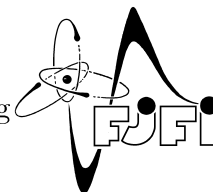




CZECH TECHNICAL UNIVERSITY IN PRAGUE  
Faculty of Nuclear Sciences and Physical Engineering



# Automatic Box Layout in I3T Tool

## Automatické rozmisťování propojených modulů v nástroji I3T

Bachelor's Degree Project

Author: **Marek Nechanský**  
Supervisor: **Ing. Petr Felkel, Ph.D.**  
Language advisor: **Mgr. Hana Čápová**  
  
Academic year: 2018/2019



- Zadání práce -

- Zadání práce (zadní strana) -

*Acknowledgment:*

I would like to thank Petr Felkel for his expert guidance and express my gratitude to Hana Čápková for her language assistance.

*Author's declaration:*

I declare that this Bachelor's Degree Project is entirely my own work and I have listed all the used sources in the bibliography.

Prague, July 8, 2019

Marek Nechanský



*Název práce:*

**Automatické rozmisťování propojených modulů v nástroji I3T**

*Autor:* Marek Nechanský

*Obor:* Aplikovaná informatika

*Druh práce:* Bakalářská práce

*Vedoucí práce:* Ing. Petr Felkel, Ph.D., České vysoké učení technické v Praze, Fakulta elektrotechnická, Katedra počítačové grafiky a interakce

*Abstrakt:* Důležitou roli při pochopení grafu hraje jeho grafická reprezentace. Vykreslení grafu by mělo být závislé na typu grafu. Hodně specifický typ grafu je používán v programu nazvaném Interactive Tool for Teaching Transformations. Tento program je používán k výuce transformací a uživatel by měl jednoznačně z vykresleného grafu poznat, co graf reprezentuje. Existuje spousta technik na vykreslování grafů, ale pouze typ rozmisťování zvaný Sugiyama-style graph drawing je vhodný pro tento typ grafu. Tato technika je hodně obecná a implementace může být navržena speciálně pro grafy v tomto výukovém programu.

*Klíčová slova:* Rozložení grafu, Rozmisťovací techniky, Sugiyama-style graph drawing, Vizualizace grafu

*Title:*

**Automatic Box Layout in I3T Tool**

*Author:* Marek Nechanský

*Abstract:* A drawing of a graph has an important impact on understanding of the graph. The drawing should be different for various types of graphs. A very specific type of graph is used in a program called Interactive Tool for Teaching Transformations. This program is used for teaching transformations and the user should definitely understand what a graph represents from a layout of the graph. There are many techniques for creating a layout of a graph, but only a technique called Sugiyama-style graph drawing is suitable for this application. This technique is very general and the implementation can be designed for graphs in this program.

*Key words:* Graph visualisation, Graph Drawing, Layout techniques, Sugiyama-style graph drawing





# Contents

<b>1</b>	<b>Introduction</b>	<b>13</b>
1.1	Interactive Tool for Teaching Transformations . . . . .	13
<b>2</b>	<b>Research</b>	<b>15</b>
2.1	Graphs in I3T . . . . .	15
2.2	Programs with similar graph structures as I3T . . . . .	18
2.2.1	Diagram creating websites . . . . .	19
2.2.2	Public Implementation of Graph Algorithm Library and Editor . . . . .	19
2.2.3	Graphviz - Graph Visualization Software . . . . .	20
2.2.4	Blender . . . . .	20
2.2.5	Unreal engine . . . . .	20
2.3	Requirements of a proper graph layout . . . . .	20
2.4	Layout Techniques . . . . .	22
2.4.1	Interactive layout techniques . . . . .	22
2.4.2	Semi-autonomous layout techniques . . . . .	23
2.4.3	Autonomous layout techniques . . . . .	23
<b>3</b>	<b>Refactorization of classes representing I3T modules</b>	<b>31</b>
3.1	Current state of the classes representing modules . . . . .	31
3.2	Changes to the classes representing modules . . . . .	32
<b>4</b>	<b>Implementation of the layout techniques</b>	<b>37</b>
4.1	Interactive techniques . . . . .	37
4.2	Semi-autonomous technique . . . . .	38
4.3	Autonomous technique . . . . .	39
4.3.1	graph structure used in the technique . . . . .	39
4.3.2	Cycle removal . . . . .	41
4.3.3	Layer assignment . . . . .	41
4.3.4	Order creation . . . . .	42
4.3.5	Coordinate assignment . . . . .	43
4.4	Testing of layout techniques . . . . .	44
4.5	Results . . . . .	44
	<b>Summary</b>	<b>47</b>

**A Results of layout techniques** **49**

A.1 Interactive layout techniques . . . . . 49

A.2 Semi-autonomous layout technique . . . . . 53

A.3 Autonomous layout technique . . . . . 54

**B contents of the enclosed CD** **59**

# List of Figures

2.1	Window of I3T with basic scene . . . . .	15
2.2	Modules with specific structures . . . . .	16
2.3	transformation modules . . . . .	16
2.4	Basic diagram created in Lucidchart . . . . .	19
2.5	Sequence cycle . . . . .	25
2.6	Screen cycle . . . . .	25
2.7	Deformed projection in a screen module . . . . .	26
2.8	An example of two modules with a different start and end edge directions . . . .	29
3.1	An example of a sender operator module . . . . .	32
4.1	An edge crossing with four modules . . . . .	42
4.2	An edge crossing with three modules . . . . .	43
A.1	Example of a usage of horizontal alignment . . . . .	49
A.2	Example of a usage of horizontal distribute . . . . .	50
A.3	Example of a usage of even horizontal distribute . . . . .	50
A.4	Example of a usage of vertical alignment . . . . .	51
A.5	Example of a usage of vertical distribute . . . . .	51
A.6	Example of a usage of even vertical distribute . . . . .	52
A.7	Example of a usage of distribute . . . . .	53
A.8	First version of the layout technique on a part of a scene called armAnimated . .	54
A.9	Final version of the layout technique on a part of a scene called armAnimated . .	54
A.10	First version of the layout technique on a scene called 01_modelTransformation- Graph . . . . .	55
A.11	Final version of the layout technique on a scene called 01_modelTransformation- Graph . . . . .	55
A.12	Laid out scene called 03_rotateAroundPoint . . . . .	56
A.13	Laid out scene called 03_rotateAroundPoint-1 . . . . .	56
A.14	Laid out scene called 05_lookAt . . . . .	56
A.15	Scene 09_frustumMultiMonitor laid out by a user . . . . .	57
A.16	Scene 09_frustumMultiMonitor laid out by the layout technique . . . . .	57
A.17	Laid out scene called 10_quaternionMatrixComparison . . . . .	57
A.18	First version of the layout technique on a scene called 10_quaternionMatrixCom- parison . . . . .	58



# Chapter 1

## Introduction

A layout of a graph has consequences on the understandability of the graph. The preferable layout can differ for different usage of a graph and properties of a graph can also change the preferable layout. The more specific are graphs, which should be laid out by a technique, the more specialized should the technique be.

Graphs, which will be studied in this project are directed graphs with exceptional directed cycles. Even though they are generally not planar, they can be structured in a way that their layout is more readable than when all the nodes are placed inappropriately. Users of various applications do not pay attention to the structure of their work and they focus on the functionality. That is why there are refactoring tools in integrated development environments or alignment features in Powerpoint. Target of those functionalities are to let user utilize correctly the application and afterwards it organizes their creation to be easier to work with in the future. The result of this thesis should be a tool that can help users in the same way as refactoring or alignment, however on the graph structures.

The objective of this project is improving the user interface for part of Interactive Tool for Teaching Transformations (hereinafter referred to as I3T) that can be described as a graph of transformations and operators. Specifically, The target is to design and implement algorithms, which can help user to lay out graphs. These algorithms should improve the understandability of the application and the effectivity of the learning process when using I3T. Layout techniques will be tested by users to make them as user-friendly as possible. To create these functionalities improvements will be necessary in the base code of I3T and will result in refactoring parts of the code concerning graphs. The target of this refactoring is to increase readability of code for future development and to remove mistakes caused by inefficient code.

### 1.1 Interactive Tool for Teaching Transformations

I3T is a software used for teaching 3D transformations and their usage in a visual and interactive way. I3T was created by Michal Foltá in 2016 as a masters thesis at the Department of Computer Graphics and Interaction, Faculty of Electrical Engineering, Czech Technical University in Prague. Since then Foltá's supervisor Ing. Petr Felkel, Ph.D. continues leading the development of the whole project [1].

I3T has a graphical user interface (hereinafter referred to as GUI) divided to two parts: a scene and a workspace. In the workspace a user can create and connect modules representing mathematical operations. Matrices, which can be result of these operations, can be used on 3D objects to change their properties. All modules are represented as boxes with data. These

boxes can be connected by curves that represent the connection of an output of one operation to an input of another operation. Transformed objects are rendered in a 3D scene view, which is interactive and the user can easily understand what the transformations did to the object.

Even though I3T is complete software in terms of functionality, there are many drawbacks in its visual appearance and its interactivity with a user. There are already drafts of visual improvements and upgrading of an interactivity is the objective of theses supervised by doctor Felkel.

## Chapter 2

# Research

In this chapter, graphs, used in I3T, are in-depth described in the Section 2.1. There are a few paragraphs dedicated to applications using similar graphs and graph structures like the ones in I3T 2.2. There is a section dedicated to requirements of a proper graph layout 2.3. After that, interactive, semi-autonomous and autonomous layout techniques are described 2.4.

