) needs to calculate each face separately and every new vertex will increase the time it takes to render the picture onto the screen. In real-time rendering, speed is important so that the program can maintain a good frame rate. If the frame rate drops too low or changes rapidly, the rendered scene will start to annoy the user's eyes and therefore lowers the enjoyment of watching the screen. Still, the models should look good and run smoothly in the program.

One good method for adding details to a model is to use surface detail mapping and texture mappings. The most basic mapping is diffuse mapping, where the model gets its surface color from a texture. This can be seen on the left in Picture 13. However, this will just give a flat look to a model's surface. A better end result can be achieved by also adding surface detail mapping to the model.

There are many different types of surface detail mapping systems. Some of them will add new vertices to the models from textures like with displacement mapping. These techniques usually give better end results, but not necessarily more performance for the program [9]. For this purpose, there is the bump mapping method, which obtains the surface details from a gray-scale height map, where darker shades of gray define the lower surfaces, like holes, and brighter shades represent higher surfaces, like bolts in an iron door. However, a better end result for this kind of mapping can be achieved with the normal mapping technique, which is a slightly improved version of height mapping or bump mapping [9]. Table 2 shows a comparison between these systems. This thesis will focus on normal mapping, because it's the most used surface detail mapping in real-time rendering.

Table 2: Comparison between different surface detail mappings

Surface detail mapping	**Effect on the model**	**From**	**Effect on performance**	**End result**
Height map Bump map	Calculate approximate normal vector for each model fragments	Gray-scale height map	Minor decrease on performance	Decent looking surface details when area is lighted.
Normal map	Contains pre-calculated normal vectors for each model fragment	RGB normal map	Minor decrease on performance	Great looking surface details when area is lighted.
Displacement map	Modify model vertices from texture	Gray-scale height map	Major decrease on performance	Great end result even when the area is not lighted

6.1 Normal mapping

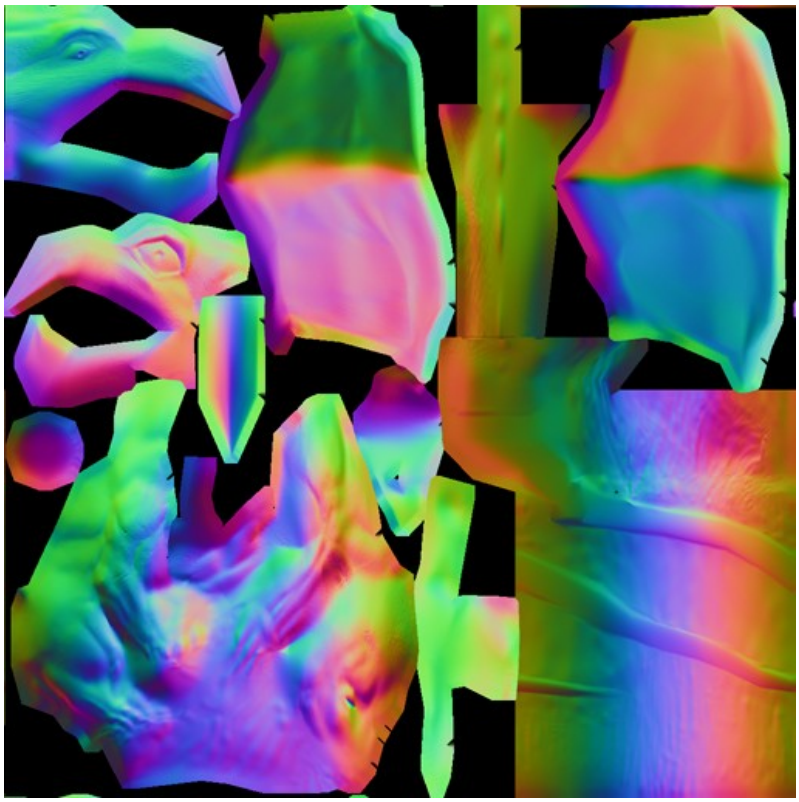
The idea behind normal mapping is really simple. This thesis has shown that each of a model's faces should have their own normal vectors, which are pointing straight away from the surface. This normal vector is used for calculating the light reflections and the angle of incidence of that surface. Normal mapping is a method where a surface can have multiple normal vectors that are obtained from a normal map texture. This way, the surface reflects lights in multiple locations and so even a flat surface can look bumpy, like can be seen in Picture 13. This effect is still fake and the illusion of the detailed surface will disappear, if the angle between the surface and the camera is small enough. Normal mapping is still effective and a good solution for many cases. Normal vectors consist of 3 components, so they can be stored in normal map

textures with RGB color values. Each color value represent the position of the normal vector on a certain coordinate axis [9]. There are 2 different kinds of normal mapping techniques.

