Scuola universitaria professionale della Svizzera italiana
Dipartimento tecnologie innovative

SUPSI

Documentation: TowerCraneSimulator 2022 & Utopia Engine

Students
Adriano Cicco
Fabio Crugnola
Igor Fontanini

Supervisor
Achille Peternier

Correlator

Client
Achille Peternier

Degree course
Ingegneria Informatica

Course module
M-I6110Z – Realtà Virtuale

Anno
2022-2023

Date
20-05-2023

Index

Abstract

In the following document, a deep analysis of a graphic engine, commissioned as a final project for the Computer Graphics course and later updated in the following Virtual Reality course is discussed along with the realization of a simple application exploiting it.
The engine uses the OpenGL graphic library, along with the GLM mathematics library, the FreeGlut, the GLEW and FreeImage libraries for window management and image loading, respectively. Special attention has been paid to ensuring the compatibility of the engine with the OpenVR framework and the LeapMotion SDK. The final purpose was realizing a proof-of-concept application able to interact with a compatible Virtual Reality Headset. This document will detail the design and implementation of the engine and the client application, as well as their capabilities and performance. The document covers only the changes that were implemented to support virtual reality. We recommend to check the old version of this document (available in this same folder) for other details.

Utopia

Utopia is the name that is used to refer to the shared library. The library comes packaged as a shared library in the form of a DLL file for 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 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 the project flexibility. By using both 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.
Moreover, Utopia contains methods that allow interpreting an OVO file and building a hierarchy of nodes that can be graphically represented with rendering pipelines. The approach used throughout the entire development process was to always design the various components while giving Utopia's users full flexibility in its usage mechanics.