### 2.1 Graphs in I3T

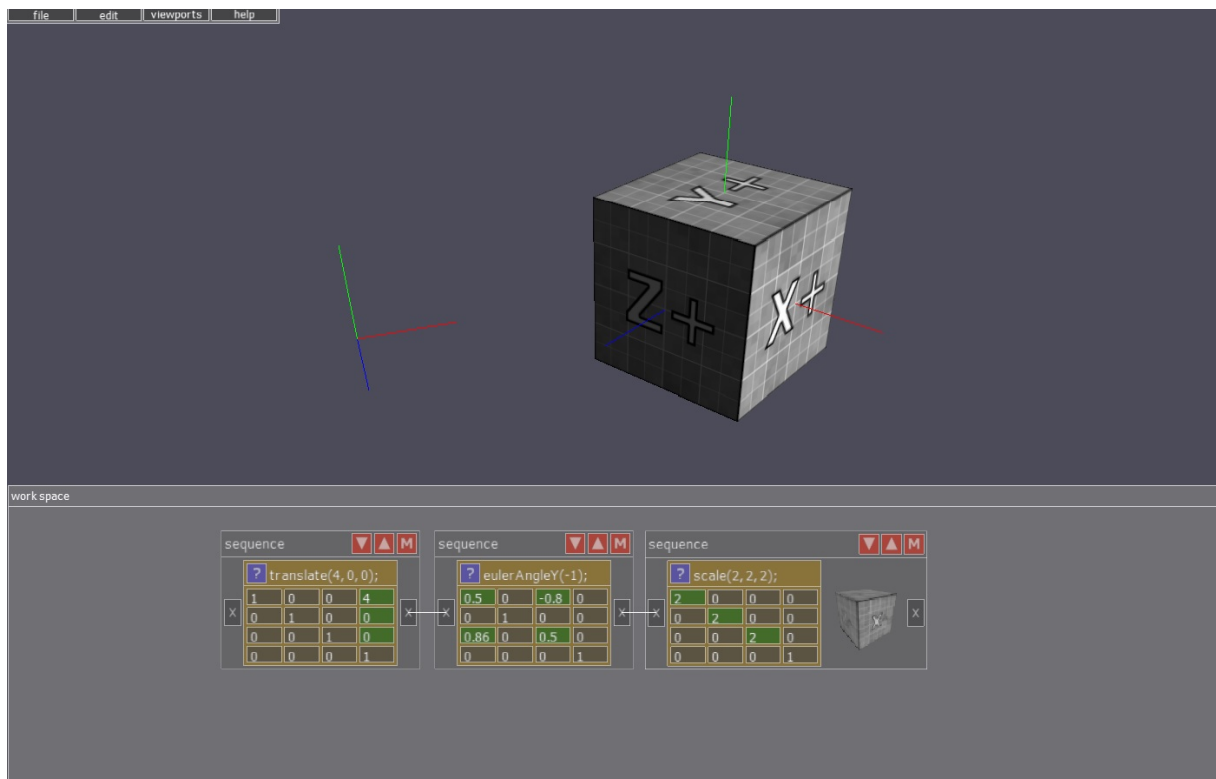
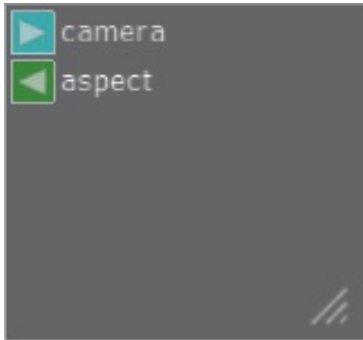
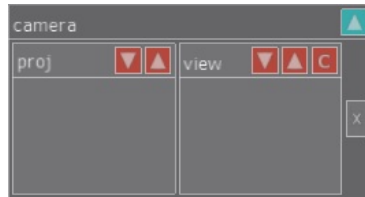


Figure 2.1: Window of I3T with basic scene

Graphs that can be seen in I3T have nodes represented by modules with a rectangular shape and edges are curves connecting modules. Modules represent mathematical operations, such as



(a) A screen module



(b) A camera module

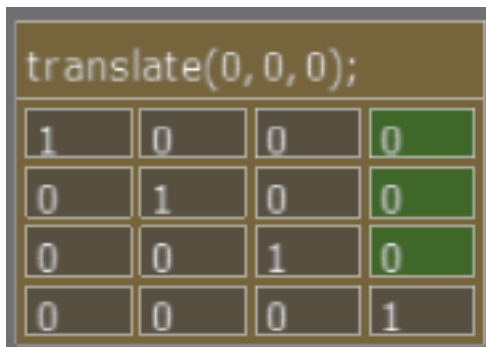


(c) A trackball module

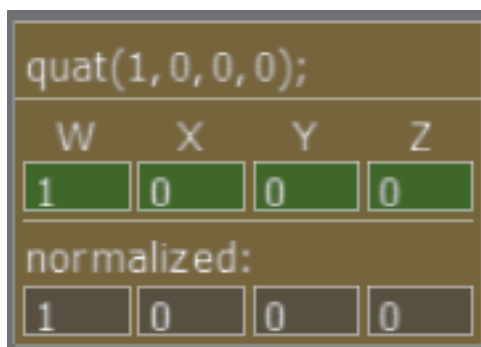


(d) A float cycle module

Figure 2.2: Modules with specific structures



(a) matrix transformation module



(b) quat transformation module

Figure 2.3: transformation modules



matrix multiplication, cosine of an angle. Modules have inputs and outputs. Inputs represent variables for the operation of the module and outputs are usually the outcome of the operation. Output of one module can be connected with a curve representing an edge to an input of another module. Most modules have 4 basic parts: an input part, output part, label part and central part. The input part contains one or more inputs and it is on the left side of a module and the output part contains one or more outputs and it is situated on the right side of a module. If a module has the label part it is always situated on the top side of a module. On the left side of the label part is a name or label of a module. On the right side of the label part can be more inputs and outputs. The central part usually displays data, which comes out on outputs of the module. For example, a matrix for a matrix multiplication or a float for a cosine of an angle. In some modules, a user can change the output of the module using GUI of the central part.

Inputs and outputs of a module are represented by rectangles with an arrow on them. The arrow represents a direction of an input/output and it changes how the curve representing edge will be drawn. Direction of these arrows is usually to the right. This fact implies the flow of the graph [8]. Therefore, an edge should have its source on the left side and its sink on the right side of a workspace. A module, which is on the end of an edge should be more on the right in a final layout than the one on the start of the edge. There are three modules with different directions of inputs or outputs. Sequence modules has an input and two outputs in the label part. Outputs direction is up and the input direction is down. The camera module has an output in the label part with the upward direction. In addition to that, there are two sub-modules in the central part, which have the same input and two outputs as a sequence module. A Screen module does not have the label part and its output is in the input part going leftwards. When an input or output of a module is mentioned in the following text, it means the module or the node, which is on the other endpoint of an input or output edge respectively. The place, where the edge begins and ends are called input tab and output tab, respectively.

The I3T workspace have three types of modules. The first type are transformation modules. They do not have any inputs and outputs and they only have a central and label part. They represent transformations such as a translation or rotation. A matrix of a transformation module is displayed in a central part and the key parts of the matrix can usually be changed. Their usage is to be put in sequence modules, which are the second type. Sequence modules represent a transformation that is applied to an object as seen in Figure 2.1. Transformation modules can be dragged inside a sequence module. The multiplication of these transformation modules is the matrix of the sequence module. An object can be bound to a sequence module. Vertices of the object are then transformed by the matrix of the sequence module. Sequence modules have a special input and output in the input and output parts. Two sequence modules can be connected by them. The matrix of the module on the start of the edge is multiplied by the matrix of the other module and then stored as the matrix of the other module. This type of input or output is called multiplication input or output. This can be done multiple times. For example, there are three sequence modules, where first module is connected to the second module and second one to the third one with multiplication edges. The matrix of the first module is the multiplication of transformations in the first module. The matrix of the second module is the multiplication of the multiplication of transformations of first module and multiplication of transformations of the second module. The matrix of the third sequence module has the multiplication of all transformations in all three sequence modules. This property can be used to create a graph of a scene with sequence modules. Sequence modules have two outputs in the label called storage output and matrix output. The storage output is a matrix, which is created by the multiplication of transformations in a sequence module. The matrix output is the matrix

of a sequence. Sequence modules have one input in the label part. It is called storage input and it takes a matrix and works with it like with matrices of transformations.

The third type of modules are operators. Each operator represent a different operation. Their inputs are used as operands of an operation and outputs are results of the operation, e.g., multiplication of two floats have two float inputs and a float output, which is the multiplication of values that are on inputs. Operators that outputs matrices can be connected to a sequence module to put the matrix inside the sequence module, in order to change properties of an object bound to the sequence module.

Special modules are camera, screen, trackball and float cycle modules, which have a very different structure and have to be used differently 2.2. The camera module has a complicated central part, which is not important for this project. It has an output tab in the output part, which is connected to all sequence modules without a multiplication input. This connection represents that all the vertices displayed on the screen have to be multiplied from the left by a perspective and view matrices. A screen module has only a central and input parts. It has one input and one output. The input is a camera module and output is an aspect ratio of the screen. In its central part, the scene, which is captured by the camera of the camera module, is displayed.

A trackball module has a unique and interactive central part, where a user can choose a rotation or create its own rotation by rotating a trackball. It has only one output and that is the rotation matrix created by the central part. The Float cycle creates a float, which is sent to the output, which starts on one value and each tick it increases its value by a step and when the target value is reached, the value is reset to the starting one. Its main output is the changing value. It has other complicated inputs and outputs, which can alternate the cycle, but they are not important for this project.

This structure of modules is very similar to how programmers change object's position and form with matrices and mathematical operations in 3D computer graphics. Mathematical operations result in matrices that are multiplied in some order and then the final matrix is multiplied by homogeneous coordinates of vertices of an object. Even though it is a good representation of operations that are used in the computer graphics programming the structure might be a little confusing, because there are matrices inside sequence modules, which do not work when they are out of a sequence module and other matrices that cannot be put inside sequence modules, but they have inputs and outputs and have to be connected by an edge to the sequence modules. Fortunately, there is a nice short video about the usage of I3T on the I3T web [1], which explains basics of the program very easily.

## 2.2 Programs with similar graph structures as I3T

There are a lot of programs or web pages that contain graphs with similar structure. UML diagrams or flowcharts and many more types of graphs have the similar properties as graphs that are used in I3T. The most important structure difference are directions of edges. The fixed position of input and output tabs within modules is not usual. Nevertheless, we can study and apply similar techniques and methods, that make those graphs more user-friendly, to graphs in I3T.

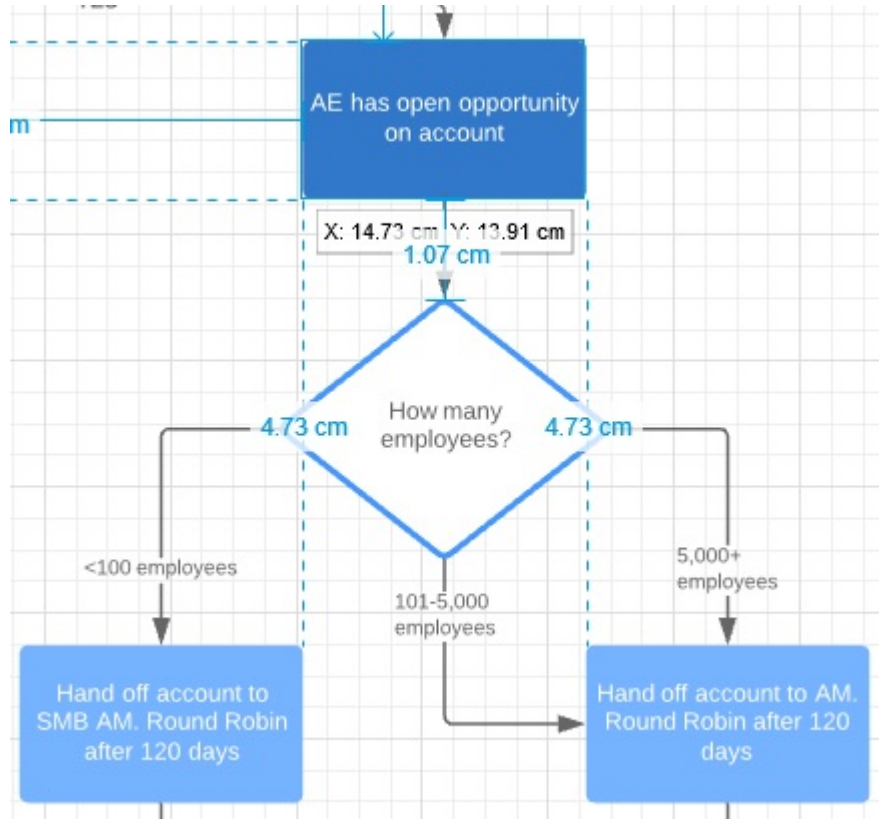


Figure 2.4: Basic diagram created in Lucidchart

### 2.2.1 Diagram creating websites

First website, whose target is creating diagrams, is Lucidchart [4]. Although Lucidchart's purpose is very different than the one of I3T, there are very useful tools that make graph creation and placement easier. The whole workspace area has grid background that is used by users to place nodes on the same line and have reasonable spaces between nodes. When node is being dragged around the workspace, lines appear to indicate with which nodes the current node is aligned as seen in Figure 2.4. Functions like this can help users to lay out nodes in a structured way. There are also various functions that arrange selected nodes and edges. "Auto Layout" is a set of functions that create new layout for the selected nodes and even though function like this might be useful in I3T it is not very well implemented in Lucidchart. Most of algorithms used for the arranging in LucidChart destroys the structure of a graph. A set of functions called "Align Objects" moves selected nodes so that they are horizontally or vertically arranged. Users might also choose which part of nodes should be on the same line and these simple functions can easily improve readability of selected parts of the graph.

