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

From Console to Canvas

A Step-by-Step Guide to Visualization with `natId`

Authors: Benjamin Bandić, Nedim Bečić
Professor: Izudin Džafić

Contents

1 Introduction

You've successfully built simulations and algorithms that work in the console. Now, it's time to bring them to life. This guide will walk you through the process of converting a console-based application into a fully interactive and animated Graphical User Interface (GUI) using the `natId` framework.

We will treat this as a template. The principles you learn here for the Port Simulation can be applied to animating an A* pathfinding algorithm, a physics simulation, or anything else you can imagine.

1.1 The Guiding Philosophy: Model-View-Controller (MVC)

Before we write a single line of GUI code, we must understand the core design pattern that prevents our code from becoming a tangled mess: **Model-View-Controller**.

- **The Model:** This is your existing simulation logic. The `Container`, `Ship`, and `Truck` classes. They hold the data and the rules. The Model knows nothing about graphics.
- **The View:** This is the visual representation. Its only job is to draw things on the screen based on data. It's a "dumb" artist. The View knows nothing about simulation rules.
- **The Controller:** This is the brain. It connects the Model and the View. It processes user input (button clicks), tells the Model to update its state, and then tells the View to redraw itself.

Our entire project structure will be built around keeping these three parts separate.

2 The Foundation - Creating Your "Visual Models"

Your logical Model (`Container`, `Truck`) is perfect for the console, but it's missing key information for a GUI, namely: "Where am I on the screen?" and "What do I look like?". This is why our first step is to create new `structs` or `classes` that bridge this gap.

Your First Task: For each logical object you want to see on screen, create a corresponding "Visual" struct. It's best practice to put these new visual structs in their own header file, e.g., `VisualModels.h`.

```
// VisualModels.h
#pragma once
#include <gui/Image.h> // We need this for images
#include <td/Point.h>  // We need this for coordinates
#include "Containers.h" // We link to the logical model

// Define some constants to avoid "magic numbers" in our code.
// This makes tweaking the layout much easier later!
const gui::CoordType CONTAINER_WIDTH = 100.0;
const gui::CoordType CONTAINER_HEIGHT = 70.0;

// The Visual Model for a Container
struct VisualContainer {
    // 1. A pointer back to the original logical object.
    const Container* logicalContainer;

    // 2. A pointer to the image that represents this container.
    gui::Image* image;

    // 3. Its CURRENT position on the screen. This is what we will animate.
    gui::Point pos; // td::Point holds an x and y coordinate.

    // 4. Its TARGET position. Where is it trying to go?
    gui::Point targetPos;

    // 5. A flag to know if it's currently in motion.
    bool isMoving = false;

    // A constructor to make creating them easier.
    VisualContainer(const Container* logical, gui::Image* img, gui::Point startPos)
        : logicalContainer(logical), image(img), pos(startPos), targetPos(startPos) {}
};
```

Listing 1: *A template for a 'VisualContainer' struct.*

3 Setting the Stage - The Core GUI Classes

Now we build the "scaffolding" of our application. This is a standard structure you will use in every `natId` project.

3.1 `main.cpp`: The Igniter

Its only job is to create and run the application. The `app.run()` function starts the infinite event loop that is the heartbeat of our program.

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

int main(int argc, const char * argv[]) {
    Application app (argc, argv);
    app.init("EN"); // Initialize with a language
    return app.run(); // Starts the event loop.
}
```

Listing 2: *main.cpp*

3.2 `Application.h`: The Factory

Its only job is to know how to create your main window. The framework calls `createInitialWindow()` to ask, "What window should I make?". This is an example of Inversion of Control.

```cpp
#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) {}
};
```

Listing 3: *Application.h*

3.3 `MainWindow.h`: The Window Frame

This is the literal window on your desktop. It sets the size/title and holds your main content, the `MainView`.

