TU WIEN Informatics

Adaptagrams/libavoid for Sprotty

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Software & Information Engineering

by

Vladyslav Hnatiuk
Registration Number 01613669

to the Faculty of Informatics

at the TU Wien

Advisor: Ass. Prof. Dr. Dominik Bork
Assistance: Dr. Philip Langer

Vienna, 3rd February, 2023

Vladyslav Hnatiuk	Dominik Bork

Erklärung zur Verfassung der Arbeit

Vladyslav Hnatiuk

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 3. Februar 2023

Vladyslav Hnatiuk

Acknowledgements

I would like to thank my supervisor Ass. Prof. Dr. Dominik Bork for support and guidance throughout the creation of this thesis. Also, I would like to thank Dr. Philip Langer for his support and valuable input on the implementation part, especially the integration of results of this work in `sprotty` project.

I am also grateful to my wife, who supported me through this venture and the whole study, and my parents for their unceasing encouragement and attention.

Abstract

More and more tools move to the web platform, at least as an optional way of their use. With the advance of technologies for web development, more complex tools are implementable. This is also true for software for graphical modelling. Compared with text editors, algorithms for fundamental tasks such as a representation of a diagram or editing actions are needed in all of them.

Different algorithms for solving these tasks exist for desktop applications. One of the most popular set of libraries for solving common problems in diagram building and editing, such as routing of elements, is Adaptagrams created by Tim Dwyer and Michael Wybrow.

If someone wants to re-use these libraries in web application, the main problem is that they are written in C++, which is not supported in the web environment by default. The most obvious solution way is to rewrite the libraries in Javascript or Typescript. Still, at the same time, it is also very time-consuming because 4 libraries in Adaptagrams include many complex algorithms.

A new technology called WebAssembly was designed and implemented in all modern browsers to make code porting from other platforms to the web possible.

In this work, one part of Adaptagrams, namely 'libavoid' library, is ported via WebAssembly for usage in web applications and the whole process of creating the'libavoid-js' library as a Javascript version of 'libavoid' is described. Also, a showcase is provided, including a demonstration of the integration of 'libavoid-js' in the existing framework for developing interactive graphical diagrams called Sprotty, as well as a performance comparison of the routing algorithm from libavoid-js and built-in Sprotty.

Contents

CHAPTER 1

Introduction

Models are a popular approach to represent complex systems by splitting them into models and address all relevant concerns. In software engineering, model-driven engineering(MDE) became a popular methodology, which allows to reduce complexity and to fill the gap between high-level concepts used by domain experts and low-level abstractions provided by programming languages. It's possible due to different modelling techniques and generating system artifacts. Usually, MDE solutions have domain-specific concepts, this helps to connect the problem space in which domain experts work, and the programming space(implementation) [BCOR15].

MDE has been actively researched since 90s, and Antonio Bucchiarone with co-authors in their "Grand challenges in model-driven engineering: an analysis of the state of the research" publication split this period into two parts: to around 2007 and after. In the first part modelling language issues were dominating, for example, UML was considerably changed, and there was plenty of research on both modelling languages and metamodelling. Significant research challenges such as language engineering, language workbenches, model management, model analysis, models at runtime, modelling repositories, scalability across different dimensions and others were identified in the second part. Besides, there are also technical challenges as well. One of them is the tool and implementation challenge, and it is often mentioned as a key aspect which makes the adoption of MDE more harder [BCPP20].

MDE tools are complex solutions, and their development touches many different areas and includes many tasks, some of which can be challenges on their own. One such example is the development of a visual editor for graphical models. There are tools for that, for example, Sprotty framework in case of solution for the web platform. Visual editor consists of many parts, and one of the essential parts of a graphical diagram editor is the optimal routing of elements of the diagram. Some commercial products have built-in solution of this problem, but there is little information about how they work and almost no open-source reusable and easily integrable solutions.

Michael Wybrow and co-authors created and explained in their articles "Seeing Around Corners: Fast Orthogonal Connector Routing" [MW10] and "Orthogonal connector routing" algorithms for polyline and orthogonal routing of diagrams and implemented them in `libavoid` library [KM14].

On the time of writing this work there were 3 implementations of routing algorithms used in libavoid:

- libavoid - original C++ library
- webcola - JavaScript implementation of part of adaptagrams algorithms. Routing algorithms are tightly coupled with layout constraints, it is hard to test routing algorithm separately, for this reason it is not used for comparison in this work
- webcola-wasm - fork of webcola, that is partially rewritten in Rust and compiled to WebAssembly. Routing algorithms are kept in JavaScript, so this library is not used for comparison in this work

Our goal is to compile the original libavoid library to WebAssembly, compare the performance of C++ and JavaScript libraries and integrate it into Sprotty framework. Sprotty includes routing algorithms as well, but they are too primitive for advanced solutions; namely, they can find the route between nodes and don't support object avoiding and related features to make diagrams as understandable as possible.

In this work, we describe C++ to WebAssembly compilation process(Chapter 2), give an introduction to libavoid structure and its usage(Chapter 3), show how compiled libavoid-js can be used in a JavaScript project and compare its performance with the original libavoid, explain how a new functionality such as router can be integrated into sprotty framework(Chapter 4). present real-world application of a new libavoid-js router in sprotty-based modelling tool BIGER(Chapter 5). In the last chapter 6, we summarize the results of this work, make conclusions and describe our vision of further work.

CHAPTER 2

C++ to WebAssembly compilation

JavaScript is the only built-in language of the Web, and it cannot meet all the requirements of modern applications that become more complex every day and require better performance and security. Especially, it is not optimal as a compilation target. These points motivated engineers of major browser vendors, and in cooperation, they designed a portable low-level bytecode called WebAssembly [HRS+17]. Because WebAssembly is a low-level language, it is mostly generated from more high-level languages like C, C++, Rust, etc.; in other words, code in a more high-level language is compiled to WebAssembly.

In the history of WebAssembly development, many different tools were able to compile code from C++ to WebAssembly. Nowadays, the only open-source and actively developed toolchain is emscripten [Conb]. It supports not only C and C++ but also all languages that can be compiled using LLVM [Fou], such as Julia, Rust, Objective C, and others. The toolchain is available for local installation on a PC and for use in a Docker container [Ems].

The main part of emscripten is `emcc` compiler that can compile C/C++ code to WebAssembly. It can be used either directly with source files or as a compiler in existing C/C++ projects and reuse project build configuration. Still, in this work, it was used directly due to the simple structure of the library and no need for complex build configuration.

There are two main use cases of emscripten: creating a new library with a WebAssembly port and a WebAssembly port of an existing library. For the first use case, `embind` [Cona] library is included in emscripten. Additionally to library code, simple glue code with definitions of the external interface should be written. Then the library can be compiled with `emcc` compiler without any additional steps.

For the second case, which was also chosen for libavoid, the usage of emscripten is also simple. If a code with one or few functions with parameters that have simple data types should be compiled and used in Javascript, it can be achieved by passing correct parameters such as names of functions to export, but in the case of libavoid library, with advanced interface that includes many classes, methods and functions it should be additionally described. For this purpose, WebIDL[Groa] language can be used.

WebIDL

WebIDL is a language used in emscripten to describe the existing codebase's interfaces. Let's assume there is the following C++ code, and we would like to create WebAssembly bindings for it using the WebIDL definition.

```
namespace Avoid {
enum RoutingParameter
{
    segmentPenalty = 0,
    anglePenalty
};

class Router {
    public:
        Router(const unsigned int flags);
        void moveShape(ShapeRef *shape, const Polygon& newPoly);
        void setRoutingParameter(
            const RoutingParameter parameter,
            const double value = chooseSensibleParamValue
        );
};

}
```

WebIDL definition for this piece of code is following:

```
enum Avoid_RoutingParameter {
  "Avoid::segmentPenalty",
  "Avoid::anglePenalty"
}

[Prefix = "Avoid::"]
interface Router {
  void Router(unsigned long flags);
  void moveShape(ShapeRef shape, [Ref] Polygon newPolygon);
  void setRoutingParameter(
        Avoid_RoutingParameter parameter,
```

```
        double value
    );
}
```

There are enum `RoutingParameter` and class `Router` in a namespace `Avoid`.

