Scuola universitaria professionale della Svizzera italiana
Dipartimento tecnologie innovative

SUPSI

Documentation: TowerCraneSimulator 2022 & Utopia Engine

Students
Fabio Crugnola
Lorenzo Di Folco
Adriano Cicco

Supervisor
Achille Peternier

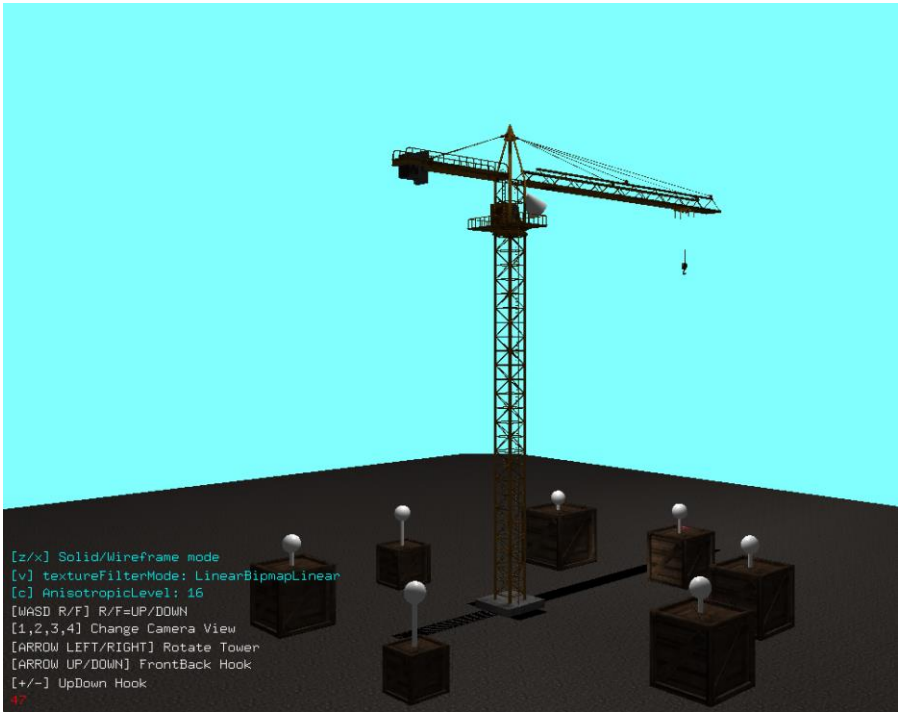
Correlator
Amos Brocco
Giancarlo Corti
Marco Paoiello

Client
Achille Peternier

Degree course
Ingegneria Informatica

Course module
M-I5070 – Grafica

Anno
2022-2023



Date
6-01-2023

STUDENTSUPSI

Index

Abstract

In the following document, a deep analysis of a graphic engine, commissioned as a final project for the Computer Graphics course gets discussed along with the realization of a simple application exploiting the above-mentioned graphic engine.
The engine uses the OpenGL graphics library, along with the GLM mathematics library and the FreeGlut and FreeImage libraries for window management and image loading, respectively. Special attention has been paid to ensure the compatibility of the engine with both Windows and Linux operating systems, allowing for a wider range of usage and deployment. The document will detail the design and implementation of the engine and the client application, as well as their capabilities and performance.

Utopia

Utopia is the name that is used to refer to the shared library. The library is comes packaged as a shared library that comes in the form of a .so file under Linux and .dll under Windows.
It has been decided to use the C++ 11 standard. In particular, the library takes advantage of modern C++ with features such as smart pointers. Smart pointers are an evolution of traditional pointers, providing automatic memory management and increased safety compared to traditional pointers. Most of the classes of which the library consist of are implemented using the PIMPL idiom. The PIMPL pattern, allows for hiding the complexity of a class's implementation behind a pointer, improving code readability and project flexibility. By using both of these tools, the development process and the resulting code are more robust and inherently safer to use when compared to similar implementation written with old techniques.
To sum up, Utopia contains methods that allow interpreting an OVO file and building a hierarchy of nodes that can be graphically represented through the use of rendering pipelines. The approach used throughout the entire development, was to always design the various components while giving to Utopia’s users full flexibility in its usage mechanics.