### 2.2.2 Public Implementation of Graph Algorithm Library and Editor

Public Implementation of Graph Algorithm Library and Editor or Pigale in short is a c++ program containing graph algorithms [5]. It uses many different drawing and layout techniques such as orthogonal layout and visibility layout. These techniques are usable on I3T graphs and they can create user-friendly and readable layout. On the other hand these layouts cannot be

used on graph with as graphical representation as those in I3T. In Pigale, nodes are represented by a rectangular with variable size. Algorithms can work with the size to create a better layout.

### 2.2.3 Graphviz - Graph Visualization Software

Graphviz is an open source graph visualization software [6]. It creates graphs and diagrams using commands. It has many different features and functionalities for different types of graphs. Most important tool is *dot*. It creates layered drawings of directed graphs. Its purpose is to create readable and user-friendly layout from an input. Why is this tool important for this project is described in the Section Autonomous layout techniques 2.4.3.

### 2.2.4 Blender

Program containing graphs, that are closer to I3T in terms of theme, is Blender. Blender is a free and open source 3D creation set [2]. Structure of graphs in Blender is the same as the structure of the operator structure in I3T. This type of graph is used to adjust properties of some entity, e.g., properties of material or texture. Blender does not have many usable function on its own, but there are plugins helping with graph editing and laying out.

Notable plugin is called NodeArrange. It has many functions for rotating scaling and moving nodes. Very useful from user perspective is a function that distributes nodes. It destroys collisions between nodes and creates space between them. This can be also used in horizontal or vertical form, which moves all node from barycenter, which creates more space for edges and makes the graph more readable. NodeArrange can also create simple layout, which respect directions of edges and it tries to move nodes connected by an edge closer. It is implemented by similar algorithm as in *dot*. Even though the graphs from Blender are similar to graphs in I3T in this layout function the graph is worked with like the graphviz graphs and even though the idea is correct, it cannot be easily applied on I3T graphs. Another plugin that is very helpful when laying out graphs is Node Wrangler. Its main functionalities are aligning nodes. This type of alignment is also used in Lucidchart and they are applied well on the structure of Blender graphs.

### 2.2.5 Unreal engine

Comparable graphs are used in program called Unreal engine [3]. It is a game engine and it uses for events such as mouse clicks, where the nodes contain functions and edges indicate parameters of nodes. Unreal engine's graphs have a lot of in common with Blender's graphs. There are also implemented aligning functions with adjustable space between nodes and also many different types of alignment. For example nodes can be aligned to the top node or to the bottom node. Also the nodes can be aligned by their center, top or bottom. And these functions are also present for the vertical usage. There are also distribute functions for horizontal and vertical usage, which create space between the nodes. In short Blender and Unreal engine has very similar graphs and also almost same functions for graph editing.

## 2.3 Requirements of a proper graph layout

The target of layout techniques is to fulfill requirements of the proper graph layout. Most of these requirements should be fulfilled by autonomous layout techniques. Interactive and semi-autonomous techniques should help a user to fulfill them, but the result is dependent on the work of the user. The main target of techniques is readability of the graph, but it has to be

separated to achievable components. These components are often same for all graphs, but a few components are specific for certain type of graph or certain usage of graph [20]. All requirements have their exceptions, but these exceptions are so rare that the technique cannot always detect them. Another problem is with fulfilling these requirements simultaneously, because usually there is no perfect layout and a technique has to decide which is more important for the layout. Therefore, these requirements are meant to improve the vast majority of graph drawings. All the requirements that are not special for I3T graphs are from articles [9, 19, 20].

1. Basic requirement that is present for all types of graphs is the minimization of edge crossings. If the graph is planar then the drawing should not have edge crossings. If the graph is not planar the technique should find the least amount of edge crossings and use that layout. Deciding edge crossings is not a trivial problem for most graphs. For graphs with special properties, such as graphs in I3T, this requirement can be simplified. This simplification is used in the autonomous layout and it is described in the Section 2.4.3.
2. Another basic requirement is the minimization of the area of a graph. Even though this requirement is very important, in many cases it is solved first and then it might be worsen by fulfilling other requirements. This minimization can be achieved by dividing this requirements into two smaller requirements. First is the minimization of the lengths of edges. This can be easily achieved by moving nodes that are connected by edge as close to each other as they can. Second requirement is that the Nodes in a graph should be distributed evenly and the graph should be as symmetric as possible. The target is to avoid unused space inside the area of a graph. This cannot be always achieved, because if the directions of edges are in upwards direction, the graph will not be symmetric and it will be more filled in the top position.
3. Another requirement is to keep the edges as straight as possible and if a bend or a curve is needed, then the bend should not be sharp or the curve should be smooth. This can be achieved by prioritizing straight edges and if the edge cannot be straight the technique tries to give the edge enough space to make the bend smooth. This can create a problem with previous requirement and the priority between them has to be chosen.
4. Some usages of graphs prefer having the graph in the rectangular shape with specific aspect ratio. This requirement can be used, for example, when the graph should fit on a paper or on a slide of a presentation. This is not a problem for I3T graphs. Workspace has zoom, therefore observing whole graph is not the problem and a user can move around the workspace with mouse and focus on important parts.
5. If there is a size of a representation of a node in a graph, then there should be enough space between the nodes. The space should be big enough so representations of nodes are not colliding. The space between node representations should be big enough to fit edge representations and some more space to increase readability.
6. In I3T, sequence modules have special meaning in the graph and they should be visible and the previously mentioned requirements should be prioritized for them. They should be connected by a straight edge and the distance between them should be very short. Edges in I3T have constant direction therefore the graph should respect this direction. Also the edges in I3T are not represented by straight lines, therefore there has to be more space between nodes.

7. The starting and ending direction of curves representing edge in I3T is often to the right. This means that for each node in an I3T graph its outputs should be more on the right than the node. This requirement is a good starting condition. If a layout, where this requirement is fulfilled, is created, other requirements can be processed without changing the basic structure.

## 2.4 Layout Techniques

Layout techniques we decided to add to the I3T can be separated into three different categories: interactive, semi-autonomous and autonomous. The purpose of interactive techniques is to help the user to move multiple nodes in a precise way. Semi-autonomous layout techniques are used for larger parts of a graph. They should transform a graph in a way that it looks similar, but some requirements are done better. Autonomous techniques change the whole layout of the graph and user cannot be sure what the result will be before using it. They should fulfill most if not all the requirements for a proper graph layout for the targeted graph. First two types of techniques are usually used on parts of a graph, but autonomous techniques are meant to be used on a whole graph. This division also indicates the complexity of different categories, because interactive techniques are usually easy to implement and autonomous techniques contain complicated algorithms.

### 2.4.1 Interactive layout techniques

As mentioned above, interactive layout techniques are simple and they change the graph in a way that the user intends them to. These techniques should help the user to create structured graph with less steps. Specifically, they should arrange nodes selected beforehand by the user to an easy and understandable structure. These techniques are not meant to be used on whole graphs or a lot of modules, because their goal is to help with the movement of small groups of nodes, such as walks within the graph or a small sub-graphs of a graph. There are a lot of different interactive techniques because every software discussed above has some. Majority of them are not very useful in I3T and therefore only the techniques that help with the specific structure of graph the I3T uses are mentioned.

First interactive technique is alignment in one direction (vertical or horizontal). This technique aligns all selected nodes to one line (vertical or horizontal). There are many different points that could be on one line and there should be different techniques for these. For horizontal alignment, the top, bottom, center of all selected nodes or their connection (edge) should be aligned. It can be used after creating modules to have them nicely in one line before thinking about more complex structure.

Next technique distributes selected tabs in one direction (vertical or horizontal). It finds a node in the center of selected nodes in chosen direction, and moves all other tabs away or closer to it in a way that the gap between a node and a node next to it is always the same. It does not change position in other direction than the chosen one. The goal of this technique is similar to the first one's and it only helps move the modules quickly and precisely. Last interactive technique can be also vertical or horizontal. It distributes modules in the specified direction. The difference is that nodes with the lowest and highest coordinate in the direction will stay on their place and other modules are evenly laid out between these nodes with same space between them. This might be the more usable technique than the normal distribution. It can be used, when selected nodes have a boundary, where they should be laid out.

### 2.4.2 Semi-autonomous layout techniques

Semi-autonomous layout techniques have a few properties similar to the interactive layout techniques. They also tries to move modules the same way as the user would. They are more complicated and its functionality might vary on the implementation. Its purpose is not that specific as for the interactive techniques. The layout after the application of this technique might be different than expected by a user. The technique fitting I3T distributes nodes in both directions. This can be done in many ways and it can be used for more purposes. The most suitable technique is the one that does not change the order<sup>1</sup> of the nodes and only tries to create more space between them. It is done by finding the middle module and moving all the modules away from it. The problem is how far they should be moved. If one module is not moved enough there could still be collision between modules. If one module is moved too far it could destroy the whole mental map of the graph [10]. Modules are processed from the ones closer to the middle module to the ones that are far away. This method could still break the mental map if the graph is complicated, but the purpose of this method is not to lay out complex graphs, but to help improve the distribution of a part of graph that can be moved to the right position afterwards.

### 2.4.3 Autonomous layout techniques

Autonomous layout techniques read a structure of the given graph and output a drawing of the graph. The output contains positions of nodes and descriptions of curves in the graph [9]. Curves in the I3T are described by positions of connected nodes, therefore we use these techniques to place the nodes to the optimal layout. These techniques do not preserve mental map [10] and therefore user cannot expect previous layout to influence the layout created by this type of technique. Despite that, autonomous techniques follow rules that are used by users when creating their own layout, which are implications of requirements of a proper graph layout. If a technique is designed and implemented well, the user's expected result should be similar or worse than the result of the technique.