Enum is defined similarly to C++, but the list of values is a list of strings with a namespace at the beginning split by '::'. The namespace should also be prepended to the enum name, but split by '_'.

Class is defined as an interface with methods. The namespace is set as a prefix above the interface. WebIDL language has its own set of data types[Grob], they are similar to the C++ types, but there are also differences, e.g. there is no 'int' type, and 'long' should be used instead. Pointers are passed as values without '*', and for references, there is a `[Ref]` decorator [Cond].

After defining the interface, `WebIDL Binder` tool should be applied on .idl files(WebIDL definitions), and it produces two files: `glue.cpp` and `glue.js`. The first one should be compiled with other sources via `emcc`, and the second one included in JavaScript library, `emcc` has `--post-js glue.js` parameter for that.

Using generated code in JavaScript environment

`emcc` tool generates two files: a JavaScript module and a WASM module. The name of the javascript module can be specified with '-o' argument(e.g. '-o portedLib.js'). WASM module will have the same name but with a '.wasm' extension. This module includes an exported default function that should be used to initialise the WASM module. The JavaScript module imports and initialises WASM module, the only thing that the developer should do with the WASM module is to check that a web server supports hosting WASM modules and generated WASM module is available.

The current JavaScript standard(ES2021) doesn't support importing WebAssembly(WASM) modules in the same way as JavaScript modules yet, what would be a good and simple way to use WASM modules, but such proposal exists [Ros]. It is also important to note that instantiation of WASM modules is asynchronous, so using it as an ESM module is possible only with a top-level await statement, which is also not supported yet, but proposed for including JS standard [BSEB].

```
import Module from 'portedLib.js';

Module().then(module => {
  // after initialization the module is available for usage
  // instantiating compiled C++ class 'Router'
  const classInstance = new module.Router();
  // calling compiled C++ function
  // 'calculateDistance(int x1, int y1, int x2, int y2)'
  const distance = module.calculateDistance(12, 18, 39, 55);
```

```
});
```

Limitations and challenges with WebIDL

The compilation is a complex process, and it makes requirements for tools high both from the user perspective and internal implementation. Emscripten has extensive documentation and clear tutorials. The only important aspect, if a compilation error occurs, is to carefully check the source of the error. It can be wrong idl definition, wrongly generated code from `WebIDL Binder`(a bit more about this in the next paragraph), compiler parameters etc.

WebIDL is an open standard, and there is detailed specification but only a few basic examples and tutorial on the web. The two largest open-source WebIDL consumers are web browsers Mozilla Firefox and Google Chrome, and there are many real examples of WebIDL usage in their repositories. But they also have their dialects of IDL with additional functionality, which `WebIDL Binder` doesn't support. Generally, `WebIDL Binder` doesn't support all constructions needed to describe interfaces of C++ code, even not very complex. It was also the case with libavoid, so we forked `WebIDL Binder` and extended it with some additional functionality. Details are available in our repository [HtEA]. Also, all WebIDL bindings created for libavoid are available in libavoid-js repository [Hnab].

Working on libavoid bindings, we encountered the following limitation with overloading in emscripten: only functions/methods with a different number of arguments can be overloaded. If a function/method is overloaded with the same number of arguments but different types, there is no way to compile both using WebIDL+emcc without custom code.

Another important aspect to keep in mind is that generated library doesn't check the correctness of members' usage like function calls, whether a number of arguments is correct, they have correct types. And errors in case of wrong usage are not always self-explaining. To solve this problem at least partially we:

- generated API documentation for libavoid-js to have documentation not only for the original C++ library but also for generated JS library
- implemented TypeScript typings for libavoid-js

Interface of C++ Arrays in JavaScript

In some cases, JavaScript interface doesn't correspond interface that the developer may expect. One simple example of such case is `std::vector`. `Avoid::Polygon` class has attribute `ps` of type `std::vector` and in JavaScript, it is an array-like structure, but access by index like `polygon.ps[1]` doesn't work correctly, and methods `set_ps(<index>, <value>)` and `get_ps(<index>)` should be used.

Pointers to C++ objects

Assuming there is a C++ class `ConnRef`(it is a real example from libavoid library, we will take a look in more detail in the next chapter) for connection(edge) and it has a method `setCallback(void (*cb)(void *), void *ptr)` to set callback which is called when a connection was changed. We set Javascript function as a callback and pass the second argument, which will be passed to the callback as a parameter.

```
function connCallback(connRefPtr) {
    const changedConnRef = Avoid.wrapPointer(
      connRefPtr,
      Avoid.ConnRef
    );
}

const connRef = new Avoid.ConnRef();
connRef.setCallback(connCallback, connRef);
```

We pass `connRef` in `setCallbck` as `ConnRef` instance, however, `connCallback` gets pointer identifier(number) to `ConnRef` instance instead of the instance itself. To be able to get the class instance by pointer identifier, generated Javascript module, `Avoid` in this example, has `wrapPointer(identifier, class)` function that takes pointer identifier and class of an instance as parameters.

Debugging

To get logs from C++ code and simplify debugging, `printf` works in generated WebAssembly code by default without additional build parameters.

To allow more advanced debugging, `emcc` can also generate a source map for .wasm file if `-g4 --source-map-base http://localhost:8080/` parameters are passed. More details on how `libavoid-js` can be debugged are available in its repository.

CHAPTER 3

libavoid: Architecture and Usage

In this chapter, we will explain libavoid architecture and its features and show a usage example. API of libavoid C++ library is described in API documentation [Wyb]. It consists of several classes such as Router, Rectangle, Shape etc. But there is only one basic documented example and short descriptions of classes and their methods, making entry-level high. To simplify this for new users or developers who will work with JavaScript version of libavoid(it has the same API), in this chapter, we share an introduction and some lessons we have learned working on this thesis. All relevant classes are also visualized on class diagram 3.1.

libavoid includes tests that are used to check functionality after changes in the library. We made a fork [Hnaa] of libavoid with refactored tests to allow developers to use them as usage examples. Also, support for a more modern build system 'CMake' was added to simplify development and usage in modern IDEs, CI/CD was configured, and one of the most important - routing algorithm was improved to better handle hierarchical diagrams.

3.1 Router

The main class that is used to instantiate a router at the beginning, configure parameters and perform incremental actions with a diagram. On instantiation, you should select either orthogonal or polyline routing type(see `RouterFlag` enum in router.h). It can be overwritten for individual connections later. This class has three main methods for configuration: `setRoutingParameter`, `setRoutingOption`, `setRoutingPenalty`(see enums in method definitions for possible values). Also, `Router` class has a list of methods for incremental changes such as `moveShape`, `deleteShape` and others.

There are two modes of routing: immediate and queued. By default, queued is used for maximal performance. `setTransactionUse` method can be used to switch the

mode. If queued mode is used, `processTransaction` method should be called after modifying the diagram to apply all changes.

There are also two methods to output the result of routing either in a text file or in an SVG file: `outputDiagramSVG` and `outputDiagram`. They can be very useful for debugging and checking whether your canvas implementation works correctly.

3.2 Diagram elements(nodes)

The base class for diagram elements(nodes) is `Polygon`. It consists of a list of points and can have arbitrary geometry. There is also `Rectangle` class to simplify the creation of blocks that are rectangles. Using a polygon-based object should be created an instance of `ShapeRef` that becomes part of the diagram.

There is also `Cluster` to group the nodes. Then edges between two child nodes inside of the cluster will be connected so that the edge is only inside of the cluster if possible. If a child node is connected with a node outside the cluster, the router tries to connect them so that edge intersects the cluster border only once.

Another kind of diagram element is junctions. A junction is a fixed or free-floating point connections can be attached to. A router can also improve the positioning of free-floating junctions, and there is `recommendedPosition` method in `Junction` class for this purpose.

3.3 Element connections(edges)

A connection between two nodes or edges is represented as a Connector object, an instance of `ConnRef` class. The connector has two endpoints: source and destination, that are represented as `ConnEnd` instances. `ConnEnd` instances are created used points(instances of `Point` class) or `ShapeConnectionPin` instances. `ShapeConnectionPin` instances are points that belong to a certain shape. They can be either fixed relative to the parent shape or attached, for example, to the centre or side of the parent shape. If a point belongs to a node, it's highly recommended to use `ShapeConnectionPin` because the router can provide a much better result and use all configuration parameters correctly. If you use `Point` and place it on the position so that it visually is in shape, routing may be wrong. That's why the proper design of the whole solution, in which all needed data for router, nodes and edges are available, is important to achieve the best routing result.