Structure

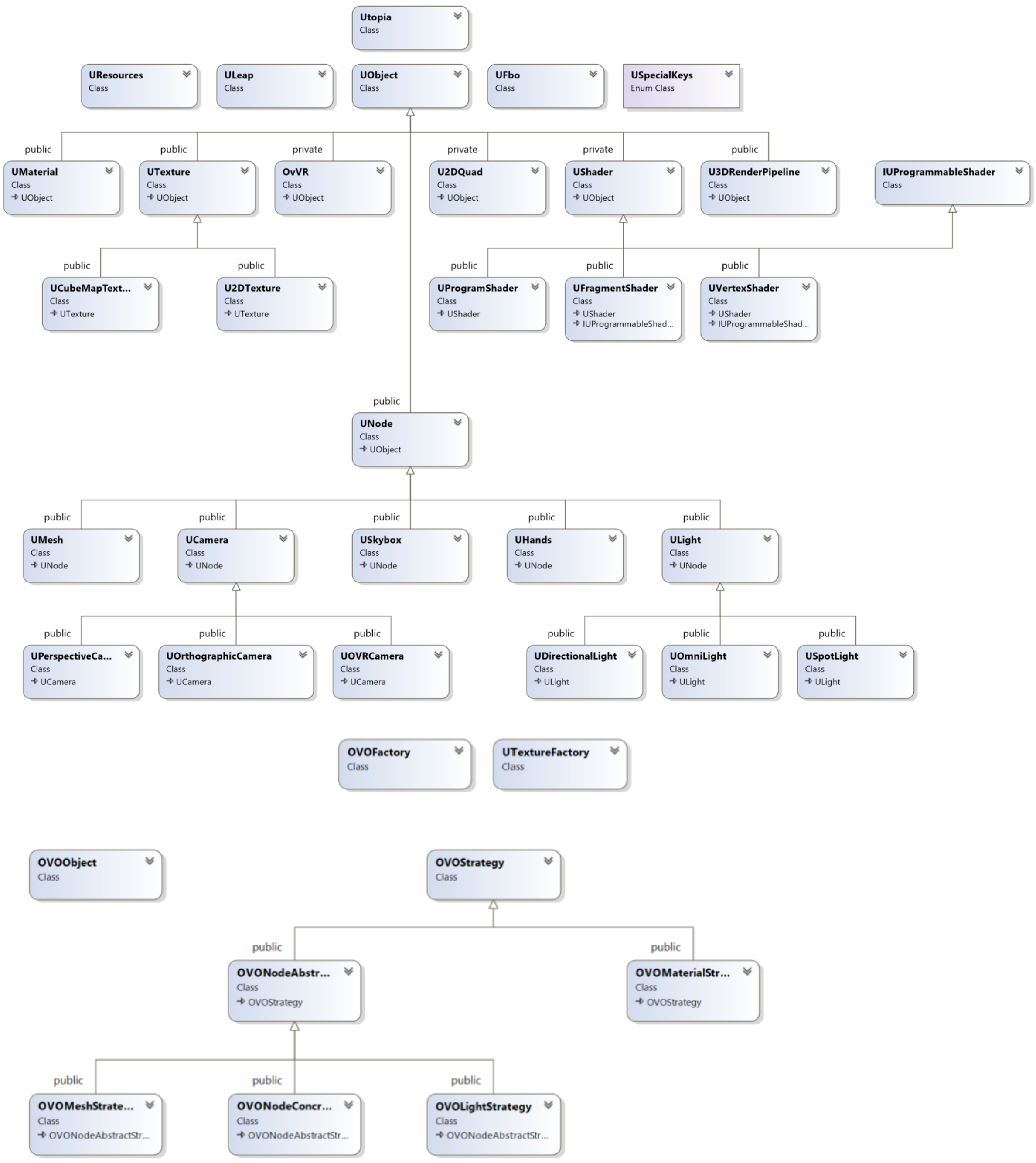


Figure 1 Utopia class diagram

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

In the following lines, the logic and the elements composing the various block that got changed for the virtual reality implementation will be described in more details.

ULight

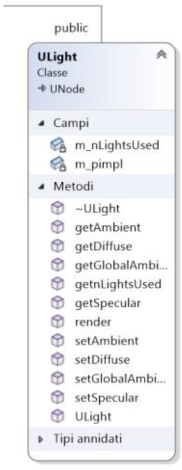
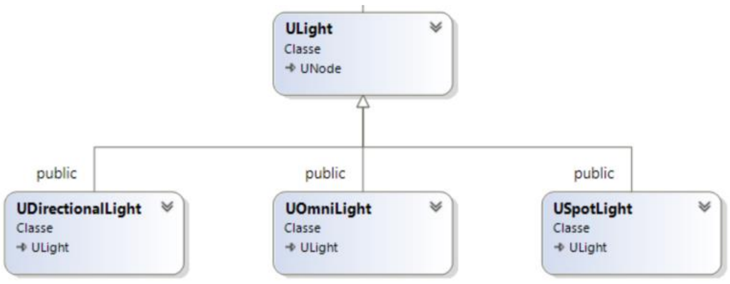


Figure 2 Inner structure of the ULight class

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 ULight class contains an unsigned int containing the number of lights present in the scene, useful within the shaders to divide the emission and ambient of the various lights by the number of lights present at a given time. This strategy has been used because of the use of multipass rendering, where ambient and diffuse components cannot be summed NLightUsed times.
The ULight render is responsible for passing information to all the illumination shaders, such as the number of lights, the ambient, diffused and the specular field of light.
Before calling the ULight render, the rendering of the inheriting classes is called.

These classes are characterized by their own fragment and vertex shader, useful to represent different types of illumination.
ULight is inherited by 3 light types: USpotLight, UDirectionalLight and UOmniLight.

Figure 3 Detailed view of the lights hierarchy



UDirectionalLight It does not need to pass any additional information that is not already provided by ULight to the shader. It is considered that the directionalLight always points downwards.

UOmniLight In addition to the parameters provided by ULight, this type of light also requires the position of the light in eye coordinates to operate.
USpotLight provides information to the fragment shader regarding the light position, spotlight direction, cutoff and linear and quadratic attenuation.
Inside the shader to determine the intensity of the light is used a parabolic function, to ensure a greater intensity in the center of the beam Luminous decreasing the intensity exponentially in its extremities

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 there is 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.
Each material must have an associated texture, when creating a material, the base is set a white texture that can be replaced later (for example with the one loaded from the OVO file).
Inside the rendering of the material is passed information to the currently running shader regarding the emission, ambient, diffuse, specular and shininess of the material.
Each material instantiated during the program execution is added to a static list under this class with the purpose of keeping easy and global access to each of them. In particular, with the *forEach()* static method accepting a lambda function, is possible to apply to each of these materials the same operations.

