Oct 2013

Project Book

AR on QR

Augmented reality above QR Codes

Aviya Cherevatsky (Levy)

Supervisors:
Prof. Ben-Chen Mirela
Mr. Elster Constantine
Mr. Vardi Dan

Introduction

Overview

In this project an android application was developed. The application displays augmented reality advertisements on top of visual markers. The application detects visual markers on every frame, calculates the markers transformations and displays a related augmented reality commercial on the frame.
AR on QR is a camera application which detects, decodes, and augments QR Codes. The QR Code content determines the AR model to be displayed. The application is able to recognize and decode the QR Code under various transformations, and able to deal with acute angles of the view point. As far as we know there is no commercial barcode scanner which is able to recognize and decode QR Codes under acute projective transformations.

Figure 1: The application screenshots

Background

What Is a QR Code?

QR code (abbreviated from Quick Response Code) is the trademark for a type of matrix barcode. Applications include product tracking, item identification, time tracking, document management, general marketing, and much more. A QR code consists of finder pattern, timing pattern, alignment pattern, format information, coded data and quite zone, which are key components/elements of a Quick Response code.

Each component has its own role, the finder pattern is a block with central black square, with white and black alternate Square outside of it (concentric squares of alternate colors). There are such three blocks available in each corner except bottom right. This finder pattern tells the scanner/reader that image which is being decoded/scanned is QR code. Finder pattern does not contain any data.

Transformation representation

Rigid transformation of a vector $x \in \mathbb{R}^N$ consists of a rotation and translation. The mathematical form of transformation of vector x, is $R \cdot x + T$ where R is the rotation Matrix ($R \in \mathbb{R}^N \times \mathbb{R}^N$) and T is the translation vector ($T \in \mathbb{R}^N$).

We can use a homogeneous coordinates representation (add 1 as last coordinate and get $x' \in R^{N+1}$). In this representation, transformations can be represented by a single matrix. In this form, concatenating of transformation is a simply multiplication of matrix.

$$\begin{bmatrix} \begin{pmatrix} R11 & R12 & R13 \\ R21 & R22 & R23 \\ R31 & R32 & R33 \end{pmatrix} & \begin{matrix} t1 \\ t2 \\ t3 \end{matrix} \\ \begin{matrix} 0 & 0 & 0 \end{matrix} & 1 \end{bmatrix} \bullet \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} R11x + R12y + R13z + t1 \\ R21x + R22y + R23z + t2 \\ R21x + R32y + R33z + t3 \\ 1 \end{bmatrix} \doteq \begin{pmatrix} R11 & R12 & R13 \\ R21 & R22 & R23 \\ R31 & R32 & R33 \end{pmatrix} \bullet \begin{pmatrix} x \\ y \\ z \end{pmatrix} + \begin{pmatrix} t1 \\ t2 \\ t3 \end{pmatrix}$$

Figure 2: representation of a rigid transformation in R3 by a single matrix.

ARToolKit

ARToolKit is a multiplatform software library for building Augmented Reality (AR) applications (open source). The ARToolKit video tracking libraries calculates the real camera position and orientation relative to physical markers in real time. The application is able to detect a predefined set of markers, and each marker has to be defines in advance as an input in form of image file. Several restrictions hold for the marker to be valid. The marker must have a bold black frame and any simple black icon inside. The user can create markers using a free On-Line software.

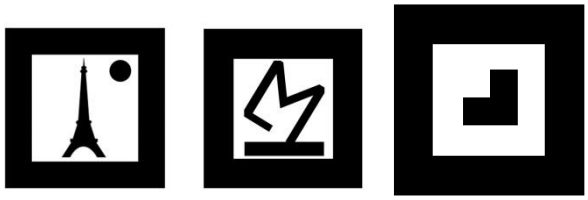


Figure 3: ARToolkit Markers

It is possible to recover the camera coordinate system transformation relative to each marker in the frame with ARToolKit.
The frame and the AR model rendering process is made by OpenGL. There is a correlation between ARToolKit coordinate systems and OpenGL Coordinate systems: The OpenGL world coordinate system is the marker coordinate system and the OpenGL View coordinates system is set to the real camera coordinate system by the camera transformation. In this setup, rendering the virtual model is very comfortable- the model is simply rendered on the world coordinate system origin, and the OpenGL view coordinate systems changes according the camera transformation.

ARTooKit camera and marker coordinate systems

The camera coordinate systems:
The X axis heading right, Y axis heading down and Z axis heading forward
The marker coordinate systems:
The X axis heading right, Y axis heading forward and Z axis heading up. The marker is located on XY plane. Both coordinate systems are right handed.