Picture 13: Cube with diffuse mapping (left) and cube with diffuse and normal mapping (right)

6.2 Object space normal mapping

In the first method, normals are stored in a world or object space where the normal can point to any direction from the surface and surface normals are calculated in the same object space where the model exists. Object space normal mapping can be recognized from the rainbow colored texture. This method is simpler and slightly faster than tangent space normal mapping, but it has disadvantages. If the model that contains world or object space normal mapping is rotated or gets deformed, its normals will point toward a wrong direction and so has an incorrect end result [11]. This is why tangent space normal mapping is a more commonly used method. Object space normal mapping can be seen in Picture 14.



Picture 14: Object space normal map [26]

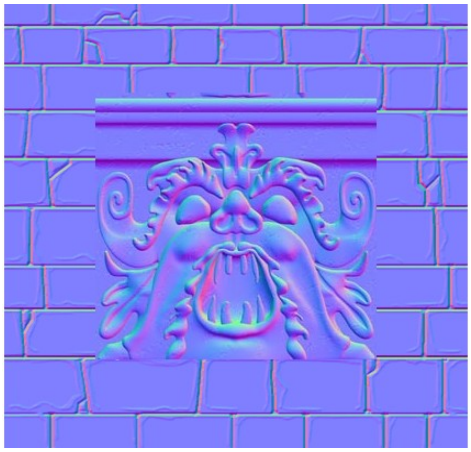
6.3 Tangent space normal mapping

With tangent space normal mapping, normals are stored in a tangent space and the normal vectors are all closely pointing outwards towards the positive z-axis. These vectors can hold values between -1 and 1. Normal vectors can be stored in a texture as RGB values. However, color values have to be stored to the texture between 0 and 1. For these normal vectors, values must be converted to be between 0 and 1. This can be achieved with Equation 21. When the normals are read from the texture they have to be converted back to the -1...1 system [10].

$$r = (x + 1)/2$$

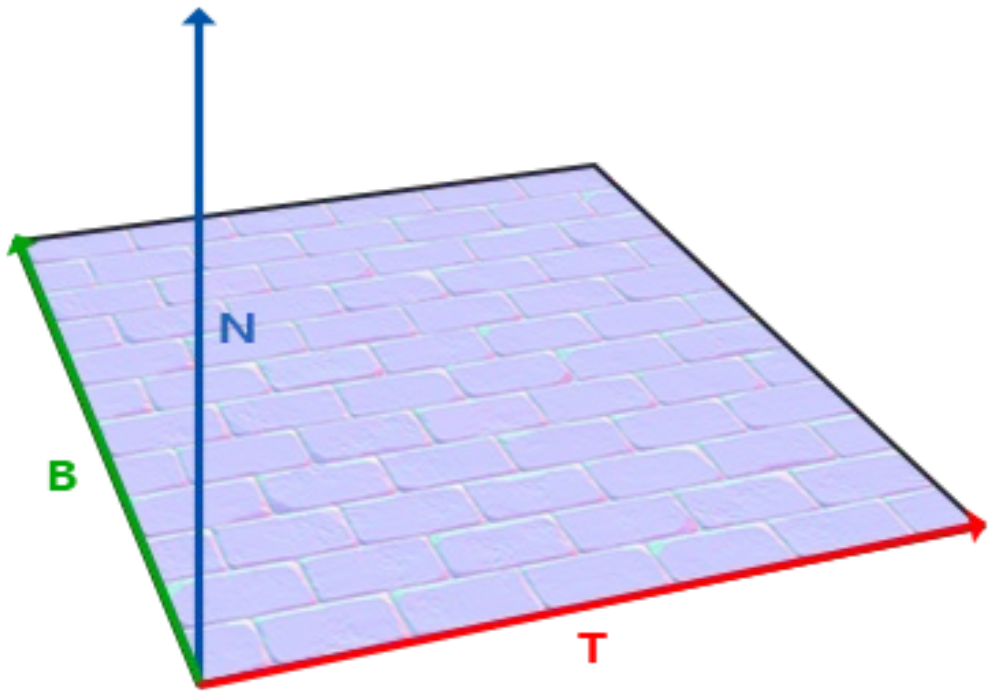
Equation 22: Normal map texture conversion

The z axis is the last coordinate axis in 3D space and blue is the last color in the RGB color space, which is why tangent space normal mapping contains a lot of blueish colors. This system's advantage is that the normals exist in tangent space, and distorting or rotating the model does not affect the direction of the normals. However, this requires that before any calculations, the normals have to be converted to the same object space where the model exist [9]. A tangent space normal map can be seen in Picture 15.