Structure

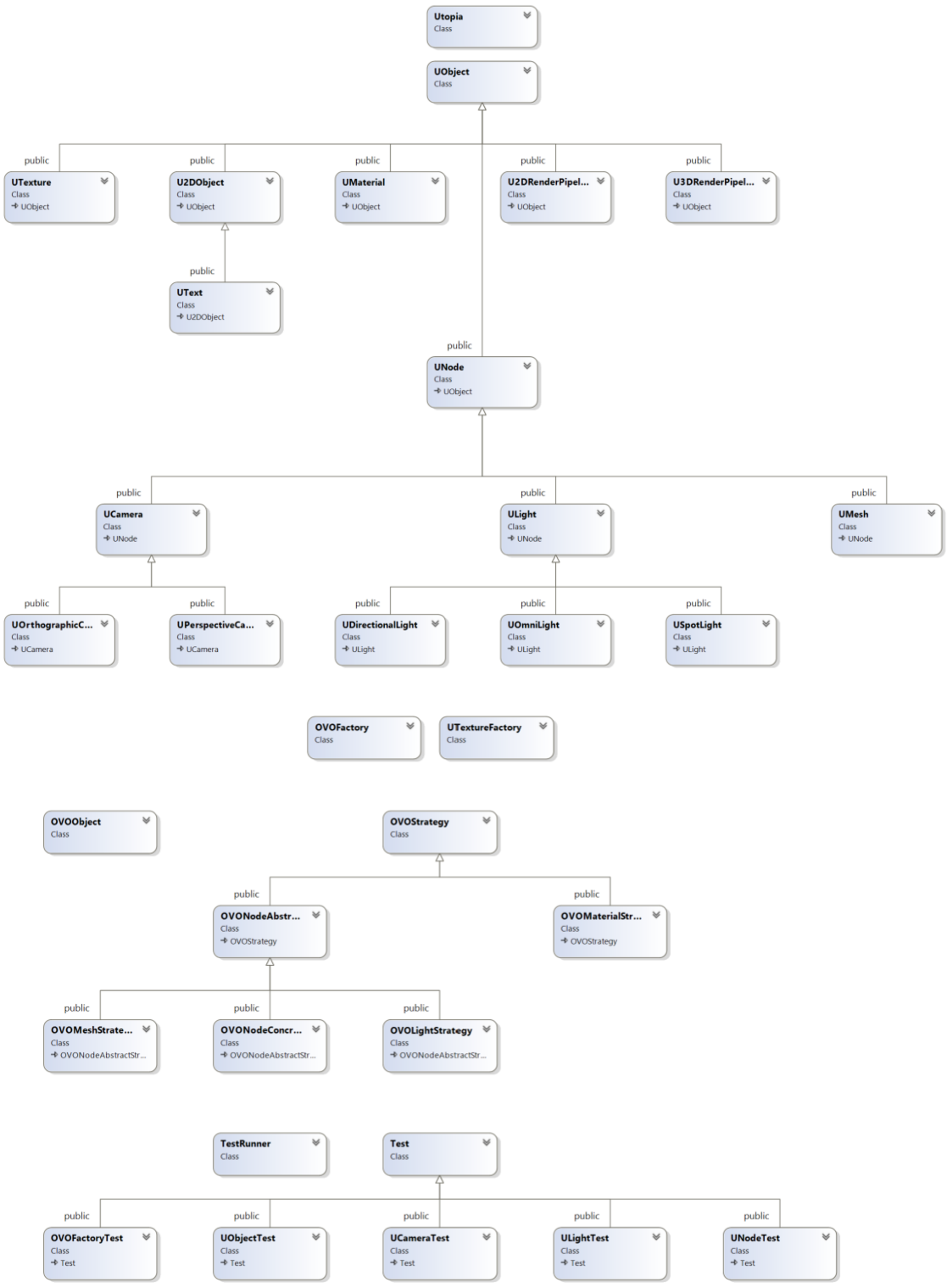


Figure 1 Utopia class diagram

As it can be seen in Figure 1, the Utopia Engine can be divided in four main blocks loosely based on its class hierarchy:

- **The "UObject" block**: that consist of all the classes that inherits directly the UObject class. The UObject class is at the base of the class hierarchy of the Utopia Engine.
- **The "UNode" block:** that holds all the classes that are part of the 3D scene graph.
- **The "OVO" block:** that consist of all the classes dedicated to the decoding of an OVO file
- **The "Test" block:** that holds the logic used to implement all the unit tests and feature tests of the engine

In the following lines, the logic and the elements composing the various block will be described in more details.

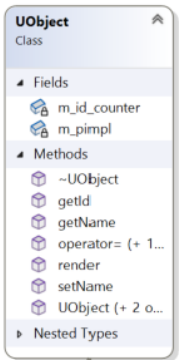
Utopia

The Utopia class is implemented with the Singleton pattern. The singleton instance is used as the main point of access to most of the graphics engine configuration related stuff. Consequently, most of the method calls made to it will appear at the begging of the client application and in the rendering main loop. The most important methods of this class are:

- ***init()***: This is the first function that must be called at beginning of a program using Utopia. It initializes the OpenGL context with the Glut library helper functions and creates the one and only window that will be used Utopia.
- ***clear()***: This function is used to clear the display, usually at the end of the render loop.
- ***free()***: This function is called at the end of the execution of the program with the purpose of freeing correctly the memory used by OpenGL.
- ***mainLoop()***: This function is used to notify to the internally used freeGlut library to execute all the procedures needed before we start drawing object on the screen. It needs to be called inside the main loop that a developer using our library will define in its client application.
- ***swap()***: This function swaps the buffers of the current window since OpenGL in Utopia is by default configured with two video buffers.