The next step after routing is getting its result and displaying it on canvas. To get routes, there is `displayRoute` method in `ConnRef` class. It means all `ConnRef` instances should be saved, and then the result of the call is a `Polygon` instance that can be iterated, and each point is accessible via `at` method by its index.

We also encountered a problem [Hnac] in `ShapeConnectionPin` usage. Its position is set once on instantiation and cannot be changed later. The only way to do this is to delete the pin and create a new one with the same properties but another position.

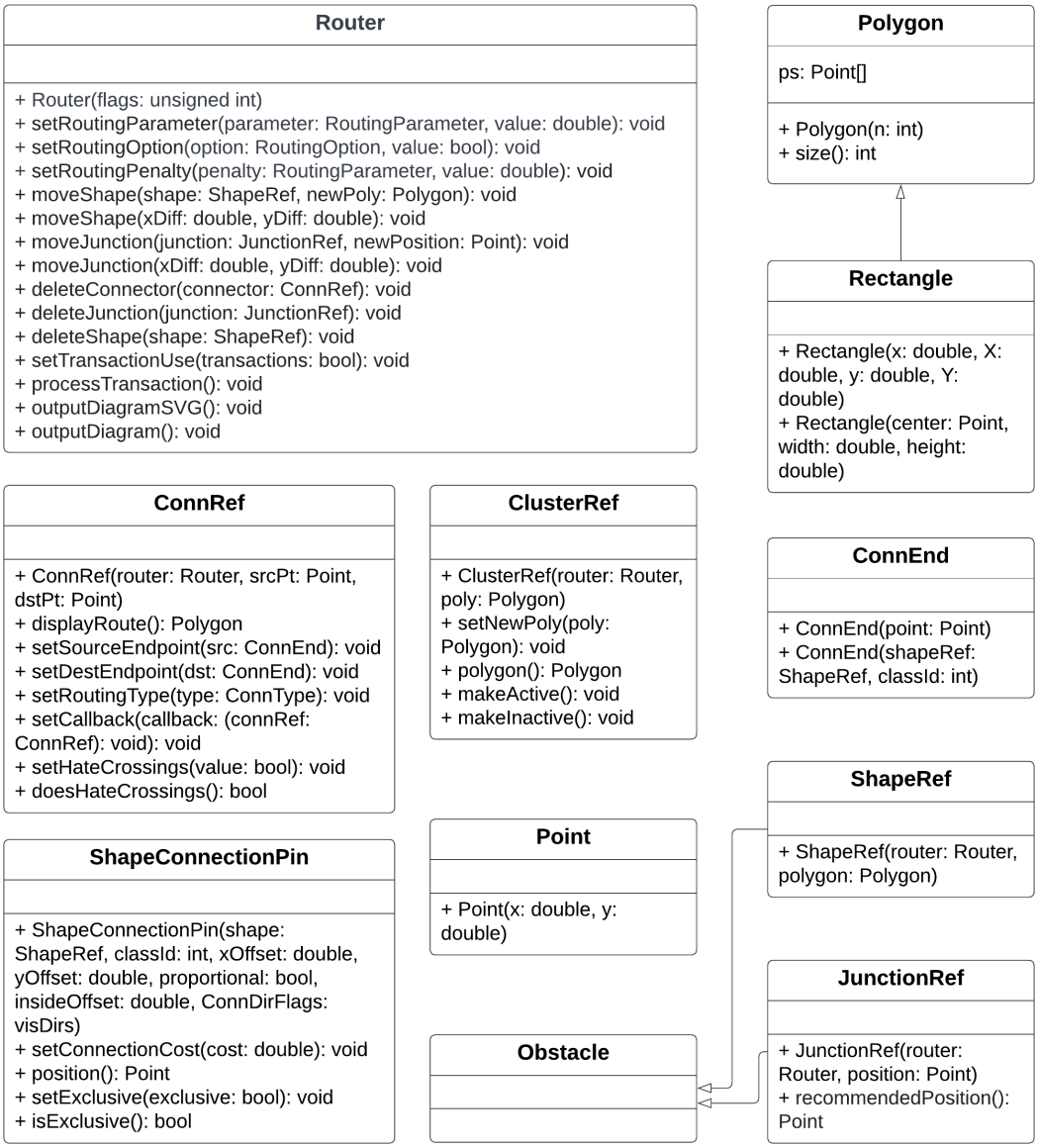

Figure 3.1: libavoid class diagram

3.4 Usage Example

Now we will take a look at step-by-step usage example of C++ libavoid. There are two parts: creating a diagram and applying incremental changes to the diagram.

3.4.1 Create a diagram and get routed edges

```
#include "libavoid/libavoid.h"

using namespace Avoid;

/* 1. Instantiate a router. By default(without arguments), it is
 * instantiated in polyline mode by passing `OrthogonalRouting`
 * parameter we change the mode to orthogonal
 */
auto router = new Router(OrthogonalRouting);

/* 2. Set router parameters, options and penalties(optional
 * step, all parameters also have default values described in
 * the documentation). List of all parameters, options and
 * penalties is available in the documentation as well.
 */
router->setRoutingParameter(
    RoutingParameter::shapeBufferDistance, 4
);
router->setRoutingOption(
    RoutingOption::nudgeOrthogonalSegmentsConnectedToShapes,
    true
);
router->setRoutingPenalty(
   RoutingParameter::segmentPenalty, 50
);

/* 3. Create an instance of Rectangle with the top left corner
 * (-100, 100) and right bottom corner (220, 350).
 * A rectangle stores only placement information about figures
 * and has no impact on routing.
 */
Rectangle shape1Rectangle({ -100, 100 }, { 220, 350 });

/* 4. Instantiate a Shape on a base of a rectangle
 *    `shape1Rectangle`.
 * It is automatically added to the router, and from now
 * the router takes it to account(e.g. avoids if needed) on
```

```
 * calculating edges.
 */
ShapeRef *shape1 = new ShapeRef(router, shape1Rectangle);

/* 5. Create another rectangle and shape
 */
Rectangle shape2Rectangle({ 300, 400 }, { 400, 500 });
ShapeRef *shape2 = new ShapeRef(router, shape2Rectangle);

/* 6. For the definition of connection pins on shapes, there is the
 * ShapeConnectionPin class. Shape connection pins can be absolute
 * or relatively positioned in shape. Create a shape pin with class
 * id 1 and absolute position (0, 14) in the 'shape1'. The fifth
 * parameter means whether the pin has a relative position, the sixth
 * is inside offset of the pin, and with the last one, we set possible
 * edge directions from the pin. In this case, the first edge segment
 * from this pin can go only to the left.
 *
 * Shape class id helps to classify shape pins and allows later for
 * example to create an edge from/to one of the pins with class id X.
 * The best pin for this class will be chosen automatically.
 */
new ShapeConnectionPin(shape1, 1, 0, 14, false, 0, ConnDirLeft);
new ShapeConnectionPin(shape1, 1, 1, 0.1, true, 0, ConnDirRight);

/* 7. Create another ShapeConnectionPin instance with class id 2,
 * relative position (0.5, 0) that is the centre of the top side of a
 * node, and allow the first segment to go only to the top.
 */
new ShapeConnectionPin(shape2, 2, 0.5, 0, true, 0, ConnDirUp);

/* 8. To create a connection between two shapes, we need two
 * ConnEnd instances, one for source and one for the destination,
 * that are created by passing parent shape and class id to
 * which this end belongs.
 */
ConnEnd srcPtEnd(shape1, 1);
ConnEnd dstPtEnd(shape2, 2);

/* 9. An edge is represented as a ConnRef instance. Its constructor
 * takes router and two ConnEnd's as arguments.
 */
ConnRef *connection = new ConnRef(router, srcPtEnd, dstPtEnd);
```

```
/* 10. We set general parameters for the router in steps 1
 * and 2, and we can also overwrite some of them for individual
 * connections.
 */
connection->setRoutingType(ConnType_PolyLine);

/* 11. By default, the router queues all changes and applies
 * them when 'processTransaction' method is called. All actions
 * are grouped and processed together for efficiency.
 */
router->processTransaction();

/* 12. Get edge points.
 */
std::vector<Point> actualRoute = connection->displayRoute().ps;

/* 13. Save the diagram to an SVG file to get a visualization
 * of the result.
 */
 router->outputDiagramSVG("usage_example");
```

After execution of this code, we get the diagram illustrated on Figure 3.2.

