Crane

Virtual Reality - SUPSI 2022/2023

Mattia Dell'Oca, Manuele Nolli

Introduction

Crane is a simulation video game written in C++ that allows the user to control a crane through a *Virtual Reality Head Mounted Display* and an *Optical Hand Tracking* module. The user is able to rotate the *jib* (working arm) and move the trolley back and forth to position the hook correctly on a container to pick it up and move it around. In addition, it is possible to lower and higher the crane's hook to pick up a scene object and move it to another position.

This project consists of two different components bound together:

- **3D graphics engine**, a dynamic library (.dll) used to provide an API and a series of high-level classes that interact with OpenGL and other internal libraries.

- **Crane client,** which uses the graphics engine to create the entire *Crane* environment, such as the 3D model loading, the application logic and the user interaction handling.

Main features

- VR and Optical Hand Tracking
- Multiple lights with Multiple-pass Rendering (Blinn-Phong reflection model)
- Sphere culling
- Skybox support
- Shadows support
- Interactions such as arm rotation, trolley and hook movement
- Hooking objects in the scene

External dependencies

Libraries

- FreeGLUT (*Free OpenGL Utility Toolkit*), https://freeglut.sourceforge.net
- GLM (*OpenGL Mathematics*), https://github.com/g-truc/glm
- FreeImage, https://freeimage.sourceforge.io
- Leap Motion, https://www.ultraleap.com

- OpenVR, https://github.com/ValveSoftware/openvr
- Glew (OpenGL Extension Wrangler), https://glew.sourceforge.net

Tools

- Doxygen v1.9.7, https://www.doxygen.nl/index.html

Development

The development of the Crane project involved creating two main components: the 3D graphics engine and the Crane client.

The 3D graphics engine is a dynamic library that provides an API and high-level classes for interacting with OpenGL and other internal libraries. It was developed using C++ and was designed to be used by other applications to easily incorporate 3D graphics with or without VR functionality.

Indeed, as will be explained in more detail further in the paper, the developed system can operate in *Standard* mode, where 3D objects can be rendered in a simple 2D window, and *Stereoscopic* mode, where interactions via VR and hand tracking are possible.

The Crane client uses the graphics engine to create the entire Crane environment, including the 3D model loading, the application logic, and the handling of user interactions.

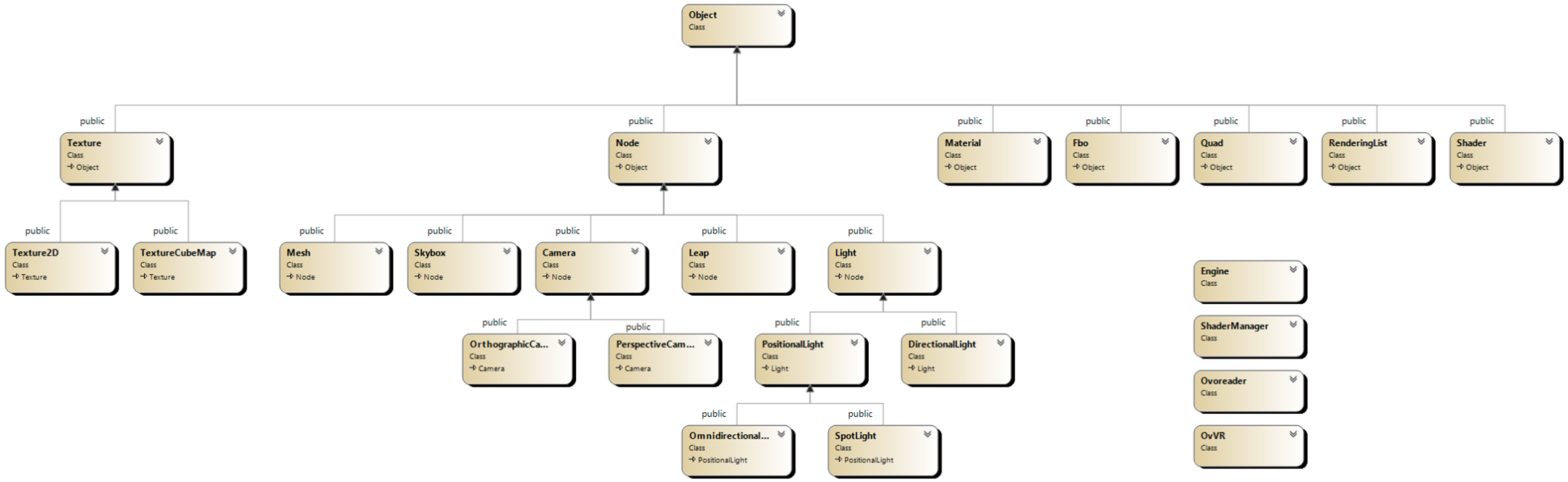
Doxygen was used during the development process of the graphics engine and the client with the aim of generating readable high-quality code documentation.