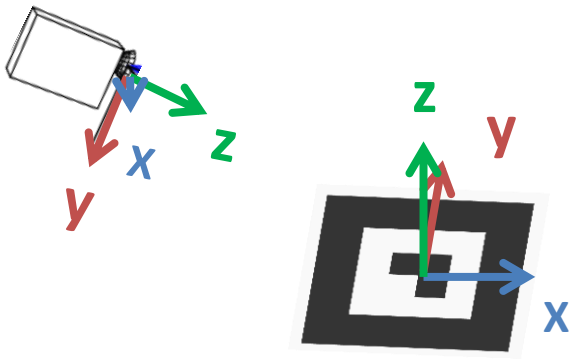


Figure 4: The camera and ARToolKit marker coordinate systems

Using ARTooKit for QR Code detection, and extracting camera transformation.

ARToolkit markers frame and finder pattern looks quite similar. By creating a marker of finder pattern we can detect a finder pattern by ARToolkit. ARToolKit calculates the camera transformation relative to each finder pattern; our goal is to calculate the camera transformation, relative to the QR Code, out of the three camera transformations.
There are two challenges in this approach:

1. How to extract the correct rotation and translation
   out of the 3 transformations.
2. The finder pattern marker is symmetrical therefor there are 4 possible solutions for the rotation matrix, ARToolKit selects arbitrary one of the solution.

QR Code coordinate systems

The QR Code coordinate system defined similar to the Marker coordinate system. The X axis heading right, Y axis heading forward and Z axis heading down (right handed coordinate system)

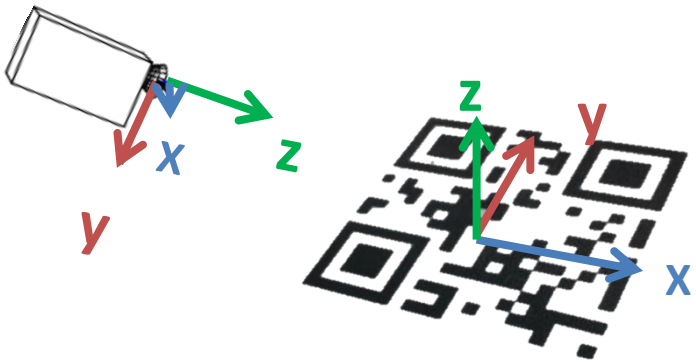
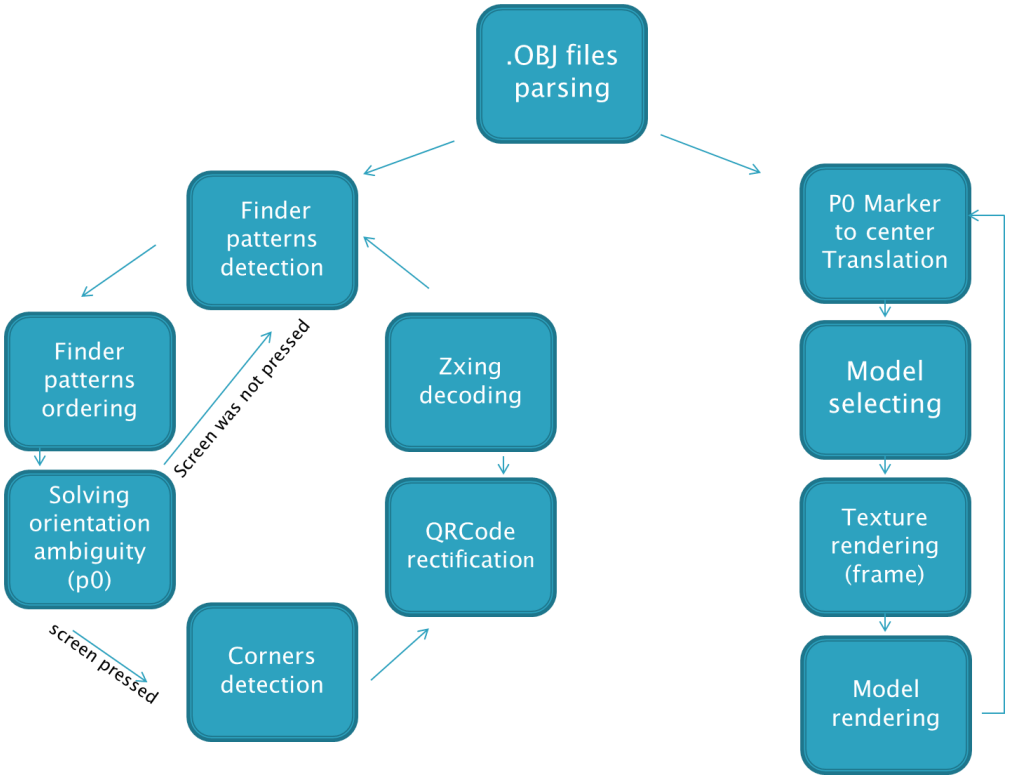