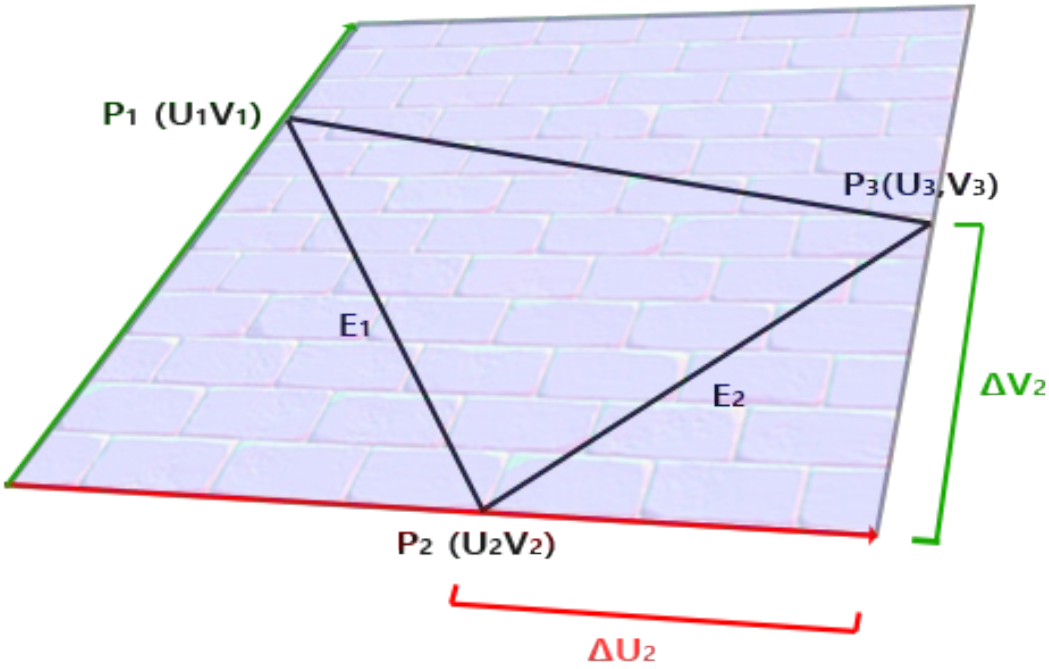
Picture 15: Tangent space normal map

Tangent space can be thought of as the local space of the normal map vectors. Models can be converted to different object spaces by multiplying the model transformation matrix with a proper transformation matrix. This matrix is called TBN, where the letters depict the tangent, bitangent and normal vectors [9]. These 3 vectors are aligned in the surface of normal map, so that the normal vector points out from the surface. The tangent vector points to the right of the normal map and the bitangent points straight up.



Picture 16: TBN vectors [27]

Calculating the tangent and bitangent vectors is a little bit more complicated than calculating the normal vector. The tangent and bitangent vectors align on the 2 edges of the normal map surface, as can be seen in Picture 16. This fact can be used to form an equation that can be used to form these vectors.



Picture 17: Tangent space UV map [28]

Picture 17 shows that the texture coordinate differences of an edge *E2* of a triangle shaped face (denoted as *ΔU2* and *ΔV2*) are expressed in the same direction as the red tangent vector *T* and green bitangent vector *B*. So, both edges *E1* and *E2* can be written as a linear combination of the tangent vector *T* and the bitangent vector *B* [9].

Here is a start equation that can be formed:

$$E_1 = \Delta U_1 T + \Delta V_1 B$$

$$E_2 = \Delta U_2 T + \Delta V_2 B$$

These equations can also be opened to a form:

$$(E_{1x}, E_{1y}, E_{1z}) = \Delta U_1 (T_x, T_y, T_z) + \Delta V_1 (B_x, B_y, B_z)$$
$$(E_{2x}, E_{2y}, E_{2z}) = \Delta U_2 (T_x, T_y, T_z) + \Delta V_2 (B_x, B_y, B_z)$$

E vectors can be calculated as difference vector between two vector positions and ΔU and ΔV as the texture coordinate differences.

Last equations can be written in a different form: that of matrix multiplication:

$$\begin{bmatrix} E_{1x} E_{1y} E_{1z} \\ E_{2x} E_{2y} E_{2z} \end{bmatrix} = \begin{bmatrix} \Delta U_1 \Delta V_1 \\ \Delta U_2 \Delta V_2 \end{bmatrix} \begin{bmatrix} T_x T_y T_z \\ B_x B_y B_z \end{bmatrix}$$

Bouth sides can now be multiplied with inverse of ΔU ΔU matrix.

$$\begin{bmatrix} E_{1x} E_{1y} E_{1z} \\ E_{2x} E_{2y} E_{2z} \end{bmatrix} \begin{bmatrix} \Delta U_1 \Delta V_1 \\ \Delta U_2 \Delta V_2 \end{bmatrix}^{-1} = \begin{bmatrix} T_x T_y T_z \\ B_x B_y B_z \end{bmatrix}$$

Inversed delta texture coordinate matrix can now be calculated and tangent T and bitangent B can be solved from this equation.

$$\begin{bmatrix} T_x T_y T_z \\ B_x B_y B_z \end{bmatrix} = \frac{1}{\Delta U_1 \Delta V_2 - \Delta U_2 \Delta V_1} \begin{bmatrix} \Delta V_2 - \Delta V_1 \\ -\Delta U_2 \Delta U_1 \end{bmatrix} \begin{bmatrix} E_{1x} E_{1y} E_{1z} \\ E_{2x} E_{2y} E_{2z} \end{bmatrix}$$

This is the end formula that can be used to calculate Tangent vector T and bitangent vector B for a single triangle.

