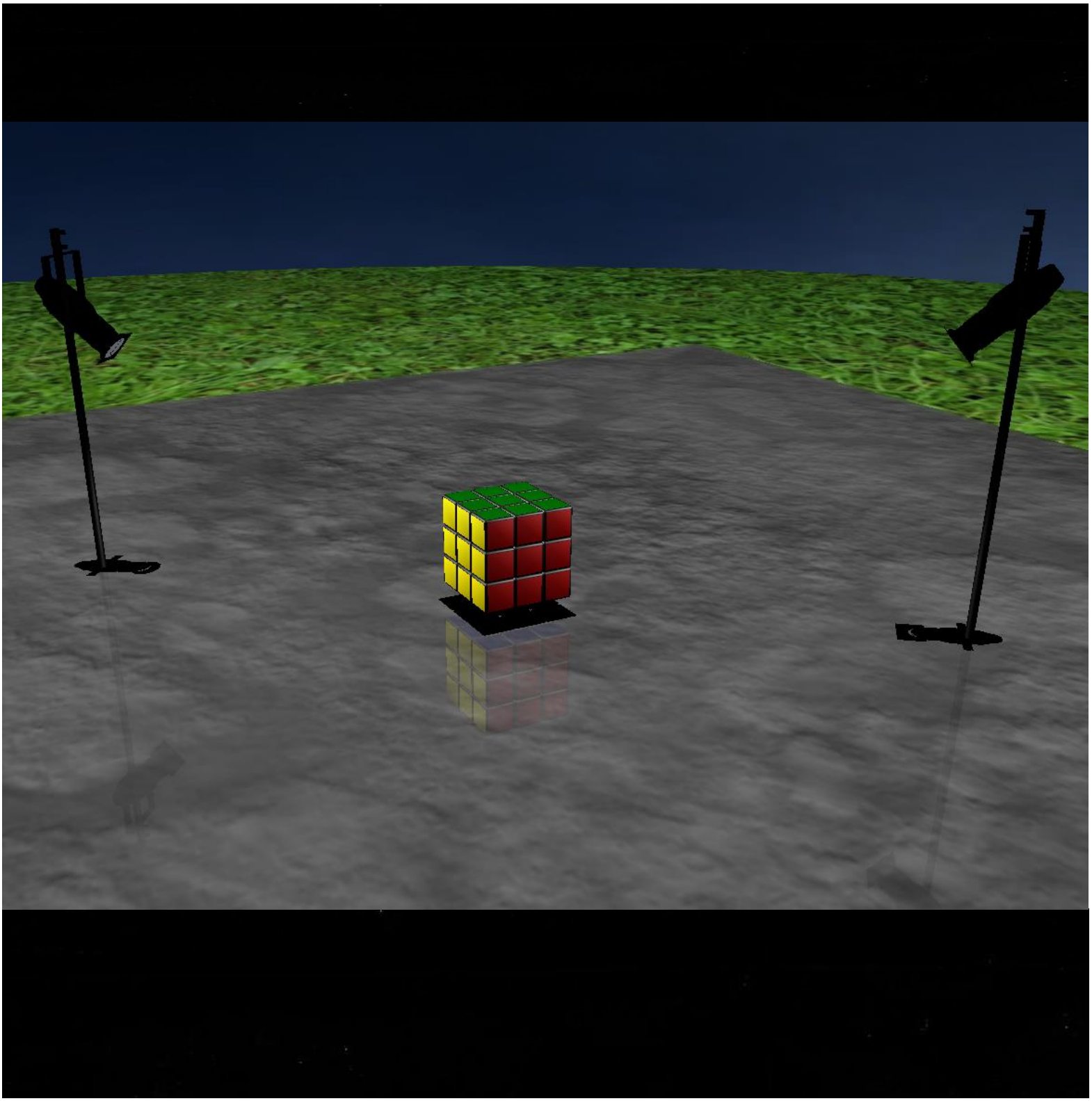
Scuola universitaria professionale
della Svizzera italiana
SUPSI

Blind Engine

Jorge Esteves, Niko Storni & Marko Pacak



Index

Abstract

Concerning BlindEngine

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

The objective is to create an interface between OpenGL and the user.

