22 GENNAIO 2016

TARRASQUE ENGINE

COMPUTER GRAPHIC

Students:

Farine Antoine

Maric Adrian

Ribeiro da Costa Nuno Miguel

Professor:

Achille Peternier

Sede:

SUPSI DTI Manno

Summary

1. Introduction

For the course of Computer Graphics we were asked to develop a basic graphic engine. The requirements were that our graphic engine should be able to load a 3D scene from file and reproduce the entire scene, rendering every object and his details, in addition, manipulate the loaded scene graph. The graphic engine should be distributed as .dll on windows and .so on linux. We were also challenged to develop an application which would let a user play with a rubik's cube using our graphic engine as developing tool.

To help us in the task at hand we used the libraries we learnt in class such as assimp, freeimage, freeglut and 3ds Max: assimp is used to parse a 3D object file, in our case we use a collada file; feeimage is used to load the texture images to put on our meshes; freeglut is used to access the OpenGL context and functions and finally 3ds Max is used to model our models and scene.

To design our graphic engine project we first designed a class diagram before starting implementation, what we came up with was basically what we ended up programming. The final version of the class diagram as designed thanks to visual studio looks like this.

2. Engine

2.1. Dynamic/Shared library configuration

In order to compile a dll in windows we need to setup our classes so that the compiler knows what to export when building the dll.

We need to define two macros, one for defining the export function while exporting or importing and the other one to define the library entry point. The first macro is defined in the class TE_Object and TE_Engine and is as follows:

Then in the implementation of TE_Engine we set the dll entry point, we don't use this function but needs to be implemented in order to be able to use the dynamic library under windows systems. The function is implemented as follows:

Notice that both are inside #ifdef WIN32 blocks, under linux we define the LIB_API macro but it doesn't do anything.

2.2. TE_Object

TE_Object is an abstract class extended by TE_Node, TE_Material and TE_Texture, which guarantees univocal id, define an abstract method to render object and another one to obtain the object type.

To determinate the object type, we specified an enum which contains values such as OBJECT, NODE, LIGHT, CAMERA, MESH, MATERIAL and TEXTURE.

Considering that this class is extended by the others, in his header we include all the necessary library mentioned in the Introduction.

```
class LIB_API TE_Object
{
public:
    virtual void render() = 0;
    int get_id();
    Object_type get_type();

    TE_Object();
    ~TE_Object();
protected:
    static int staticId;
    int m_id;
    Object_type m_type;
};
```

2.3. TE_Node

```
class LIB_API TE_Node : public TE_Object
{
public:
    void render(){};
    void link(TE_Node*);
    void unlink(TE_Node*);
    void init_mat();
    TE_Node* find_node(string);
    vector<TE_Node*>* get_children();


    void translate(TE_VEC3);
    void scale(TE_VEC3);
    void rotate(float, TE_VEC3);
    void trasform(TE_MAT4);

    TE_Node* get_parent();
    string get_name();
    void set_name(string);
    void set_parent(TE_Node*);
    bool has_children();

    TE_MAT4 get_position();
    void set_position(TE_MAT4);

    TE_Node(string);
    ~TE_Node();

protected:
    string m_name;
    TE_Node* m_parent;
    vector<TE_Node*>* m_children;
    TE_MAT4 m_position;
};
```

TE_Node represents a node of our scene graph, it provides function to manipulate the node transformation matrix and find information about the scene graph. We can look for a node by using its name using the find_node function, this will parse the scene graph looking for a node with the same name as passed in the function. The transformation functions translate, scale and rotate perform operations on the node before its transform matrix, the transform function instead applies the passed matrix after the matrix transformation; to get a matrix of a transformation a user can use the macros te_inverse, te_scale, te_rotate and te_translate which use glm functions to generate a matrix. It also provides functions to link and unlink nodes between them, the link function will link the node passed as an argument to the node calling the function, if the node passed as argument already has a parent, it will be unlinked from it.

2.4. TE_Mesh

TE_Mesh is a class which stores information to describe a mesh. Information stored is the TE_Material, vertexes, normals and the texture coordinates, the last three are saved as vectors of TE_VEC3, for x, y and z. This class extends TE_Node and must override the render method.