Equation 23: Formula for calculating tangent and bitangent

Storing the bitangent vector is optional. This vector can also be easily calculated with a cross product from the normal and tangent vectors. This can be useful, if the goal is to reduce program memory consumption. Many 3D file formats can store pre-calculated tangent and bitangent vectors and 3D model importers like Assimp can calculate them when the model gets loaded into the program [9]. This phase is done on the CPU side of the program. These 3 vectors can now be sent to the GPU, where the program will render the image.

The GPU uses shader programs, where the vectors can be used to form a TBN matrix, which can be used to change the vectors to the tangent space.

There are 2 different ways the TBN matrix can be used for normal mapping. The first way is to create a TBN matrix that will convert any vector from tangent space to world space. With this matrix, a normal vector that is obtained from a normal map can now be converted to world space [9].

TBN matrix from tangent to world space in right handed cordinate system:

world space matrix = M
tangent unit vector $= \hat{T}$
bitangent unit vector $= \hat{B}$
normal unit vector $= \hat{N}$

world space tangent vector $= \hat{WT} =$ normalize(M * $\hat{T}$)
world space bitangent vector $= \hat{WB} =$ normalize(M * $\hat{B}$)
world space normal vector $= \hat{WN} =$ normalize(M * $\hat{N}$)

$$TBN = \begin{bmatrix} \hat{WT}_x & \hat{WB}_x & \hat{WN}_x \\ \hat{WT}_y & \hat{WB}_y & \hat{WN}_y \\ \hat{WT}_z & \hat{WB}_z & \hat{WN}_z \end{bmatrix}$$

Equation 24: Tangent space to world space matrix

This however is usually a less efficient way to use normal mapping. Each model fragment has its own individual normal vector that is obtained from the normal map. In Picture 17 is a triangle that gets formed between 3 vertices. This triangle is textured with a diffuse texture and a normal map. With this method, the world space matrix multiplication must be done for each fragment that are inside the triangle which is a lot of multiplications.

The second way is to create a TBN matrix that converts vectors from world to tangent space. This matrix can be used to convert all calculated vectors like lights, camera, and model vertices to tangent space. After this, lighting can be calculated in tangent space [9]. These conversions need to be done once for each vertex. In a triangle, there are 3 vertices and in this case there are total of

9 conversions. There are hundreds of conversions in the first system. There are almost always more fragments than vertices, and this is why the second system is usually a better option than the first system [9].

TBN matrix from world to tangent space in right handed cordinate system:

world space matrix = M
tangent unit vector $= \hat{T}$
bitangent unit vector $= \hat{B}$
normal unit vector $= \hat{N}$

First calculate inversed world matrix by inverting and transposing world space matrix

normal matrix $= M^{-T}$

After this vectors get multiplyed with this inversed matrix:
tangent space tangent unit vector = $\hat{TT}$ = normalize($M^{-T} * \hat{T}$)
tangent space bitangent unit vector = $\hat{TB}$ = normalize($M^{-T} * \hat{B}$)
tangent space normal unit vector = $\hat{TN}$ = normalize($M^{-T} * \hat{N}$)

$$TBN = \begin{bmatrix} \hat{TT}_x & \hat{TB}_x & \hat{TN}_x \\ \hat{TT}_y & \hat{TB}_y & \hat{TN}_y \\ \hat{TT}_z & \hat{TB}_z & \hat{TN}_z \end{bmatrix}$$

World space to tangent space TBN matrix can be created by transposing TBN matrix

$$TBN^T = \begin{bmatrix} \hat{TT}_x & \hat{TT}_y & \hat{TT}_z \\ \hat{TB}_x & \hat{TB}_y & \hat{TB}_z \\ \hat{TN}_x & \hat{TN}_y & \hat{TN}_z \end{bmatrix}$$

Equation 25: World space to tangent space matrix

7 NORMAL MAPPING IN HACTENGINE

This thesis is a small part of the HactEngine project. It is a documentation of the research process for implementing support for normal mapping into the HactEngine game engine. After this research process, HactEngine supports the surface detail mapping: normal mapping.

7.1 HactEngine introduction

HactEngine is a multi-platform 3D game engine developed by the Finnish company Indium Games. The engine's core is written in C++ and uses the popular SDL library as its base. The engine uses the Lua scripting language, which is extremely easy to learn, but also incredibly powerful. Thanks to Lua, almost everything in the game can be modified while the program is running, saving a lot of development time for the engine's user. HactEngine uses OpenGL as its graphical framework. The engine got Tekes funding in 2015, and the company will release the engine as open source after the engine is finished.

7.2 Entity and Properties

HactEngine has a system where every object in the game world are stored in a hierarchic container class called Entity. Entities can be anything in the game world, like a game state, a 3D model, a camera, or a light or sound source. An entity can also be a combination of multiple resources. Entities are based on a hierarchic structure where the first existing element is called the root entity. The root entity can contain child entities, which can have their own child entities and so on. The engine calls the currently assigned root entity to update and render each frame. These function calls are recursive, so that they are called for each child entity in the entity hierarchy.