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. Our graphic engine works across multiple platforms thanks to the C++ and C standard libraries on which we relied. OpenGL API was used to accomplish the rendering in 2D and 3D.

Additionally, the following libraries were used:

- FreeGLUT (Allows for a better interfacing with OpenGL)
- GLM (Several math tools and operations provided)
- Assimp (A parser that allows us to quickly import a scene from a file)
- FreeImage (A tool used for quick and reliable managing of images)

Project Architecture

Solution Structure

BlindEngine is divided into two sub-projects:

1. The engine library itself
2. The demo that implements the library

The first one is not executable alone, it's in fact a library (either .dll or .so) and it can be interfaced with projects such as our demo.

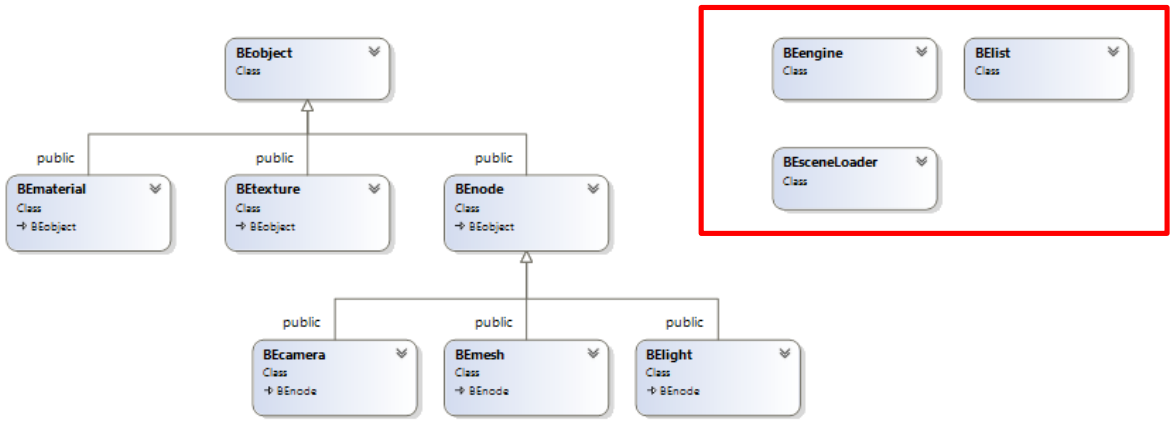
The demo is the implementation of a simple game (the Rubik's Cube) where the user is able to interact and solve a Rubik's Cube loaded through the engine library. The scene was previously compiled in 3D Studio Max and exported.

The engine itself doesn't have any intelligence over how the cube is formed or how the faces are rotated, however it gives the necessary tools for the developers to accomplish such tasks.