The Utopia class act also as the main dispatch center for all textures and materials. Every material and texture loaded from files or instantiated during the program execution will always be available here and accessible through their assigned names. A wide range of callbacks can be set through various methods to be able to handle different events such as keyboard events, reshape of the window, closing of the window and so on. Moreover, it offers the ability to set and start a timer with a dedicated callback to handle asynchronous operation. The main principle used while developing this class was extreme flexibility. The wide range of methods offered gives the user the ability to handle as better as he thinks the main loop of its application by combining the various Utopia class method's calls.

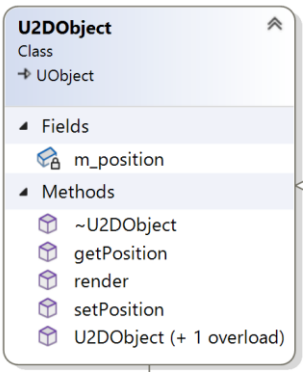
UObject



As mentioned earlier, the UObject class is at the base of most of the class hierarchy of the Utopia Engine. Every UObject have a mandatory name, a unique identifier represented with an integer and a virtual *render()* method.
The UObject gets its ID assigned by a static field called *m_id_counter* which gets incremented every time an UObject is instantiated. The render method is supposed to be called during the scene render phase.
An UObject cannot be copied since it is uniquely identified with its ID. To simplify the management of object instances through all the project it has been decided to remove any kind of default copy constructor and copy assignment operator.

Figure 2 Inner structure of the UObject class

U2DObject



An U2DObject is a simple UObject (class from which directly inherits from) used to represent an object on the screen and therefore using only 2D coordinates.
An U2DObject has got only a field which is *m_position* and it stores the 2D space position coordinates x and y with a *glm::vec2*. Like its parent class UObject, it cannot be instantiated.

Figure 3 Inner structure of the U2DObject class

UText

UText is an U2DObject used to represent text on the screen.
The class can be passed to an **U2DRenderPipeline** to display text on the screen at the chosen position.
It is possible to customize the color of the text within the RGB color space by using the *setColor()* method.
An additional feature is the possibility to choose (if they are available on the chosen platform), a custom bitmap font with the method *setFont()*, which will take as a parameter one of the possible values of the *UText::Font* enum.

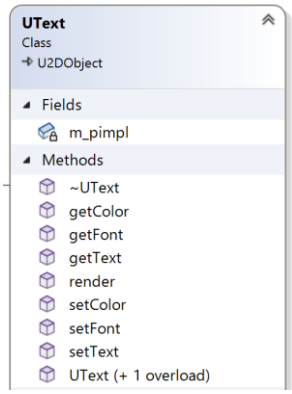


Figure 4 Inner structure of the UText class

UNode

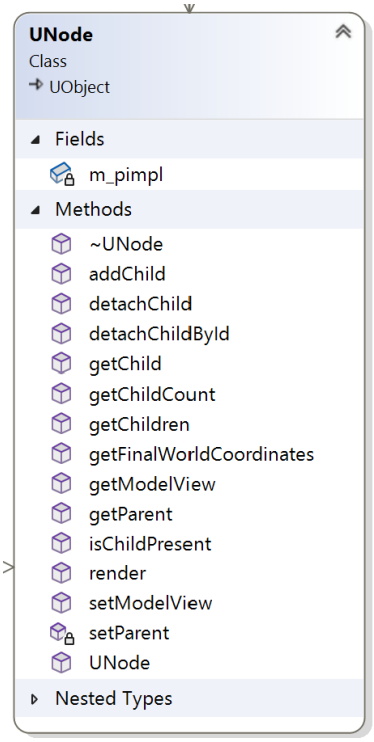


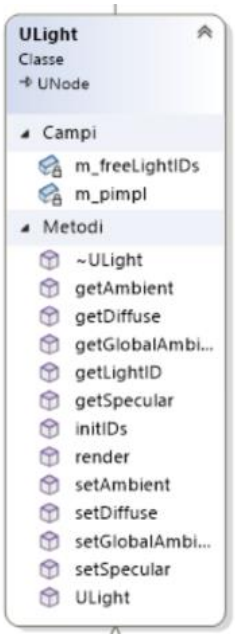
Figure 5 Inner structure of the UNode class