Figure 5: Camera and QR Code coordinate systems

The application state machine



The first step is initialization, loading and parsing the models files.
After all initializations, the application activates two synchronous threads, the first thread is responsible for the rendering process, and the second is responsible for the application logic. The two threads work independently. The thread which is

responsible for the logic calculates the transformations and the decoded string out of the QR Code and passes this information to the other thread.

APIs and Tools

Android-
Android is an operating system based on the Linux kernel, and designed primarily for touchscreen mobile devices such as smartphones and tablet computers.
Applications are developed in the Java language using the Android software development kit (SDK). The SDK includes a comprehensive set of development tools, including a debugger, software libraries, a handset emulator, documentation, sample code, and tutorials.
Many devices, such as smart-phones and tablets are supported by the android OS.

ARToolKit for android (AndAR)-
ARToolKit is a C library which is ported to android using the JNI framework.

Android NDK-
The android NDK contains a set of tools and build files used to generate native code libraries from C and C++ sources. A recompilation of ARToolKit C code needed due to several changes in ARToolKit logic.

ARToolkit model viewer project –
An open source project, which based on AndAR and has an additional feature - dynamic loading of models. The source contains an .Obj file parser, the parser was integrated into the ARonQR application to support .Obj file rendering.

ZXing-
ZXing is an open-source, multi-format 1D/2D barcode image processing library implemented in Java, which is ported to android. The library can be used to encode and decode barcodes.

OpenCV
OpenCV is a library which contains many computer vision algorithms, implemented in c++ and has a java interface. The library is used in the project for QR Code rectification.

OpenGL
OpenGL (Open Graphics Library) is a cross-language, multi-platform application programming interface (API) for rendering 2D and 3D computer graphics.
We use OpenGL to render the frame and the 3D model on screen.

Logic and algorithms

Finder pattern detection
We use ARToolKit for detection of the QR Code three finder patterns.

For each finder patterns ARToolKit estimates the rotation matrix and translation vector of the camera relative to the finder pattern.
Our task is to find the rotation and translation vector of the QR Code as one entity.
There are two problems with the original algorithm of ARToolKit:

1. In case there are several identical markers in a frame the original detection detects jest one of them. And therefore calculate the transformation only to one marker.

2. In case we have a squared shape marker, it is invariant under rotation of $\frac{\pi}{2}$ multiplications. Therefore there are four possible solutions for the rotation matrix. In that case the original implementation of ARToolKit will calculate one of the four possible solutions `arbitrary`.

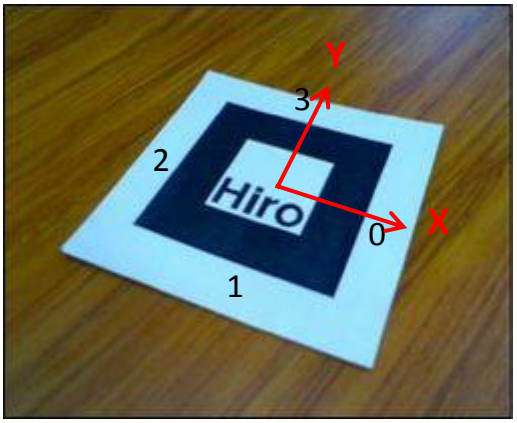
The first problem is easy to solve. The marker that ARToolKit retrieve as an output is the marker that fits best to the original pattern. The change is to retrieve the three best fits.
The solution of the second problem will be displayed at the next section.

Solving the orientation ambiguity

For each marker, a sequence of two phases is performed.
In the first phase the application detects the marker and creates a structure that describes the marker. In the second phase, the application estimates the transformations of each detected marker.
The first phase output is a structure that contains some properties of the detected markers such as:

1. The marker center location
2. The four line equations of the markers border
3. The four border corners
4. The index of the line at the lines array, which crosses the Y axis on its positive side.