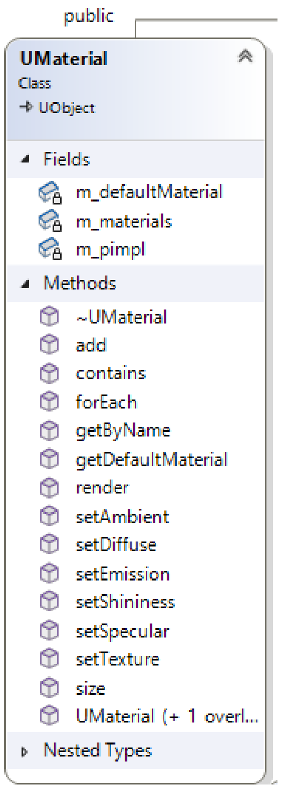


Figure 4 UMaterial inner structure

UTexture

UTexture is an abstract class containing the *TexId* of the texture it refers to. When the *render()* method of UTexture is called, the bind to the *TexId* is made.

Each texture is loaded using *glGenerateMipmap()*, allowing the use of "bitmap filters" without having to read textures from the disk again. Moreover, each texture has multiple methods associated to it which can be used to improve its aspect under the various render conditions. Some of these methods (all of them are in Figure 9) are:

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

Currently in Utopia there are two possible variants of UTexture:

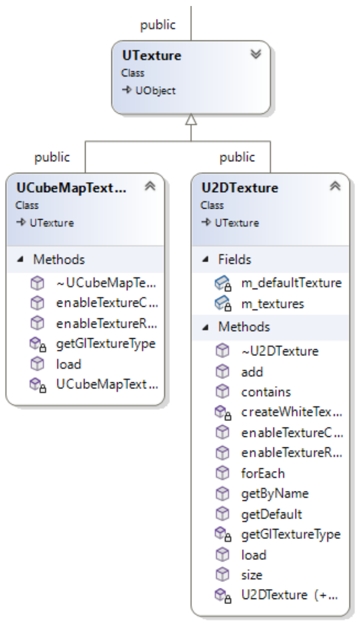
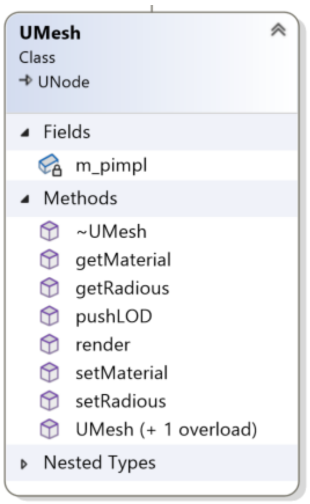


Figure 5 Inner structure of the UTexture class

- **U2DTexture** that is used to represent textures belonging to UMesh with UMaterial. Each U2DTexture instantiated during the program execution is added to a static list under this class with the purpose of keeping easy and global access to each of them. In particular, with the *forEach()* static method accepting a lambda function, is possible to apply to each of these textures the same operations such as the one defined in the precedent paragraph.
- **UCubeMapTexture** that is used to represent six textures belonging to each face of a USkybox.

UMesh



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. The UMesh class now uses **VBO**s (Vertex Buffer Objects) to store the vertices of the Mesh in the GPU. This makes it possible to avoid the overhead of communication between RAM and VRAM when the Mesh is to be rendered, and thus to be more performant. **Face index arrays** are also used to avoid saving the same vertices several times but defining their order by the index assigned to them. Another mechanism used is the **VAO** (Vertex Array Object), so that it can be used as a container for VBOs and Face index arrays. This makes it possible to return to the context of the current mesh only via the id of the VAO.

