**Blind Engine**

## Niko Storni, Marko Pacak & Jorge Estevez

****

****

****

**Index**

**Abstract …………………………………………………………………………………………………………………………………………**

Concerning Blind Engine …………………………………………………………………………………

Development & Libraries …………………………………………………………………………………

**Project Architecture …………………………………………………………………………………………………………**

Solution Structure ………………………………………………………………………………………………

Class Diagram ……………………………………………………………………………………………………………

Working Principles ………………………………………………………………………………………………

**Classes Overview** ……………………………………………………………………………………………………………………

Engine Classes …………………………………………………………………………………………………………

BEengine …………………………………………………………………………………………………………

BElist ………………………………………………………………………………………………………………

BEsceneloader ……………………………………………………………………………………………

Scene-graph Classes ……………………………………………………………………………………………

BEobject …………………………………………………………………………………………………………

BEtexture ………………………………………………………………………………………………………

BEmaterial ……………………………………………………………………………………………………

BEnode ………………………………………………………………………………………………………………

BElight ……………………………………………………………………………………………………………

BEmesh ………………………………………………………………………………………………………………

BEcamera …………………………………………………………………………………………………………

Rubik Cube Classes ………………………………………………………………………………………………

Rubik …………………………………………………………………………………………………………………

Main ……………………………………………………………………………………………………………………

**Issues ………………………………………………………………………………………………………………………………………………**

Transparency ………………………………………………………………………………………………………………

Multi Mesh ……………………………………………………………………………………………………………………

Cube Rotation ……………………………………………………………………………………………………………

**Abstract**

**Concerning Blind Engine**

Blind Engine is a graphic engine written in C++ that throught the cross-platform vector-graphics API widely known as OpenGL ® allows the upload of graphical scenes in DAE format.

Blind Engine was written by students Niko Storni, Marko Pacak and Jorge Esteves as final project for the course Computer Graphics.

**Development & Libraries**

The project was entirely written using Microsoft ® Visual Studio and ultimately Code::Blocks. Blind Engine functions only thanks to the C++ and C standard libraries and as such can be run both on Microsoft ® Windows and Linux OS’s.

Blind Engine works specifically thanks to the functions offered by OpenGL ®.

Moreover the following additional libraries were used:

* FreeGLUT
* GLM
* Assimp
* FreeImage

**Project Architecture**

**Solution Structure**

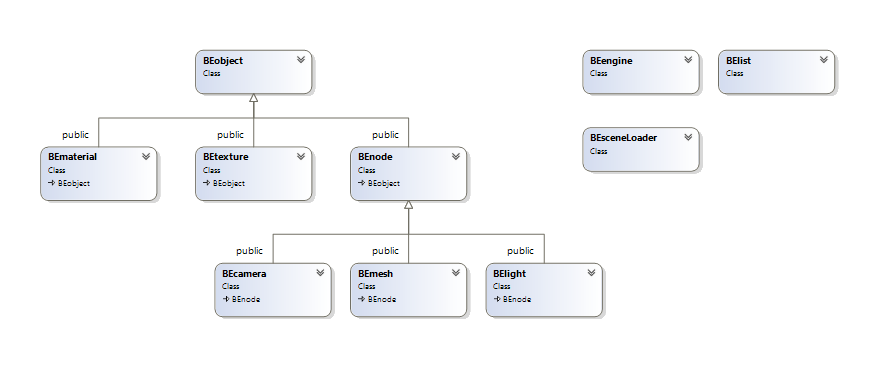
Blind Engine is divided into two sub-projects.

The Blind Engine part is exported as DLL (dynamic link library) and SO, ergo as library files containing functions and data accessible by third party programs.

The Rubik Cube executable part is a demonstration-program meant to illustrate the functionalities of the engine.