There are many types of graphs and each of them has different structure and optimal layout. For example the layout for a rooted tree is different than the layout for Directed acyclic digraphs [9]. Also the meaning behind nodes and edges is important for the layout. Graphs, whose nodes represent countries and edges are between countries that are neighbors, can be laid out using force-directed algorithm [11], which uses forces between nodes and iterates until the layout is consistent. This approach cannot be used on a graph, where the direction of the edge has some meaning. That is why there has to be a different layout for a different usage. Another approach is to detect fragments in the graph, for example cycles, sequences or branching. These fragments are usually easy to order and lay out and recursively this could be done for smaller fragments. Unstructured fragment of a tree could be lay out by some basic algorithm [18]. Unfortunately this approach does not work well with I3T graphs, because of the number of different unstructured fragments and their common appearance. This algorithm is used on the graphs that have only one source and one sink nodes. Another idea was to recreate these rules to fit the graphs we use, but due to the complexity of the I3T graphs this could not be done in better method than the one used.

I3T graphs have similar structure to directed acyclic graphs. Even though directed cycles occur in these graphs, they are very specific and easy to handle. They have fixed starting and ending directions of a curve representing edges. These graphs can be laid out with hierarchical drawing

---

<sup>1</sup>An order is a part of a layout of nodes. It defines relative positions of nodes. For example, a first node is on the left of a second node and a third node is above the second node.

or sometimes called layering-based drawing or Sugiyama-style drawing. Hierarchical drawing puts nodes on vertical resp. horizontal lines called layers depending on their input/output edges, then finds the best permutation of nodes in the layers, which is called order and finally places nodes on coordinates [9]. Other approach are grid-based algorithms, which require one input nodes and one output nodes (nodes that only have multiple outputs or inputs) [12].

According to the experimental study [12] layering-based algorithm, which is used in a Graphviz tool called *dot*, has the best results for all the main criteria. That is total area of the layout, total and max edge length, edge-crossing and total edge bends. This algorithm does not respect screen ratio, but graphs in I3T do not work with screen ratio and its proportions are not always the same, therefore this attribute is not very important for us. Area of the graph for the algorithm is comparable to the area of the visibility algorithm (grid-based algorithm). Even though area of the graph should be minimized, general readability is more important and because of the size of modules in I3T, area of the graph considering nodes as points is not relevant.

For reasons written above the most suitable option for the autonomous layout technique in I3T is the algorithm from *dot*. It is a specific algorithm using the Sugiyama-style drawing [13]. The Sugiyama-style drawing has four general steps 1. In the first step it removes cycles from the graph. Specific algorithms of the drawing do not work properly if not used on an acyclic graph, therefore to prevent mistakes of the algorithm, this has to be the first step. In the second step it creates layers and organizes nodes of the graph to the layers. The basic idea behind layers is that each node in a layer with index  $i$  has its inputs in a layer with index smaller than  $i$  and outputs in a layer with index higher than  $i$ . Rank of a node is the index of a layer, which contains the node. It could not be accomplished if there was a cycle in the graph. Second step also creates dummy nodes and edges, which are added between nodes with long span edges. The purpose of dummy nodes and edges is to make space for a long edge in the graph and because edges do not take any space in the layout technique, nodes have to be made. The third steps finds the best permutation of nodes in layers to minimize the edge crossing. Layers with arranged nodes in them are called order in a graph. Very often algorithms using this drawing iterates over all or most permutations and with decides if the permutation is better than the other ones or not. In the forth step the technique gives coordinates to nodes in a graph according to the order of the graph and edges in the graph. Then the dummy nodes and edges are deleted. This step can be interpreted in many ways and the algorithms usually differ in this step.

---

**Algorithm 1** Steps of Sugiyama-style drawing

---

```

1: procedure ARRANGE()
2:   REMOVE_CYCLES()
3:   ASSIGN_LAYERS()
4:   CREATE_ORDER()
5:   ASSIGN_COORDINATES()

```

---

The first step is the preprocessing of a graph, where we need to make it acyclic. A cycle in the graph makes the rank assignment inconsistent [7]. Optimal rank assignment of a graph is the one that each node in a graph has all inputs in the layer with smaller index and outputs with larger index than the node, but if there is a cycle in the graph this condition cannot be fulfilled. In *dot*, cycle removal uses depth-first search. This algorithm marks nodes in a depth-first search using the orientation of edges and when it reaches a marked node there has to be a cycle in the used walk. When the cycle is found, the last edge the walk is temporarily removed from the graph and added back after the whole process is completed. This process is done until the whole



graph is traversed. Another approach of the depth-search first method is to reverse an edge that goes to the marked node instead of removing it. In some cases reversing the edge could lead to the problems with the final layout. For example, there might be a wrong rank assignment, because the algorithm does not know which node of the cycle should have the smallest rank. If the edge is removed it cannot influence the layout, therefore if the removed edge is important for the drawing of the graph reversing method should be used. *Dot* uses removal of the cycle edges because in the simple structure, that it provides, these edges are not important for the final layout. Another method used for a cycle removal is a simple heuristic algorithm [8,15], which goes through all the nodes of the graph and reverses incoming edges or outgoing edges if there are more of them in the graph. An edge cannot be reversed multiple times therefore all edges that are connected to the processed node are temporarily removed. This algorithm runs fast and is much easier to implement than the algorithm used by dot algorithm but the reversal of that many edges might influence the final layout too much and therefore this cycle removal algorithm is useless for our purpose.

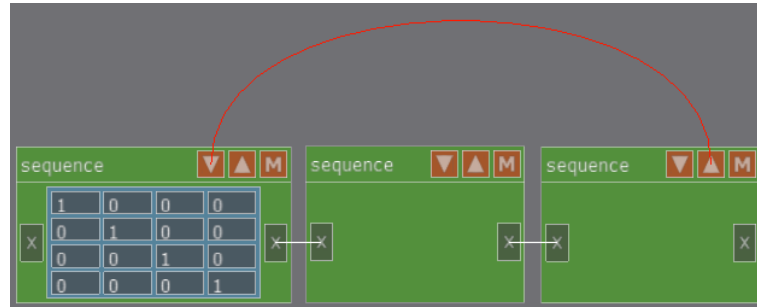


Figure 2.5: Sequence cycle

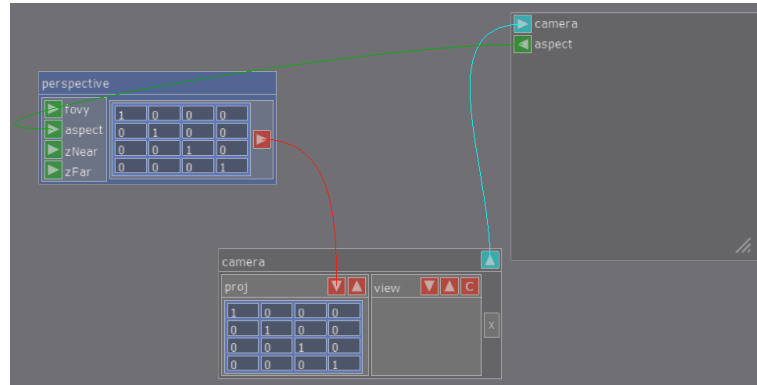


Figure 2.6: Screen cycle

Cycles are very special and unique in I3T, because of the purpose of the nodes. For most modes a cycle would mean that the value in a module would be affected by its value and therefore it would iterate to its limit, but this error is prevented by not allowing user to create this type of a cycle. This is true for the most of the nodes, because most of the nodes represent mathematical operations and their inputs and outputs are operands in this operation. Unfortunately, there are two exceptions to this statement. The first occurrence of a cycle is when there is a walk of sequence modules connected with multiplication inputs and outputs. One module on a walk outputs matrix, which is stored in it, to the sequence that is before it in the walk 2.5. This

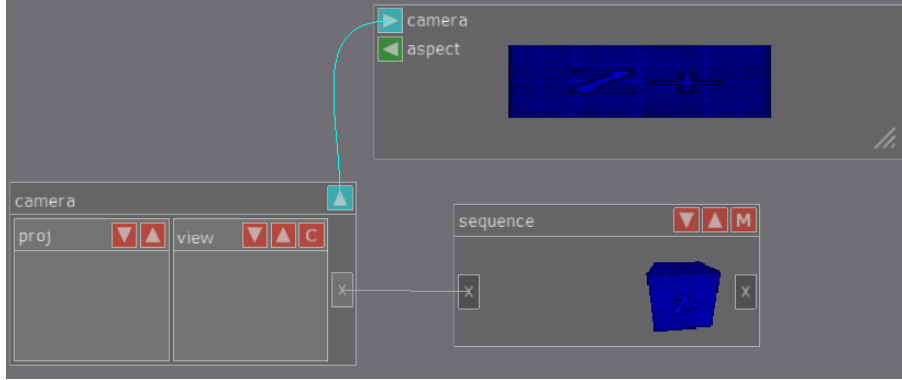


Figure 2.7: Deformed projection in a screen module

output is not using its multiplication and it is only using the transformations that are in the sequence or the matrix input and therefore its value is not affected by modules on the walk. To remove this cycle without destroying the structure, the edge that is the matrix output of the sequence that causes the cycle should be reversed. It is not always trivial to spot the edge that should be reversed but reversing wrong edge could affect the layout. If the right edge is reversed the structure is better than before reversal, because the node on the start of the edge should have higher rank than the node on the end of the edge and also these nodes should be directly connected to ensure an influence of this connection in the final layout. A cycle can also be created while using screen and camera modules 2.6. The input of the screen module is the camera module, the screen module outputs aspect of the screen, which is float and it can be connected to a operator module and the operator module or its directed successor can then be input for the camera module as a matrix input for its view or projection transformation. This cycle should be in every usage of the screen and camera modules, because the projection matrix in the camera module should use aspect of the screen in order not to show the projected object deformed 2.7. This cycle can be removed by reversing the edge from a screen module to the operator module. Screen module is supposed to have higher rank than the operator therefore, reversing helps the structure. The disadvantage is that the input of an operator module will be an edge going from the right part of the layout to the input part of a module that is on the left in the layout, which does not look good. This cannot be solved by a layout technique and it is the problem of the I3T, which is hard to solve and it is not target of this project. Using these two methods we can eliminate all the cycles that are in the I3T, but at least one general cycle removal algorithm should be implemented to ensure that in the future versions of I3T, where might be more modules, this part of the layout algorithm still works. This step is very essential, because next steps would fail or might not stop running if the graph had a cycle.