Render() method first of all calls the render() of the material variable, then proceed to build the mesh. In case that the material is transparent, checked by the method isTransparent(), we have to first render the back face, so that we enable the front face culling and iterate over the number of vertices, with step value of 3. At each iteration we draw a triangle, for every vertex of the last one, we draw a normal, the vertex and we set the texture coordinate; at the end of the loop, we set back the back face culling. Only transparent meshes need to differentiate between front and back face culling and loop them, that's why we put the loop before in an if check. The second loop works same as the last one, but just rendering the front face (back face culling active by default).

2.5. TE_Material

TE_Material stores data about the material properties and the relative texture. Texture gives more details about the surface and is stored as type TE_Texture. Material's properties are reflective characteristic such as ambient, diffuse, specular and emissive, which defines along with the shininess property, how material react to lights. If the material is undefined for a mesh, we set his properties to show a light-gray color. An additional information is the opacity, this parameter has to be set by the user with set_transparent(float) method after the mesh is instantiated, because the loading from file is bugged. To determinate if it's transparent, we offer the method is_transparent(). The engine applies the opacity to a mesh by using the alpha channel of reflective properties, in fact, we force that value to the opacity specified by the user.

Since the class extends TE_Object, we override the method render(). If there's a texture, we call his render method, however we issue freeglut to apply the material's properties with glMaterialfv(...)

2.6. TE_Texture

```
class LIB_API TE_Texture :public TE_Object
{
public:
    void render();

    TE_Texture(std::string,int);
    ~TE_Texture();
private:
    unsigned int m_tex_id;
    int m_anis_level;
    FIBITMAP *bitmap;
};
```

TE_Texture class loads and stores a texture by giving the path to the 2d file format. It's also possible pass the anisotropy level, otherwise its value is set to 0. Texture sources are placed in the same folder of the executable, but it would be better if we put them in another apposite folder. During the instantiation we generate an id bound to the next texture loaded, that will be used to swap in the texture while rendering again, then we load the image from file and convert it to 32 bits, to ensure compatibility, and it's useful for alpha channels to store transparent textures. After loading, we tell freeGLUT to load the texture by passing the id, and with gluBuild2DMipmaps(...) we build a mipmap, or rather a series of pre-sized texture maps of decreasing resolutions.

Texture rendering is done by the overriding the method render(), first step is enabling texture mapping and loading the image by id, then we instruct freeGLUT on how behave with resizing. While a face has texture coordinates not in range [0 .. 1], we repeat the image. To avoid aliasing by zooming in or jittering by zooming out the image, we tell freeGLUT to use the mipmap. Finally, if it's specified, we set the anisotropic level.

When the instantiation get destructed, we unload the relative image and delete the mipmap to free allocated memory.

2.7. TE_Camera

```
class LIB_API TE_Camera : public TE_Node
{
public:
    void render(){};
    TE_MAT4 inverse();
    TE_Camera(string);
    ~TE_Camera();

private:
    TE_MAT4 m_camera_pos;
};
```

TE_Camera manages information about the scene's point of view. An instantiated camera is placed in origin, all the meshes, to be seen, have to be placed at inverse matrix of the camera, for this reason we defined a method to retrieve the inverse matrix. Since the camera could be a node in a scene graph, this class extends TE_Node, but we don't manage and export cameras in the scene

2.8. TE_Light

TE_Light maintains data about light source in our scene. Common properties of lights are ambient, diffuse and specular, which define the color and intensity. While instancing, we pass previous properties, the light type and the cutoff angle, the latter must be first converted from radians to degrees, and thus is between 0 and 360°, we have to split it in half for freeGLUT, this parameter isn't necessary for the directional light. Every instantiated light is bind to an GL_LIGHT#, we hold an univocal light id which is used to get the element from the lights array that contains all 8 ordered GL_LIGHT#, where # is the index of the array. A Matrix position has to be saved to establish the light position. This class offer set_direction(...) to define the target of the spot light and provide set_attenuation(...) method to specify the attenuation, which works only for omnidirectional and spot lights.

Rendering contains a check to identify which light we're about to render, depending on the light type. For all the lights we specify theirs position with W value at 1, except for directional/infinite type, which is zero, because a number divided by zero get an infinite value. For omnidirectional/point and spot lights we must

set the cutoff angle, the angle from the light direction axis, which is 180 for the first one (360 overall), and a variable value between 0 and 90 (180 overall) for the second one, the latter require also a target position.

