University of Sarajevo
Faculty of Electrical Engineering
Department of Data Science and Artificial Intelligence

Laboratory Exercise 3

Artificial Intelligence

Authors: Benjamin Bandić, Nedim Bečić
Professor: Izudin Džafić

Contents

1 Goal of the Exercise

The goal of this laboratory exercise is to build a graphical user interface (GUI) and animation for the port logistics simulation developed in the previous exercise. Students will apply their understanding of Object-Oriented Programming to a visual context using the `natId` C++ framework.

This exercise will provide hands-on experience with core GUI programming concepts, including event-driven programming, 2D graphics rendering, and animation loops. By the end of this exercise, students will have a deeper understanding of how to separate application logic from its visual representation and how to manage a real-time visual simulation state.

2 Theoretical Introduction to `natId` Visualization Concepts

2.1 Understanding CMake: The Build System

Before writing C++ code, it's essential to understand how it's compiled and linked into a runnable application. For this, we use CMake.

What is CMake?

CMake is a cross-platform, open-source build system generator. It does not build the project itself; instead, it generates the project files for a specific environment (e.g., Xcode projects on macOS, Visual Studio solutions on Windows, or Makefiles on Linux). This allows developers to maintain a single, simple set of build scripts that can be used on any platform.

How does it work?

You write instructions in a file named `CMakeLists.txt` (or `.cmake`). The key commands you will use are:

- `set()`: Creates a variable.
- `file(GLOB ...)`: Finds all files matching a pattern and stores them in a variable.
- `add_executable()`: Defines your final program and lists the source files needed to build it.
- `target_link_libraries()`: Links your program with external libraries, such as the `natId` framework libraries (`mu` and `natGUI`).

```
set(Lab3_NAME Lab3)

# Find all of our project's source and header files
file(GLOB Lab3_SOURCES ${CMAKE_CURRENT_LIST_DIR}/src/*.cpp)
file(GLOB Lab3_INCS  ${CMAKE_CURRENT_LIST_DIR}/src/*.h)

# Create the executable program
add_executable(${Lab3_NAME} ${Lab3_SOURCES} ${Lab3_INCS})
# Link our program with the necessary natId libraries
target_link_libraries(${Lab3_NAME}
    debug ${MU_LIB_DEBUG}
    debug ${NATGUI_LIB_DEBUG}
    optimized ${MU_LIB_RELEASE}
    optimized ${NATGUI_LIB_RELEASE}
)

# Use natId helper functions to configure it as a GUI app
setTargetPropertiesForGUIApp(${Lab3_NAME} ${Lab3_PLIST})
```

Example of a CmakeLists.txt file

2.2 gui::Canvas

The `gui::Canvas` is the fundamental building block for custom 2D graphics in `natId`. Think of it as a digital board. To create a custom visualization, you must create a new class that inherits from `gui::Canvas`.

Key Concepts:

- `onDraw(const gui::Rect& rect)`: A `protected virtual` method that you must override. The framework calls this method whenever the canvas needs to be repainted. All drawing code must be inside this method.
- `reDraw()`: A `protected` method used to schedule a repaint of the canvas. Because it is protected, you must create a public wrapper (e.g., `refresh()`) to call it from outside classes.
- `onResize(const gui::Size& newSize)`: A `virtual` method called when the canvas size changes. This is the only reliable way to get the current width and height for dynamic layouts and centering. To use it, you must first call `enableResizeEvent(true);` in your canvas's constructor.

```
#include <gui/Canvas.h>

class SimulationCanvas : public gui::Canvas {
protected:
    int canvasWidth = 0;
    int canvasHeight = 0;

    void onResize(const gui::Size& newSize) override {
        this->canvasWidth = newSize.width;
        this->canvasHeight = newSize.height;
```

```
        this->reDraw();
    }

    void onDraw(const gui::Rect& rect) override {
        // ... all drawing code goes here ...
    }

public:
    SimulationCanvas() {
        // This is required to make onResize work.
        enableResizeEvent(true);
    }

    // A public method that allows outside classes to request a redraw.
    void refresh() {
        this->reDraw();
    }
};
```

A skeleton for our SimulationCanvas class.

2.3 `gui::Shape`, `gui::Rect`, and `gui::CoordType`

The `gui::Shape` class is a versatile object for creating and drawing geometric figures.