UNode as mentioned above is an abstract class derived from UObject and from which all the objects that can be put in a 3D scene graph inherits. UNode in addition to having the characteristics of an UObject stores the node's model view matrix which is used to apply transformation to the node such as translations or rotation by using the GLM library. The model view matrix uses the *glm::mat4* type to store its information.
Given that an UNode is an element of the scene graph, this means that it also might have an UNode parent, which is represented with a raw pointer since it can also not have it (and therefore be nullptr), and multiple UNodes children. UNodes' children are stored within a vector of shared pointers to UNodes. Within the class we can find all the various public functions for managing the vector of child nodes and the various getters and setters for its fields.
It is possible to add a child to an UNode with the method *addChild()*, which accept in input a shared pointer to an another UNode. A UNode cannot be a child of multiples UNodes therefore if an UNode that is already a child of another one is added, Utopia will throw a runtime exception resulting in the termination of the execution.
The same thing happens when someone tries to add a parent UNode to one of its children, an UNode is added as a child to itself or an UNode gets added again as children of its parent.

A particular mention is needed also for the *getFinalWorldCoordinates()* method which return the world coordinates of a particular node by keeping into account of all the hierarchy of the scene graph of which might be part.

To avoid inconsistencies all the getter methods that returns a child of an UNode returns it as a weak pointer. In this way the ownership of the UNode is always of its parent and corruption of the internal state of a parent is avoided.

ULight



The abstract class ULight is an UNode and contains all the general settings used in the various specialization of lights. The general settings are: light's ambient, diffuse, specular, global ambient values, and light ID.

The class also includes a static stack of integers called *m_freeLightIDs*, which is a used to keep track of available light IDs. The *initIDs()* function initializes this stack with the maximum number of available lights. The constructor takes care of removing and assigning the top element in the stack to the new ULight object.
The ULight's destructor, adds the ID back to the stack before removing the object.
ULight overrides the *render()* function coming from the UNode class. In addition to what gets done by UNode (which is basically render all of its children), it sets the general settings and enables the light.

Figure 6 Inner structure of the ULight class

ULight is inherited by 3 light types: USpotLight, UDirectionalLight and UOmniLight.

UDirectionalLight implements the same methods as the ULight class with only one difference in the *render()* function where it needs to manage some internal stuff related to the freeGlut library.

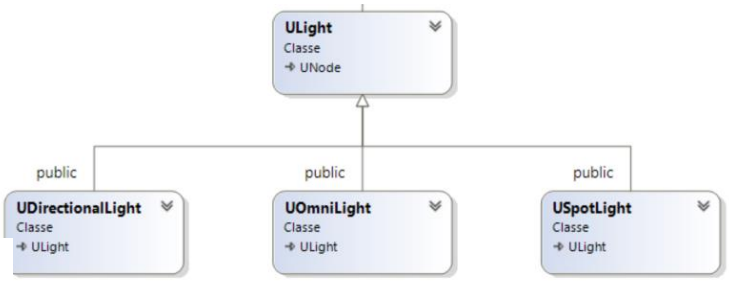


Figure 7 Detailed view of the lights hierarchy

UOmniLight unlike UDirectionalLight has 3 attenuation parameters: constant, quadratic and linear.
All these parameters are set together with the cutoff value (default for this type of light to 180) in the usual *render()* method.

USpotLight adds to the parameters of UOmniLight the cutoff value, which is not predefined but can take a value between 0 and 90, and a *glm::vec3* vector representing its direction.

UMaterial

This class is an UObject used for handling materials. It is possible to create a material using 2 constructors. The first constructor contains the name of the material, it initializes the various emission and ambient vectors to default values. The second constructor allows to pass the attributes of a material along with the name.
Inside UMaterial we can find a static variable called *m_defaultMaterial* which holds a default material. The user can request the default material at any time through a static getter, this material does not have to be initialized by the user and will be used automatically if the UMesh that needs to be displayed does not have an associated material.

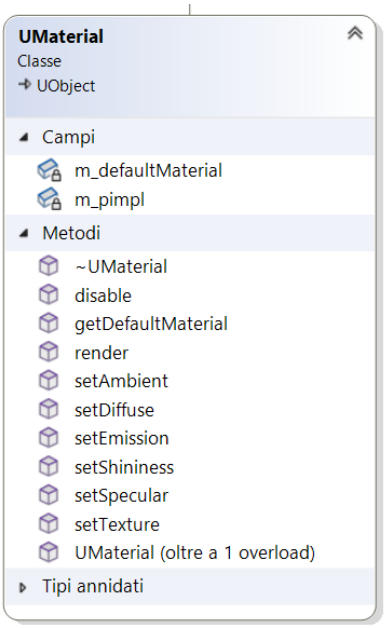


Figure 8 Inner structure of the UMaterial class

UTexture

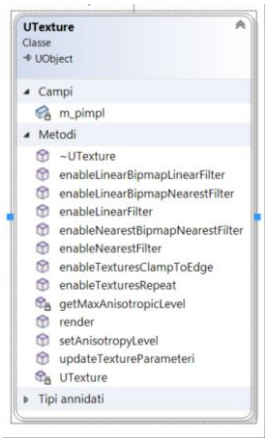


Figure 9 Inner structure of the UTexture class

UTexture contains the *TexId* of the texture it refers to. When the *render()* method of UTexture is called, the bind to the *TexId* is made and a *glEnable(GL_TEXTURE_2D)* is invoked. At the end of the *render()* method of UMaterial, a *glDisable(GL_TEXTURE_2D)* is always called so that the context of the textures does not affect UMaterial that do not contain them.