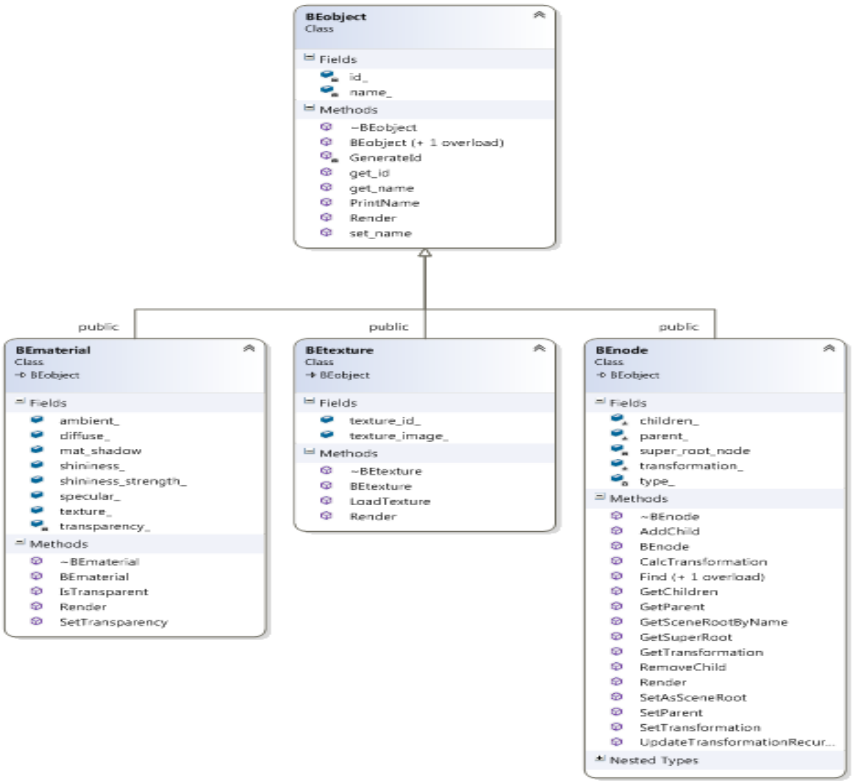
Classes

Class Diagram

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



The abstract class Object operates as base class for all the other classes responsible for holding the data and the structure of the scene. It implements the pure virtual method Render() inherited by all subclasses which will force them to implement their own render method. The node class acts as a generic node in the scene graph. Most of the methods implemented in BEnode are then overridden by the subclasses.



Engine Classes

BEengine

BEengine is the entry class of our application. Through BEengine we are able to load a COLLADA file (.DAE extension), analyze the scene graph and call all relative methods for rendering our scene.

This class is constructed using the singleton pattern, this means that there can ever only be one instance of BEengine.

Among all the methods it's important to note the following ones:

- Init(): Initializes the context of FreeGLUT, FreeImage and OpenGL.
- LoadScene(): Calls the loader and passes the path given by the user to load a scene from a previously exported model.
- Start(): Calculates the world coordinates of each element to render and passes over the control to FreeGLUT by entering the main loop.

BEengine acts as an interface to programmers for interacting with the implementation of the whole graphic engine.

BElist

This class contains multiple lists of objects that must be rendered.

Each object, either mesh, camera or light, is also paired with their own world coordinates where they should appear. Because meshes must be sorted, we can't store them in a map, therefore we have an internal structure of "Meshes" that bundle together the pointer to the mesh and the coordinates. This allows us to store this object in a vector and sort it.

Among all the methods it's important to note the following ones:

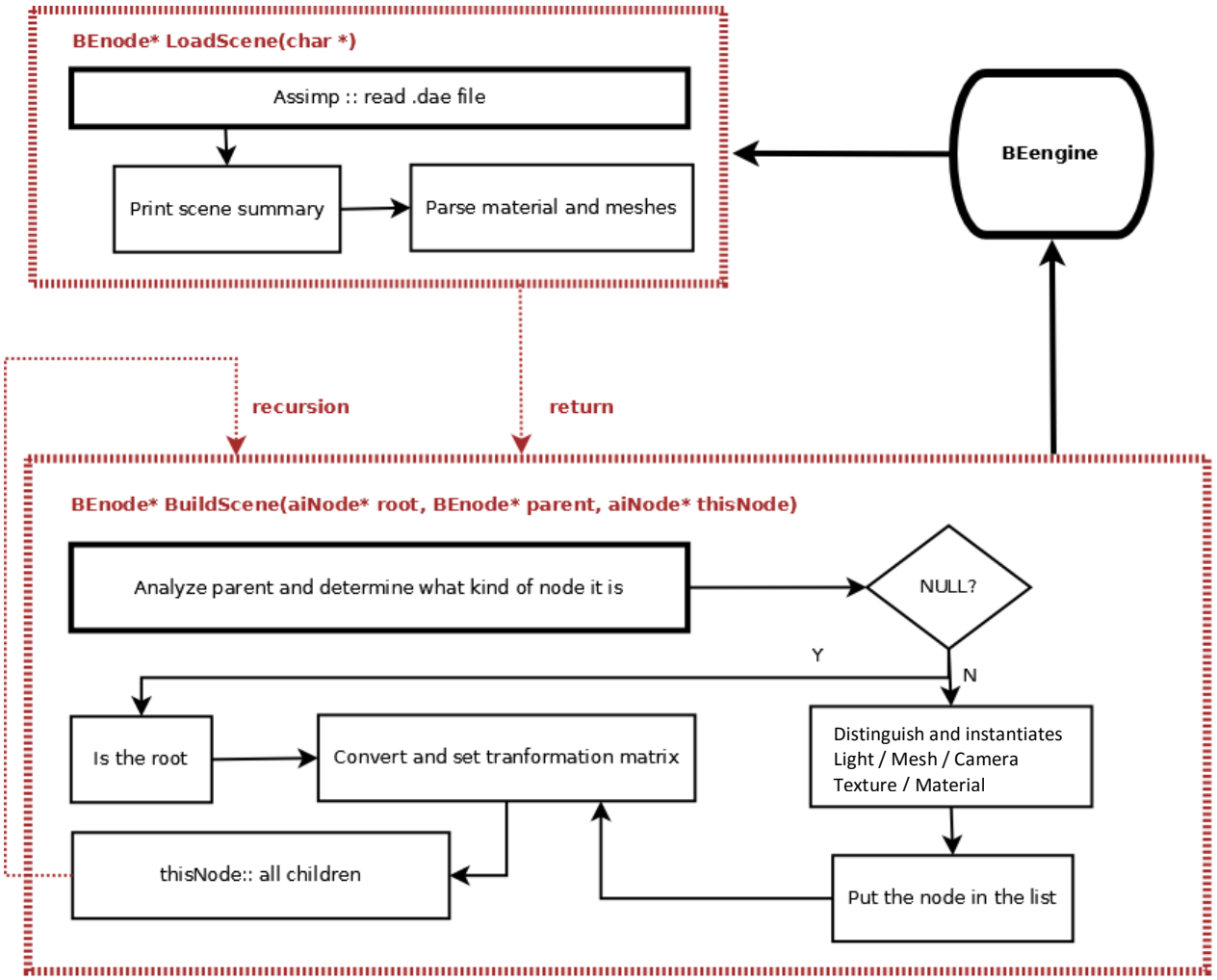
- RenderAll(): calls the rendering method of each type of node
- RenderMeshes(): initializes the stencil buffer, defines the area where reflections should be drawn, draws reflections, draws opaque objects, draws transparent objects.
- Pass*(): All methods having pass in their name are used to update the world coordinates of the nodes in the lists.
- DeepSort(): Used to sort transparent meshes from the furthest to the closest one. A lambda expression is used to comply with C++ 11 standards.

BEsceneLoader

This class is necessary for the loading of a .DAE exported model. Assimp reads the file and builds its own scene graph which we then parse ourselves and use to build our own internal structure. Assimp is capable of understanding and loading many different extensions, however for our engine we will be limited to the above mentioned one.

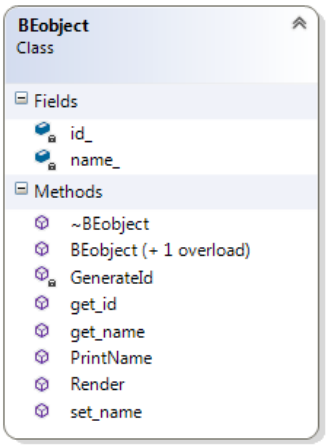
Among all the methods it's important to note the following ones:

- LoadScene(): reads the scene through Assimp, prints debug information about it and then calls the relevant parsing methods.
- ParseMaterials(): goes through all materials found in the scene and retrieves the info in which we're interested and then builds the relevant BEmaterial nodes.
- ParseMeshes(): same as the above method, it's important to note that we also parse submeshes (which are meshes composed by other meshes)
- BuildScene(): Starting from the objects created in the previous methods, a scene graph is built mirroring the structure used by Assimp.