The second step tries to minimize the sum of the lengths of the edges and to create an optimal division of nodes of a graph to layers. A rule for layers is that each node should have its inputs in previous layers, therefore in the layer with smaller index and outputs in foregoing layers. To ensure the minimization of lengths of edges in a graph, the difference between a rank of a node and a rank of its input should be minimal. In the position assignment step, all nodes in the same layer get similar x coordinate, therefore it is very important to create the best layers in order to achieve a good layout. Following algorithms fulfill these criteria with the best time complexity and optimal result. Unfortunately optimal result might be subjective concept and it is different for various graph types and usages. And the algorithms described in this paragraph are used for directed acyclic graphs with similar structure as I3T graphs. *Dot* uses

the network simplex algorithm. It starts with finding a feasible spanning tree, which contains edges with minimal slack, which is the difference of minimal length and the actual length. Then finds optimal spanning tree, where no cut is negative using edge weights. If the cut is negative, it divides the tree to a tail and head components and finds an edge with a minimal slack in the current state, which connects these components. If there is not a negative cut in the tree, ranks are assigned to the tree and corresponding layers are created [7]. This algorithm is not proven to run in polynomial time but in practice, it is fast. Its disadvantages are its unpredictability and unknown run-time. Another algorithm with similar results is the longest path algorithm [8, 14], which finds all sources, which are nodes with no inputs, and puts them in the first layer. Then all the unassigned nodes that have all their inputs assigned to the already processed layers are put in the next layer. If there is no node that can be put in the current layer, the layer is finished and we start filling the next one. This process goes until all the nodes are assigned. An advantage of this algorithm is its simplicity. Although we might not get the optimal minimization of lengths of edges, it guarantees the minimal difference between ranks of nodes on an edge. The procedure is understandable and applicable by the user, which increases the readability of the graph. Another algorithm is called Coffman-Graham layering algorithm [14]. This algorithm tries to create the optimal hierarchy with the maximal size of layers and is very similar to the previous algorithm. Maximal size of layer is not a very useful property in our case, because choosing the maximal size could influence the drawing without improving any requirement of a proper layout. Therefore, the most suitable algorithm is the longest path algorithm. here is a disadvantage to its approach. All the sources are in the first layer, therefore they all have the same x coordinate in the final position assignment. It might be an advantage for some graph types, but I3T graph sources often are not placed on the first x coordinate in the graph. This can be fixed with a simple addition to the algorithm. After the assignment of layers, all nodes are gone through again, but we start from nodes in penultimate layer and we try to find the maximal rank of the node, which does not break the rule for layers. This part of the algorithm does not change properties of layers and it still fulfills conditions of layers and it shortens unnecessarily long edges. In the foregoing steps dummy nodes, which are representing an edge going through multiple layers, will be needed. They have same properties as normal nodes but they have limitations. A dummy node always has one input and output, which can be the node on the start of the long edge or another dummy node or the node on the end of the long edge. They have fixed size that the edge needs in order not to be covered by some module. Even though the width of a curve representing the edge in the graph drawing is in most cases very thin, usually the size of the dummy node is bigger. This property helps a user to see the curve properly without searching for it between other modules. To find these long edges we need to go through all edges in a graph and find all edges, whose endpoints are not in adjoining layers. If we find the edge, dummy node is created for each layer that is between the start and end nodes of the edge then we set all the connections between the nodes and continue searching other edges.

---

**Algorithm 2** The third step of the layout technique from

---

```

1: procedure ORDER_LAYERS()
2:   best_order = INIT_ORDER()
3:   for iteration = 1  $\rightarrow$  max_iteration do
4:     order = ORDER_BY_BARYCENTER(iteration)
5:     if EDGE_CROSSINGS(order)  $\leq$  EDGE_CROSSINGS(best_order) then
6:       bestOrder = order
7:   order_of_graph = best_order

```

---

The third step orders nodes in layers. The purpose is to minimize edge crossing in the layout. Vertex ordering or Multi-layer crossing minimization are NP-hard problems even for the simplest cases like two layer graphs. Therefore, heuristic algorithms are used to solve these problems [7]. The most used solutions to these problems are the barycenter or the median functions [8]. The pseudo-code of the barycenter method is in Algorithm 2. First we need an initial order of nodes in each layer and this order is stored as the best order. That means layers are now represented by a partially ordered set of nodes, where each node has its index within its layer. Then we iterate an integer from zero to a chosen constant. In even iterations nodes are processed starting from second layer to the last layer and in each layer all nodes are sorted. nodes in a layer are sorted by a median or barycenter of the position of their inputs in the previous layer. In the odd iterations we go from the penultimate layer and end in the first layer and in each layer all nodes are sorted by a median or barycenter of the position of their outputs in the next layer. Due to dummy nodes, inputs of nodes are always in the previous layer and outputs of the are always in the next layer. In each iteration, crossings of the best order and current order are compared and the better one is stored as the best order. In some implementations, the algorithm exchanges the adjacent nodes and if the order with the exchange has less crossings it is preserved otherwise the nodes are changed back. This is before comparing the best order with current order in each iteration. This part of the algorithm needs the algorithm to use more iterations to find the best order, as its changes might not reduce the number of crossings immediately, but after a few iterations it could lead to a better order. Unfortunately, storing and testing all orders with their future state would be expensive, therefore this part is not using its full potential. Despite that, the method statistically helps the layout and in can be very fast. The biggest drain is the edge crossings when the exchange is completed. It has to be well optimized, because it is called for each node in each iteration. In practice techniques with and without the last part are used and it depends on the structure of the graphs whether it is needed. It does not make the layout worse if enough iterations are used. In the best case last iterations should result in the same order with the least amount of edge crossings, because it means that all the nodes are in the are perfectly placed to the order and they do not need to be moved. unfortunately, for the more complex graphs with a lot of branching, this does not happen and it means that there is not the best order and the last one that has the least edge crossings is chosen and worked with in the next step.

The last step places nodes on their final position using the order created in the steps described above. This is the step that differs for different usages of the graph, and also where the target and parts of algorithm differs from the one in *dot*. The purpose of this step is to use the order, which fulfilling layout requirements, to place nodes without breaking any important properties and fulfilling those requirement of a proper graph layout that are specific for the usage of the graph. On the one hand, placing nodes on the plane using order might be a trivial task, on the other hand, there are more complex targets of the placement such as minimizing the size of the graph, making sure that long edges are as straight as possible and also maintaining good properties of the order [8]. The method used in Sugiyama drawing is called Priority method. First it gives nodes basic coordinates, which are computed from the order and sizes of nodes. After this step, we only work with the  $y$  component of node position. Then it iterates in the same way as in the previous step. Each node in a layer gets a priority depending on how many inputs or outputs they have. Inputs and outputs are considered in odd and even iterations, respectively. Then node tries to move to the barycenter or median of their inputs/outputs positions moving only nodes with smaller priority. Important requirement is that node cannot jump over another node, it can only push it [17]. Therefore, the order created in previous step is conserved. This is the

basic idea behind the method, but there can be added more methods to create better layout. In *dot* after each iteration the sum of lengths of the edges in the graphs are compared and the better variant is chosen for the next iteration. This might be very useful if the requirement of minimization of the area of the graph is very important. However, for the purpose of I3T graphs readability is much more important and giving graph more space is not a big drawback and risking that the graph would be cramped is not worth for our application. In *dot* there are also more functions creating better layout for general directed acyclic graph. Most of their methods are unusable in I3T as they do not respect requirements of a proper I3T graph layout.

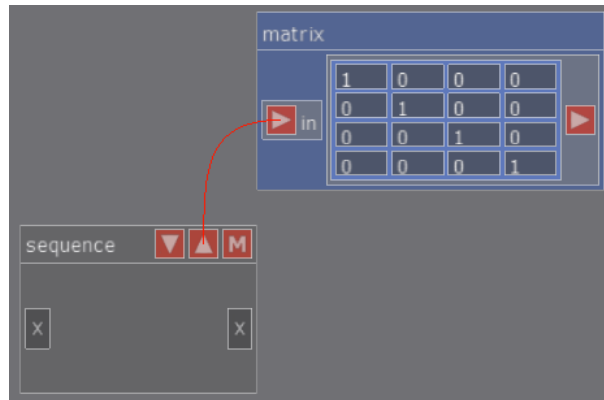


Figure 2.8: An example of two modules with a different start and end edge directions

One of the biggest differences between I3T graphs and the graphs in *dot* is that edges connecting nodes have constant direction, therefore algorithm has to respect this direction and create a layout that uses this direction as the uniform direction for most edges. The basic properties of the edge is that it begins on the right side of a module with a direction to the right and ends on the left side of a module with a direction also to the right. There are special cases, such as, sequence modules that have inputs that are on the top side of the module and their direction is down. Also they have outputs that start on the top side of a module and their direction is up. The Screen module has its output on the left side of its module and its direction is to the left. When creating order in the layers and placing nodes on the plane we have to consider these direction. For example, the node that has its output on the top side of it module and it is connected to the left side of another module has to have lower  $y$  coordinate as well as it should be lower in the layer than without considering these directions2.8. With this property, there are a lot of problems, because Sugiyama drawing does not work with directions and adding more constrictions is not always easy. How this problem is handled is described in detail in the Implementation chapter.



## Chapter 3

# Refactorization of classes representing I3T modules

I3T is a complicated software with a complex class hierarchies. The development of I3T continues and the implementation of main functionalities need to be understandable by a developer. We chose to refactor the implementation of modules in I3T, in order to improve readability and expandability of the current structure. This first half of this chapter is about problems and possible solutions of the class hierarchy. In the second half, the implementation of the chosen solution is described.

### 3.1 Current state of the classes representing modules

There are a few different types of modules in I3T 2.1. Almost all the modules have a common parent called `HintForm`, which represents a rectangular module with a label part. `HintForm` defines the visual structure of the modules and it is very general. Its successors are more specific and they represent different types of modules. There are `OperatorForm`, `MatrixFormBase`, `CameraTransformationForm` and `TransformationForm`. The only module that is not a successor to `HintForm` is `SceneTab`, which represents screen modules.

Even though the class structure of modules defines its objects perfectly, there are a few things that could be done better. First problem is with the number of classes, which represent individual modules. There are 84 classes for operator modules and 13 classes for transformation modules. Only a few of those classes have very different structure from others with the same ancestor. Modules have labels for inputs, outputs and for the whole module. These labels cannot be changed by the code, hence not even in the GUI. Naming of some classes is inconsistent and sometimes the names do not relate to what are they representing.

`OperatorForm` represents all operator modules and unusual modules like the trackball and the float cycle. Classes representing operator modules that directly inherit from `OperatorForm` are usually called `OperatorForm` with a suffix containing the name of the operator, for example: `OperatorFormDeterminant` for the operator module creating a determinant of a matrix or `OperatorFormMatrixSender` for a matrix sender. These operators are almost perfectly defined by `OperatorForm` and thus they only need three methods to distinguish themselves from other operators. These methods are as follows: constructor, `getCopy` and `updateTrasmitterValues`. Constructor needs a lot of information about the module in order to work. This information consists of number of inputs and outputs and their types, a label, a keyword of the module and a tag text. These values are hard-coded to individual constructors. The `getCopy` method's

purpose is to create a new instance of its module and return it. The method needs all the constructor parameters and the type of the operator. The `updateTransmitterValues` method tells module how to change its values when its inputs or input values are changed. This method differs substantially for each module.