After all we instruct freeGLUT to set the lights properties and attenuation values, and activate as well the local viewer position for a more accurate computation of specular highlights

This class in addition provide methods to check if a light is enabled, and other two to turn it on or off.

2.9. TE_List

```
class LIB_API TE_List
{
public:
    void render(TE_MAT4);
    void pass(TE_Node*);
    vector<TE_List_istance>* get_istances();
    vector<TE_List_istance>*  get_depth_sorted_transparent_objects();

    TE_List();
    ~TE_List();
private:
    vector<TE_List_istance>* m_istances;
    vector<TE_MAT4>* m_mat_stack;
};
```

TE_List represents our list of istances to render, inside this class we store a vector of TE_List_istance. It provides the pass method to calculate recursively the final world coordinates of each object in the scene graph by parsing the hierarchy. It also has a render method which renders every object inside it's vector, rendering is done in 3 phases:

1. Rendering lights: at first we render the lights in the vector
2. Render solid objects: We render all solid objects.
3. Render transparent objects: We render all transparent in order from the furthest to the nearest, every object is rendered in two phases: first its back faces and then its front faces.

TE_List_istance

This class represents an object stored inside the TE_List vector, inside it we store a pointer to a TE_Node and a TE_MAT4 for the world coordinates.

```
class LIB_API TE_List_istance{
public:
    TE_Node* m_node;
    TE_MAT4 m_world_matrix;

    TE_List_istance(TE_Node*, TE_MAT4);
    TE_List_istance(){};
    ~TE_List_istance(){};
};
```

2.10. TE_Engine

TE_Engine represents the main object of this project, this class lets us manage the OpenGL context and its properties. This class implements a singleton pattern in order to initiate only one instance of a GL context inside our application, calling the init() function an object will be instantiated and a GL context will be set up.

A client can set up its GL callback functions with the setters provided (set_display_callback, set_keyboard_callback, set_idle_callback_ and set_special callback), all of these functions get a function pointer as argument

To manage a scene a user can load a scene using the load_scene function which takes a filename as parameter.

This method utilizes the library ASSIMP (Asset Import Library) which loads 3d file formats, such as .OBJ and .DAE extensions, the latter format include all the minimum requirements to describe a scene.

At frst we utilize Assimp::Importer to load the scene from the given path, then we instantiate a TE_Node which will contain the root node and his matrix, before saving the matrix, we have to transpose it, because 3DSMax has a different convention.

Before returning the root node to the invoker, we iterate over all root's child nodes, we link them to the root and call the recursive load_node() method on every child, which saves the nodes information.

This method takes an Assimp node as parameter and returns a TE_Node. As we did with the root node, we store a transposed matrix, successively we check which kind of node we're handling.

In case the parameter mNumMeshes is different from zero, the node is a mesh, therefore we loop over the number of vertices and for each one, we store the vertex, the normals, and if a texture is present, we save also the texture coordinates; in three different vectors. Following, we get material reflective properties and opacity, the latter is bugged in Assimp, so that we force its values to 1 and leave to the user the choice to set material's transparency. If the mesh has a texture, we create a TE_Texture with the texture's name. After all, we instantiate a TE_Material with those properties and the given texture and that instance will be passed along with the vectors to build the TE_Mesh.

If the node is not a mesh, we verify if it is a light, by checking the name of the considered node with those in the array mLights of the scene, but first, if the node is named "*.Target", we skip it, because that's the spotlight target and we can retrieve the same information from the spotlight node. When found, we instantiate a TE_Light with ambient, diffuse, specular vectors, and also the light angle. Since we get different values between Windows and Linux, we set ambient and specular as multiples of the diffuse value. Successively we set the instance matrix position, the light coordinates and his attenuation. In case that the light is a spotlight, we must also store the target.

In case that the node considered is not a mesh, even not a light, we store it as a simple TE_Node and store his matrix.

In all cases, shortly before returning the node, we have to iterate over the children as the same way as we did for the root node.

After loading a scene and having it passed inside a TE_List, the user can tell the engine what to render, by passing a camera node and the list to the method render() the engine will know the point of view of the scene and what to render on it. To start rendering a user has to call the method start(), this will start the glut main loop.