Scene-graph Classes

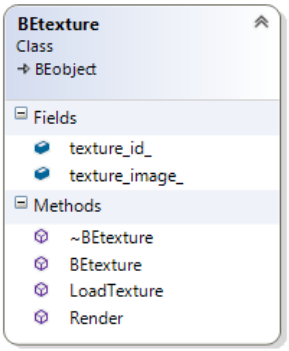
This section illustrates all classes involved in the construction of the scene.



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() that will allow all objects to draw themselves in the world; moreover, it stores basic information useful to distinguish each single object.

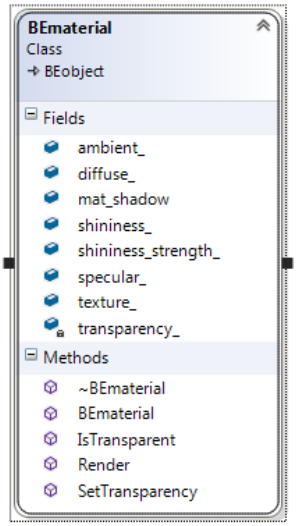
BEtexture



This class stores the texture loaded on construction. This texture is then drawn on the screen when the render method is finally called.

LoadTexture() is responsible for retrieving the image from the disk and storing it in a standardized format (RGBA).

BEmaterial



This class holds the material parameters such as ambient, diffuse and specular.

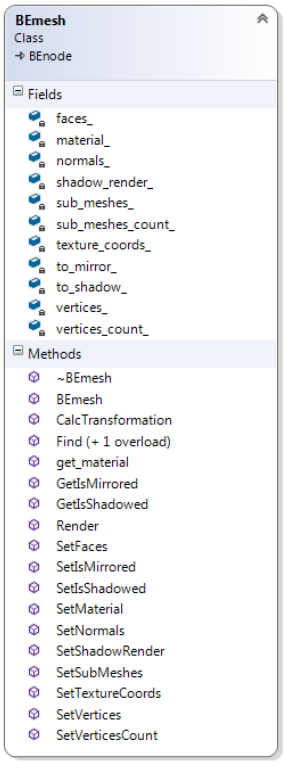
Ambient is the color a material is rendered when not directly illuminated.

Diffuse is the color resulting from light scattered on the surface

Specular is the reflection of the light source itself.

The material might also possess a texture. If a texture is stored, when the material is rendered, the texture render method is called as well.

BEmesh



A mesh (also known as Object or Model or Primitive) stores coordinates useful for the rendering of the objects:

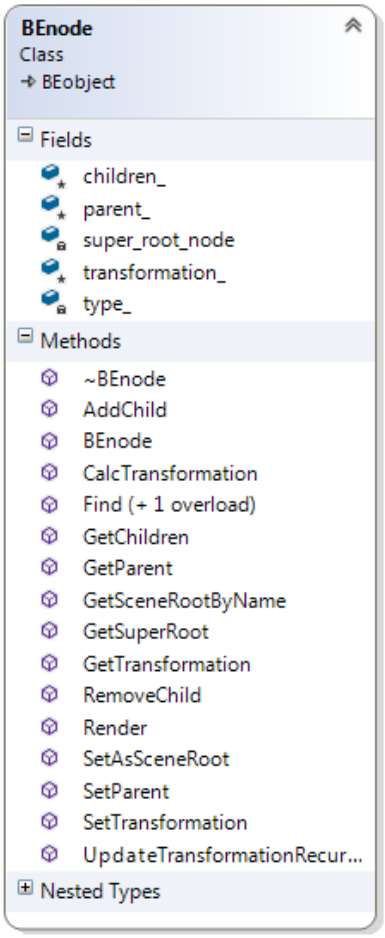
Vertices, norms, texture coordinates. These are passed to the drawing loop pictured below and serve for a correct rendering.

