Crane

Computer Graphics - SUPSI 2022/2023

Luca Di Bello, Mattia Dell'Oca, Manuele Nolli

Introduction

Crane is a simple simulation video game written in C++ that allows the user to control a crane to interact with objects present in the virtual environment. 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. With specific keymaps 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, .so) 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

Crane is available for both Linux and Windows.

Main features

- User is able to rotate the arm, move the trolley and the hook
- Textured environment using *mipmapping* technique
- Multiple lights and shadows support (Blinn-Phong reflection model)
- Multiple available static cameras and one dynamic camera
- Real-time simulation (always > 24 FPS)
- The hook can pick up and move objects present 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/
- Google Test, http://google.github.io/googletest/

Tools

- Doxygen v1.9.5, 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 functionality.

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. The client was also developed using C++ and is available for both Linux and Windows operating systems.

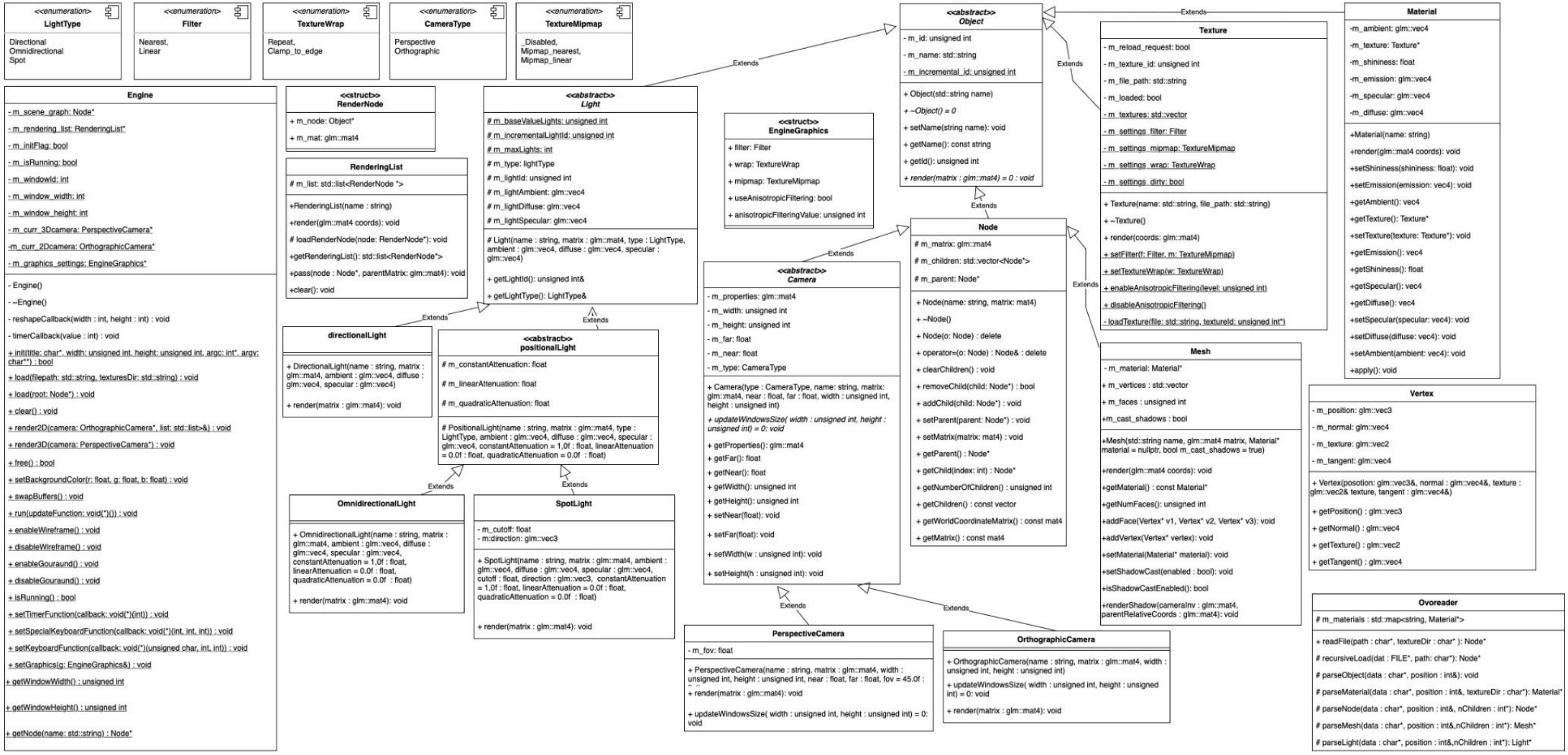
Both the graphics engine and the Crane client were developed using a combination of coding and testing and, during the development process, the team used Doxygen to generate readable code documentation, and libraries such as FreeGLUT, GLM, and FreeImage. Once the code was complete and tested, it was compiled into the final products: the dynamic library for the graphics engine and the executable file for the Crane client.