Key Concepts:

- `gui::CoordType`: A special data type (like `float` or `double`) used for all coordinate and size values in the `natId` GUI. For type safety, you should use `gui::CoordType` for any variable that will be used in a `gui::Rect` or `td::Point`.
- `gui::Rect`: Represents a rectangular area. Its constructor takes four coordinates: `(left, top, right, bottom)`. To create a rectangle from a position and size, you must calculate the `right` and `bottom` coordinates.
- `gui::Shape`: An object that can be configured to represent a shape.
  - `createRect(rect)`: Configures the shape to be a rectangle.
  - `drawFill(color)`: Fills the shape's interior with a solid color.
  - `drawWire(color)`: Draws only the shape's outline.

```
// Inside the onDraw method
gui::Shape myShape;
gui::CoordType x = 50.0;
gui::CoordType y = 50.0;
gui::CoordType width = 100.0;
gui::CoordType height = 70.0;

// Correctly create a Rect using (left, top, right, bottom)
gui::Rect myRect(x, y, x + width, y + height);

// Configure and draw the shape
myShape.createRect(myRect);
myShape.drawFill(td::ColorID::Red);
```

Correctly creating and drawing a rectangle.

2.4 Animation and Event Handling

Animation in `natId` is built directly into the `gui::Canvas` and should be used instead of external timers for visual updates.

Key Concepts:

- `startAnimation()`: A public method on `gui::Canvas` that tells the framework to call your `onDraw()` method repeatedly at a high frame rate.
- `stopAnimation()`: Stops the repeated calls to `onDraw`.
- `isAnimating()`: Returns `true` if the animation loop is running.
- **The Animation Loop:** The standard pattern is to place a simulation update function (e.g., `step()`) at the beginning of `onDraw`. This function is responsible for updating the positions of your visual objects for the current frame.
- `onClick(gui::Button* pBtn)`: User interaction is handled by overriding event methods. This is where you would call `startAnimation()` or `stopAnimation()`.

```
class MainView : public gui::View {
protected:
    SimulationCanvas canvas;
    gui::Button btnStart;

public:
    MainView() { /* ... setup buttons and layouts ... */ }

    bool onClick(gui::Button* pBtn) override {
        if (pBtn == &btnStart) {
            if (canvas.isAnimating()) {
                canvas.stopAnimation();
            } else {
                canvas.startAnimation();
            }
            return true;
        }
        return false;
    }
};
```

A simplified animation-ready MainView.

```
void onDraw(const gui::Rect& rect) override {
    // If the animation is running, update object positions before
   drawing.
    if (isAnimating()) {
        step();
    }
    // ... all drawing code follows ...
}
```

The animation hook inside SimulationCanvas::onDraw.

3 Putting It All Together: Project Structure

To create the final application, several classes must work together. Here is a breakdown of the key files and their roles, along with their source code.

3.1 `main.cpp`: The Entry Point

This file is the starting point of the entire application. Its only job is to create an instance of your `Application` class and run it.

```
#include  Application.h
#include <gui/WinMain.h>

int main(int argc, const char * argv[])
{
    Application app(argc, argv);
    app.init( EN );
    return app.run();
}
```

3.2 `Application.h`: The Application Class

This class inherits from `gui::Application` and is responsible for creating the main window of your program.

```
#pragma once
#include <gui/Application.h>
#include  MainWindow.h

class Application : public gui::Application {
protected:
    gui::Window* createInitialWindow() override {
        return new MainWindow();
    }
public:
    Application(int argc, const char** argv) : gui::Application(argc,
   argv) {}
};
```

3.3 `MainWindow.h`: The Main Window

This class inherits from `gui::Window` and acts as the main frame of your application. Its primary role is to create and display your central view, which will contain all other UI elements.

```
#pragma once
#include <gui/Window.h>
#include  MainView.h

class MainWindow : public gui::Window {
protected:
    MainView view;
public:
    MainWindow() : gui::Window(gui::Geometry(50, 50, 1440, 800)) {
        this->setTitle( Port Logistics Simulation );
```

```
        this->setCentralView(&view);
    }
};
```

3.4 MainView.h: The Conductor

This is the most important class for control and logic. It inherits from `gui::View` and is responsible for:

- Creating and managing the layout (buttons and the canvas).
- Holding the simulation state and all logical objects (`Ship`, `Port`).
- Handling button clicks (`onClick`).
- Owning the visual data (`visualContainers`, `visualTrucks`).
- Containing the core `step()` function that drives the simulation's logic.

```
#pragma once
#include <gui/View.h>
#include <gui/Button.h>
#include  SimulationCanvas.h
// ... other includes

enum class SimState { Idle, UnloadingToxic, /* ... */, Done };

class MainView : public gui::View {
protected:
    Ship ship;
    Port port;
    std::vector<VisualContainer> visualContainers;
    std::vector<VisualTruck> visualTrucks;
    SimulationCanvas canvas;
    gui::Button btnStart, btnStep, btnReset;
    SimState currentState = SimState::Idle;
    // ... other members

public:
    MainView();
    void resetSimulation();
    void step();
    bool onClick(gui::Button* pBtn) override;
};
```

A snippet of MainView.h showing its structure. The full logic is in the provided solution.

4 Main Task: Visualizing the Port Logistics Simulation

The main goal is to create a visual representation of the direct unloading portion of the port simulation. You will use the `natId` framework to draw the scene and animate the movement of priority containers from the ship directly to trucks.

- **Task 1: Create the Simulation Scene**

  - Create a `SimulationCanvas` class that will be your main drawing area.
  - Inside `onDraw`, define and draw two distinct rectangular zones for the **Ship** and **Trucks**.
  - Use `gui::DrawableString` to add labels above each zone.
  - Load and draw a static `ship.png` image anchored to the bottom of the "Ship Cargo" zone.
  - The final layout should be managed by a `MainView` class containing the canvas and control buttons ("Start/Pause", "Step", "Reset").

- **Task 2: Visualize the Initial State**
  - Create a `resetSimulation()` method in your main view or canvas.
  - This method should create the logical `Ship` and `Container` objects.
  - For each logical container, create a corresponding `VisualContainer` object, storing its initial position on top of the ship image in two neat columns.
  - Use different images for each type of container.

- **Task 3: Animate Priority Unloading**
  - In your view that manages the simulation, create a `step()` method and a `SimState` enum to manage the simulation flow.
  - The `step()` function will be called repeatedly by the animation loop. It should first animate any object that is currently moving. If no object is moving, it should decide which container to move next.
  - Implement the logic for the `UnloadingToxic` and `UnloadingRefrigerated` states.
    * For each container in these categories, a new `VisualTruck` should be created in the "Trucks" zone.
    * The container's animation `targetPos` should be calculated to be on the bed of the truck.
    * The simulation must only move **one container at a time**.

5 Bonus Tasks

5.1 Bonus Task 1: Visualizing the Sorting Yard

Extend your simulation to visually implement the sorting yard logic from the previous lab for non-priority containers.

- Add a new rectangular zone **Sorting Yard**, between the two existing ones.
- **Update Animation Logic:** Create a new `SimState` called `UnloadingToSort`. After all priority containers are unloaded, the simulation should enter this state. All `OrdinaryContainer` and `FragileContainer` objects should be animated from the ship to the "Sorting Yard" zone.

- **Visual Stacking:** The `targetPos` for these containers should be calculated so they form neat stacks within the sorting yard's boundaries, stacking from the bottom up and adhering to the weight-stacking rule (a heavier container cannot be placed on a lighter one).
- **Loading from Yard:** Create a `LoadingFromSort` state. In this state, your logic should find the heaviest available container from the top of any stack, create a "Standard" truck, and animate the container from the yard to the truck.

5.2 Bonus Task 2: Visualizing the Departure Terminal

Enhance the simulation to include the departure terminal and inspection logic from the previous lab.

- **Update Truck Visuals:** Add visual indicators to the trucks to represent their state (`Waiting`, `In Inspection`).
- **Create New Zones:** Draw two new small rectangular zones within the "Trucks" area: a "Waiting Queue" and an "Inspection Lane".
- **Update Animation Logic:** When a truck is full, animate it into the "Waiting Queue". If an inspection lane is free, animate the first truck from the queue into the "Inspection Lane". After a simulated time (a certain number of frames), animate the truck off-screen to signify its departure.