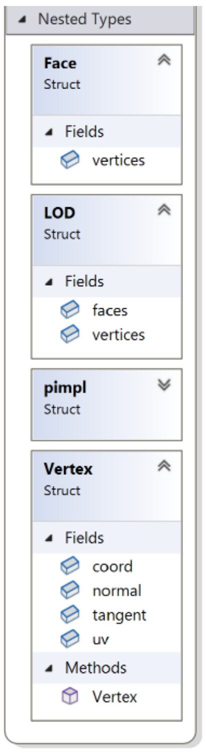
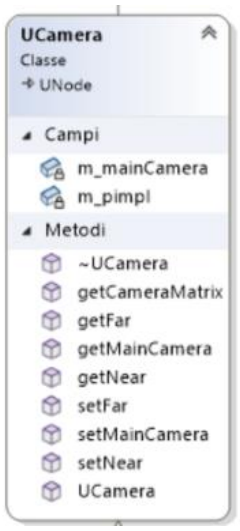


Figure 6 UMesh internal types

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 7 UCamera internal diagram

UOVRCamera is a subclass of UCamera whose instances are used as main cameras when the OpenVR mode is enabled in the file "conf/global.conf". The camera is internally set as main by the engine in the *init()* phase. The camera matrix and the model view are internally managed by the OpenVR framework, and the developer shouldn't have the necessity of messing with them.

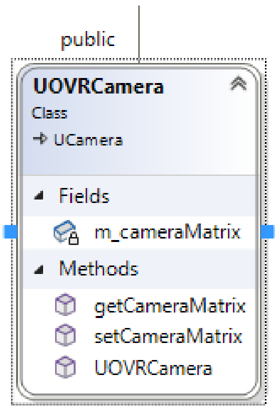


Figure 8 UOVRCamera internal diagram

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.

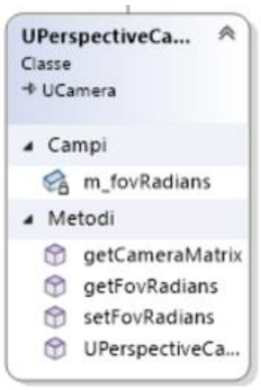

UPerspectiveCa...
Classe
UCamera
Campi
m_fovRadians
Metodi
getCameraMatrix
getFovRadians
setFovRadians
UPerspectiveCa...

Figure 9 UPerspectiveCamera 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.

UFbo

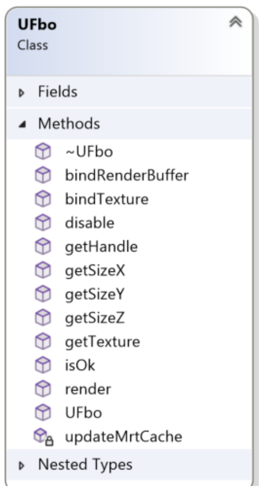

UFbo
Class
Fields
Methods
~UFbo
bindRenderBuffer
bindTexture
disable
getHandle
getSizeX
getSizeY
getSizeZ
getTexture
isOk
render
UFbo
updateMrtCache
Nested Types

Figure 10 UFbo class internal structure

UFbo is a class that makes it easier to use **FBO**s (Frame Buffer Objects). FBOs allow rendering output to be redirected into a dedicated buffer. This makes rendering faster because it is all done on the GPU.
This class allows to create an FBO and assign textures and render buffers to it. The method *bindTexture()* allows the texture to be associated with the FBO, it will be the colour buffer. The bindRenderBuffer() method creates the Render Buffer and associates it with the FBO. It will be the depth buffer. The *isOk()* method is useful for checking buffer states. The *render()* method simply binds the FBO and specifies the buffers to be used. In opposition, the *disable()* method unbinds the FBO. It is also possible to receive texture information such as the texture id from the *getTexture()* method and its size from the *getSize*()* methods.
Moreover, there’s a way to find out the id of the FBO via *getHandle().*