3.4.2 Applying incremental changes and getting the new result

The next part after creating a diagram is applying changes like moving the shapes, removing them etc. The following code snippet shows how this can be done in libavoid. Note that this is a continuation of the code above, it works only after the execution of the first part above.

```
/* 1. Change shape size: moveShape method of the router can be
 * used for both moving a shape and changing its size.
 * Change size of shape1 to ({-50, -50}, {180, 250}).
 */
router->moveShape(shape1, Rectangle({-50, -50}, {180, 250}));

/* 2. Add a new shape with the top left corner (-300, 260) and
 * bottom right corner (-110, 310).
 */
Rectangle shape3Rectangle({-300, 260}, {-110, 310});
ShapeRef *shape3 = new ShapeRef(router, shape3Rectangle);
new ShapeConnectionPin(shape3, 6, 0.5, 0, true, 0, ConnDirAll);
```

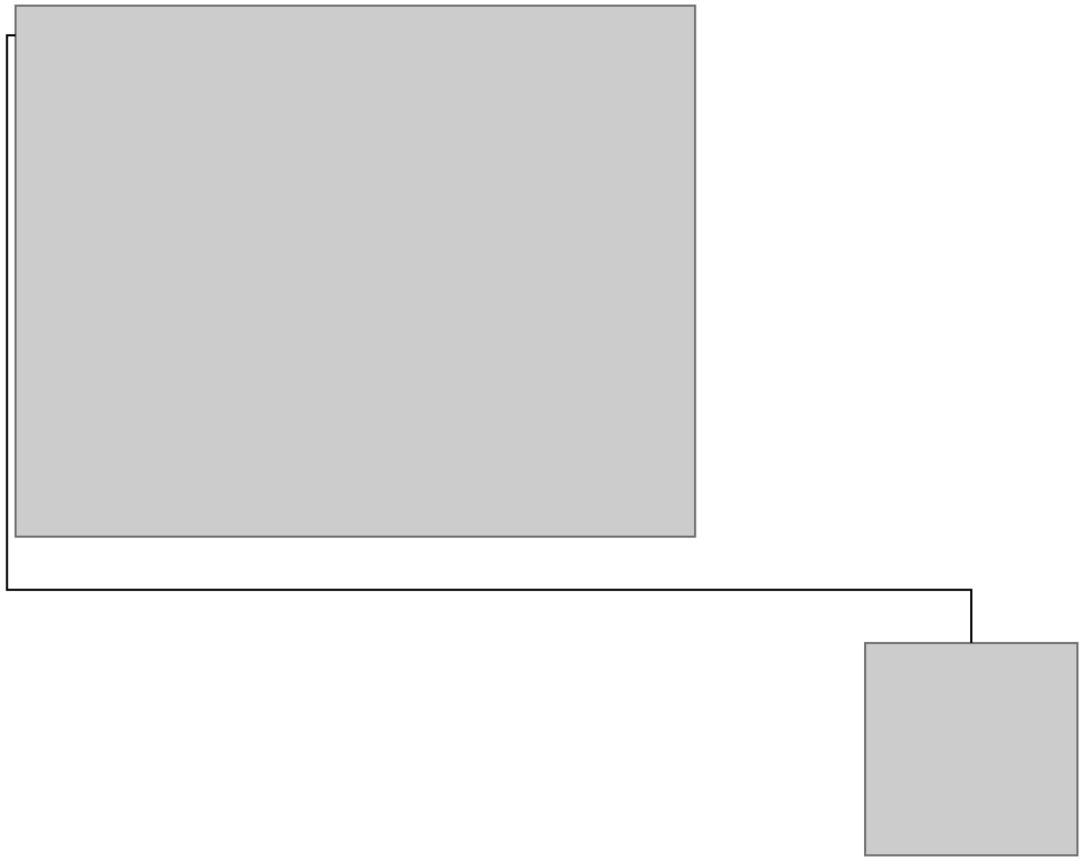

Figure 3.2: Result diagram

```
/* 3. Move shape3 by 20 horizontally and by -100 vertically.
 */
router->moveShape(shape3, 20, -100);

/* 4. Add a new connector between shape1 and shape3
 */
ConnEnd srcPtEnd(shape1, 1);
ConnEnd dstPtEnd(shape3, 3);
ConnRef *connRef2 = new ConnRef(router, srcPtEnd, dstPtEnd);

// 5. Apply all changes and save the result diagram in
//    'incremental_changes_part1.svg'
router->processActions();
router->outputDiagramSVG("incremental_changes_part1");
```

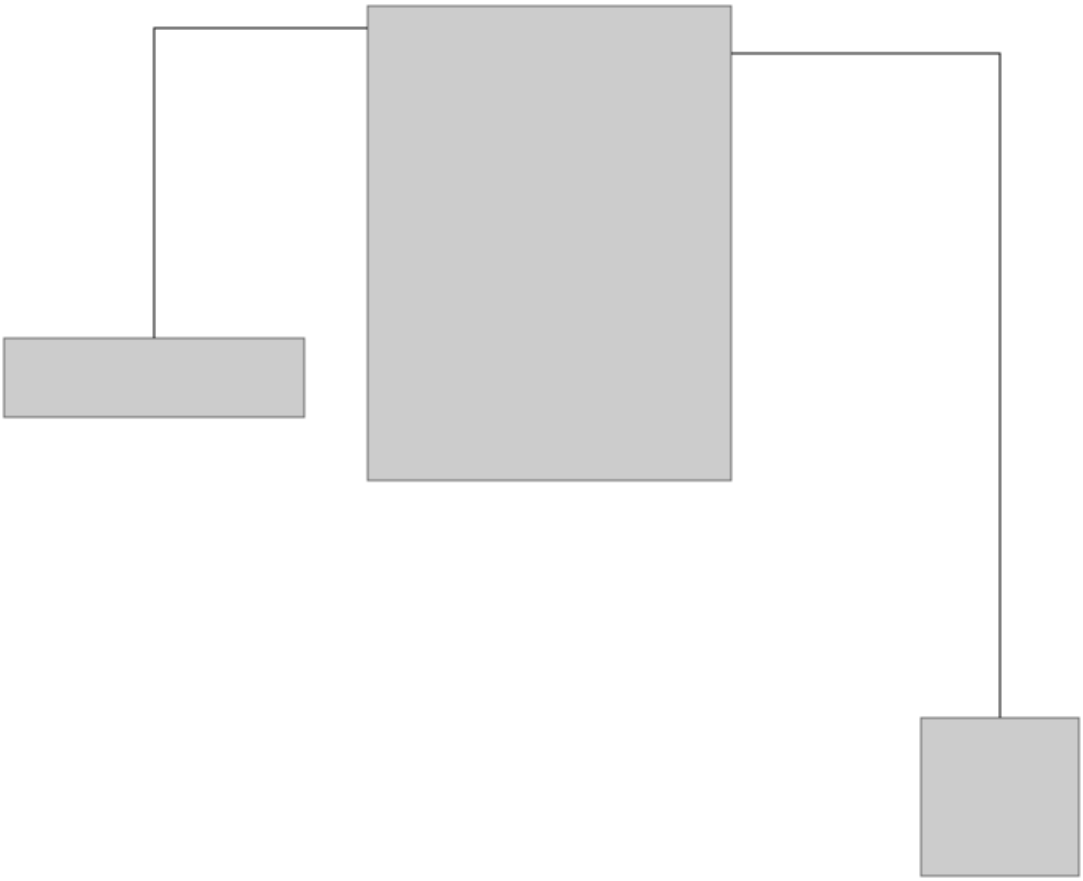

Figure 3.3: Incremental change: part 1

```
/* 6. Delete shape2.
 */
router->deleteShape(shape2);

// 7. Delete the connector which was created in the previous
//    example
router->deleteConnector(connection);

// 8. Apply all changes and save the result diagram in
//    'incremental_changes_part2.svg'
router->processActions();
router->outputDiagramSVG("incremental_changes_part2");
```