The fourth parameter determines the marker rotation that will be calculated in the next phase. For example, in case of the following frame and line indexes, the index of the line which crosses the Y axis on its positive side is 3.



For finder pattern there is no ability to find the upper line without further information due to symmetry, thus this parameter is set arbitrary and ruins the next phase of transformation calculation.

Luckily, when searching for QR Code there are two additional markers, and by combining the information from the three of them we can detect the line that crosses the Y axis on its positive side, and fix the fourth parameter.
From now on we will denote the upper right finder pattern - p1, the upper left finder pattern - p0 and the bottom left finder pattern - p2.

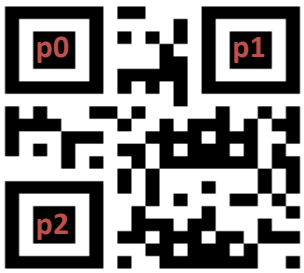


Figure 6: Labeling the QR Code finder patterns

Solving marker p0 ambiguity

At this stage we will find the correct orientation of marker p0. There is no need to correct p1 and p2 orientation as will be explained later.
The first step is to recognize each detected finder pattern as p0, p1 and p2.
Then, after recognizing the finder patterns, the fourth parameter of p0 will be set to the most distanced line from p2 center.

Recognizing the finder patterns

It is possible to see that for each border line of p0 there is an identical line on p1 or p2 border. The top and bottom lines of p0 contained also in p1 border, and the right and left lines of p0 contained also in p2 border.
For each line on each marker we will calculate the closest line on every other marker. The marker with the minimum distances sum will be p0.
The distance between two lines computed by the following weighted Euclidean Norm:

$$\|Li - Lj\| = \sqrt{((Li_a - Lj_a)\cdot 200)^2 + ((Li_b - Lj_b)\cdot 200)^2 + (Li_c - Lj_c)^2}$$

Factor of 100 is given to the first two coordinates to make all numbers at the same order of magnitude. And additional factor of 2 is for giving a multiple weight to the gradient element.

Finding the QR Code transformation matrix

The information we are holding up to this point is the three finder patterns transformations, where markers p1 and p2 may hold an incorrect rotation matrix.

On a rigid body transformation, every point has the same rotation while translation is different. The QR Code is a rigid body therefor p0, p1, p2 and the QR Code center has the same rotation matrix. An obvious outcome is that the three coordinates systems of the three finder patterns are related by a pure X/Y translation. The QR Code center location will be the average of p1 and p2 location.
By combining it all together we can build the QR Code transformation matrix:

$$Trans(p0)=\begin{bmatrix}\begin{pmatrix}&&\\&R0&\\&&\end{pmatrix}&\begin{matrix}0\\0\\0\end{matrix}\\\begin{pmatrix}&T0&\end{pmatrix}&1\end{bmatrix},Trans(p1)=\begin{bmatrix}\begin{pmatrix}&&\\&R1&\\&&\end{pmatrix}&\begin{matrix}0\\0\\0\end{matrix}\\\begin{pmatrix}&T1&\end{pmatrix}&1\end{bmatrix},Trans(p2)=\begin{bmatrix}\begin{pmatrix}&&\\&R2&\\&&\end{pmatrix}&\begin{matrix}0\\0\\0\end{matrix}\\\begin{pmatrix}&T2&\end{pmatrix}&1\end{bmatrix}$$

$$Trans(QRCode)=\begin{bmatrix}\begin{pmatrix}&&\\&R0&\\&&\end{pmatrix}&\begin{matrix}0\\0\\0\end{matrix}\\\begin{pmatrix}\frac{T1+T2}{2}\end{pmatrix}&1\end{bmatrix}$$

Computing Euler angles from a rotation matrix

We start off with the standard definition of the rotations about the three principle axes:

$$R_x(\psi) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\psi & -\sin\psi \\ 0 & \sin\psi & \cos\psi \end{bmatrix}$$

$$R_y(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix}$$

$$R_z(\phi) = \begin{bmatrix} \cos\phi & -\sin\phi & 0 \\ \sin\phi & \cos\phi & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

The rotation matrix can be thought of a sequence of three rotations, one about each principle axis. Since matrix multiplication does not commute, the order of the axes which one rotates about will affect the result.
For this analysis, we will rotate first about the x-axis, then the y-axis, and finally the z-axis:

$$\begin{bmatrix} \cos\theta\cos\phi & \sin\psi\sin\theta\cos\phi - \cos\psi\sin\phi & \cos\psi\sin\theta\cos\phi + \sin\psi\sin\phi \\ \cos\theta\sin\phi & \sin\psi\sin\theta\sin\phi + \cos\psi\cos\phi & \cos\psi\sin\theta\sin\phi - \sin\psi\cos\phi \\ -\sin\theta & \sin\psi\cos\theta & \cos\psi\cos\theta \end{bmatrix}$$