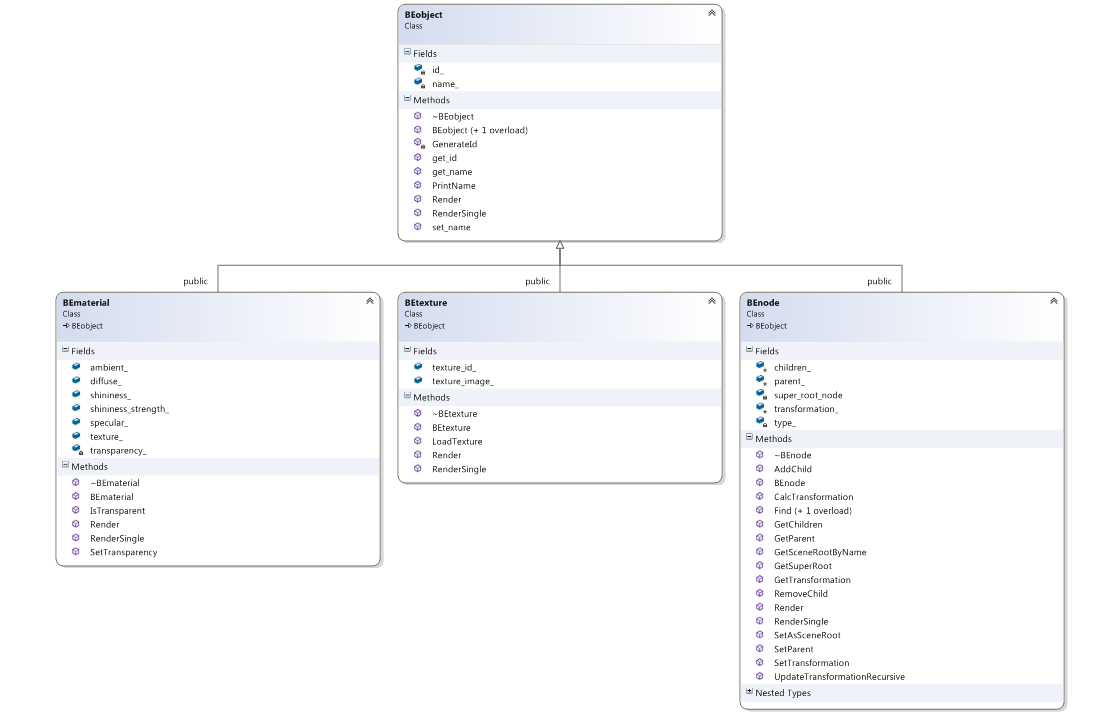
**Class Diagram**

The following diagram briefly illustrates all classes present in the project.



The abstract class Object operates as base class for all other classes present in the Blind Engine world. It implements the pure virtual method RenderSingle(glm::mat4) which will allow all heirs to draw themselves. Inherit from Object the classes Material, Texture and Node. Particularly Node acts as base class for the physical objects Camera, Light and Mesh.

Besides overriding RenderSingle, Node offers all methods for the calculation, positioning and drawing of any single physical object. We will analyze such methods in a while.



**Working Principles**

As the program is launched, Rubik’s main instanciates and initializes a BEengine object which takes care of uploading the file “scene\_final.DAE”.

The object Rubik Cube is then created thrught our engine method calls before finally starting the engine which will allow our program to enter freeGLUT’s main loop.

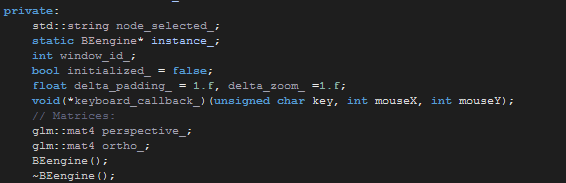
Keyboard Callbacks are dicretly implemented in the main file.