Each entity holds a property container that is used for storing data for each entity. These properties can be vectors, matrices, integers, floats, strings, or

references to engine assets. Each entity can be modified by adding new properties to them. A user can create a camera from any entity by adding a camera matrix to it. These entities can be modified in the engine's scripting language, Lua. The engine provides various different classes, which create entities that hold the correct property values for certain use-cases. These will speed up development time when the user can create entities like cameras and lights without worrying about the correct property values for them.

Lua does not support object oriented programming by default. HactEngine provides this feature to Lua. HactEngine's Lua objects can use inheritance and they can also be inherited from C++ classes. The user can call C++ functions from inherited Lua classes in the same way that the user would call standard Lua functions. This is made possible with Lua metatables and the SWIG wrapper generator software. Metatables will also make it easy to get and set property values with a metatable call. The user can read a property value by calling the metamethod and create a new property or set the value of an existing property by assigning a value to it.

HactEngine is thread-safe, and any property can be read or written to from any thread. This way, a user can send data from one thread to another, and rendering, input and game logic can all run on different threads. Threading is a powerful tool, where the user can separate some expensive calculations like physics simulation from other calculations. This way, a separate thread can calculate different calculations simultaneously and one thread does not need to wait for another thread to finish its calculations. HactEngine makes it easy for the user to use threading. A new thread can be created just with one line of code in Lua scripting. Threading also enables the user to run and load new content to the game with a hidden loading thread. This way, the user does not need long loading screen during game sessions. Properties have a render mask, which tell if the property value should be sent to a certain shader program.

Entities can also be created with 3D modeling software, by exporting a 3D scene into the game engine. The engine currently supports more than 40

different 3D file formats. The engine can create automatic entity objects from unmerged 3D objects existing in the scene. This way, single objects become a separate entity, which can be controlled inside the engine. The user can even design a whole game scene inside a modeling program. Some file formats can store UV-coordinates and different kinds of surface detail mappings like normal mapping. HactEngine supports multi-texturing and is able to parse different kinds of surface detail mappings. The engine can also read cameras and different light objects from these scenes.

7.3 Materials

Some file formats can store the materials of 3D meshes. Materials are various variables, which are used to define the visual behavior of 3D surfaces. These variables are sent to a shader program, where they can be used for calculations. Creating different materials can be used to improve the visual quality of a 3D scene. HactEngine can read these materials from 3D file formats and link them to the 3D object meshes. Materials are automatically sent to the shader program.

7.4 Mesh

HactEngine stores each 3D object's vertices as Mesh object, where the vertices are indexed to minimize the memory consumption of the game engine. The class also stores UV-coordinates, normals, tangents and bitangents of each of the model's vertices. The Entity class holds a property reference value to a Mesh object, so that multiple 3D objects can use the same 3D mesh. This way, the game scene can have multiple copies of the same 3D object with minimal memory consumption. The Mesh class sends vertex data to the shader program, where it can be used to render the model to the screen. When there are no more entities that hold a reference to a Mesh object, the engine uses its asset management system and removes the mesh from memory.

7.5 Asset manager

HactEngine has an AssetManager class, which manages the loading and unloading of game assets. These can be e.g. 3D model meshes, music, or textures. The AssetManager maintains a system where the asset will be unloaded automatically, if it has not been in use for a certain time period. If the asset is requested again after unloading it, the engine uses a separate thread to load that asset back into the program automatically. This way, the engine user does not need to worry about the memory and asset management during the development process.

HactEngine uses Assimp (Open Asset Import Library) as its 3D model loader [15]. Assimp is a popular open source asset importer that supports more than 40 different 3D model file formats [15]. It also has a lot of useful features that can be enabled and disabled by the user. One of these features is that the library calculates normals, tangents and bitangents automatically for every model vertex that gets loaded with the library. Using Assimp as a model loader, the developer can ignore the tangent and bitangent vector calculation that can be seen in Equation 22. The engine handles the sending of normal and tangent vectors into the shader program. The engine does not store or send the bitangent vector to the shader. This vector gets calculated inside the vertex shader program with a simple cross product.

To simplify the example code in this thesis, shader programs are written for a forward rendering system. In this system, every light gets calculated for all vertices that are loaded into the shader. A more efficient way is to use deferred rendering, which is implemented in HactEngine. This is a more complicated system, but enables the use of huge amounts of different light sources. In this system, the first shader run will just create the geometry of the scene and the lighting is calculated with a second shader run. This way, lighting can be calculated for every pixel in the screen instead of every fragment of every face.

7.6 C++ implementation

HactEngine has a namespace called AssetManager::Model, which is used to load 3D models into the engine. Implementing normal mapping into the engine required very little additional code into the C++ side of the game engine. Assimp calculates the normal and tangent vectors for each model automatically and the only modification was to save these 2 vectors into the Mesh class and sending them to the shader program. HactEngine uses GLM (OpenGL Mathematics) as its mathematical library [8]. GLM is a popular mathematic library that provides support for variables like vectors and matrices to the C++ environment. HactEngine also provides GLM on the Lua side of the engine. This makes it really fast to modify OpenGL code or shader programs while the game is running.