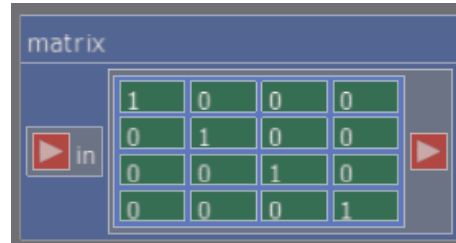


Figure 3.1: An example of a sender operator module

`OperatorFormSender` is a class derived from `OperatorForm` and it represents a modules, whose data can be changed using the user interface and sent along edges to other modules. These modules are called senders. Typical senders are those that send data types of I3T, e.g., a matrix or float 3.1. All of these have a specific class with the same methods as operators, but these methods are very similar and only a few words are changed for each sender. Atypical senders are the float cycle and the trackball represented by classes `OperatorFloatCycle` and `OperatorOrbitRotate`, respectively. Even though these two classes inherit from `OperatorFormSenders`, their structure is complicated and their visual and interactive usage is different from other senders, which can be seen in Figures 2.2c and 2.2d. That is why they have to be treated separately.

`MatrixFormBase` is a class derived directly from `HintForm` and its successors represent all transformations which are currently in I3T. Its abstract sucesor is `MatrixForm`, which represents matrix transformations and its chlidren are individual transformation modules 2.3a. These modules have a different number of methods, different member data and the methods they share have different implementations. Another sucesor to `MatrixFormBase` is the class called `MatrixFormQuat`, which represents a transformation module that performs a rotation using quaternions 2.3b. `MatrixFormQuat`'s structure has the same problems as the other transformations modules and its different visual representation creates even more diversity.

## 3.2 Changes to the classes representing modules

Classes mentioned in the previous section are problematic. They might slow down the future developement of the software, because of their unflexibility and unnecessary complexness.

The main problem that is present in all of the mentioned classes is a duplicate code. The major phenomenon is the duplication of classes with the same methods, which are not even that much different and they often differ only in constants that are hard-coded to individual classes. If the `getCopy` method was to be changed just by one line, the rework would have to be done separately in each class representing an operator module. Adding a new operator module requires a lot of copy-pasting and therefore a developer is more inclined to make mistakes.

For the number of classes that are present in the hierarchy, reading and learning from the documentation is not easy. There are more than ten different descendants of `OperatorForm`, it is easy to miss the important descendant called `OperatorFormSender`, which is very different from other descendants that are almost the same. Even though this problem is mainly in the classes of modules, it spreads more around the project.



Another example of a duplicate code is present in functions creating scenes from files. Parsing files is done separately for each class representing a module and it is often hard-coded very inefficiently. An example is the method reading an operator module from a stream, which contains the two same lines being used for each operator.

Programs that need objects with very similar but not always the same properties have usually three approaches. The first approach is to make one class that has all the similar properties and many descendants that contain the differences. Another one is creating a class that is very general and the differences are passed by a constructor or in class methods. The last solution is to use a template on one general class, which will hold dissimilarities.

The first approach would be good if these objects do not have much in common. There has to be a lot of data and methods, which are specific for individual classes. This is implemented in I3T and it is not efficient for this purpose.

The second one could be good if the methods were not different. Differences would lead to pointers to methods and individual methods implemented and passed by a constructor to the class. All the data that is different can be stored and passed through the constructor or stored as constants and used by the constructor. This approach could lead to the problem of detection of a module that the class represents and in case of refactoring the I3T tool, it would create a lot of changes not just in the hierarchy of classes, but also in the functionality of the program, because in other solutions individual objects can be detected by dynamic cast.

The last option is creating templates, which represent different modules. Working with templates is usually harder than with the code of previous solutions. Debuggers usually do not have understandable errors for templates and development environments usually do not detect problems with templates. Therefore the template has to be implemented properly in order not to slow down the development. Despite that it has many advantages. Only the methods that are different have to be implemented and we do not have to work with function pointers. Data for different modules can be stored separately as an array of constants and passed to the class by an index in the template parameter. Adding more objects is easy, because only the differences have to be implemented. Problem with templates is that the parameter of a template cannot be a variable, therefore, there has to be hard-coded creation of each module, which takes a lot of code. This is not a refactoring problem and it would need reworking a lot of functionalities to be solved.

Before creating templates, a lot of small refactoring had to be done. A lot of the same code was replaced by simple functions. As a result, the function reading a specific operator module from a file is an if statement checking the keyword, a call of the method with a parameter of the type of the operator (the parameter was later replaced by the template) and the general code for all modules. All the `getCopy` methods were reduced to a single macro with a few parameters, which was inserted instead of the function. Even though this macro removed a lot of a duplicate code and overall reduced the size of the whole structure, it did not make the code better. The programmer, who is not familiar with the function of the macro and its usage, might be confused. Unfortunately, this refactoring could not be used by any approach mentioned above. That is why this change had to be reverted.

As there are many ways how to create templates for this problem, the main idea of templates will be now introduced and then there will be more specific information for different types of modules, which are operator, sender and transformation modules, in separate paragraphs. For each module type, an enum was created. Then there is a structure holding all the data of a module. The last part is an array of instances of this structure with specified values. Enums are used as an index to the array and also as the template parameter for a template class. The index

is used for each method to know what data the method should access. Most of the methods are general for all modules of a particular type. The usage of specific data and different methods are made as specification of the method.

Due to the structure of the float cycle and the trackball modules, we decided not to change them and refactor only the typical sender modules. The current hierarchy contains an unchanged class `OperatorFormSender` and a class `SenderNode`, which inherits from the `OperatorFormSender`. `SenderNode` is a template class and its parameter is an enum called `SenderType` that indicates which sender is the current instance, e.g. `SenderType::FloatSender` is the enum for a float sender module.

The new `Sender` structure was created and it contains: a keyword of the sender, its default label, data type of the sender and its input tab name, which is defaultly set to "in". `SenderType` also provides indices for the static vector of instances of the structure `Sender` called `senders`. There is no specialization in the template and also neither switch nor if statements were necessary to distinguish the individual senders. All data (float for float sender, matrix for matrix sender) is stored as a 4x4 matrix and each sender knows which part of the matrix it is using, e.g. a float sender module uses the first element of the first row, a vector of 3 floats sender module uses the first three floats of the first row of the matrix. This approach is not the most efficient, but there is no other way to store a general value in the project. This inefficiency means that for each sender we might maximally use 15 more floats, which is not much compared to the size of the whole structure. Each sender has its default name, which is displayed on the top of the module. This name can be changed from outside of the class or set in the constructor. There are major differences between senders and the other operators and thus they cannot be merged without using too much specialization.

Even though all senders could be represented by a nontemplate class, we decided to create the template to make it easier to add a new sender module and also to respect the structure of the program, which now contains templates.

Templates for operators were done very similarly. A new template class `OperatorNode` was created. The enum is called `OperationType`, a new structure is called `Operation` and a vector of `Operation` instances is called `operations`. `Operation` contains a keyword, a default label, a number and types of inputs and outputs, a default tag and vectors of input and output names. The constructor and the `getCopy` methods are general for all operators. Only the `updateTransmitterValues` method has to be specialised for each operator. Variable names of input and output tabs were added to the logic of operators. Before this change the labels were hard-coded. Now, there are default type names (for a float or a matrix), default labels of input and output tabs for each operator. Names can be also set with a parameter in the constructor or changed during the module's existence. The tag is now default for each operator module and also can be changed using the constructor or accessed later.

The problem with transformation modules is much more complicated. Two templates classes had to be created: `AddTransformationForm` and `TransformationNode`. `AddTransformationForm` is the class for the dialog creating and editing transformations. They both use the same enum as a template parameter. The name of the enum is `TransformationType`, for `TransformationNode` the structure is called `TransformationData` and the vector of the data is called `transformationData`. Another structure and a vector had to be created for `AddTransformationNode` and they are called `AddTransformationData` and `addTransformationData`, respectively. `TransformationData` contains a title of the transformation, a vector of indices of elements that are enabled in the matrix, a matrix type and five booleans, which serve for locking and synergies. Explaining them is not important for this project. `AddTransformationData` contains a name of the dialog,

names of the fields in the dialog box and dimensions of the dialog. AddTransformation has only few specializations, which are as follows: a constructor for a lookAt transformation (all other transformation dialogs have rows with two columns containing a name of the field and an input field, but lookAt has rows with four columns, where first is a name, then there are input fields for individual floats of a vector). SetToDegrees and setToRadian methods are different for transformations that uses angles. In the general constructor there are if statements for each type of transformation, because the data of the dialog has to be stored in the same space. This space is a matrix and in a constructor each transformation needs to know how to store this data. In TransformationNode there is a lot of specialization. Each transformation module needs getMatrixFromValues and getTitleString methods, because they cannot be generalised. This is not a problem, because they are very short. For transformations using angles, there are setToRadians and setToDegrees methods specialisations. There are few transformations with unique specialization, such as modules representing rotations having methods setLimits or calculateSynergyValues specialised for their unique functions for rotations. During the refactoring of transformations one functionality was changed in the AddTransformationNode. That is when in the settings the used angles are changed to degrees or radians, the numbers in the dialog are changed too to fit the same transformation before the change in settings. Before the change of the functionality the value would stay the same, which represents different angle. In addition, when the user tries to edit a transformation module, the AddTransformationNode dialog shows the numbers used in the transformation instead of the numbers that were stored during the last usage of the dialog. These changes are very minor and they were agreed to be improvements to the program.



## Chapter 4

# Implementation of the layout techniques

This chapter is about implementation details of layout techniques described in Section 2.4. Even though the techniques were already explained, there are parts of algorithms, which might not be evident from the description or which are special for this implementation, that should be clarified.

In all techniques need a class representing all the modules. The Tab class represents every rectangular area with a border frame in I3T. It contains all the necessary data, such as position and size of the area. If we need more specific information about a module, we use if statements with dynamic casts. This allows us to access data specific for a certain module. We have to use Tab pointers to work with general modules, because of the SceneTab, which does not inherit from HintForm.

An implementation of each technique has 3 steps. The first step are popups for layouts that are implemented in the SpaceTabPopup class. These popups make sure that a layout technique can be selected from the right click on the workspace and then when the technique is chosen the second step is called.

The second step is a method of TransformationSpaceScrollTab. This method uses selected tabs and calls the third step or if the technique is trivial, it lays out all the selected tabs and skips the third step. After the tabs are layed out a static method named TabSpace::changeMessage has to be called in order to save a current state of the scene to the undo/redo system.

The third step is the function or a set of functions that lay out tabs, which they get from their parameters. Positions of modules and also curves between the modules are updated in this step. Functions of this step are the implementation of a layout technique. These functions have their own header and source files if needed or they are in layout.h and layout.cpp files. There are functions already implemented for updating curves of all the modules. Using dynamic cast on a tab the curve can get updated after a change of any module.

### 4.1 Interactive techniques

As mentioned above, interactive techniques are trivial and their implementation is often not very interesting. Despite that, the algorithms will be briefly described. All the interactive techniques have two usages: horizontal and vertical. The horizontal usage of techniques will be described in this section. The only difference in the usages is the coordinate, which is used. When the vertical technique changes the x coordinate, the horizontal changes the y coordinate.