The angles ψ,ϑ, and φ are the Euler angles.
Starting with R31, we find that:

$$\begin{aligned} \theta_1 &= -\sin^{-1}(R_{31}) \\ \theta_2 &= \pi - \theta_1 = \pi + \sin^{-1}(R_{31}) \end{aligned}$$

To find the values for ψ, we observe that

$$\frac{R_{32}}{R_{33}} = \tan(\psi) \quad \Longrightarrow \psi = \operatorname{atan2}(R_{32}, R_{33}),$$

A similar analysis holds for φ

$$\frac{R_{21}}{R_{11}} = \tan\phi.$$

Note: when $|\sin\theta| = 1, (\cos(\theta) = 0)$ there is an alternate solution.

QR Code rectification

The motivation to rectify the QR Code is to make ZXing to be able to decode the QR Code.
The decoding algorithm of ZXing works only if the QR Code is rectified, and can't work when the QR Code is rotated at some arbitrary angle.
In order to rectify the QR Code, a projective transformation has to be recovered from the QR Code plane to an imaginary plane which is parallel to the camera plane.
This transformation is also called Homography.

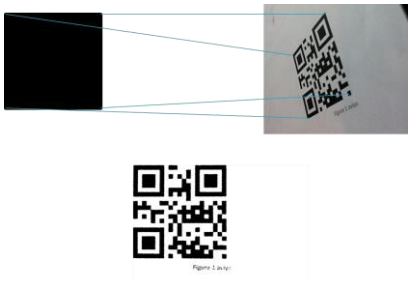


Figure 7: Using homography for QR Code rectification.
In the right upper corner-The original QR Code. In the bottom – the rectification result.

At least four corresponding points required for homography computation. These four points in our case are the corners of the QR Code border. AR Toolkit is able to give us the three finder pattern centers, and we use them to estimate the coordinates of the four corners. This part of the algorithm is explained in submission of the first part of the project in details.

OpenGL ES2.0 rendering

OpenGL ES2.0 is an API for full-function 2D and 3D graphics on Embedded Systems. It is a subset of desktop OpenGL. OpenGL ES2 has a programmable pipeline which gives the ability of writing vertex shaders and fragments shaders. Beginning with Android 2.2 (API Level 8), the framework supports the OpenGL ES 2.0 API.

OpenGL ES1X VS. OpenGL ES20

- Performance -
  In general, OpenGL ES 2.0 provides faster graphics performance than the ES 1.0/1.1 APIs.
- Coding Convenience -
  The OpenGL ES 1.0/1.1 API provides a fixed function pipeline and convenience functions which are not available in the ES 2.0 API. Developers who are new to OpenGL may find coding for OpenGL ES 1.0/1.1 faster and more convenient.(fog, matrixes)
- Graphics Control -
  The OpenGL ES 2.0 API provides a higher degree of control by providing a fully programmable pipeline through the use of shaders. With more direct control of the graphics processing pipeline, developers can create effects that would be very difficult to generate using the 1.0/1.1 API.

Although ARToolKit contain an OpenGL ES1 renderer and it could have been easier to implement the same simple renderer, we choose to implement an OpenGL ES2 renderer due to performance and curiosity

.OBJ file parsing

We use .OBJ files for 3D model rendering. The OBJ file format is a simple data-format that represents 3D geometry alone — namely, the position of each vertex, the UV position of each texture coordinate vertex, normals, and the faces that make each polygon defined as a list of vertices, and texture vertices.
Naïve OBJ files parser written in the "AndARModelViewer" project (support only triangle mashes), the parser was integrated into the application to support 3D object rendering.
After parsing the OBJ file the model runtime representation contains list of groups, each group contains vertices list, normal list, and material properties.
This information is sent to the Vertex Shader.

The Vertex and Fragment Shaders

The vertex shader inputs are: Vertex, Model view projection matrix, Vertex Normal, Material properties, Lights properties, Texture Coordinates, and the rendering mode (3D or 2D – for rendering the 2D frame). The output parameters are vertex position in clip coordinates, and some output to the fragment shader such as Vertex color (after calculating the vertex color with light). The output of the fragment shader is the fragment color.