7.7 GLSL implementation

The GLSL side of the normal mapping implementation required the rewrite of the vertex and fragment shaders. The shaders are written for a forward rendering system in order to demonstrate the use of normal mapping as it is presented in this thesis. HactEngine uses a more advanced version of this rendering system, called deferred rendering. In deferred rendering, the TBN matrix calculation is made on the first run of the vertex and fragment shaders, when the scene gets formed without any lighting. This way, the light calculation can be done in world space and the TBN matrix can be ignored on second vertex and fragment shader run.

Appendices 1 and 2 show the vertex and fragment shaders used for normal mapping. The GLSL implementation uses the world to tangent space matrix that is calculated during the vertex shader stage. This TBN matrix is used to transform the camera, light and model vertex position into tangent space. After this, the vertex shader calculates the vertex position in screen space so that the

image can be seen on the screen. After this, it will send vertex data and the converted attributes into fragment shader.

During fragment shader stage, the first step is to obtain the normal vector from the normal map and turn it from the stored 0 to 1 system to the correct -1 to 1 system. This shader also calculates gamma correction for the diffuse texture. After this, the vector from the surface to the camera gets calculated in tangent space. The final step is to calculate the color of a fragment with the Blinn-Phong shading model and the final color is returned from the shader.

8 CONCLUSION

The use of different surface detail mapping methods are a popular optimization method in both offline and real-time rendering. The theory behind them is very broad and therefore this thesis just scratches the surface of surface detail mapping. The same goes for the different shading models. HactEngine provides a various amount of different shading models including models that are showcased in this thesis. The Blinn-Phong shading model is old, but still a great example of a 3D shading model. Understanding the functionality of this model will surely help the user to understand and to learn new shading techniques. Game engines provide a lot different types of shading models, where the developer can use them without knowing the theory or mathematics behind them. But it will surely never give the user trouble by knowing what's going on inside the shader programs.

HactEngine now supports normal mapping, but in the future it should provide support for various amounts of different surface detail mapping models. However, normal mapping is one of the most popular of these mapping models and is still one of the most effective optimization methods in 3D graphics. Indium Games uses normal mapping for their upcoming game, by mixing 3D models and 2D skeleton animations. With normal mapping, even a flat 2D skeleton animation can obtain more details from the normals obtained from a normal map and by interacting with lights around the game scene.

REFERENCES

[1] Intel converts ET: Quake Wars to ray-tracing, [www document]. available http://www.tgdaily.com/trendwatch-features/37925-intel-converts-et-quake-wars-to-ray-tracing 2008. (read 28.09.2015)

[2] Fixed Function Pipeline, [www document]. available https://www.opengl.org/wiki/Fixed_Function_Pipeline 2015. (read 28.09.2015)

[3] Tom Dalling, Modern OpenGL Series, [www document]. available http://www.tomdalling.com/blog/modern-opengl/08-even-more-lighting-directional-lights-spotlights-multiple-lights 2014. (read 29.09.2015)

[4] Puhakka, Antti. 2008. 3D-grafiikka.Talentum Media Helsinki: 29 - 40s. ISBN 978-952-14-1192-2.

[5] Coordinate Systems in OpenGL, [www document]. available http://www.matrix44.net/cms/notes/opengl-3d-graphics/coordinate-systems-in-opengl 2016. (read 23.03.2016)

[6] Vector, [www document]. available http://www.wildbunny.co.uk/blog/vector-maths-a-primer-for-games-programmers/vector/#Dot. (read 23.03.2016)

[7] OpenGL, [www document]. available https://www.opengl.org/sdk/docs/. (read 23.03.2016)

[8] GML, [www document]. available http://glm.g-truc.net/. (read 23.03.2016)

[9] Learn OpenGL, [www document]. available http://learnopengl.com/. (read 23.04.2016)

[10] Puhakka, Antti. 2008. 3D-grafiikka.Talentum Media Helsinki: 201 - 258s. ISBN 978-952-14-1192-2.

[11] Jonathan Kreuzer, Object Space Normal Mapping with Skeletal Animation Tutorial, [www document]. available http://www.3dkingdoms.com/tutorial.html. (read 23.04.2016)

[12] Tom Dalling, OpenGL in 2014, [www document]. available http://www.tomdalling.com/blog/modern-opengl/opengl-in-2014/. (read 18.04.2016)

[13] GLSL, [www document]. Available https://www.opengl.org/wiki/Core_Language_(GLSL). (read 20.04.2016)

[14] What's the Difference Between a CPU and a GPU, [www document]. Available https://blogs.nvidia.com/blog/2009/12/16/whats-the-difference-between-a-cpu-and-a-gpu/ (read 19.05.2016)

[15] Assimp official web page, [www document]. Available http://www.assimp.org/ (read 21.05.2016)

[16] Polygon mesh, [www document]. Available https://en.wikipedia.org/wiki/Polygon_mesh#/media/File:Dolphin_triangle_mesh.png (read 21.05.2016)