Each texture is loaded using *gluBuild2DMipmaps*(), allowing the use of "bitmap filters" without having to read textures from the disk again. The UTexture::*updateTextureParameteri()* method allows to update the *texParameter* of the texture by passing a global void method containing the various *glTexParameter* that needs to be set. Some preconfigured static methods to pass as a parameter to the above-mentioned function, are also defined in UTexture. Some of these methods (all of them are in Figure 9) are:

- UTexture::enableNearestFilter()
- UTexture::enableNearestBipmapNearestFilter()
- UTexture::enableLinearFilter ()

There is a method with similar functionality to UTexture*::updateTextureParameteri* that allows to set the anisotropic filter of the texture called UTexture::*updateAnisotropyLevelTextureParameteri()*.

Since an UTexture is directly related to a file on disk, currently, the only way to instantiate an UTexture is through its factory class **UTextureFactory** which is explained more in detail in the OVO file decoding paragraph.

UMesh

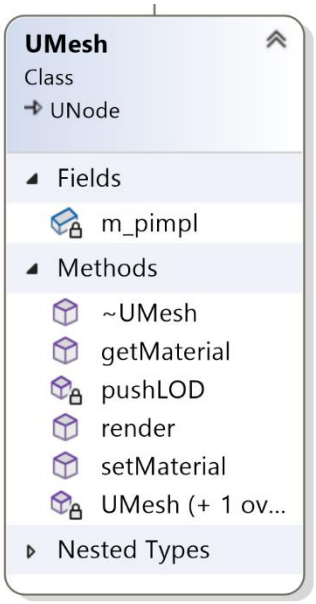


Figure 10 UMesh internal diagram

An UMesh object as the name might suggest, is a 3D mesh in the scene graph and therefore it inherits all the members and methods of the UNode class.
An UMesh can also have an UMaterial and therefore an UTexture.
To handle the triangles of which a mesh consist of by definition, some additional structures visible in Figure 11 have been designed. These structures are used internally to organize and simplify the 3D mesh rendering process with the OpenGL API: Each triangle indeed must be rendered in the correct order, and it comes with other information which are put together in a data structure called **Vertex**.
A vertex holds the three points being the triangle in *coord*, the normal vector used to distinguish the front and the back of the triangle, the *uv* needed to correctly apply the material and the *tangent*. Vertex are organized then in **Faces** which divides them in the correct order.
It is possible to have the same face represented with a different number of vertexes with the aim to enhance details or reduce them for performance reasons.

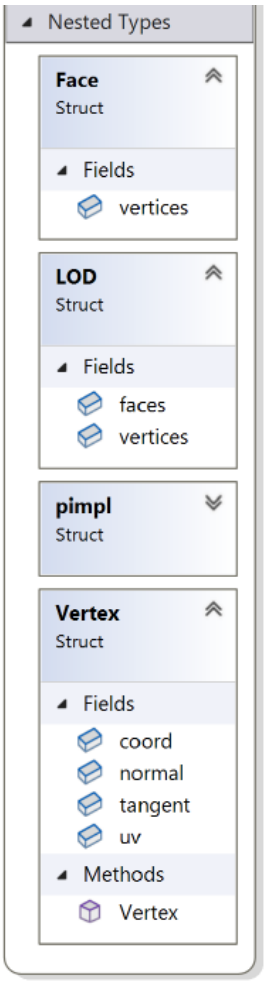
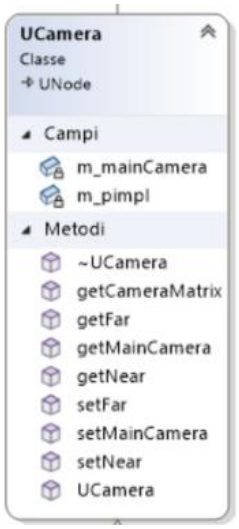


Figure 11 UMesh internal types

Therefore, in UMesh, faces are collected in **LOD**s (Level of Detail) which is a data structure used to organize faces that belongs to the same mesh and that have a specific number of vertexes. Currently, to render a Umesh, the first LOD available in the UMesh is used without distinguishing them by number of total vertices. An UMesh can at the moment be instantiated only through the use of the OVOFactory class.