U3DRenderPipeline

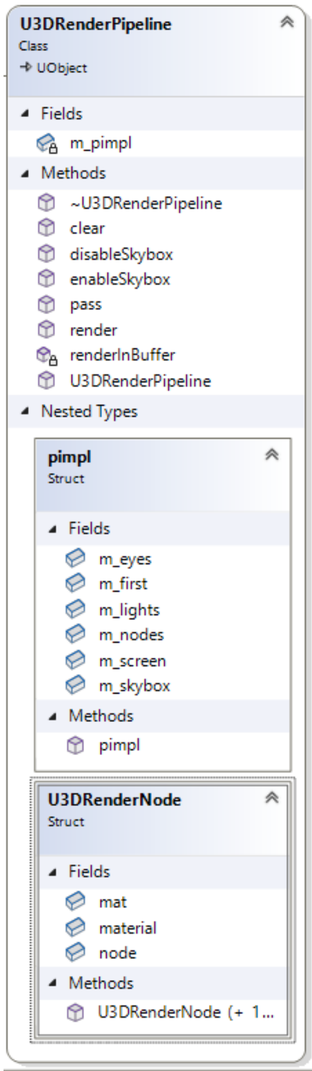

U3DRenderPipeline
Class
→ UObject
Fields
m_pimpl
Methods
~U3DRenderPipeline
clear
disableSkybox
enableSkybox
pass
render
renderInBuffer
U3DRenderPipeline
Nested Types
pimpl
Struct
Fields
m_eyes
m_first
m_lights
m_nodes
m_screen
m_skybox
Methods
pimpl
U3DRenderNode
Struct
Fields
mat
material
node
Methods
U3DRenderNode (+ 1...

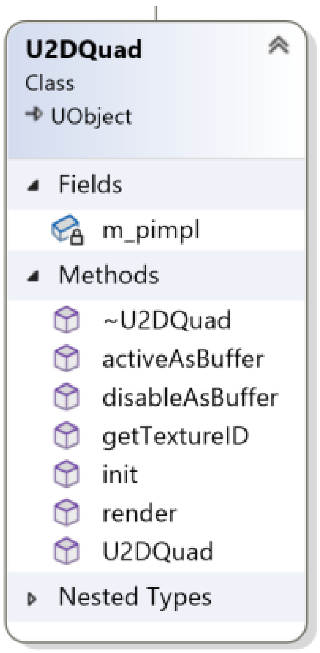
Figure 11 Inner structure of the U3DRenderPipeline

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 with 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 also can 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 an UMesh.
The same UNode pointer can be passed multiple times to the pipeline. By using different model views at each *pass()* call, it is possible to render the same object with different transformations or colors. This technique developers save a huge amount of memory which would be otherwise wasted by duplicating every UNode that needs to be rendered more than one time.
The *render()* method when called proceeds to draw on the screen by using the multipass render technique. In particular, for each of the ULight nodes passed it will render the scene with each "enabled" and then melt together the result. Depending on whether the OpenVR mode is enabled (by modifying the "conf/global.conf" file) the scene might be rendered in two buffers (for each "eye" or better said screen of the VR headset) or just one with specific attention to the various model views and camera matrices.
By using the *enableSkybox()* and *disableSkybox()* methods it's possible to configure a USkybox for the rendered scene. Specifically, the *enableSkybox()* method accept as parameter a UCubeMapTexture and cube modelview (principally used to scale the cube to accommodate the whole scene correctly).
It is possible to remove every element in the pipeline by using the *clear()* method.

Sphereculling

To achieve sphere culling, the mean value between the near and the far plane was calculated, the result was assigned as z coordinates with inverted sign of a vector position in Eye coordinates. The position vector defined above will be the center point of the rendering sphere with a radius equal to the average between the near and the far plane.
Each mesh present in the scene is characterized by a radius that, starting from the pivot of the object, will create a sphere containing all the vertices of the mesh.
If the sphere created by the mesh is located inside or intersects the rendering sphere the mesh will be displayed, if the mesh sphere does not touch and is not inside the rendering sphere the object will not be rendered.
To calculate the position of a given object in Eye coordinates the final world coordinates of the mesh is multiplied by the inverse of the camera and then the values related to the translation amateur of the object have been taken