[17] Cartesian Coordinates.PNG, [www document] Available https://commons.wikimedia.org/wiki/File:2D_Cartesian_Coordinates.PNG

[18] Cartesian coordinate system.JPG, [www document] Available https://en.wikipedia.org/wiki/Cartesian_coordinate_system#/media/File:3D_Cartesian_Coodinate_Handedness.jpg (read 21.05.2016)

[19] Object transformation.PNG, [www document]. Available http://www.matrix44.net/cms/wp-content/uploads/2011/03/object_rot_trans_1.png (read 21.05.2016)

[20] Ogl vertex life.PNG, [www document]. Available http://www.matrix44.net/cms/wp-content/uploads/2011/03/ogl_vertex_life.png (read 21.05.2016)

[21] Rgb-light2.PNG, [www document]. Available http://www.tomdalling.com/images/posts/modern-opengl-06/rgb-light2.png (read 21.05.2016)

[22] Cross product.PNG, [www document]. Available https://encrypted-tbn2.gstatic.com/images?q=tbn:ANd9GcR-Joka0-PqDoJqUrQFacpa7pusAlfMd6cPg8uDeKf5hhe6b0nCFA (read 21.05.2016)

[23] Phong components version 4.PNG, [www document]. Available https://en.wikipedia.org/wiki/File:Phong_components_version_4.png (read 21.05.2016)

[24] aoi_min_max.PNG, [www document]. Available http://www.tomdalling.com/images/posts/modern-opengl-06/aoi_min_max.png (read 21.05.2016)

[25] Blinn Vectors.SVG, [www document]. Available https://commons.wikimedia.org/wiki/File:Blinn_Vectors.svg (read 21.05.2016)

[26] Gargoyle-uv world space.JPG, [www document]. Available http://www.surlybird.com/tutorials/TangentSpace/images/gargoyle-uv_world_space.jpg (read 21.05.2016)

[27] Normal_mapping_tbn_vectors.PNG, [www document]. Available http://learnopengl.com/img/advanced-lighting/normal_mapping_tbn_vectors.png (read 21.05.2016)