Graphics Engine

The development of a 3D graphics engine was a crucial aspect of the Crane project. The engine, implemented in C++, is a dynamic library that offers an API and high-level classes for interacting with OpenGL and other internal libraries. It was designed to be easily incorporated into other applications, providing a simple and flexible solution for adding 3D graphics functionality.

The graphics engine was developed in parallel with the Crane project, with the goal of creating an intuitive interface. The following chapters provide further details on the structure of the project and the implemented features.

UML Diagram



All classes, methods, and attributes have been documented in detail using Doxygen comments. The generated documentation can be accessed in the project repository, within the "docs" directory.
Note: This UML is truncated thus allowing the reader to understand the general structure of the graphics engine codebase. The original UML diagram, with attribute and methods information is attached to this document.

Features

The engine has been designed in such a way that allows for the creation of any desired 3D environment through the use of simple APIs. This design allows for the rapid creation of a 3D scene within a window with minimal lines of code required for implementation.

In addition, two engine execution modes are possible:

- *Standard*: simple 3D rendering of a model with keyboard interactions
- *Stereoscopic*: VR rendering using OpenVR and interactions with Leap Motion

List of the main features of the graphics engine:

- Blinn-Phong lighting model with unlimited light support (Multi pass rendering)
- VR and hand tracking support
- VBO and VAO based rendering to increase performance
- Customized shaders
- Skybox rendering
- Sphere culling to improve performance
- Load OVO files, which are 3D scenes created in 3ds Max and exported using a custom plugin, and import meshes, textures, materials, and lights into the engine
- Implement hard shadows, which are created using a linear application that scales the mesh and projects all vertices onto the XZ plane, causing a black shadow to appear under the object
- Set up special callbacks, such as timers, collision checking and display functions, to control the behavior of the engine
- Window interaction, including clearing the screen, swapping buffers and changing the background color
- Acquire window information such as the height and width.

Technical information

Shaders

The engine was developed with the support of OpenGL version 4.4. This made it possible to develop custom shaders for rendering objects. More specifically, 10 shaders were written:

- Vertex shader for mesh rendering
- 3 different Fragment shaders, one for every light type
- Vertex and Fragment shader for Leap Motion (hand tracker)
- Vertex and Fragment shader for sky box rendering
- 2 utility passthrough shader

VBO, VBA

In order to increase performance, it was decided to store the mesh information within the GPU memory buffers, i.e. the Vertex Buffer Objects, at application start-up. The data saved per mesh are vertices, normals, texture coordinates and face information. Thus a total of 4 VBOs per mesh, which for simplicity have been grouped into a single VAO (per mesh).

FBO (VR)

Since it is necessary for a Virtual Reality system with head mounted display (HMD) to create two different images per frame that differ by an interocular distance (IOD), FBOs (Frame Buffer Object) have been used. They allow the output to be redirected and saved in GPU memory, without the need to use the front buffer. In addition, OpenVR library is well suited to use FBO, since it is required to pass two images, one for each eye.

Sphere Culling

To improve the performance of the graphics engine, it was decided to use the technique of sphere culling. It is based on the fact that each mesh is enclosed by a sphere box. Moreover, a non-visible sphere is created from the near plane to the far plane, in the direction where the camera is looking. When rendering, a check on each mesh is performed to evaluate whether or not the latter is contained in the previously mentioned sphere. This avoids rendering non-visible meshes and improves performance.

Skybox

In order to improve the immersion and presence inside the simulation a skybox has been implemented. A skybox is a cuboid texture that gives the impression of being in a larger space. The texture cubemap can be defined client-side.

Optical hand tracking

In order to create an immersive Virtual Reality experience, a hand tracking system has been integrated. This is made possible through the use of an optical tracking device, the Leap Motion Controller. The sensor tracks the hand's position with a good field of view and creates a 3D model of vertices representing the fingers, palm, wrist and elbow of each hand of the user. Through the dedicated APIs, the engine can acquire the hands position of the user and represent them in the Virtual Environment. A callback can be defined in the client to

determine how to handle collision between hands and scene objects. It has been decided to only check collisions for the index fingers.
This feature is only available in Stereoscopic mode.

Engine API structure

The API for the engine allows users to interact with the key features of the graphics engine. The subsequent chapters provide a list of all the API methods, along with a brief description of their functions. Detailed documentation on the engine can be found in the "docs" directory within the project repository.