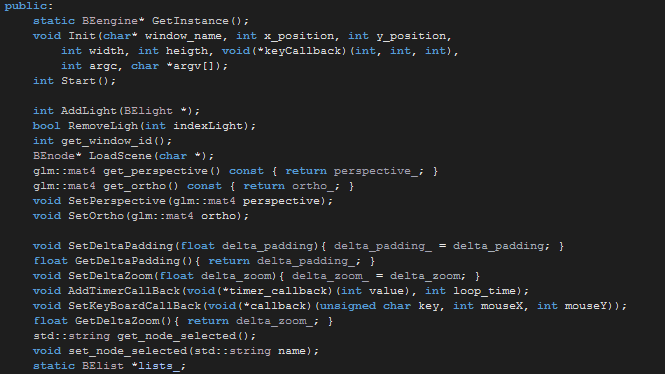
**Classes**

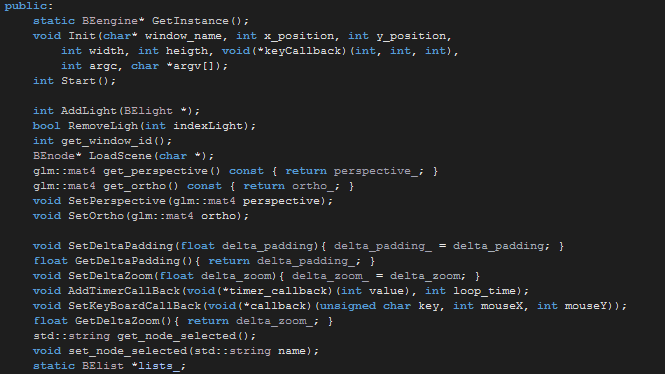
**Engine Classes**

**BEengine**

BEengine is the entry class of our application. Throught BEengine we are able to upload a COLLADA file (.dae extension), anaylize the scene graph and call all relative methods for rendering our scene. Let’s analyze our singleton class by starting with the private fields and methods.



Besides the private constructor and destructor (as usually required by a singleton class) BEengine holds the ID of the window, the pointer to the only BEengine instance, some other fields and utility and the two main matrices.



Within the public fields and methods, we can notice:

* getters and setters
* methods for initalizing and starting the engine
* a static pointer to a BElist (discussed afterwards)
* the LoadScene method

Noteworthy is cleary the method LoadScene. What does it do? It simply istanciates a BEsceneLoader variable and throught it it loades a scene.

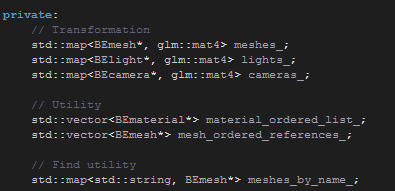
Let’s remind ourselves that BEengine will be instanciated in any program using our engine and that be\_engine->loadScene(“scene.dae”) will be called in order to launch the program correctly.

At the end BEengine is nothing more than an interface between the launching program and all classes doing the hard work in the background.

**BElist**

BElist acts as containter for all objects present in our scene graph: those rendered and those yet to be rendered. As a matter of fact, BElist is a copy of the scene graph, with the difference that there is no hierarchical order for the nodes contained in it. Moreover, whereas nodes’ matrices in the actual scene-graph hold coordinates relative to the father node, here in the BElist nodes hold world coordinates.

Below we can see the collections holding all objects.

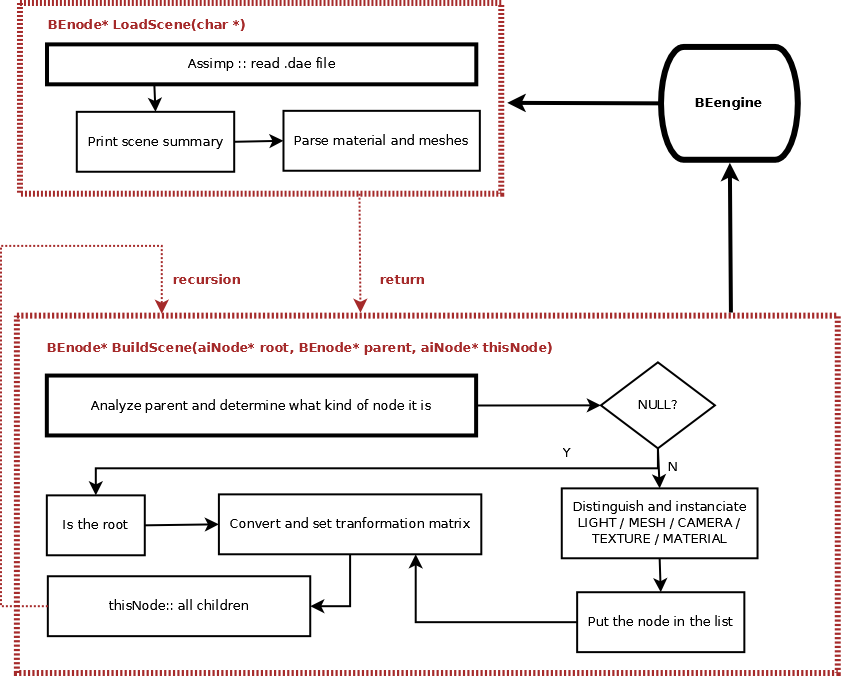


As scene-graph copy, BElist implements all rendering methods for each kind of object.