```cpp
#pragma once
#include <gui/Window.h>
#include "MainView.h" // It needs to know about its content

class MainWindow : public gui::Window {
protected:
    MainView view; // The main content of our window
```

```
public:
    MainWindow() : gui::Window(gui::Geometry(50, 50, 1440, 800)) {
        this->setTitle("My Awesome Simulation");
        this->setCentralView(&view); // Plug the view into the window
    }
};
```

Listing 4: *MainWindow.h*

4 The First Milestone - Drawing a Static Scene

Let's get something on the screen! Our goal is to draw the initial, non-moving state of the simulation. All of this work happens in your `SimulationCanvas.h` class.

4.1 Loading and Assigning Images

Before we can draw, we need to load our images from the `assets` folder.
Key Principle: Load images **once** in a constructor, draw them **many times** in `onDraw`.Loading from disk is slow and should never be done in a loop or drawing function.

The best place to manage this is in your Controller, the `MainView` class.

1. **Declare `gui::Image` members** in `MainView.h` to hold the loaded image data.

2. **Load the images** from disk in the `MainView` constructor using a relative path.

3. **Assign pointers** to these loaded images when you create your `VisualContainer` objects in a setup method like `resetSimulation()`.

```
// In MainView.h
class MainView : public gui::View {
protected:
    // Members to HOLD the loaded image data
    gui::Image imgToxic, imgRefrigerated, imgOrdinary, imgFragile;

public:
    MainView() {
        // Load each image from disk into its corresponding object
        imgToxic.load("path_to_the_assets_folder/container_toxic.png");
        imgRefrigerated.load("path_to_the_assets_folder/container_refrigerated.png");
        // ... load other images ...

        resetSimulation();
    }

    void resetSimulation() {
        // ... create logical containers ...

        // Assign pointers to the correct, already-loaded image
        for (const auto& c : ship.get_toxic_cargo()) {
```

```
            visualContainers.emplace_back(&c, &imgToxic, /* startPos */);
        }
        for (const auto& c : ship.get_ordinary_cargo()) {
            visualContainers.emplace_back(&c, &imgOrdinary, /* startPos */);
        }
    }
    // ...
};
```

Listing 5: *The Load-and-Assign pattern for images in* `MainView`.

4.2 The Drawing Toolbox

All drawing must happen inside the `onDraw` method. Here are your primary tools.

`gui::Rect` This object defines a rectangular area. Its constructor is `gui::Rect(left, top, right, bottom)`. To create a rectangle from a starting point `(x, y)` and a size `(width, height)`, you must calculate `right = x + width` and `bottom = y + height`.

`gui::Shape` This is the object you use to actually draw primitive shapes. You can reuse a single `gui::Shape` object to draw many things.

`gui::Image` This is used for drawing `.png` files. It's best to load images once in a constructor, and then draw them every frame inside `onDraw`.

4.3 Implementing `onDraw`

Here is a template for a basic `onDraw` method that draws a static scene.

```
void onDraw(const gui::Rect& rect) override {
    // This method is our canvas. It gets wiped clean and redrawn every time.

    // Step 1: Define the areas for our simulation zones
    gui::Rect shipArea(50, 50, 350, 750);
    gui::Rect truckArea(850, 50, 1350, 750);

    // Step 2: Draw the background and zone outlines
    gui::Shape shape;
    shape.createRect(shipArea);
    shape.drawWire(td::ColorID::Black);
    shape.createRect(truckArea);
    shape.drawWire(td::ColorID::Black);

    // Step 3: Draw static images like the ship
    gui::Rect shipDrawingRect(shipArea.left, shipArea.bottom - SHIP_HEIGHT,
                              shipArea.right, shipArea.bottom);
    imgShip.draw(shipDrawingRect);

    // Step 4: Draw the dynamic objects (our VisualModels)
    for (const auto& vc : visualContainers) {
        gui::Rect destRect(vc.pos.x, vc.pos.y,
                           vc.pos.x + CONTAINER_WIDTH,
                           vc.pos.y + CONTAINER_HEIGHT);
        vc.image->draw(destRect);
    }
```

```
}
```

Listing 6: *A template for the `onDraw` method.*

At this point, you should be able to run your application and see your static layout.

5 Bringing it to Life - The Animation Loop

Animation is an illusion created by changing an object's position slightly and then redrawing the screen very quickly. We will use a non-blocking timer to achieve this.

5.1 The Mechanism: `gui::Timer` + `step()` + `refresh()`

1. **Create a `gui::Timer` in `MainView.h`.**

```
// In MainView.h's protected members
gui::Timer timer;

// In MainView's constructor
MainView() : timer(this, 0.05, false) // Tick every 0.05s
{ /* ... */ }
```

2. **Create the `step()` function in `MainView.h`.** This function will contain ALL of your animation and state machine logic.