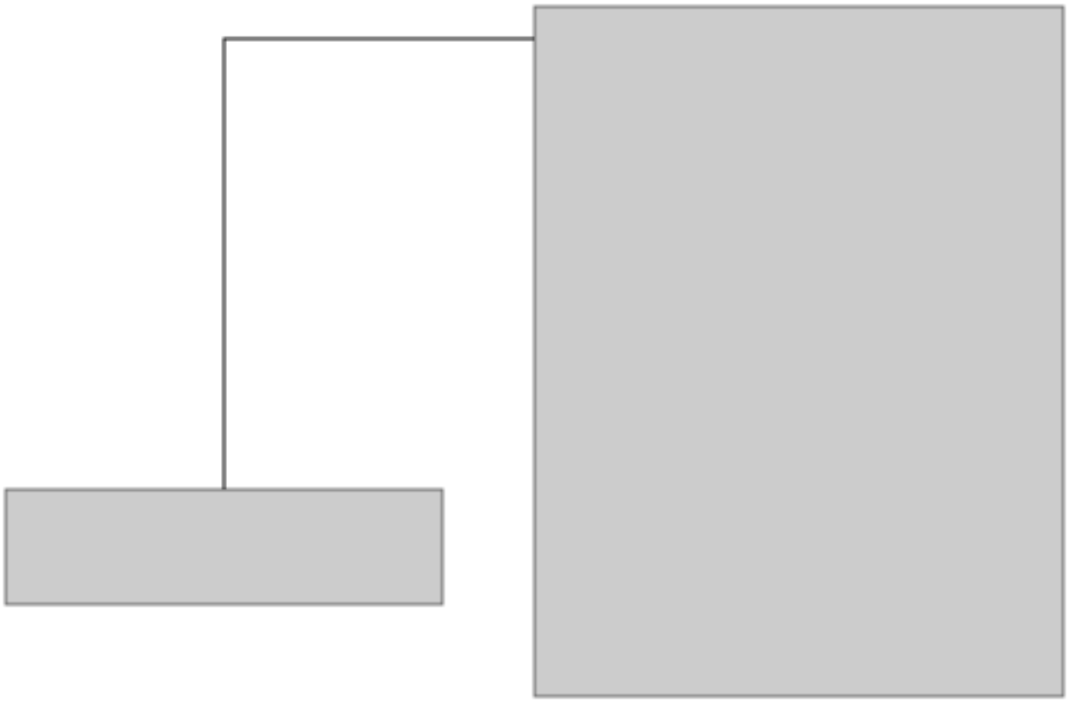

Figure 3.4: Incremental change: part 2

CHAPTER 4

Using libavoid-js in web projects

In chapter 2, we explained the basics usage of WASM modules generated using `emcc`, then in Chapter 3, we showed how the original `libavoid` library can be used in C++ and in this chapter, we will demonstrate how JavaScript library `libavoid-js` can be integrated into a web project.

4.1 Import and Usage

As we showed in chapter 2, after compilation C++ to WASM + JavaScript via `emcc`, an asynchronous function is available for initialization of the module, after which it can be used. In libavoid-js, there is no direct access to this function, instead interface of the library includes an object `AvoidLib` with the following methods:

- asynchronous `load` for loading WASM module
- synchronous `getInstance` to get an instance of the WASM module after its instantiation

So to use libavoid-js in a project, a developer should import it, call the asynchronous `load` function and wait for its execution. After this API provided by WASM module is ready for use. The code snippet looks like the following:

```
import { AvoidLib } from "libavoid-js";

async function main() {
    const avoidLib = await AvoidLib.load();
    // use API of WASM module, example:
    const router1 = new avoidLib.Router(
        avoidLib.OrthogonalRouting
```

```
    );

    // or if a module instance is needed somewhere else(another
    // function or another module), then:
    const avoid = AvoidLib.getInstance();
    const router2 = new avoid.Router(avoid.OrthogonalRouting);
}
```

Another important aspect is that destructors in C++ are not called automatically. So if they include custom logic, for example, destructor in `Router` class removes all related to router instance elements like nodes, edges etc. To call its destructor, there is a general `Avoid.destroy()` function, to which instance, in this example, the instance of the router should be passed. If the router destructor is not called, it would cause memory leaks.

4.2 Integration libavoid-js in sprotty

Sprotty framework is based on GLSP(graphical language server protocol) [REIWC18], and it has a flexible and modular architecture. All features are optional and are connected to applications with a dependency injection approach(`inversify` library is used for implementation). The router is also one of the features.

4.2.1 Router

Router implementation is represented as an injectable class that extends `AbstractEdgeRouter` and implements `IEdgeRouter` interface. All routers available in `sprotty` route each edge independently by calling `route(edge: Readonly<SEdge>, args?: Record<string, unknown>): RoutedPoint[]` method of the router. `libavoid` router has another approach, it routes all edges together. To make it possible in `sprotty`, a new interface `IMultipleEdgesRouter` was introduced. It has a new method `routeAll(edges: SRoutableElement[], parent: Readonly<SParentElement>): EdgeRouting;` that can be used to route all edges at once. How it can be used will be shown in section 4.3. Sprotty calls `routeAll` method each time when rerouting is needed, but the whole diagram should not always be rerouted, it would be inefficient if, for example, only one edge was changed. To avoid full rerouting, `LibavoidSprottyRouter` saves results after each `routeAll` call and compares data passed in the next call with the previous one to detect changes in the diagram. This way is much cheaper in the sense of resources as the rerouting of the full diagram, but if sprotty passed only changes in the diagram instead of the full diagram, the routing would be more performant. Whether the performance of the whole diagram editor would noticeably increase without adding a lot of complexity, need to be investigated in further work.

Implementation

All functions in `sprotty` related to instantiating the diagram editor are synchronous. As we explained in section 4.1, WASM module can be loaded only asynchronously, so the developer should call `load` function from `sprotty-routing-libavoid` and wait for its execution.

`libavoid-js` router is instantiated with `LibavoidRouter` and the instance has the same life time as the `sprotty` router. All parameters can be set using `setOptions(options: LibavoidRouterOptions)` method of `LibavoidRouter`.

Multiple routers can be used in one diagram simultaneously. Edges that should be routed by the router are determined by `routerKind` parameter of the edge. And in the router, there is `kind` getter that returns the appropriate kind as a string(see example below).

```
import { injectable } from "inversify";
import {
    AbstractEdgeRouter, IMultipleEdgesRouter,
    SParentElement, EdgeRouting
} from "sprotty";

@injectable()
export class LibavoidRouter
  extends AbstractEdgeRouter
  implements IMultipleEdgesRouter
{
    static readonly KIND = "libavoid";

    routeAll(
        edges: LibavoidEdge[], parent: SParentElement
    ): EdgeRouting {}

    get kind() {
        return LibavoidRouter.KIND;
    }

    <other methods and attributes>
}
```

4.2.2 Edges

There is also a possibility to extend edge information. Edge information is represented by `SRoutableElement` class and can be extended as follows:

```
export class LibavoidEdge extends SRoutableElement {
  routeType = 0;
```

```
  sourceVisibleDirections = undefined;
  targetVisibleDirections = undefined;
  hateCrossings = false;
}
```

And then used as a type for edges, one of the examples is above in `routeAll` method of `LibavoidRouter`.

4.3 Usage of LibavoidRouter

The router can be connected to the container module of the diagram editor in the following way:

```
import { TYPES } from 'sprotty';
import { LibavoidRouter } from 'sprotty-routing-libavoid';

const editorContainer = new ContainerModule((
    bind, unbind, isBound, rebind
) => {
    bind(LibavoidRouter).toSelf().inSingletonScope();
    bind(TYPES.IEdgeRouter).toService(LibavoidRouter);
}
```

In the case of the router from `sprotty-libavoid-routing`, there is also a possibility to set options of the router using `setOptions` method:

```
import { RouteType } from 'sprotty-routing-libavoid';

const router = editorContainer.get(LibavoidRouter);
router.setOptions({
    routingType: RouteType.Orthogonal,
    segmentPenalty: 50,
    idealNudgingDistance: 4,
    nudgeOrthogonalSegmentsConnectedToShapes: true,
    nudgeOrthogonalTouchingColinearSegments: true
});
```

`LibavoidRouter` supports the same options as the original libavoid router.