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 a user-friendly and 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 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, which are built on top of OpenGL. This design allows for the rapid creation of a 3D scene within a window with minimal lines of code required for implementation.

List of the main features of the graphics engine:

- Set up special callbacks, such as timers and display functions, to control the behavior of the engine
- Interact with windows, including clearing the screen, swapping buffers and changing the background color
- 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
- Ability to change graphics settings on the fly, automatically reload textures if major changes are detected
- 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
- Blinn-Phong lighting model, providing the user the ability to switch in real time between two kind of shading: flat shading and gouraud shading
- Interact with the user interface, such as displaying text on the screen
- Acquire window information such as the height and width.

Graphics settings

The engine enables users to adjust graphics settings on the fly. The following options are available:

- Change filtering: nearest filtering, linear filtering, or anisotropic filtering (if supported) with a related value
- Enable or disable mipmapping to change the texture mapping technique. If this setting is changed, the engine will automatically reload the textures
- Change the texture wrap technique to either repeat or clamp to edge

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)
```
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)
```
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 render3D(PerspectiveCamera* camera)
```
Render the 2D information:
```
void render2D(OrthographicCamera* camera, const
std::list<std::tuple<std::string, int>>& list)
```
Run the engine loop:
```
void run(void (*updateFunction)())
```
Set graphics settings profile:
```
void setGraphics(EngineGraphics& g)
```

Link 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 mouse callback function:
```
void setMouseFunction(void (*callback)(int, int, int, int))
```
Set mouse wheel callback function:
```
void setMouseWheelFunction(void (*callback)(int, int, int, int))
```
Set the mouse motion callback function. This event is fired every time the mouse moves on the screen:
```
void setMouseMotionFunction(void (*callback)(int, int))
```

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()`
Enable Gouraund shading:
`void enableGouraund()`
Disable Gouraund shading:
`void disableGouraund()`

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::render3D(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 timerCallback(value):
      fps = frame
      frame = 0

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

Rendering 2D

The following code represents the extension of the “Basic functionality” and includes the rendering of 2D information. Only the additional code is shown:

```
list<string ,int> information2D       % string = text to write
                                  % int = height position on screen

def display:
      Engine::clear()
      Engine::render3D(perspectiveCamera)
      information2D.clear()
      populate information2D
      Engine::render2D(othographicCamera,information2D)
      Engine::swapBuffer()
```

Graphic settings

The following code represents the extension of the “Basic functionality” and includes the personalization of the graphic settings. Only the additional code is shown:

```
def main:
      Engine::init(name, windowDimension)
      ...
      EngineGraphics profile
      profile.filter = Filter::LINEAR
      profile.wrap = TextureWrap::CLAMP_TO_EDGE
```