**BEsceneLoader**

BEsceneLoader holds all utility for opening a .dae file, directly parse any object and afterwards build the scene.

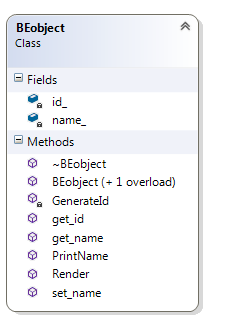
There is a lot we could say about this class, but in order to speed up the understanding of our engine let’s swiftly analyze the class through the following diagram, which sums up the functioning of LoadScene, BuildScene and all related methods.

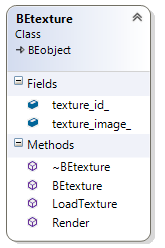


**Scene-graph Classes**

This section briefly illustrates all classes meant to be graphically rendered in our scene. If details are needed to be better understood, this section will give all necessary explanation. You will notice that each class illustrates its own Render(glm::mat4) method.

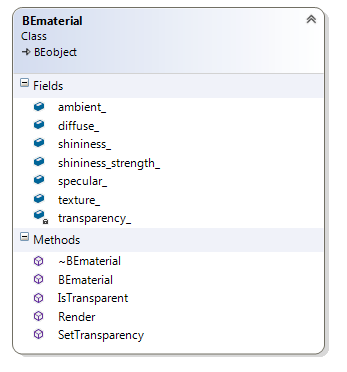
**BEobject**

As mentioned in the previous chapter BEobject act as base class for all objects present in a scene graph. It implements the pure virtual method Render(glm::mat4) that will allow all objects to “position” themselves in the world.  
Moreover, it stores basic information for distinguishing each single object.

**BEtexture**

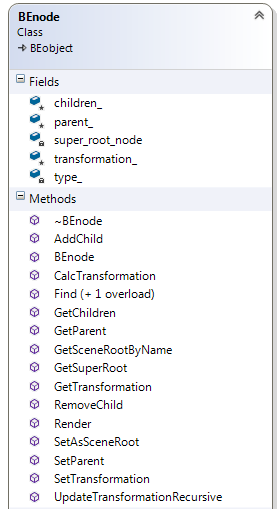
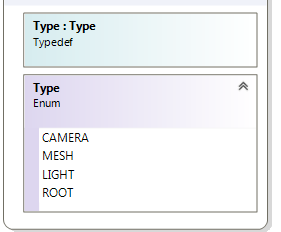
BEtexture holds storage and logic for rendering images as texture. As instanciated, it uploades an image into the field FIBITMAP\* texture\_image throught FreeImage utilities.

The Render(glm::mat4) method updates and sets the texture throught OpenGL ® calls.

**BEmaterial**

This class holds all material’s parameters such as the matrices ambient, diffuse, specular and more essentialy it holds a pointer to a texture.

The Render(glm::mat4) method checks whether a texture is needed to be rendered and then renders the material based on the proprieties throught OpenGL ® calls.



**BEnode**

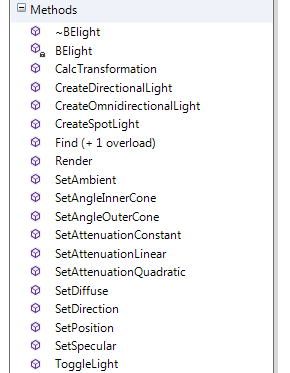
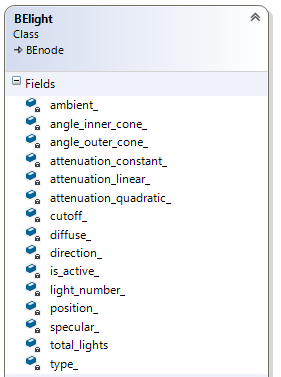
BEnode is the base for all physical objects in our graphical universe.

Each node holds a pointer to the parent as well as vector containing pointers to the children.

This class introduces some new utility functions useful for updating node’s data.

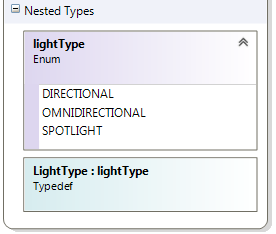
The Render(glm::mat4) method is overridden yet left emtpy as it’s not meant to be used here.

Noteworthy are the functions relative to the transformation matrix. For example UpdateTransformationRecursive(glm::mat4) updates the current node’s matrix and does the same for all children.