Edges support the same parameters as `ConnRef` in `libavoid`(they are also listed in subsection 4.2.2). Example of edge object:

```
{
  id: "edge1",
  type: "edge:straight",
  sourceId: "node0",
```

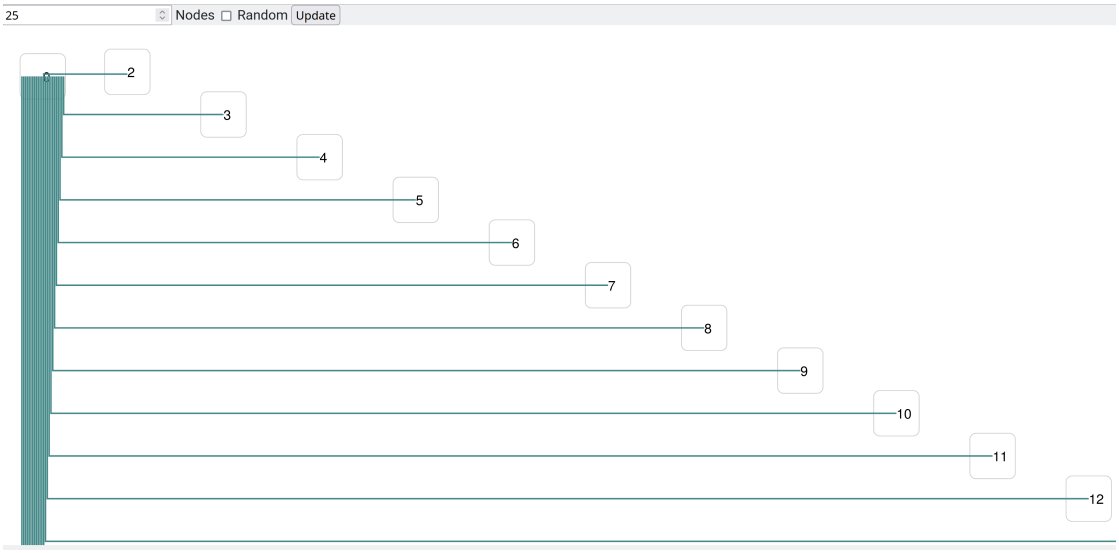

Figure 4.1: Default look of example application

```
  targetId: "node1",
  routerKind: "libavoid",
  hateCrossings: true
}
```

The whole example, which is also live demo [Hna22a], of sprotty-based application with libavoid-js router is available in our repository [Hna22b]. There are possibilities to test diagrams with a different number of nodes and also random placement.

CHAPTER 5

Evaluation

In this chapter, we will show a usage of `LibavoidRouter` from `sprotty-routing-libavoid` in a real-world application. This showcase demonstrates that `LibavoidRouter` is a production-ready tool and can be successfully integrated into real sprotty-based applications.

5.1 Performance Benchmarks

To measure the performance of JS bindings of libavoid, performance tests in `libavoid` and its port to JavaScript were implemented and performed. In this test, there are:

- 21 shapes
- 20 connections

Mean of 100 iterations(all values in ms):

Test	C++	JS
Full reroute	30	327
Moving node	21	169

Measurements show that `libavoid-js` is 8-11 times slower than `libavoid`. It's a large difference, but in our case, it is sufficient for many use cases(small, medium and large size diagrams, but not very large).

Performance of `sprotty-routing-libavoid` is even worse because the integration of `libavoid-js` router in `sprotty` router is not optimal. The router in sprotty has

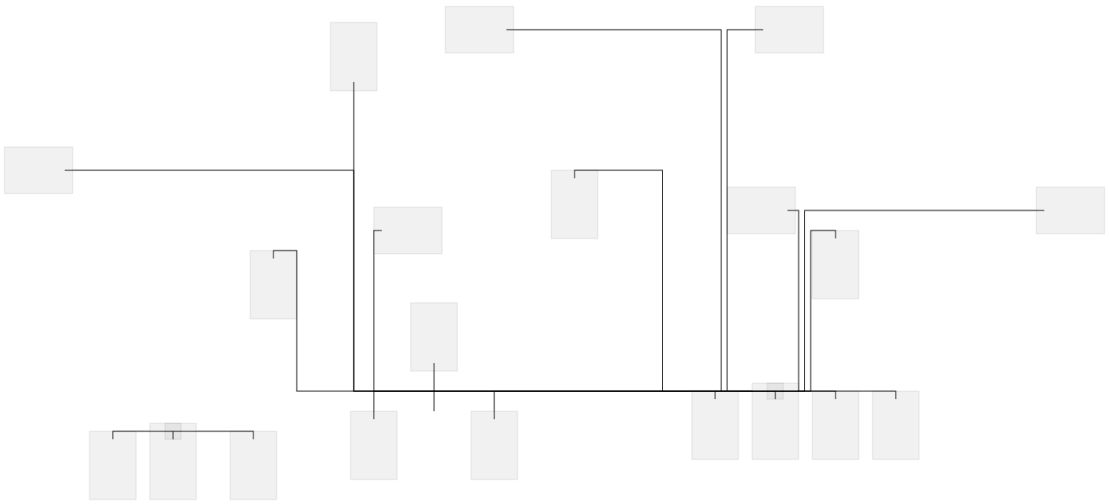

Figure 5.1: Diagram used for benchmark

a such interface, with which a diagram is rerouted after each interaction, also, if it is not needed in cases such as element hover or partial reroute(e.g. node moved) would be sufficient. `libavoid` router supports modifications of existing diagrams/incremental changes as described in section 3.1, with its usage, only affected connections would be rerouted. For the most optimal implementation, big changes in sprotty and router interface could be needed, so we solved this problem by saving of intermediate results in the router itself.

Example application with `sprotty-routing-libavoid`, which was introduced in section 4.3 on the same machine, on which benchmarks were performed, has 'good enough'(moving of element with many edges causes no lags) performance with up to 60 nodes.

5.1.1 Library Size

`emcc` supports 3 levels of basic optimizations set with '-O<num>' flag and additional optional optimizations set with other flags such as '-flto'. The meaning and effects of all of them are explained in `emcc` documentation [Conc].

We tested all 3 levels for `libavoid-js`, but only with levels 1 and 2 library worked as expected, with level 3 there were stability issues.

Generated code consists of two files: .wasm(WebAssembly) and .js(JavaScript). Their size with all 3 levels of optimizations was(all values are in KB):

Level	JS	JS(gzip)	WASM	WASM(gzip)	Total	Total(gzip)
debug(-g4)	381	49	3294	693	3675	742
0	228	24	928	240	1156	264
1	227	24	484	163	711	187
2	209	23	450	156	659	179
3	209	23	463	157	672	180

Results show that there is no noticeable difference in size between levels 1-3(less than 5% improvement), and only between levels 0 and 1 size was 30% decreased. We don't compare it with debug build because it is used only locally for debugging, and it is given in the table only to show all possible variants.

So the current size of the production version of `libavoid-js` is 659 KB and compressed(gzip) 179 KB.

5.1.2 Showcase

`LibavoidRouter` was integrated into BIGER modeling tool [GB21]. The BIGER tool is developed as a language server, for implementation GLSP stack is used, and it is distributed as Visual Studio Code extension [For22]. It used the default polyline router from sprotty with some basic routing configuration. There were two problems related to routing:

- default polyline router routes each edge independently and doesn't take nodes and other edges into account. It can lead to intersections. See 5.2 for example. To solve this problem at least partially, manual editing of edges like adding checkpoints and moving them, was available in BIGER, but it required manual work of the user. Routing of the same diagram by `LibavoidRouter` is shown on 5.3.
- Edges ends are fixed, and after moving a node in place of edge ends, there are checkpoints which the user should move manually. See 5.4 for example. After migration `LibavoidRouter` edge ends are moved together with the node, and manual beatifying by a user is not needed anymore. A diagram after the same move routed by `LibavoidRouter` is shown on 5.5.