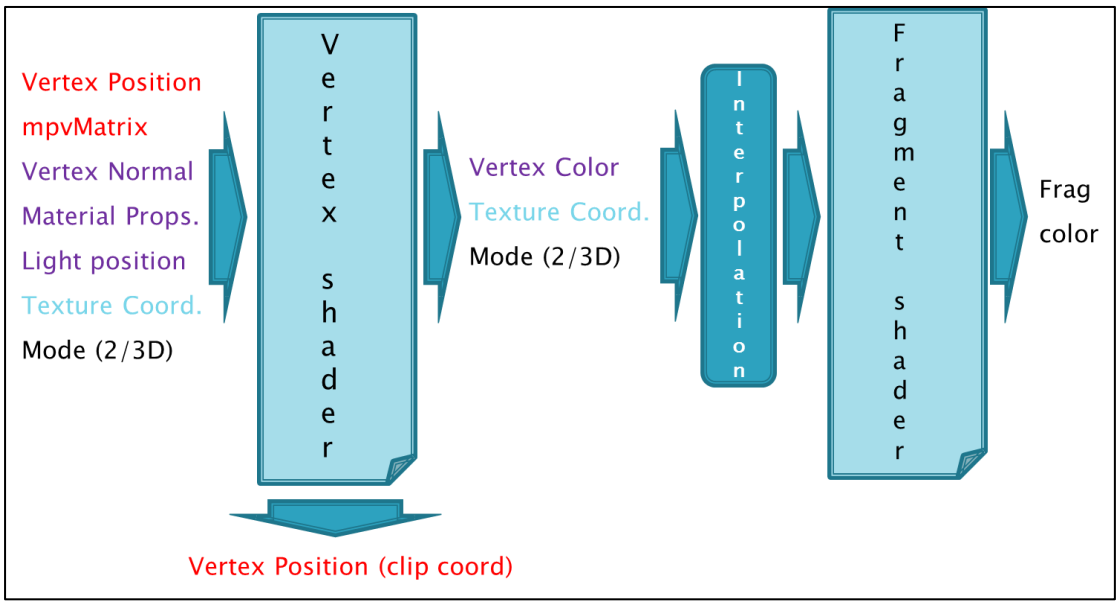


Figure 8: Shaders inputs and outputs

The vertex shader is responsible for giving the vertex color, lighting ,position in clip coordinates, and the texture matching coordinates. The color and the texture coordinates interpolated values are the input to the fragment shader. And then the

fragment shader is responsible for mixing the color and the texture RGB into one single color.
In the current implementation we have separated the implementation into two modes:

2D mode – in this mode the 2D video frame is being rendered.
3D mode – in this mode the 3D model is rendered.

In 2D mode we draw the video frame as is without lighting influence.
In the vertex shader we calculate the texture coordinates only and pass it to the fragment shader. At the fragment shader the fragment color is set to the texture RGB at the interpolated coordinates.

In 3D mode we want to draw the 3D model therefore the light influence is very important. At the vertex shader we calculate the vertex color according to the vertex normal, light position and material properties. The vertex interpolated fragment color passes into the fragment shader and the fragment color is set to this color.

Lighting

Let's begin with the basic light Properties:

Ambient light – It seems to come from all directions. When ambient light strikes a surface, it's scattered equally in all directions.
Actually, it's the color of an object when there is no direct light.
Diffuse light - Is the light that comes from one direction. Once it hits a surface, however, it's scattered equally in all directions.
Specular light - comes from a particular direction, and it tends to bounce off the surface in a preferred direction.

Material properties are the color reflection when the ambient, diffuse, specular lights hit the model surface.

Lighting formulas

Let $L = (L_x.L_y.L_z)$ be the unit vector that points from the vertex to the light position.
Let $n = (n_x.n_y.n_z)$ be the unit vertex normal.
Let $LightDiff = (LightDiff_r.LightDiff_g.LightDiff_b)$ be the Light diffuse color.
Let $ObjDiff = (ObjDiff_r.ObjDiff_g.ObjDiff_b)$ be the object color when diffuse light hit its surface.

Then the diffuse Lighting formula is

$$\max(0, dot(L, n)) \cdot LightDiff \cdot ObjectDiff$$

The Diffuse and ambient color formula is

$$\max(0, dot(L, n)) \cdot LightDiff \cdot ObjectDiff + LightAmb \cdot ObjectAmb$$

In our implementation we assume that all light materials are white ($(1,1,1)$), and there is no specular color component.