[28] normal_mapping_surface_edges.PNG, [www document]. Available http://learnopengl.com/img/advanced-lighting/normal_mapping_surface_edges.png (read 21.05.2016)

```
#version 330 core

// VERTEX SHADER (main.vert)

// Coordinates
layout(location = 1) in vec3 v_vertex;
layout(location = 2) in vec3 v_normal;
layout(location = 3) in vec3 v_tangent;
layout(location = 5) in vec2 v_uv;
layout(location = 7) in vec2 v_uv2;
layout(location = 6) in vec4 v_color;
layout(location = 8) in vec4 v_color2;

out Properties {
    vec3 f_vertex;
    vec3 f_normal;
    vec2 f_uv;
    vec4 tangentScreenPosition;
    vec3 tangentLightPosition;
} properties;

// Light properties struct
struct Light {
    vec4 position;
    vec3 color;
    float attenuationConstant;
    float attenuationLinear;
    float attenuationQuadratic;
    float ambientCoefficient;
};

// Light uniform
uniform Light light;

// Model material struct
struct Material {
    float opacity;
    float shininess;

    vec3 diffuse;
    vec3 ambient;
    vec3 specular;
    vec3 emissive;
};

// Material uniform
uniform Material material;

// Texture
struct Textures {
    sampler2D diffuse;
    sampler2D normal;
};

uniform Textures textures;

// Matrices
uniform mat4 projection;
uniform mat4 camera;
uniform mat4 model;


/*!
 * Vertex shader main function.
 */
void main(void) {
    // Set Identity matrix as TBN matrix
    mat3 TBN = mat3(1);

    // Calculate world to tangent space matrix
    mat3 normalMatrix =  transpose(inverse(mat3(model)));
    // Calcuate tangent space if it exists
```

```
73      if (v_tangent != vec3(0.0)) {
74          // World space to tangent space matrix
75          vec3 N = normalize(normalMatrix * v_normal);
76          vec3 T = normalize(normalMatrix * v_tangent);
77
78          // Use Gram-Schmidt process and re-orthogonalize tangent with respect to normal
79          T = normalize(T - dot(T, N) * N);
80
81          // T and N are in 90 degree angle cross product return unit vector
82          vec3 B = cross(T, N);
83
84          // Create matrix from TBN vectors and transpose matrix
85          TBN = transpose(mat3(T, B, N));
86      }
87
88      // Push uv cordinate to fragment shader
89      properties.f_uv = v_uv;
90
91      // Calculate vertex position in tangent space
92      properties.f_vertex = TBN * (model * vec4(v_vertex, 1.0)).xyz;
93
94      // Calculate normal for surfaces that does not have normal mapping
95      properties.f_normal = TBN * normalize(mat3(model) * normalize(v_normal));
96
97      // Obtain camera position from camera matrix
98      properties.tangentScreenPosition = vec4(-transpose(mat3(camera)) * camera[3].xyz, 1.0);
99
100     // Set camera to tangent space
101     properties.tangentScreenPosition = vec4(TBN * vec3(properties.tangentScreenPosition.xyz),
    1.0);
102
103     // Light position in tangent space
104     properties.tangentLightPosition = TBN * light.position.xyz;
105
106     // Calculate the vertex position
107     gl_Position = projection * camera * model * vec4(v_vertex.xyz, 1.0);
108 }
```

```
#version 330 core

// FRAGMENT SHADER (main.frag)


// Coordinates
in Properties {
    vec3 f_vertex;
    vec3 f_normal;
    vec2 f_uv;
    vec4 tangentScreenPosition;
    vec3 tangentLightPosition;
} properties;


// Light properties struct
struct Light {
    vec4 position;
    vec3 color;
    float attenuationConstant;
    float attenuationLinear;
    float attenuationQuadratic;
    float ambientCoefficient;
};

// Light uniform
uniform Light light;

// Model material struct
struct Material {
    float opacity;
    float shininess;

    vec3 diffuse;
    vec3 ambient;
    vec3 specular;
    vec3 emissive;
};

// Model material uniform
uniform Material material;


// Texture
struct Textures {
    sampler2D diffuse;
    sampler2D normal;
};

uniform Textures textures;

// Matrices
uniform mat4 projection;
uniform mat4 camera;
uniform mat4 model;

// Out color
out vec4 outColor;


/*!
 * Blinn–Phong shading model.
 *
 * :param normal          Surface normal.
 * :param diffuseFragment Original surface color.
 * :param surfaceToCamera Surface to camera direction.
 * :param surfacePos      Surface position.
 *
 * :return Original color mixed with light color.
 */
vec3 BlinnPhong(const in vec3 normal, const in vec4 diffuseFragment, const in vec3
surfaceToCamera, const in vec3 surfacePos) {
```

```
72      // Calculate light direction
73      vec3 lightDirection = properties.tangentLightPosition.xyz - surfacePos;
74
75      // Calculate attenuation
76      float lightDistance = length(lightDirection);
77      float attenuation = (1.0f / (
78          light.attenuationConstant + // constant
79          light.attenuationLinear * lightDistance + // linear component
80          light.attenuationQuadratic * lightDistance * lightDistance // Quadratic component
81      ));
82
83      // Normalize light direction
84      lightDirection = normalize(lightDirection);
85
86      // Ambient
87      vec3 ambient = light.ambientCoefficient * diffuseFragment.rgb * light.color.rgb;
88
89
90      // Diffuse
91      float diffuseCoefficient = max(0.0, dot(normal, lightDirection));
92      vec3 diffuse = diffuseCoefficient *  diffuseFragment.rgb * light.color.rgb;
93
94
95      // Specular (Blinn–Phong)
96      vec3 halfVector = normalize(surfaceToCamera + lightDirection);
97      float specularCoefficient = pow(max(0.0, dot(normal, halfVector)), material.shininess);
98      vec3 specular = specularCoefficient * material.specular * light.color.rgb;
99
100     // Return final color
101     return ambient + attenuation * (diffuse + specular);
102 }
103
104
105 /*!
106  * Fragment shader main function.
107  */
108 void main(void) {
109     // Obtain fragment cololor from texture
110     vec4 diffuseFragment = texture(textures.diffuse, properties.f_uv);
111
112     // Discard pixel that have opacity from depth buffer
113     if (diffuseFragment.a <= 0.1) {
114         discard;
115     }
116
117     // Normal without normal mapping
118     vec3 normal = properties.f_normal;
119
120     // Obtain normal from normal map in range [0, 1]
121     vec3 normalFragment = texture(textures.normal, properties.f_uv).rgb;
122     if (normalFragment != vec3(0.0f)) {
123         // Transform normal vector to range [-1, 1]
124         normalFragment = normalize(normalFragment * 2.0 - 1.0);
125
126         // Get normal from normal map
127         normal = normalFragment;
128     }
129
130     // Camera in tangent space
131     vec3 cameraPos = properties.tangentScreenPosition.xyz;
132
133     // set gamma correction to texture
134     float gamma = 2.2;
135     diffuseFragment = pow(diffuseFragment, vec4(vec3(1.0 / gamma), 1.0));
136
137     // Calculate tangent space surface to camera vector
138     vec3 surfaceToCamera = cameraPos - properties.f_vertex;
139
140     // Normalize camera to surface vector
141     surfaceToCamera = normalize(surfaceToCamera);
142
143     // No texture use material diffuse color
```

```
144       if (diffuseFragment.rgb == vec3(0.0f)) {
145           diffuseFragment = vec4(material.diffuse, material.opacity);
146       }
147
148       // Set empty color black
149       vec3 color = vec3(0);
150
151       // Calculate single point light
152       color += BlinnPhong(
153           normal,
154           diffuseFragment,
155           surfaceToCamera,
156           properties.f_vertex
157       );
158
159       // Final color
160       outColor = vec4(color, diffuseFragment.a * material.opacity);
161   }
```