Another improvement after migration to `LibavoidRouter` is that all edges are orthogonal. Earlier, the first part of the edge(line from a source point to the first corner) could sometimes be polyline, not orthogonal. This problem is recognizable on 5.2: edge from 'include' node to 'Course'. The new routing algorithm, practical benefits of its integration and other improvements released in BIGER with the router were also presented in the article "The bigER Modeling Tool" by Philipp-Lorenz Glaser and co-authors [GHHB22].

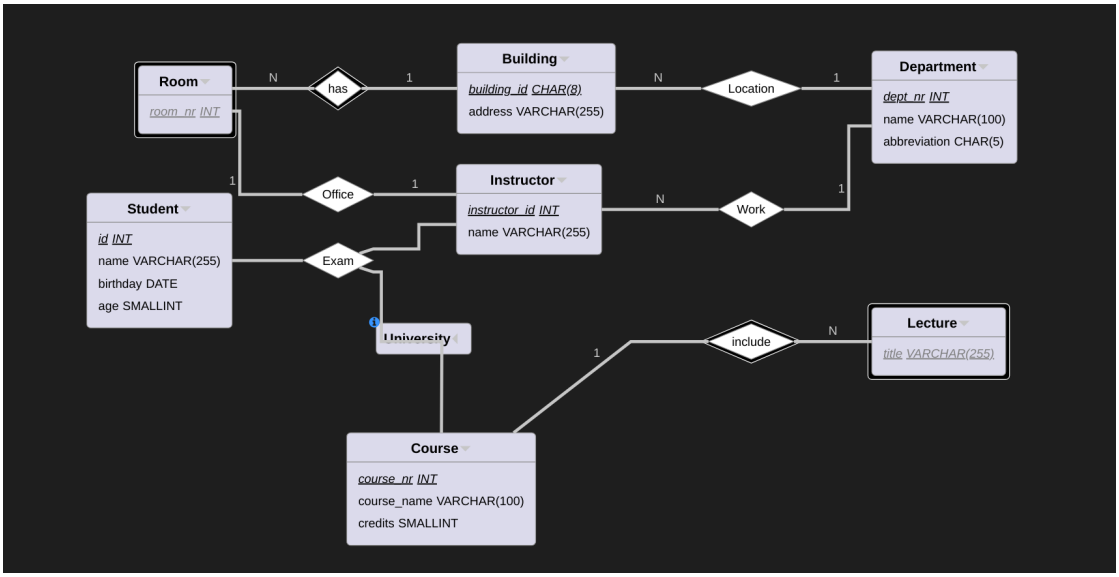

Figure 5.2: Edge from 'Exam' to 'Course' is over 'University'