The user can control what is being rendered by manipulating the nodes of the scene loaded or the GL context at hand, a user can enable/disable lighting, invert face culling, enable flat or Gouraud shading or even start a 3D rendering section in the display callback by calling tri_start() or a 2D section by calling bi_start(). A user can also write strings on screen, this needs to be done inside a 2D section though.

The init() function sets up our GL context by enabling various options such as: depth testing, face culling, lighting, normalization, alpha test and blending. This function also gets the anisotropic level available on the running machine so we can use it to render textures. It also binds the callbacks to the functions defined in the TE_Engine.cpp file. The display callback set up in the engine will always clear the buffer and then check if a user defined a function, if he has it will run his part of the display callback, after this step the engine will always render what's in the list, then it prints the fps, swaps buffers and then calls the redisplay function. The reshape callback will resize the viewport according to the new window size, all the other callbacks only check if they are defined by the user otherwise they don't do anything.

3. Rubik's cube

The Rubik's cube is the last step of this project, a client that includes and uses our Tarrasque Engine library.

The class structure is simple compared to the complexity of the application and may be resumed through the main function that initialize the engine, sets four callback functions (special, keyboard, idle and display), sets some global parameters, loads the scene from a .DAE file, passes the scene to the list, render the list and start the main loop.

Rubik matrix

A 3x3x3 matrix is used to store all the Rubik smaller cubes. The matrix is used to handle the faces rotation and the Rubik state after each movement.

Display callback

This function is called once when the main loop start and then every time there is an alteration on the scene.

It is responsible for displaying the text on the bottom left of the screen, for generating the fake shadows and the mirrored scene to simulate the reflection.

While doing that, the plane is removed from the scene and replaced with a temporary node with the same position. This is done because of the plane being one of the heaviest mesh in the scene (we need a high rate of segments for the spot light to show an almost circular target on the plane), and rendering it multiple time will result in a severe drop in frames.

To create the shadows the material is set to a black color and the root is scaled to 0 on y axis. This way our scene turns black and is flattened on the plane to simulate shadows.

To mirror the scene, the technique is similar but with a scaling of -1 on y axis. This way the scene is mirrored through the plane which is transparent and create the effect of a reflection on it.

After all the effects are created, we relink the plane, get rid of the temporary node and we render the whole scene normally.

Special callback

This function is called every time an arrow key is pressed and handle the cube rotation. For each arrow the center cube is rotated in the corresponding direction.

Keyboard callback

This function is called every time a non special key is pressed. It handles all the interaction with the cube like enabling/disabling light, rotate cube's faces, solve or shuffle the cube, ..

Key list:

[q – w]	Rotate the top face	[,]	Turn ON/OFF the sky light
[a – s]	Rotate the bottom face	[.]	Turn ON/OFF the bulb
[y – x]	Rotate the front face	[-]	Turn ON/OFF the table lamp
[1 – 2]	Rotate the back face	[k]	Solve the cube
[Q – A]	Rotate the left face	[j]	Shuffle the cube
[W – S]	Rotate the right face		

Enabling/disabling a light will also change its textures with one that use an emissive material so that the texture won't be affected by other lights.

To rotate a face basically link all the side cubes to the center of the rotating face and rotate it. To obtain that we have to use some helper node so that after linking the side cubes won't be affected by the center's face cube translation.

For the solve and shuffle method we use a list of function pointers to memorize all the moves done and their opposite so that popping them out backward will solve the cube bringing it back to the initial state and a list of pending moves so that the idle function can handle a sequence of moves.

Idle function

This function is called every time the scene is in an idle state and is responsible to handle the rotation animation and the list of pending moves.

Initially, it checks if there are pending moves and execute the first one in the pending line.

After that, the rotation angle is checked and if it is greater than 0 all the side cube are linked to the center of the rotating face which is rotated by 5° each time. The angle is decreased by the same degree and when it reaches 0 the animation is done and all the side cubes are linked again to the Rubik center.

Conclusion

During the development we had to make some decision about how to handle some parts of the code and how much freedom give to the client. In the hand we came up with a balanced solution where the complexity for the client is not too high but still has enough freedom in using the scene and the list.

This wasn't an easy task, because liberty implies complexity and coming up with a solution that grant both freedom and a easy to use environment is a really hard process.