A mesh might be composed of submeshes, if that's the case, such submeshes are rendered as well.

A mesh often contains a material. If that's the case, it is rendered.

```
glBegin(GL_TRIANGLES);
for (unsigned int i = 0; i < vertices_count_; i++)
{
    glNormal3fv(glm::value_ptr(normals_[i]));
    glTexCoord2fv(glm::value_ptr(texture_coords_[i]));
    glVertex3fv(glm::value_ptr(vertices_[i]));
}
glEnd();
```

BEnode



BEnode is the base for all physical objects in the graph.

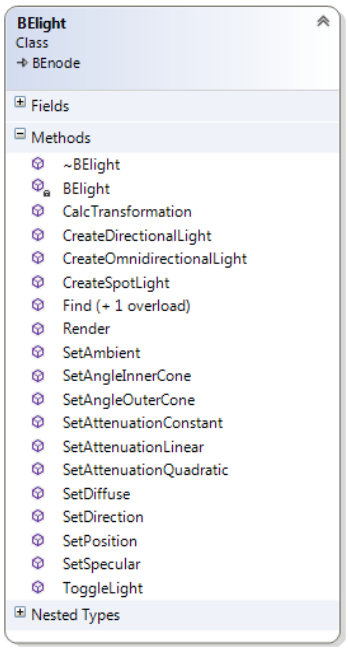
Each node holds a pointer to the parent and has a vector containing pointers to the children.

The render method must be implemented because of C++ rules, however it is never really called as it's always overridden.

Noteworthy is the function relative to the transformation:

UpdateTransformationRecursive(): it recursively computes the coordinates of the object based on the parent coordinates.

BElight



A light can be either DIRECTIONAL, OMNIDIRECTIONAL or SPOTLIGHT. The class contains all parameters necessary to define a light by the standards of OpenGL.

For each light there is a different costructor and rendering is heavily dependent on the type of light.

A maximum number of 8 lights can be added and rendered in a scene. It is suggested to only use 4 lights at maximum for performance reasons, in fact only 4 lights are processed by the GPU.

Issues

Transparency & Mirroring

During the initial stage of development, it wasn't clear how to implement such features.

Transparency was not initially working due to the Z-order of the meshes. Once we managed to sort out the mirroring, we implemented the stencil buffer to limit the area where reflections are rendered.

Though it correctly works on windows, on Linux it appears not to work at all. We couldn’t understand the reason behind this issue.

The mirroring calculation part happens in mesh's method CalcTransformation() where, as seen below, the object it’s translated twice above its pivot, it’s then scaled by a factor of -1 and then positioned in its world coordinates. This is done so that meshes with pivots positioned away from the reflective plane are correctly translated before the scaling is applied.

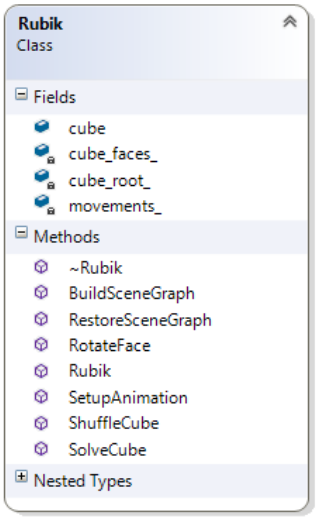
```
if (to_mirror_)
{
    glm::mat4 scale_matrix = glm::scale(glm::mat4(1), glm::vec3(1, 1, -1));
    glm::mat4 parent_transformation_inverse = glm::inverse(GetParent()->GetTransformation());
    glm::mat4 translation_offset = glm::translate(glm::mat4(), glm::vec3(.0f, .0f, -2 * parent_transformation_inverse[3].z
    glm::mat4 scaled = world_matrix * scale_matrix * translation_offset* transformation_;
    BEengine::lists_->PassMirrored(this, scaled);
}
```

Multi-Mesh

We have downloaded the Rubik’s cube model from the internet and we had problems when trying to load it because it was implemented using multi-mesh 3D models. This problem was solved by adding SubMesh references to the Mesh class.