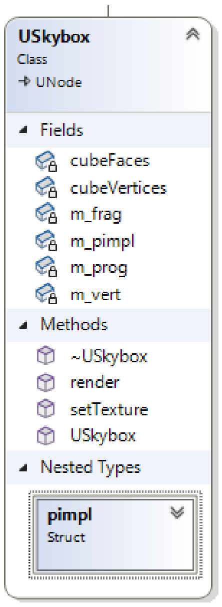
U2DQuad



The U2DQuad class allows to define quads in which rendering can be performed. So when a new U2Quad is created, must be defined the position on the screen (2D) and the orthographic matrix with which the contents of the buffers are to be rendered. The *init*() method initialises the quad where it can render using the UFbo class, which allows it to create the necessary buffers. The *activeAsBuffer*() method is required to bind the quad as a result of the render. After the whole scene has been rendered, the quad must be ubinded so the disableAsBuffer() method is called. When the Quad buffer contains the desired result, it can be rendered with the *render*() method.
This class is used in the rendering pipeline in order to render in a specific screen area. This also allows rendering in multiple areas such as those of the two eyes for the steroscopic view.

Figure 12 U2DQuad class internal structure

USkybox



USkybox is a class responsible for rendering a simple cube with an associated UCubeMapTexture. It is used as already mentioned internally to U3DRenderPipeline. The purpose of this scene object is to give developers the ability to add backgrounds to 3D scenes and create the illusion of a scene bigger than what it is. The rendered cube is customizable in its model view since it is in full effect a UNode. The principal transformation that might be applied is uniform scaling. The basic difference between a cube rendered using UMesh and USkybox is that the model view of the latter won’t be affected by translation transformations. Internally this class make use of a custom program shader to correctly attach the six textures of a UCubeMapTexture to each cube’s face correctly.

Figure 13 USkybox internal structure

UShader

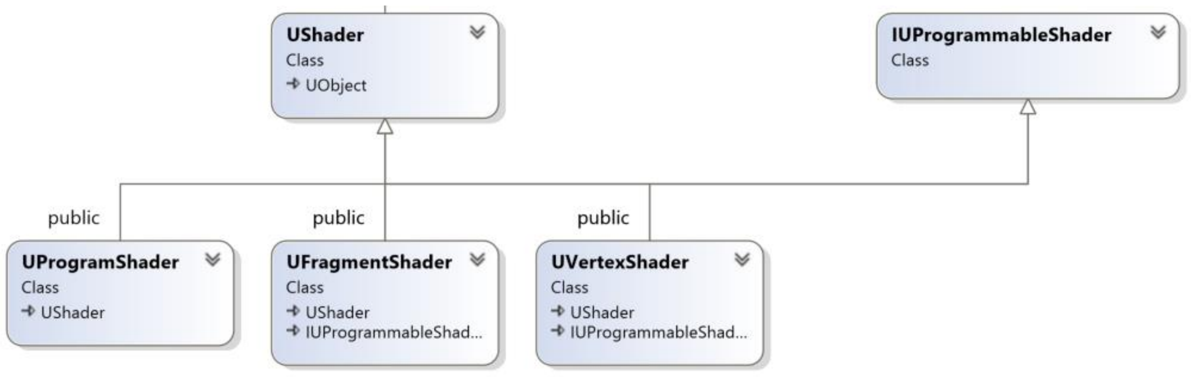
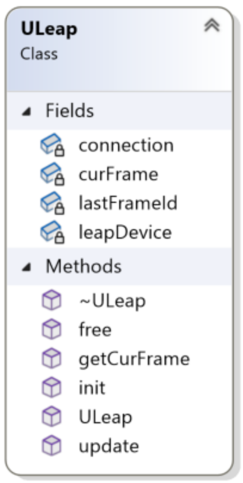