Graphics APIs

Initialize the engine:
```
bool init(const char* title, unsigned int width, unsigned int
height, int* argc, char** argv, ‘Stereoscopic’ or ‘Standard’)
```
Load a 3D model from a file:
```
void load(std::string filepath, std::string texturesDir)
```
Load a 3D model from a scene graph:
```
void load(Node* root)
```
Create a skybox loading its textures:
```
void loadSkybox(const std::string& filepath, const std::string*
cubemapNames)
```
Clear the scene graph:
```
void clear()
```
Swap the front and back buffers:
```
void swapBuffers()
```
Free the resources used by the engine:
```
bool free()
```
Render the scene:
```
void render(PerspectiveCamera* camera)
```
Run the engine loop:
```
void run(void (*updateFunction)())
```
Set graphics settings profile:
```
void setGraphics(EngineGraphics& g)
```
Set player height:
```
void setPlayerHeight(float playerHeight);
```

Callbacks

Set the keyboard callback function:
```
void setKeyboardFunction(void (*callback)(unsigned char, int,
int))
```
Set the special keyboard callback function:
```
void setSpecialKeyboardFunction(void (*callback)(int, int, int))
```
Set the timer callback function:
```
void setTimerFunction(void (*callback)(int))
```
Set the collision callback function:
```
void setCollisionCallback(void (*callback)(void*))
```

Utility functions

Search and return the first node found in the scene with the given name:
`Node* getNode(std::string name)`
Returns the current window height. The engine must be initialized before calling this method:
`unsigned int getWindowHeight()`
Returns the current window height. The engine must be initialized before calling this method
`unsigned int getWindowWidth()`
Check if the engine is running:
`bool isRunning()`
Force rendering:
`void redisplay()`
Set the background color of the window:
`void setBackgroundColor(float r, float g, float b)`
Enable wireframe rendering:
`void enableWireframe()`
Disable wireframe rendering:
`void disableWireframe()`

How to use it

Pseudocode examples will be shown below.

Basic functionality

The following code represents the minimum functionality of the engine i.e. the creation of the window, file .ovo loading and entering the main loop of the 3D scene:

```
def display:
      Engine::clear()
      Engine::render(perspectiveCamera)
      Engine::swapBuffer()

def main:
      Engine::init(name, windowDimension)
      Engine::setBackgroundColor(0.647f, 0.898f, 1.0f);
      Engine::load(pathOvoFile, pathTextureDir)
      Engine::run(display)
      Engine::free()
```

Callback integration

The following code represents the extension of the “Basic functionality” and includes the callbacks setting. Only the additional code is shown:

```
def keyboardCallback(key, x, y):
      if key == "w":
            ...
```

```
    ...

def specialKeyboardCallback(key, x, y):
    if key == KEY_UP:
        ...
    ...

def collisionCallback(data):
    glm::mat4 matrix = data;
    if distance(matrix, object) < object.radius
        ...
    ...

def timerCallback(value):
    fps = frame
    frame = 0

def main:
    Engine::init(name, windowDimension)
    ...
    Engine::setKeyboardFunction(keyboardCallback);
    Engine::setSpecialKeyboardFunction(specialKeyboardCallback);
    Engine::setTimerFunction(timerCallback);
    Engine::run(display)
    ...
```

Crane client

The Crane client is the application that brings the entire Crane environment to life. It uses the 3D graphics engine to load the 3D Ovo models, handle user interactions, and manage the application logic. The client was developed using C++ and is available for Windows operating systems. In the following sections, the development process for the Crane client will be discussed in more detail, including the challenges faced and the solutions implemented by the team.