The UMesh class is also directly responsible for the rendering of the above-mentioned UMaterial and UTexture. Inside UMesh, the *render()* method of UMaterial is called, which takes care of setting the various vectors that make up the material in the OpenGL state machine. If the UMaterial has a UTexture attached, the *render()* method *of* the UTexture gets called.

UCamera



UCamera is a UNode representing a camera pointing in a particular direction in a 3D scene graph. It contains parameters such as near and far clipping planes do define its field of view. The class also contains a static weak pointer to the main camera, which can be set and accessed through the *setMainCamera()* and *getMainCamera()* methods.

The main camera is the camera used to render a 3D scene graph in a given moment. The class has a pure virtual method that should return the camera matrix used by OpenGL as the projection matrix for a frame.
The actual cameras that are supported by Utopia are defined by the classes UOrtographicCamera and UPerspectiveCamera described below.

Figure 9 UCamera internal diagram

UOrthographicCamera is a subclass of UCamera and contains private member variables for the right, left, top, and bottom edges of the camera's view. The class also has methods to set and get these values and a constructor that takes in the camera's name, and sets the right, left, top, and bottom edges of the camera's view to the width and height of the screen. The class overrides the virtual *getCameraMatrix()* method inherited from UCamera to return a matrix that represents the orthographic camera's view using the *glm::ortho* function.

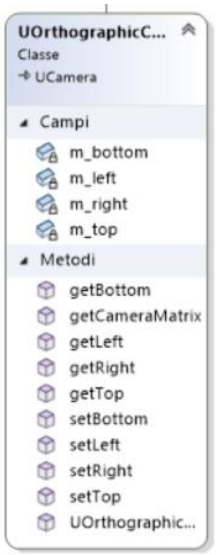
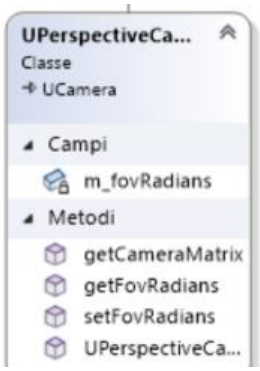


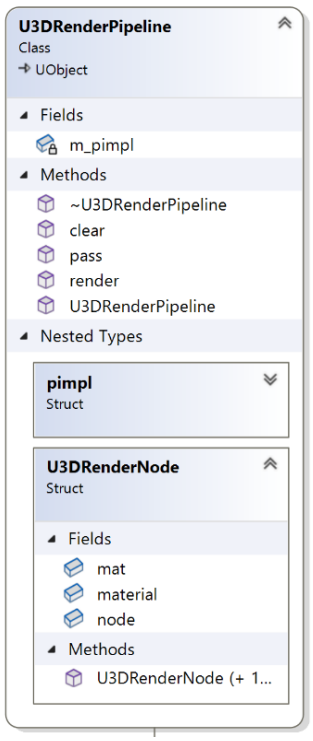
Figure 10 UOrthographicCamera internal diagram



UPerspectiveCamera is a subclass of UCamera and contains a private member variable "m_fovRadians" which represents the field of view of the camera in radians. The class has methods to set and get the field of view, a constructor that takes in the camera's name, and overrides the virtual *getCameraMatrix()* method inherited from UCamera to return a matrix that represents the perspective camera's view. The *getCameraMatrix()* method calculates the perspective projection matrix using the *glm::perspective* function, with the field of view in radians, aspect ratio, near and far values as the arguments. The aspect ratio is calculated by dividing the window width by the window height.

Figure 11 UPerspectiveCamera internal diagram

U3DRenderPipeline



The U3DRenderPipeline is a class responsible, as the name might suggest, for handling the rendering of UNodes.
The objects that the pipeline is designed to render are of UNode type. For these objects to be rendered, they need to be loaded into the pipeline through the use of the *pass()* method, which will store the UNodes in an internal *std::vector* of type **U3DRenderNode.** Along with the UNode a custom *glm::mat4* model view can be passed to *pass()*. The U3DRenderNode offers also the ability to store (by passing it along with UNode to *pass()*) an UMaterial which will be considered during the rendering phase only if the UNode passed is actually an UMesh.
The same UNode pointer can be passed multiple times to the pipeline. By using different at each *pass()* call, it is possible to render the same object with different transformations or colors. This technique let the developer save a huge amount of memory which would be otherwise waster by duplicating every UNode that needs to be rendered more than one time.
The *render()* method when called proceeds to render everything besides ULight nodes, which are rendered at the end of the procedure.
It is possible to remove every element in the pipeline by using the *clear()* method.