Client Side

Rubik



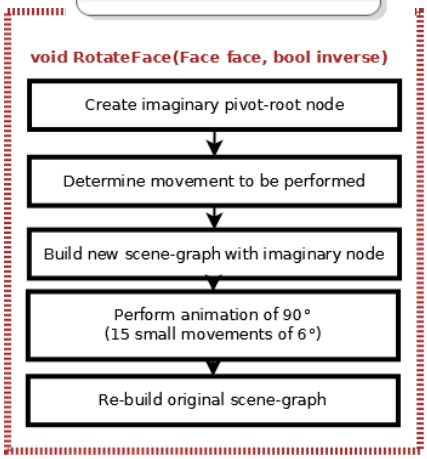
As the Demo is launched, Rubik's main instantiates and initializes a BEengine object which takes care of loading the 3D Studio Max exported scene "scene_final.DAE".

The object Rubik Cube is then created through our engine API calls before finally starting the engine which will allow our program to enter FreeGLUT main loop and therefore render the scene.

Keyboard Callbacks are directly implemented by the client and passed to the engine.

Noteworthy is the RotateFace() method which takes care of rotating a certain cube face and the Animation() method which explicitly generates the animation while the cube is rotating.

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



Perspective

It's possible to move around the scene using the following keys:

- [UP] [DOWN] [RIGHT] [LEFT] Rotate around the scene
- [PAGE UP] Zoom out
- [PAGE DOWN] Zoom in

Lights

The client provides the following operations for lights:

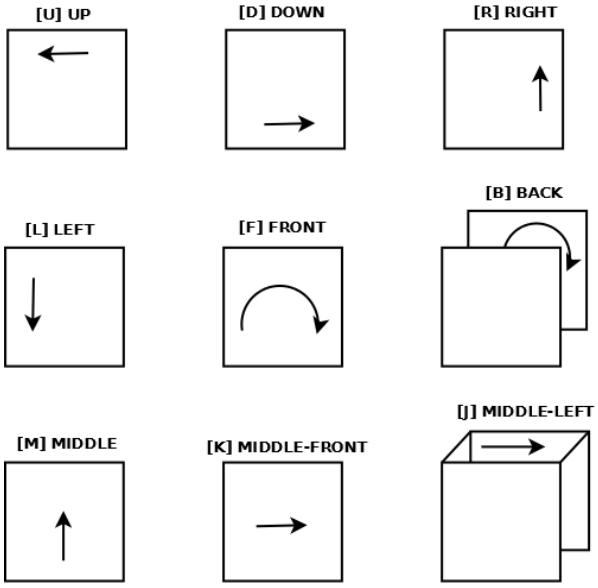
- [1]Toggle the Spot1
- [2]Toggle the Spot2
- [3]Omni light in "Dark mode"
- [4]Omni light in "Bright mode"

Cube Rotation

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

The cube-state is stored in a 3x3 matrix and it is updated after each movement.

Every rotation is logged in a vector which is used as a stack. This operation permits to solve the cube reproducing all the movements in inverse sense.



It's possible to do also those operations:

- [SPACE] Generate 10 random movements
- [P] Solve the cube

Conclusions:

Developing the engine was not an easy task, the timing was very tight and we had to work at an impossible pace to reach the deadline.

writing features such as transparency and mirroring gave us hard times, some others, such as the stencil buffer implementation were fairly easier but resulted in mixed outcomes.

During the whole span of the project we applied and improved our Computer Graphics knowledge, we also learned new approaches and solutions that were not discussed during the course.

Our C++ skills definitely got better considering that one member of the group had not used that specific language before.

The final outcome is very satisfying because we can load any scene, render it, move around it, interact with the elements and everything is handled in a clean and performant way.

There are several limitations due to the restrictions of OpenGL 1.0, but in the next semester we will be able to improve it even further with newest versions.