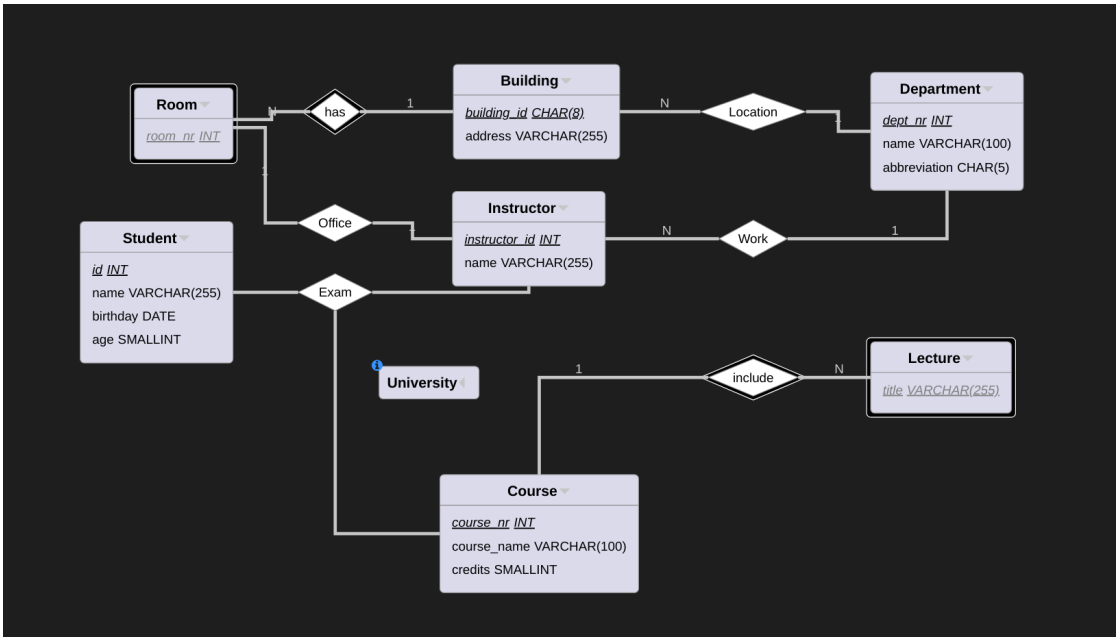

Figure 5.3: Diagram routed by LibavoidRouter

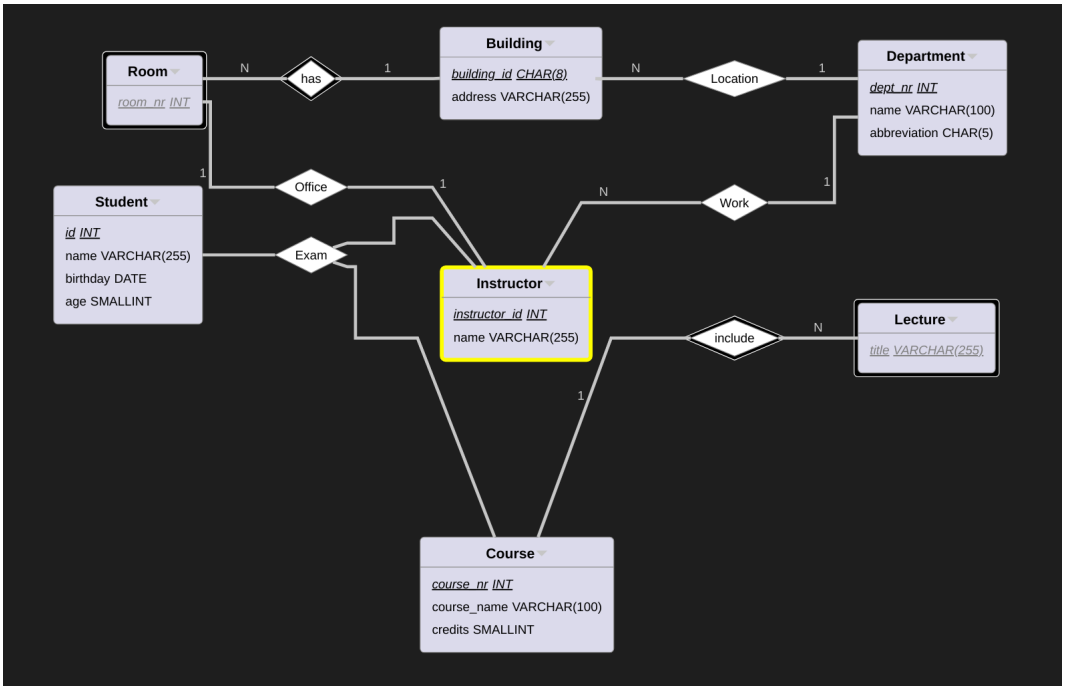

Figure 5.4: 'Course' and 'Instructor' nodes were moved

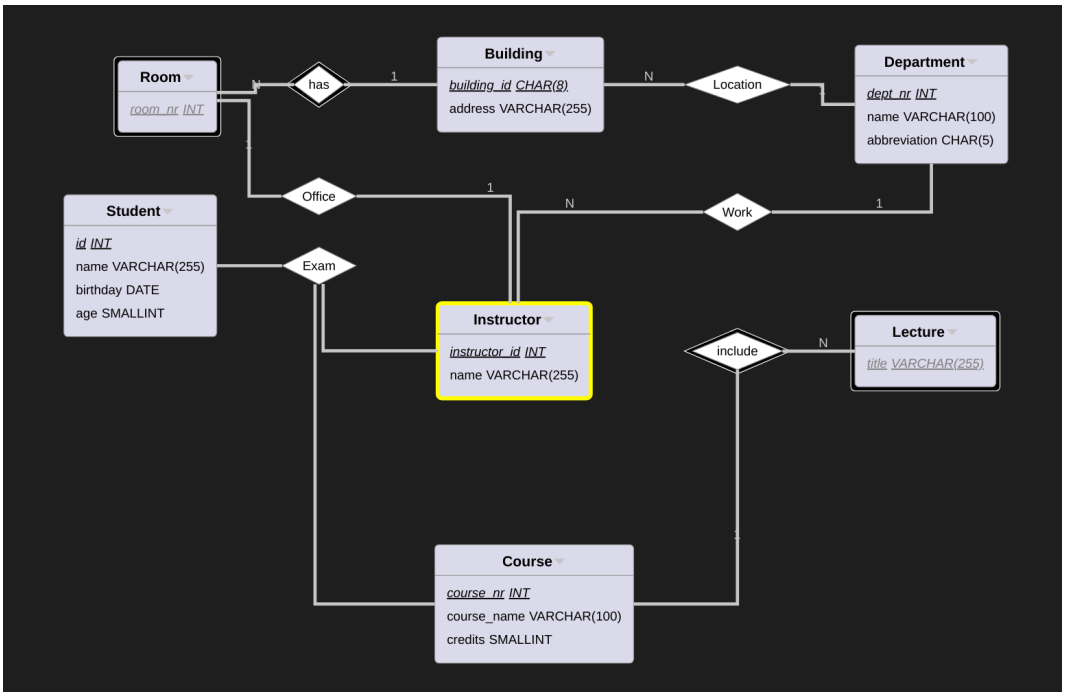

Figure 5.5: Diagram routed by LibavoidRouter

CHAPTER 6

Further work & Conclusions

In this work approach of creating WebAssembly bindings for the existing C++ code base was validated and successfully applied on libavoid library from adaptagrams package. libavoid-js library is production-ready, we showed a real-life usage example in 5, but also further improvements are also possible:

6.1 Further work

- add missing functionality in WebAssembly port of libavoid.
  The main part of libavoid functionality is included in libavoid-js, but there are two more features not included in JavaScript port of libavoid, that can be useful for some use cases: junctions and checkpoints.

- improve performance of libavoid-js by applying more compiler optimizations.
  Note: we tried to apply more optimizations in emcc, but there were problems with the stability of the result code. The reason is unknown yet.

- improve libavoid-js integration in sprotty.
  As we explained in subsection 4.2.1 section 5.1, the router interface could be improved by applying incremental changes to get better performance.

- create bindings for other libraries from adaptagrams package: libcola, libdialect, libvspc, libtopology.
  Libavoid was successfully ported to WebAssembly, so other libraries from adaptagrams package can also be ported in the same way. It would allow getting more advanced features in web stack and sprotty without big effort for implementation from scratch.

6.2 Conclusion

The aim of this work was to validate the approach of porting C++ code to WebAssembly for a library that provides routing functionality for interactive diagram editor. libavoid library was successfully ported to WebAssembly using emscripten toolset, performance of C++ and JavaScript/ WebAssembly versions were compared and then integrated into sprotty framework as an optional router.

The resulting libavoid-js library has high enough performance for many use cases, except very large diagrams. sprotty-libavoid-routing library has lower performance because of the limited interface for router integration in sprotty and is usable only for small and medium diagrams(up to 36 nodes in our example application). But it can be improved by improving the router interface in sprotty, its discussion is going on.

Also, different levels of optimizations in emcc compiler were tested to get a minimal size of libavoid-js library. The last version has a size 659 KB with optimization level 2 and 179 KB if compressed using gzip.

Bibliography

[BCOR15] Jean-Michel Bruel, Benoit Combemale, Ileana Ober, and Hélène Raynal. Mde in practice for computational science. *Procedia Computer Science*, 51:660–669, 2015. International Conference On Computational Science, ICCS 2015.

[BCPP20] Antonio Bucchiarone, Jordi Cabot, Richard F. Paige, and Alfonso Pierantonio. Grand challenges in model-driven engineering: an analysis of the state of the research. *Software and Systems Modeling*, 19(1):5–13, Jan 2020.

[BSEB] Myles Borins, Yulia Startsev, Daniel Ehrenberg, and Guy Bedford. Tc39. `https://tc39.es/proposal-top-level-await/`. Accessed: 03 February 2023.

[Cona] Emscripten Contributors. Embind. `https://emscripten.org/docs/porting/connecting_cpp_and_javascript/embind.html`. Accessed: 03 February 2023.

[Conb] Emscripten Contributors. Emscripten. `https://emscripten.org/`. Accessed: 03 February 2023.

[Conc] Emscripten Contributors. Emscripten code optimization. `https://emscripten.org/docs/optimizing/Optimizing-Code.html`. Accessed: 03 February 2023.

[Cond] Emscripten Contributors. Pointers, references and values in webidl. `https://emscripten.org/docs/porting/connecting_cpp_and_javascript/WebIDL-Binder.html#pointers-references-value-types-ref-and-value`. Accessed: 03 February 2023.

[Ems] Emscripten. Emsdk. `https://hub.docker.com/r/emscripten/emsdk`. Accessed: 03 February 2023.

[For22] Luca Forstner. Integrating glsp based tooling into visual studio code. 2022.

[Fou] LLVM Foundation. Llvm. `https://llvm.org`. Accessed: 03 February 2023.

[GB21] Philipp-Lorenz Glaser and Dominik Bork. The bigER tool - hybrid textual and graphical modeling of entity relationships in vs code. In *2021 IEEE 25th International Enterprise Distributed Object Computing Workshop (EDOCW)*, pages 337–340, 2021.

[GHHB22] Philipp-Lorenz Glaser, Georg Hammerschmied, Vladyslav Hnatiuk, and Dominik Bork. The bigER modeling tool. 3211:1–4, 2022. ER Forum and PhD Symposium 2022.

[Groa] Web Hypertext Application Technology Working Group. Webidl. `https://www.w3.org/TR/WebIDL/`. Accessed: 03 February 2023.

[Grob] Web Hypertext Application Technology Working Group. Webidl data types. `https://webidl.spec.whatwg.org/#idl-types`. Accessed: 03 February 2023.

[Hnaa] Vladyslav Hnatiuk. Adaptagrams fork. `https://github.com/Aksem/adaptagrams`. Accessed: 03 February 2023.

[Hnab] Vladyslav Hnatiuk. libavoid-js. `https://github.com/Aksem/libavoid-js`. Accessed: 03 February 2023.

[Hnac] Vladyslav Hnatiuk. Shapeconnectionpin api issue. `https://github.com/Aksem/adaptagrams/issues/8`. Accessed: 03 February 2023.

[Hna22a] Vladyslav Hnatiuk. sprotty-routing-libavoid-demo. `https://aksem.github.io/sprotty-routing-libavoid-demo/`, 2022. Accessed: 03 February 2023.

[Hna22b] Vladyslav Hnatiuk. sprotty-routing-libavoid-demo source. `https://github.com/Aksem/sprotty-routing-libavoid-demo`, 2022. Accessed: 03 February 2023.

[HRS+17] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the web up to speed with webassembly. *SIGPLAN Not.*, 52(6):185–200, jun 2017.

[HtEA] Vladyslav Hnatiuk(fork) and the Emscripten Authors(original). Webidl embdingen. `https://gitlab.com/Aksem/webidl-embindgen`. Accessed: 03 February 2023.

[KM14] M. Wybrow K. Marriott, P.J. Stuckey. Seeing around corners: Fast orthogonal connector routing. *In Proceedings of the 8th International Conference on the Theory and Application of Diagrams(Diagrams 2014)*, page 31–37, 2014.

[MW10] P.J. Stuckey M. Wybrow, K. Marriott. Orthogonal connector routing. *In Proceedings of 17th International Symposium on Graph Drawing (GD '09)*, page 219–231, 2010.

[REIWC18] Roberto Rodriguez-Echeverria, Javier Luis Cánovas Izquierdo, Manuel Wimmer, and Jordi Cabot. Towards a language server protocol infrastructure for graphical modeling. In *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, pages 370–380, 2018.

[Ros] Andreas Rossberg. Esm integration proposal. `https://github.com/webassembly/esm-integration`. Accessed: 03 February 2023.

[Wyb] M. Wybrow. Adaptagrams api documentation. `https://www.adaptagrams.org/documentation/annotated.html`. Accessed: 03. February 2023.