Figure 12 Inner structure of the U3DRenderPipeline

U2DRenderPipeline

The U2DRenderPipeline has the same role of the U3DRenderPipeline, but it is strictly built for rendering U2DObjects, which the *pass()* method accept as a parameter along with . The internal implementation is simpler than the one seen in the earlier paragraph.
When pass is called a U2DObject is added along with its screen coordinates of type *glm::vec2* (x and y) to an internal list implemented with a *std::vector* of U2DRenderNode.
The *render()* method when called, switch to a “2D Rendering Mode” by pushing onto the *glMatrix* projection stack an ortographic projection matrix and by disabling the lighting system. It then loops through the list of available U2DObjects and renders each of them with the coordinates passed earlier to the *pass()* method. At the end of the procedure, it pops all the used matrix from the stacks so that the program execution can continue.
It is possible to remove every element in the pipeline by using the *clear()* method.

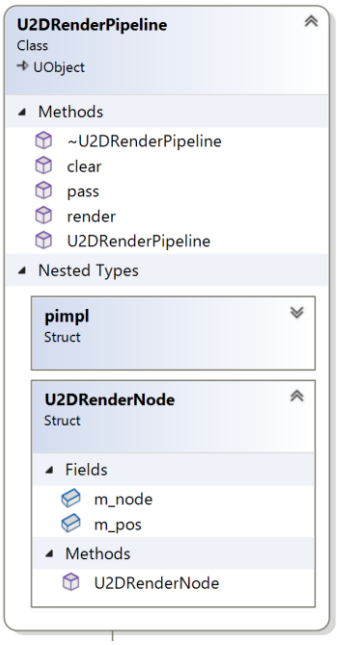


Figure 13 Inner structure of the U2DRenderPipeline

OVO file decoding

Figure 14 OVO File decoding classes hierarchy

OVO is the name of the custom file format used by the OverVision graphics engine, and it has also been chosen as a first-class citizen 3D file format in Utopia Engine.
The decoding of an OVO file from a user standpoint happens through a call to the *fromFile()* method of the **OVOFactory** class: the only argument needed is the path where the file is located.
OVOFactory will look by default under a folder called “assets” (found in the same path as the executable using the Utopia Engine) if the given OVO file exist.
The “assets” folder structure cannot be changed. This design has been chosen to make testing this feature easier.
At the end of the decoding procedure, *fromFile()* will return an UNode object (or one of its specializations) as defined in the OVO file. If the user tries to read an OVO file containing only UMaterials, the method will return a NULL pointer and the read materials will be available through the use of the *Utopia::getMaterialByName()* method. If any error happens while trying to read a file a runtime exception gets thrown by the *fromFile()* method.
The actual file interpreter has been structured using the Strategy and the Singleton patterns.
Each strategy is represented by specific specializations of the **OVOStrategy** class which are implemented as singletons since an instance of a strategy doesn’t have a particular state and there is no need to have multiple instances of them.
The various strategies as can be seen in Figure 14 loosely follows the Utopia’s 3D scene graph object hierarchy.
OVOStrategy holds the basic structure needed to read a chunk of byte from an OVO file. This part of logic has been put there since it is common to all the various type of chunks (and therefore strategies)

that can be decoded. The class internally implements a Buffer structure used to navigate through all the various field of an OVO file.
The strategy pattern makes supporting only specific pieces or chunks of the OVO Format reliable.
In the Utopia Engine the following OVO Chunks are supported and decoded:

- **A generic OVONode,** with the **OVONodeAbstractStrategy** class and its descendants.
- **A generic OVOLight,** using the **OVOLightStrategy** class
- **A generic OVOMaterial,** by the **OVOMaterialStrategy** class
- **A generic OVOMesh,** through the use of the **OVOMeshStrategy**

Each singleton strategy instance is associated to a number based on the OVO File Format Documentation which is defined in **OVOObject::Type** enum.
The key-value pair that can be formed with the above-mentioned correlation is then put in an *std::unordered_map* located inside OVOFactory.
OVOFactory is implemented as a Singleton too since there's no need to have multiple instances of a factory at this development stage.
When OVOFactory opens the file stream, it starts to parse the file with the *recursiveFetch()* method: it reads the 4 bytes representing the OVOObject number and based on that it fetches from the unordered_map, with an O(1) complexity, the right strategy needed to decode the current chunk of bytes.
At this point there are three possibilities:

- I**f the current chunk is an OVONode** then, after decoding the various information, the method *populateWithChildren()* implemented in all the OVONodeAbstractStrategy will internally keep on reading recursively the filestream with the right strategies by literally calling for all the possible children the *OVOFactory::recursiveFetch()* method.
- **If the current chunk is not a OVONode**, then the right strategy after decoding the various information and store them in the right place, won't return an UNode but *nullptr* and if the file stream still has got available bytes to read, the while loop in the fromFile() method will call again recursiveFetch() to decode the rest of the available chunks.
- **If the current chunk is not associated with a supported strategy**, then the byte chunk is simply skipped, and the next one gets read.