Game description

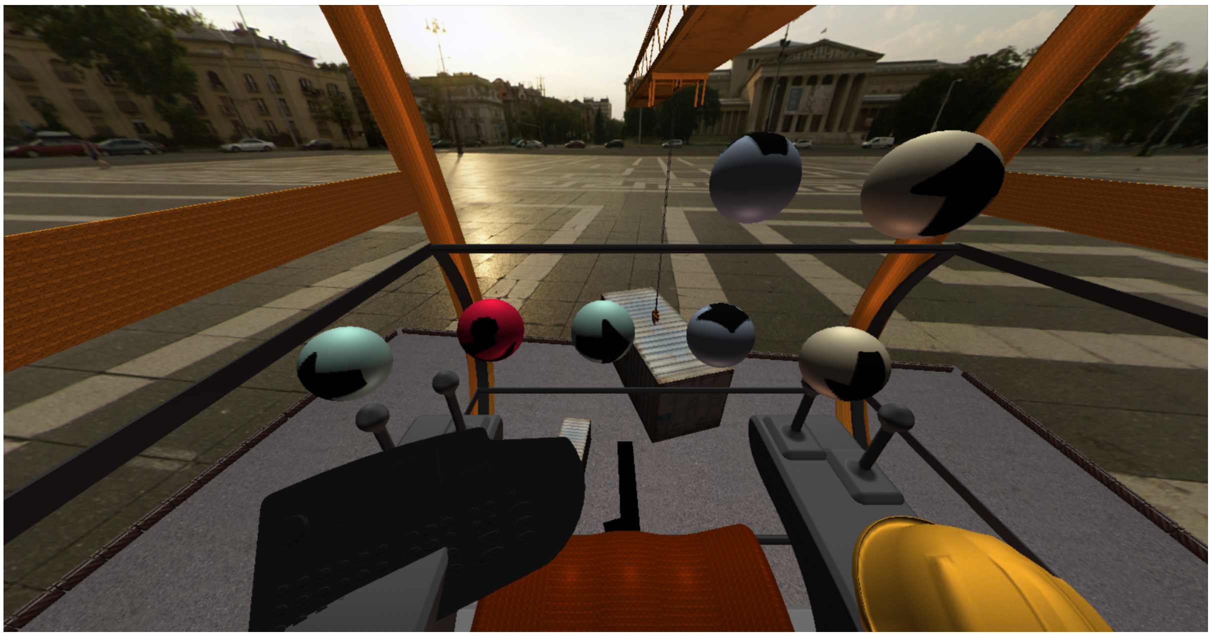
In the virtual crane simulation, the operator has the ability to control the movement of the crane with a high degree of accuracy. The controls for rotating the crane, moving the trolley back and forth on the working arm, and adjusting the height of the hook are all achievable in two methods, depending on which mode is the engine running:

- *Standard:* keyboard mapping
- *Stereoscopic:* interaction with spheres

For simplicity, the execution mode can be set from an external file called *app.config*.

The virtual environment includes two containers that can be picked up and moved using these controls. To enhance the user's experience, the simulation offers three different camera views (changeable only in standard mode). The first person view offers a realistic perspective from within the operator's cabin (Stereoscopic main camera). The hook view provides a close-up of the hook and its surroundings for precise movements. The dynamic camera allows for movement around the virtual environment using WASD keys and adjusting the view using the arrow keys. This camera is particularly useful for examining the containers and the surrounding area in greater detail.

This is a game screenshot:



Game engine integration

The engine handles much of the work, utilizing only a few API calls to build the dynamic 3D environment. In fact, most of the logic is implemented through callbacks, such as timer and display. The engine APIs are primarily used for tasks like loading the 3D scene and associated textures, adjusting graphics settings, manipulating scene graph nodes, and initiating rendering. Overall, the Crane relies on a minimal number of engine API calls to function effectively.

Application flow

The code in main.h imports all the necessary components from the engine library to be used in the virtual crane simulation. The engine has been designed to minimize the amount of repetitive code that needs to be written, allowing developers to focus on creating the gameplay and other features of the simulation.

The first thing it does is initialize the engine with a window title, window width and height, command line arguments and the execution mode (Standard or Stereoscopic). Then, based on the execution mode decided, the client will set up the interaction methods.

Next, the code sets a default graphics profile with specified texture filter, wrap, mipmap, and anisotropic filtering settings. It then sets these graphics settings in the engine and loads in the assets for the simulation, including the crane and its various components and the skybox information.

The code then searches for nodes in the scene graph with specific names and assigns them to variables. It also disables shadow casting for the plane, and, if Standard mode is set, creates two additional camera nodes to be used in the simulation. These camera nodes are added as children to their respective objects in the scene and linked together in a circular list.

Finally, the code starts the rendering loop and, when the application is closed, cleans up memory by freeing the engine and deleting the camera nodes when it is finished.

Hand’s collisions handling

The engine allows the client to specify how to handle collisions between the 3D models representing the user's hands and scene objects. At each frame, a check for both index fingers is performed to determine whether or not they are colliding with specific objects. The client can determine which object should be considered for collision handling and the action to perform on collision.
In the crane simulation the user can interact with a total of 7 buttons placed inside the cabin: each button represents an action performed by the crane. This feature is only available in Stereoscopic mode.

Cyclic camera logic

The development team has decided to implement a custom linked list of CameraNode objects in order to eliminate the need for mapping individual keyboard keys to each camera. At the moment, this is a method used only for *Standard* mode since the VR experience is obtainable from a single point of view. This linked list will function as a cyclic list, with the last element (tail) always pointing to the head of the list (first element). This will allow the user to use the same keyboard input to view the scene through different cameras. To proceed to the next camera, the following line of code can be used:

```
currentCamera = currentCamera->getNext()
```

Attachments

- Crane source project
- Graphics engine source project (Visual Studio)
- Pre-compiled *engine.dll, crane.exe*
- Graphics Engine UML
- Doxygen graphics engine documentation