---

**Algorithm 3** The last part of the horizontal distribution algorithm

---

```
1: previous = center
2: for  $i = ((\text{number\_of\_tabs} - 1)/2) - 1$  to 0 do
3:    $\text{sorted\_tabs}[i] \rightarrow x = \text{previous} \rightarrow x - \text{sorted\_tabs}[i] \rightarrow \text{size}.x - \text{gap\_size}$ 
4:   UPDATE_CURVES( $\text{sorted\_tabs}[i]$ )
5:   previous =  $\text{sorted\_tabs}[i]$ 
6: for  $i = ((\text{number\_of\_tabs} - 1)/2) + 1$  to number_of_tabs do
7:    $\text{sorted\_tabs}[i] \rightarrow x = \text{previous} \rightarrow x + \text{previous} \rightarrow \text{size}.x + \text{gap\_size}$ 
8:   UPDATE_CURVES( $\text{sorted\_tabs}[i]$ )
9:   previous =  $\text{sorted\_tabs}[i]$ 
```

---

The align technique only sets the same y position for all the selected modules. The technique is used to align the objects on one line. Therefore, the position that is chosen is not important. Modules are already selected and moving all the selected objects is in I3T. To prevent unnecessary movement, the first selected module's position is chosen. These techniques have their second and third step in one function in the TransformationSpaceScrollTab. Both functions from the first and the second and third steps are called verticalAlign or horizontalAlign in the files mentioned above. The next implemented technique is a basic distribution. They start with finding the center, which is the module, whose x coordinate is the median of x positions of all the selected modules. The center is found by sorting the vector of selected tabs and using the one with an index equal to a number of selected tabs minus one, all divided by two and this value is converted to an integer and used as the index of the center. Then we move tabs by method from the Algorithm 3. Functions implementing this techniques are called horizontalDistribution and verticalDistribution. The last technique is the even distribution. First the vector of selected tabs are sorted by x coordinate. Then the space between the first and last modules in the sorted vector is computed and subtracted by the sum of the sizes of the selected modules and divided by the number of modules minus one. This value is used as the gap between every two modules adjoining in the sorted vector. Therefore the first and the last modules stay in place and adjoining modules have the same gap between them. This technique is implemented in functions called evenHorizontalDistribution and evenVerticalDistribution.

## 4.2 Semi-autonomous technique

The only semi-autonomous technique is the distribute technique. It starts with finding a module that is closest to the barycenter of the positions of all the selected modules. This module is called center. All the modules are sorted by the distance from the center module. Then all the modules are processed in sorted vector from the first to the last. They move away from the centre until they do not collide with any already processed module. The difficult part in the implementation is the moving away during the collision. In this part the moved module is supposed to be moved minimally out of the collided module plus a little more to create a gap between them. Hence, the exact position has to be computed. This is done by finding the size of a rectangle, which is the intersection of the modules. Then we compute how many steps of unit vector going from the center to the moved module we need. Then the module is moved by the number of steps and also moved by a constant to create a gap. This procedure is done until the module is no longer in collision with any other already processed module. All the major functions

are called distribute and in the layout.h and layout.cpp is the function called removeCollision, which implements the difficult part described above.

## 4.3 Autonomous technique

The autonomous technique, which has been described in Section 2.4.3, is much more complicated than the other techniques. Even though the steps of the techniques were already explained, the implementation is sometimes different.

### 4.3.1 graph structure used in the technique

Before the technique can be implemented, we have to create a graph structure. A graph structure of modules is already in I3T, but creating a graph with nodes that are special for this purpose is much more practical, because much data has to be stored in the nodes. Adding this data to module classes wastes resources during normal run-time. There is also problem with edge usage. Different modules use different objects to represent input and output tabs. Furthermore the connection between output and input tab is not universal for all module classes. This makes the usage of a graph unnecessarily difficult. An example of this problem is an edge between two operator modules. In this example we have a pointer to the first module and we want to get the other module pointer. For this example there are only two modules in the scene and only one edge.

First we have to go through all the OperatorCurveTab pointers in the operatorOutputs of the first module. These pointers are tabs, which represent the output tab on the first module. Then we call the getOutComponent method on all the OperatorCurveTab pointers. Then we search for the vector of CurveTab pointers, which we get from this function, that is not empty. The size of the vector is one, because there is only one edge going from the module, and the CurveTab, which it contains, is the input tab on the second module. Then we have to cast the CurveTab to the OperatorCurveTab and call its getOperator method. This method returns an Operator pointer, which has to be casted to the OperatorForm. The pointer that we get from the cast is the second module. Even though the procedure is similar for all edge, it is not the same and the usage this procedure for each usage of an edge is very inefficient.

To avoid this inefficiency, we created a new graph structure, which is used only for this layout technique. Nodes of a graph are instances of the structure called Node, which contains a Tab pointer, data necessary for the subsequent steps and two vectors of objects representing input and output edges. The Tab pointer has the address of the module, which is represented by the Node. The node data needed in the layout algorithm is: the position and the size of the module, an input and output priority, an index of the current layer and the boolean indicating if this node is a sequence module without the multiplication input, which is called possibleDFSStart. The structure representing an edge is called Connection. Each module on the edge has its own connection therefore, there are two connections for each edge. The Connection class contains a Node pointer, which points to the Node on the other side of the edge, an outgoing and incoming direction of the edge, y difference and connection index. The incoming and outgoing direction are the directions of the the input or output tab in the connection for the node using the connection and for the node, which is on the other side of the edge. the y difference variable is later used to tell if the module using this connection wants to be above or below the Node on the other side of this connection. The connectionIndex is the same for two connections of one

node only if the connection comes out of the same output tab. This information is used to count edge crossings in the later part of the algorithm.

All the parts of this implementation are in files `autonomouLayout.h` and `autonomousLayout.cpp`. All the functions and structures are implemented inside a class called `AutonomousLayout`. It contains previously mentioned structures, vector of Tab pointers, two vectors of Node pointers, and a vector of vectors of Node pointers. The Tab pointers are addresses of the modules, which should be laid out. The first vector of Node pointers are the nodes representing the modules. The second vector of Node pointers are the dummy nodes, which are created after the second step of the layout technique. The vector of vectors of Node pointers is the matrix of nodes, which all the nodes will be laid into. It helps all the modules to be assigned in their positions. In other words, the matrix serve as layers in the technique. It is used in the second, the third and the fourth steps. `AutonomousLayout` contains constants, which define the number of iterations in the third and the fourth steps, and functions implementing the technique.

The problem of creating connections between all the nodes in the graph can be separated to three parts. The first part is how to get from the Tab pointer of one module to all the CurveTab pointers, which represent input tabs of modules that are on the other side of an edge, which starts in the first module. The second part is almost the same, but we try to find all the CurveTab pointers, which represent output tabs of modules, which represent the start of an edge going to the first module. The third part is how to get a Tab pointer representing module from a CurveTab Pointer, which represents the output or input on the module.

First two parts are very similar. They are implemented in a `setOutputs`, `setInputs`, `setTransformationInputs` and `setTransformationOutputs` methods, which are member methods of `Node`. They involve running through vectors of CurveTab pointers and calling `getOutComponent` or `getInComponent`. One output tab can have multiple edges and that is why `getOutComponent` returns a vector of CurveTab pointers and `getInComponent` returns CurveTab pointer. The procedure is a little different for four types of modules. The types are operator, sequence, camera and scene modules. In These steps we declare `connectionIndex` to each connection. These steps always calls a `getConnection` method, which returns a connection, which represents the edge. This connection is stored to the input or output vector inside the Node object. The order in which are different CurveTab pointers processed is very important, because the order of connections in input and output vectors is important for the edge crossing counting in the third step. The first connection in the output vector means that the module on the other side of this edge is supposed to be in the highest position out of all outputs to avoid edge crossing. The second connected module should be below the first one and so on. It works the same way for the inputs vector. The possibleDFSStart variable is assigned in the `setOutputs` method, when creating connections.

The third part is implemented in the `getConnection` method. This method gets a CurveTab pointer of an input or output tab of a module, which is supposed to be stored in the connection, and returns the Connection pointer representing an edge. First the CurveTab pointer is casted to an `OperatorCurveTab` pointer. Then we call a `getOperator` method on the `OperatorCurveTab` pointer. It returns an `Operator` pointer, Which can be a Tab pointer of the module or a pointer to a part of the module. That is why we have to work with the pointer differently if it is a Tab, an `OperatorMultiOut` or an `OperatorStorage` pointer. From these pointers, getting a pointer to the tab representing the module is a call of few methods. This method also assigns directions of the start and the end of the connection. This part also assigns the y difference variable, which is computed differently if the module is on the end of the directed edge or on the beginning of the directed edge.



When we are processing the output edge of a module and the edge is going upwards from the module and it goes to the right, when it enters the other module as represented in Figure 2.8. When we are processing the edge from this perspective, the first module should be below the other one, therefore the y difference should be negative. When we are processing the edge from the perspective of the other module. The y difference of this connection should be positive. Even though the directions are same in both ways, they are just exchanged and that creates the difference between an input and output connection. The y difference is a sum of the two directions if the module storing this connection is the one on the end of the edge. It should be the opposite value otherwise.

This procedure has two big exceptions. The first one is that in every sequence module, there is an invisible edge between its matrix storage output and an invisible multiplication input. It is used for the functionality of the module and it is not used in GUI. We do not want to store this connection, therefore we have to check if an output of a sequence module is not the same module. If it is, we discard this connection. The second exception is with a multiplication output of a camera module, because this edge cannot be detected from the module on the other side of the edge. Therefore we add the connection representing this edge to the other module when processing the camera module.

### 4.3.2 Cycle removal

The first actual step in the technique is a cycle removal. This step has three parts. In each part a different type of a cycle is solved. The first one is a cycle with the a scene module. This is solved by finding all nodes representing a scene tab and running a depth first search. If it ends with finding the same node again, we reverse the edge going from the scene module in the graph. This step is implemented in the `findAndReverseSceneCycle` and the `findSceneCycle` methods. The second step handles cycles created with sequence modules. We start a depth first search in all nodes, which we have the variable `possibleDFSstart` set to true. If the search finds the node, which has been already processed, it reverses the last edge it used to get to this node. This step is implemented in the `findAndReverseSequenceCycle` method. The third step is the general cycle removal. In this step a depth first search is used on all the nodes. If any node is found twice during the depth first search run, the last used edge is removed and the search continues. This step is implemented in the `findAndRemoveCycle` method. The methods removing specific cycles are called from the `removeCycles` method.

### 4.3.3 Layer assignment

The second step is a layer assignment. It is implemented in a function called `assignLayers`. We want to put all the nodes in the vector of vectors of Node pointers using an algorithm. The implementation follows the Longest path algorithm and there are no special or interesting implementation details. After this step we have to remove unnecessary long edges and create dummy nodes to represent long edges in layers. The implementation of the removal of long edges goes through nodes in the layers. It is important that we start from the last layer and continue with the nodes in the previous layers, because this assures the determinism of this algorithm and also it can reduce the edge length in one run. We run through all the outputs of the processed node and we keep track of the minimal layer index  $I$  of output nodes. After we processed all the outputs, we move the node to the layer with the index  $I$  minus one. This is implemented in the function called `removeLongEdges`.