```
void step() {
    // We'll fill this in soon.
}
```

3. **Handle the Timer's "Tick" Event.**

```
bool onTimer(gui::Timer* pTimer) override {
    if (pTimer == &timer) {
        step(); // 1. Update the simulation logic
        canvas.refresh(); // 2. Request a redraw
        return true;
    }
    return false;
}
```

4. **Create the `refresh()` Wrapper.** We need a `public` way to call the `protected` `reDraw()` method.

```
// In SimulationCanvas.h
public:
    void refresh() {
        reDraw(); // This is the safe way to request a redraw.
    }
```

5.2 The Animation Event Chain

This is the most important concept to understand:

`Timer Ticks` → `onTimer` → `step()` → `canvas.refresh()` → `onDraw` → `...repeat`

5.3 Implementing the `step()` Function

The `step()` function has two jobs, which should always be in this order:

```
void step() {
    // JOB 1: Animate anything that is already in motion.
    bool isAnythingMoving = false;
    for (auto& vc : visualContainers) {
        if (vc.isMoving) {
            isAnythingMoving = true;
            // Move position a little closer to the target position
            vc.pos.x += (vc.targetPos.x - vc.pos.x) * 0.1;
            vc.pos.y += (vc.targetPos.y - vc.pos.y) * 0.1;

            // Check if it has arrived
            if (std::abs(vc.pos.x - vc.targetPos.x) < 1.0 &&
                std::abs(vc.pos.y - vc.targetPos.y) < 1.0) {
                vc.pos = vc.targetPos; // Snap to the final position
                vc.isMoving = false;
            }
        }
    }
    if (isAnythingMoving) return; // If something is moving, do nothing else!

    // JOB 2: If nothing is moving, decide what should move next.
    // This is where your state machine logic goes!
    switch (currentState) {
        case SimState::Unloading:
            // Find the next container that needs to move.
            // Calculate its targetPos.
            // Set its isMoving flag to true.
            break;

        // ... other states ...
    }
}
```

Listing 7: *The two-part structure of the ‘step()‘ function.*

6 Adding Interactivity

The final piece is letting the user control the simulation. This is done in `MainView` by handling button clicks.

```
// In MainView.h
protected:
    gui::Button btnStart, btnReset;

public:
    // ... in constructor, create and lay out buttons ...

    // The event handler for all button clicks
    bool onClick(gui::Button* pBtn) override {
        if (pBtn == &btnStart) {
            // Toggle the timer
            if (timer.isRunning()) {
                timer.stop();
            } else {
                timer.start();
            }
            return true; // We handled this event
        }
        if (pBtn == &btnReset) {
            resetSimulation(); // Call your setup function
            return true;
        }
        return false; // We didn't handle this event
    }
```

Listing 8: *Handling user input in the 'onClick' method.*

7 Conclusion: Your Project Checklist

For any new visualization project, follow these chronological steps:

1. **Create Visual Models:** Define 'structs' (`VisualObject`) that hold visual information (`pos`, `image`) and link back to your logical objects.
2. **Set up GUI Classes:** Create the standard `main.cpp`, `Application.h`, `MainWindow.h`, `MainView.h`, and `SimulationCanvas.h` files.
3. **Implement Static Drawing:** In `SimulationCanvas::onDraw`, write the code to draw your initial, non-moving scene. Get this working first!
4. **Set up the Animation Timer:** Add a `gui::Timer` to `MainView` and create the `onTimer` event handler.
5. **Create the `step()` function:** Implement the two-part logic: the animation loop (Job 1) and the state machine (Job 2).
6. **Add User Controls:** Create buttons in `MainView` and handle their events in `onClick` to start/stop the timer and reset the simulation.

By following this template, you can systematically build any animated simulation, confident that each part is separate, organized, and doing its job correctly.