As it can be seen, *fromFile()* always try to return an UNode, because with OVO this is the most common case, but still there are some situations where an OVO file might just not have any kind of node in it, as specified in the file format documentation and therefore a *nullptr* is returned instead.

A particular mention is needed for the **OVOMaterialStrategy**, which is also responsible for loading when needed additional textures images. The texture image loading happens internally to this strategy with the use of the **UTextureFactory** class. It happens here since an UTexture is directly associated as already mentioned in the other paragraphs with an UMaterial. This class is implemented as a singleton too for obvious reasons, and it just uses the FreeImage library to load in memory the requested texture which as happens with other files must be "freely" placed under the "assets" folder.
The strategy pattern, along with the singleton one, have been helpful to implement this feature without which there would have been a horrible and unmaintainable 800 or more lines class parsing the whole file.
Organizing the logic in this way makes supporting more features from an OVO Files straightforward with the simple definition and instantiation of additional strategies.

TowerCraneSimulator2022

Structure

TowerCraneSimuator2022 uses Utopia to create a simulation of a tower crane, allowing the tower to move boxes, rotate on itself, lower, raise and move the hook forward and backward. In the application window, in the bottom left corner, there is also an FPS counter that indicates at which frame rate the 3D scene is being rendered. TowerCraneSimulator also implements the management of shadows, the simulation of gravity with a uniform rectilinear motion and the management of cameras with the rotation of one of them (the “free” one) that can happen through the movement of the mouse.
Other controls (such as the one of the tower crane) are handled by the keyboard and suggested with text on the screen.
The client is composed of 4 main classes: Box, BoxesManager, Tower and ClientUtility.

Box

A Box is a wrapper class that stores multiple nodes with the aim of representing an object that can be hooked by the crane in the 3D scene.
It needs to specify three nodes of which the box should consist of. The most important are the ground node, which is the mesh to be hooked and the hook point which represent the point where the crane will hook itself. Usually, the hook point is the parent node of the other two nodes.

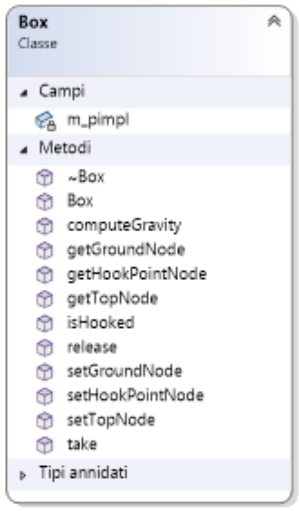


Figure 15 Inner structure of the Box class

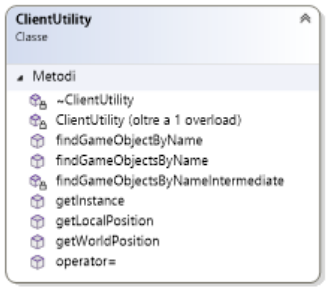
BoxesManager

BoxesManager manages all the boxes in the scene. It contains a method to apply the gravity of the boxes that is responsible for and a method that returns the box that is closer to the hook.

Tower

Tower manages all the mechanisms related to the tower crane. It stores references to the node of the tower, hook, hook cart and the root node of the simulation world. When the tower grabs a box, it detach the box nodes from the floor and adds them as children of the hook. As soon as the box is released it is attached back again to the root.

ClientUtility



ClientUtility is a singleton and supplies methods used to ease some of the operations that the client needs to perform on the scene graph. It allows to obtain the local and final position of a node and search for children nodes by name. The methods defined in this class are not provided directly by the graphics engine because they are more related to what is usually handled in a Game Engine, and Utopia it’s definitely not one.

Figure 16 Inner structure of the ClientUtility class

MainLoop

Utopia provides maximum flexibility to its users by not relying on the *displayCallback()* of freeGlut but by using the *mainLoopEvent()* of freeGlut. Therefore, in the entry point function of TowerCraneSimulator2022, after the configuration of the various Utopia settings, the creation of the various rendering pipelines, the creation of cameras, and the other needed objects, the program enters the main loop. Inside the loop, the gravity of the boxes is calculated, the shadows are rendered along with the 3d elements and the text.

Conclusions

In conclusion, the implementation of an OpenGL based graphics engine has been a successful endeavor. The use of OpenGL has allowed for efficient and high-performance rendering of 2D and 3D graphics. The engine has been tested with various models and has shown to be capable of handling large amounts of data while maintaining a consistent frame rate. The flexibility of the engine has been demonstrated through the ability to easily add new features and the good performance of TowerCraneSimulator2022. Overall, the development of this OpenGL based graphics engine has been a valuable learning experience and has supplied a solid foundation for future projects.

The developers,
Fabio Crugnola, Lorenzo Di Folco, Adriano Cicco