**BElight**

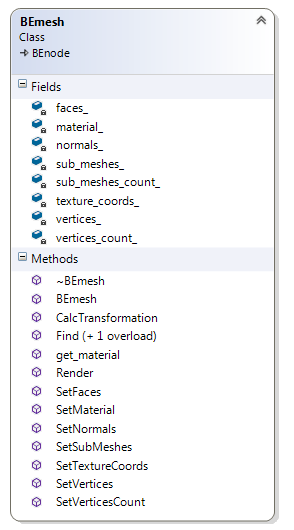
A light may be DIRECTIONAL, OMNIDIRECTIONAL or SPOTLIGHT and such file is defined by an enum. The class contains all parameters necessary to define a light by the standards of OpenGL ®.

The instantiation of a single light happens throught a global method which then calls one of the three factory methods.

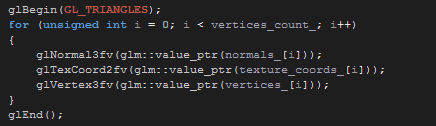
Before calling the specific method, the number of total lights is increased, the light is activated throught OpenGL ® and the paramters are set.

The Render(glm::mat4) method updates some common parameters (AMBIENT, DIFFUSE & SPECULAR), then depending on the light type others parameters such as the cutoff are updated.

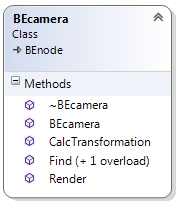
**BEmesh**

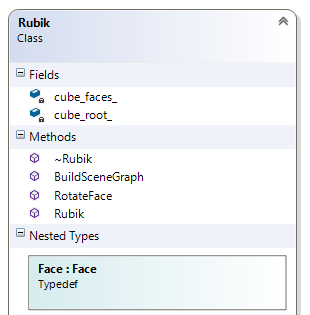
A mesh (also knows as Object or Model or Primitive) holds storage for different types of coordinates useful for object modeling such as vertices, normals, texture coordinates…

A mesh implements a material as well as an array of possible sub-meshes.

The Render(glm::mat4) method loads the matrix, renders a material (if present) and then draws the triangles in the OpenGL ® way.

It then renders any sub-mesh if present.

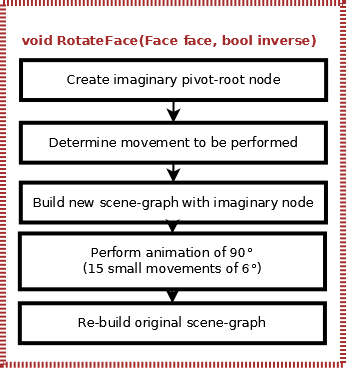
**BEcamera**

**Rubik Cube Classes**

**Rubik**

Rubik’s class simply holds an enum as reference to the faces as well as a matrix of BEnode’s (cube’s faces).

As we instanciate the cube throught the cube root, all faces are instanciated by calling the Find(std::string) method.

Noteworthy are the RotateFace(Face, bool) method which takes care of rotating a certain cube’s face and the Animation(int) method which explicitly generates the animation while the cube is rotating.

The diagram on the right side explains what RotateFace(Face, bool) does.

**Main**

Main simply acts as entry point for the Rubik’s Cube Application. It implements the callbacks for the keyboard.

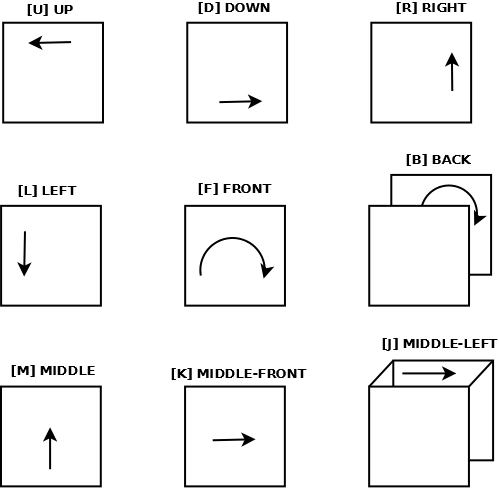
**Issues**

**Transparency**

**Multi-Mesh**

**Cube Rotation**

The rotation was implemented following a counterwise-logic where each movement holds an inverse-part.

As found difficult to implement a standard function for all kind of movements we hardcoded it into different