```
profile.mipmap = TextureMipmap::MIPMAP_LINEAR
profile.useAnisotropicFiltering = true
Engine::setGraphics(profile)
Engine::load(pathOvoFile, pathTextureDir)
...
```

Unit testing

Unit testing is an important part of the development process for any software project and is no exception for the engine. To ensure proper functionality and high quality, a subproject called "engine_test" utilizing the Google Test framework has been created. Various aspects of the engine are systematically tested to ensure proper operation.

The range of functions covered includes basic operations, such as creating, adding, and removing elements within the engine, as well as interaction with the scene graph, to ensure the core functionality is working correctly and is free of defects.

In addition, non-essential operations involving OpenGL and FreeGLUT are also tested to ensure proper interaction with these third-party libraries and the ability to take advantage of their features. Control operations, such as the maximum number of lights supported by OpenGL, are focused on, as they are critical for the performance and functionality of the engine.

The unit testing performed on the engine is crucial for ensuring reliability, efficiency and high quality, and issues are caught early in the development process for them to be fixed before becoming major problems.

Crane client

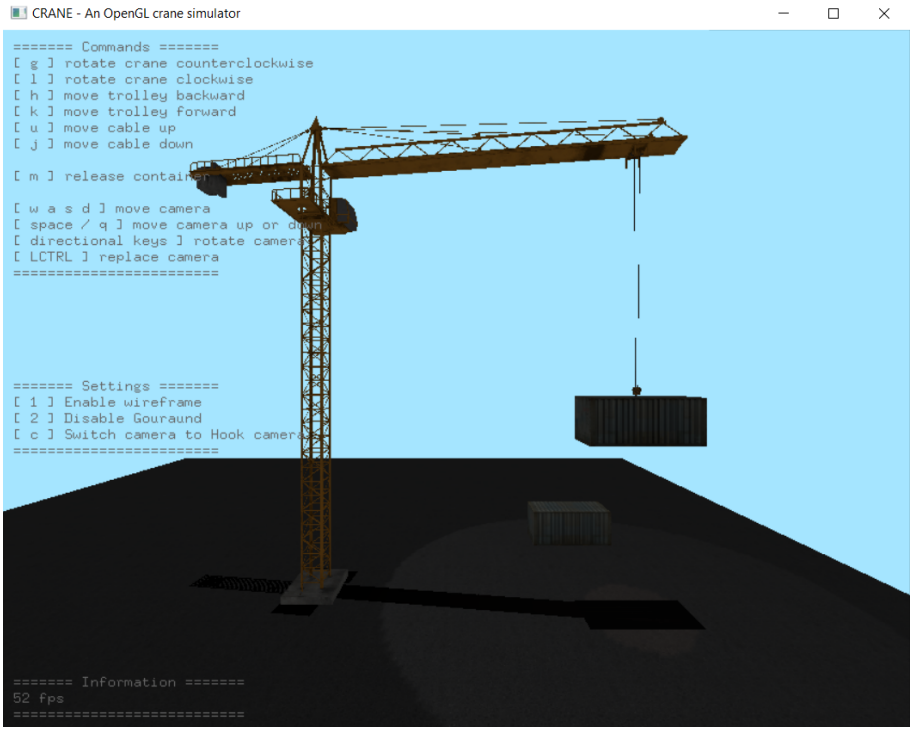
The Crane client is the application that brings the entire Crane environment to life. It uses the 3D graphics engine to load the 3D Ovo models, handle user interactions, and manage the application logic. The client was developed using C++ and is available for both Linux and Windows operating systems.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

Game description

In this 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 using the computer's keyboard. 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. The first person view offers a realistic perspective from within the operator's cabin. 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.

The game UI always shows the user all the keybindings for the various controls and the current frame rate (in FPS).

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 this virtual crane simulation. These components include the base engine class, various camera classes (orthographic and perspective), node and light classes (directional, omnidirectional, and spot). With these components imported, the setup of the game can be concise and efficient. 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, and command line arguments. Then, it sets up keyboard and special keyboard callback functions, as well as a timer callback function. The callback functions allow the engine to respond to input and perform certain actions at regular intervals.

Next, the code sets the background color of the window and creates 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.

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 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.

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. 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, both Visual Studio and Codeblocks
- Graphics engine source project, both Visual Studio and Codeblocks
- Pre-compiled *engine.dll, engine.so, crane.exe* and *crane*
- Graphics Engine UML
- Doxygen graphics engine documentation