Figure 14 UShader hierarchy

UShader is the class that defines a generic shader. In fact, the child classes UProgramShader, UFragmentShader and UVertexShader are the ones that go more specific. A shader is a separate program that must be compiled at runtime and is loaded into the GPU and executed in order to render. The parent class UShader allows parameters to be set by bind the attribute names to a shader location with the *bind()* method and receive it with the *getParamLocation()* method. And then set these attributes of various types to their value via set methods.
The child class UVertexShader represents a vertex shader. A vertex shader takes in a stream of vertices and produces an output by applying calculations on the primes. In contrast, UFragmentShader represents a fragment shader. A fragment shader takes the interpolated output of the vertex shader is writes pixels to the output buffer. Since one is linked to the other, they are handled by OpenGL within single program. The latter is handled through the UProgramShader class, which allows a program shader to be built via the *build()* method by specifying a vertex and fragment shader. UProgramShader also keeps track of the active shader in the engine. To decide which program shader to use just call the *render()* method that sets it as active.
All these classes are for easier management of shaders. Due to the fact that these classes also implement IUProgrammableShader they can load shaders from file via the *loadFromFile()* method. The shaders are saved as resource files in the project. Different shaders are used for this project for different types of cases such as the default render, for each type of light, for screen quads or skyboxes.

ULeap



ULeap is a class for interfacing with the Leap Motion device. This device makes it possible to keep track of the position of a person's hands. First of all, it is necessary to connect to the device, and to do this, the *init()* method must be used. If the connection has worked, the position of the various hand and arm joints can be obtained using the *getCurFrame()* method. The latter method returns the positions since the last update made to obtain them. In order to obtain the most recent ones, simply call the *update()* method first.

Figure 15 ULeap internal structure

UHands

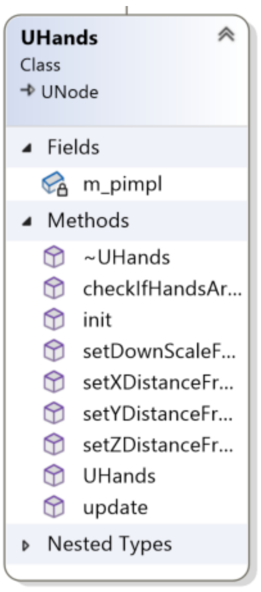


Figure 16 UHands internal structure

UHands is the class that handles the hand simulation within the scene. It does this by using the class ULeap to obtain information on the position of the user's arms and hands. The UHands class is a child of UNode so that it can be easily rendered in a scene by simply attaching it as a node. The latter, based on the main camera, will always position the hands in front of the user's view.
Since it uses an external device to function, it is necessary to initialise it with the *init()* method to ensure that it has a connection to it. In the same method it builds the model of the hands by placing spheres in the joints.
Then these 2 models will be connected to the UHands Node based on the number of hands detected by Leap, so they will be rendered. This control plus updating the position of the hands is done by the *update()* method. This is done in a different method than rendering because updating the position of the hands is time consuming. This is because it asks the Leap Motion to update (See ULeap).
The distance of the hands from the camera can also be set so that they can be configured to be at the same distance as the actual hands from the user's view.
The ratio of the transformation from mm (Leap Motion measurement) to m (virtual environment measurement) can also be changed. This is in case of models with different units of measurement.

Since the user will have to interact with some objects the simplest method is by figuring out whether his hand is within the radius of an object. The checkIfAndAreIn() method does this by checking whether the index of either hand has entered the space of the object passed as a parameter.

OpenVR Framework

The OpenVR Framework is implemented in Utopia by using the internal OVR class, which is a simple wrapper around the essentials function of OpenVR. It gets initialized in the *init()* method of Utopia only when in the “conf/global.conf” file is present the “openvr=true” setting. The class helper has multiple methods that are used to obtain model view, projection and eye to head matrices that are used in U3DRenderPipeline to render correctly in each VR headset screen a 3D scene. The class has other methods that let, for example, handle additional features such as joystick controllers, the tracking system and such.

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. TowerCraneSimulator also implements 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.
With virtual environment integration, the user's view can be moved with the movement of the head when wearing the headset. While to operate the crane, simple spheres have been inserted into the cabin that allow it to be operated as if they were buttons. The user will simply have to insert his or her index finger into one of the spheres in order to perform a movement of the crane.
The client is composed of 4 main classes: Box, BoxesManager, Tower and ClientUtility.

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 rendering pipeline, 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, and the 3D meshes are rendered. In OpenVR mode, only one camera is available, and it is placed inside the cabin.

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 assessed with various models and it has shown to be capable of handling copious 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 satisfactory 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,
Adriano Cicco, Fabio Crugnola, Igor Fontanini