Adding dummy nodes is done by going through all long edges. That means edges, which are not between nodes in adjacent layers. We have to go through all outputs of all nodes to find these edges. If we find this edge, we create enough dummy nodes for the edge to have its representation in form of dummy node in each layer between nodes connected by the long edge. When the dummy node is added its input and output priorities are set to max. This is used in the fourth step to keep the long edges straight. The new dummy node always have their input and output direction going to the right. Therefore, directions and yDifference variables have to be recalculated for nodes connected by this long edge. The size of any dummy node is zero for  $x$  coordinate and a constant for the  $y$  coordinate. The  $x$  coordinate is zero, because we do not want to change the size of the graph in the  $x$  coordination with the nodes that does not need any space. The  $y$  coordinate of the size creates a little vertical space for the edge. Creation of dummy nodes is implemented in the addDummyNodes and createDummyNode methods.

#### 4.3.4 Order creation

The third step is creating an order in layers. It is done by ordering nodes in the layers using a barycenter method. We follow a technique described in Section 2.4.3, but there are few changes to it. The first change is when computing a barycenter. To improve the order we need to take a few properties into consideration. An order of edges going from and to a module and a direction of an edge. We can improve the layout if add a number to the barycenter variable if the edge is not the first one. This is done by adding the subtraction of the connection index to the number of connections. In the same way we solve the direction of edges. If the connection has a positive  $y$  difference, then the we add to the barycenter 0.1. The numbers that are used could be altered. These values created the best layout on a tested scenes. If these additions create an edge crossing it would not be the best order and the change might be reverted in the next iteration.

After we have a vector of barycenter variables for a whole layer. We sort the layer in a way that the lowest barycenter is on the first index. The problem is when we have a module without an input or output. The barycenter of the inputs of the module without inputs is zero by this implementation. To prevent having these nodes on the first index, we mark them and then preserve their index in the layer. This could create another problem, which are the nodes that do not have any inputs or outputs, which should have the lowest index. This is done by checking this property and then putting them on the lowest indices of the layer.

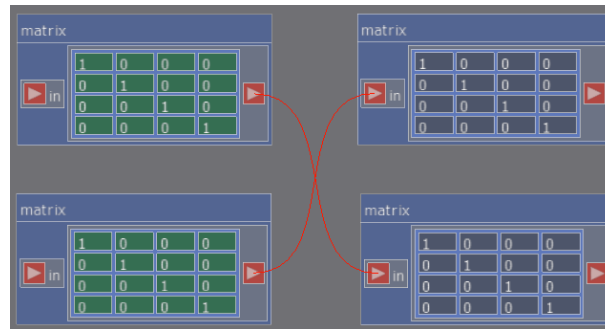


Figure 4.1: An edge crossing with four modules

The last interesting function in this step is a function that counts edge crossings. There are three types of crossings in I3T. The first crossing is between four modules is seen in Figure 4.1. The second one is when two outputs of a module are in the opposite order than the order of

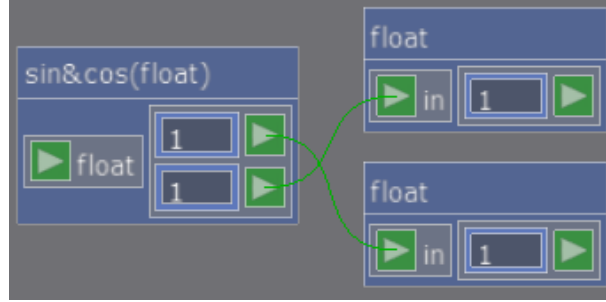


Figure 4.2: An edge crossing with three modules

output tabs on the module as seen in the Figure 4.2. The last one is the same as the second one, but for inputs. These three types have to be solved individually. In all three types we run through all the modules and check the property of their inputs or outputs. When the crossings is compared between the current one and the best one, we assign the current one to the best one even if the number of crossings is same. This is not typical for this drawing technique, but even if the crossing is the same, the structure of the order should be better. Both options with and without the equality option were tested. And the one we chose to implement found the better layout.

Functions used in this step are called `orderLayers`, `barycenter`, `orderLayersByInputs`, `orderLayersByOutputs`, `countEdgeCrossings` and `isInOrder`.

#### 4.3.5 Coordinate assignment

The last step in the technique is a coordinate assignment. We use the priority method. First we place all nodes on a grid, which fulfills a order created in the previous steps. The layer starts at the  $x$  coordinate defined by a size of nodes in previous layers. The nodes have their  $x$  coordinate centered between their layer and the next layer. Then we assign the input and output priorities to all the nodes. All the dummy nodes have max input and output priority and all the other nodes have their priority equal to the number of inputs or outputs for the input priority or output priority, respectively. We add one to the input and output priority to the nodes, which represent sequence modules. This means that the sequences have a priority when being moved, but they cannot alter the order. Modules, which have much more inputs or outputs will still have the priority to be moved more than the sequence with one input or output. This achieves better placement for the subgraph created by sequences connected by multiplication inputs and outputs, but it will not move the more complex and important parts of the graph.

Then we iterate over all layers forth and back. In the forth iterations we use input priorities and barycenters of  $y$  coordinates of inputs of modules as their target position and in the back iterations we use the output ones. In each iteration we sort all layers by their priority. When there are two nodes with the same priority, we can compute their target position to sort them better. If these target positions are above their current positions, we can put the one with a lower index in their layer higher to the sorted layer. This makes sure that the node, which moves first moves away from the other one and creates some space to the other node. This method is also used for other options. This will reduce the amount of iterations needed for a layout.

When we have a sorted layer, we run through all the nodes in the sorted layer. We compute the target position of a node and then decide if the node should move up or down. There is a function for each direction that will try to move the node, but if there is a node in the way

of the move, it will call this function on the node in the way and compute its target position. The target position is the target position of the first node plus the size of the node (the first one when moving down and the second one when moving up) plus a constant, which creates a space between nodes. This goes on until some node can be placed on their target position. In this case all the nodes try to move as much as they can with the space created by the move of all the nodes. The requirement is that only the node with the lower priority than the first node can be moved.

This step is implemented in functions called `assignCoordinates`, `setXCoordinates`, `setBasicYCoordinates`, `setNodePriorities`, `compareByInputPriority`, `compareByOutputPriority`, `compareByPriority`, `barycenterOfInputs`, `barycenterOfOutputs`, `moveNodeUpByInputs`, `moveNodeDownByInputs`, `moveNodeUpByOutputs` and `moveNodeDownByOutputs`.

After this step all positions of modules are updated and also the edges between the modules are updated. Then the graph created for this technique is deleted from memory.

## 4.4 Testing of layout techniques

All the layout techniques in I3T were tested. Three users of I3T were testing them. Interactive techniques have trivial implementation. The testing involved choosing techniques, which would be best addition to the program. The semi-autonomous technique was chosen for its general usability. The first implementation was rejected and the part of the technique, which moves the collided module had to be reworked. The previous implementation does not have good results for larger graphs and the mental map was often destroyed.

The autonomous layout technique was tested during the development. The first and second steps were not changed. These steps do not have any subjective properties and they are working properly. The third and forth steps were tested together. Users judged layouts of graphs, which are used as results in the next section. The main improvement is the behavior of dummy nodes, usage of edges with strange starting and ending directions. The different number of iterations in the last two steps were tried. The best result is when there are 24 iterations in the third step and 5 iterations in forth step. 24 iterations are enough to create a good order and it is fast enough. If more than 5 iterations were used in the last step, the chance that the technique would find a consistent placement is low. Graphs are either usually placed in less than 6 iterations or they have a structure, which cannot be placed perfectly. The difference between the final implementation and the implementation before testing can be seen in Figures A.8 to A.11, A.17 and A.18. Even though the algorithm works on most tried scenes, there are always special scene, which are hard to lay out. In Figure A.15 is a graph laid out by a user. In Figure A.16 is the same scene laid out by the technique. This scene have special requirements for a proper graph layout. And it is hard to detect these type of graphs.

## 4.5 Results

Results of an interactive technique are not subjective and they either work properly or they do not. From Figures A.1 to A.3 we can see that the implementation is the expected result of the techniques studied in research. Results of an semi-autonomous technique are subjective. The semi-autonomous technique implemented to the I3T definitely resolves collisions between modules and it moves modules in an intuitive way. This technique struggles on graphs with more modules. The usage of this technique can be seen in Figure A.7.

The target of the autonomous technique is to create a layout, which is usable and readable by a user. These criteria are subjective. The technique fulfills or at least tries to fulfill all the important requirements of a proper graph layout. In addition to that, all the unexpected behaviors were tracked and solved. After the completion, scenes, which are used for the demonstration of the usage of the I3T tool, were laid out by the technique and then tested by users. This testing helped finalizing constants in the third and the forth steps. These constants change the priority of a sequence over all other modules, the influence of a direction of an edge on the functions sorting nodes in the layer and also on the function computing the position of a module using the barycenter method. This influence had to be compared to the influence of the order of inputs and outputs on the module. In Figures, the comparison of the layout before and after testing can be seen. Some scenes, which were used for testing, laid out by the algorithm can be seen in the Figures A.9, A.11 to A.14 and A.17.



# Summary

The main purpose of this project was to research layout techniques that can be implemented in I3T. It includes find requirements of a proper graph layout for graphs that are used in I3T. The the technques, which are used on graphs similar to the graphs in I3T were researched. The result is a set of layout techniques implemented in I3T.

Interactive and semi-autonomous layouts were found usefull by users. Their implementations are fast and results are optimal. The main technique implemented in I3T is the autonomous layout technique. The technique is called Sugiyama drawing. This technique had to be changed in order to create a proper layout for I3T graphs. The importance of a sequence module had to be implemented. The technique had to be able to work with the starting and ending direction of edges, the fixed starting and ending positions of an edge on a module had used. And other smaller properties of the I3T graphs had to be considered in the implementaion.

The small part of this project was refactoring of classes representing modules. It reduced the amout of code and also the number of classes was decreased. The readability and extendability of I3T code was increased.





# Appendix A

## Results of layout techniques

### A.1 Interactive layout techniques



Figure A.1: Example of a usage of horizontal alignment



Figure A.2: Example of a usage of horizontal distribute

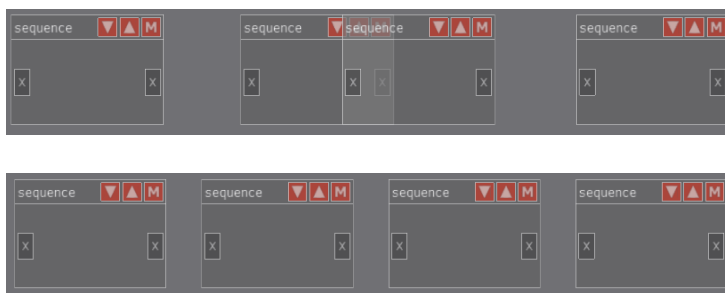


Figure A.3: Example of a usage of even horizontal distribute

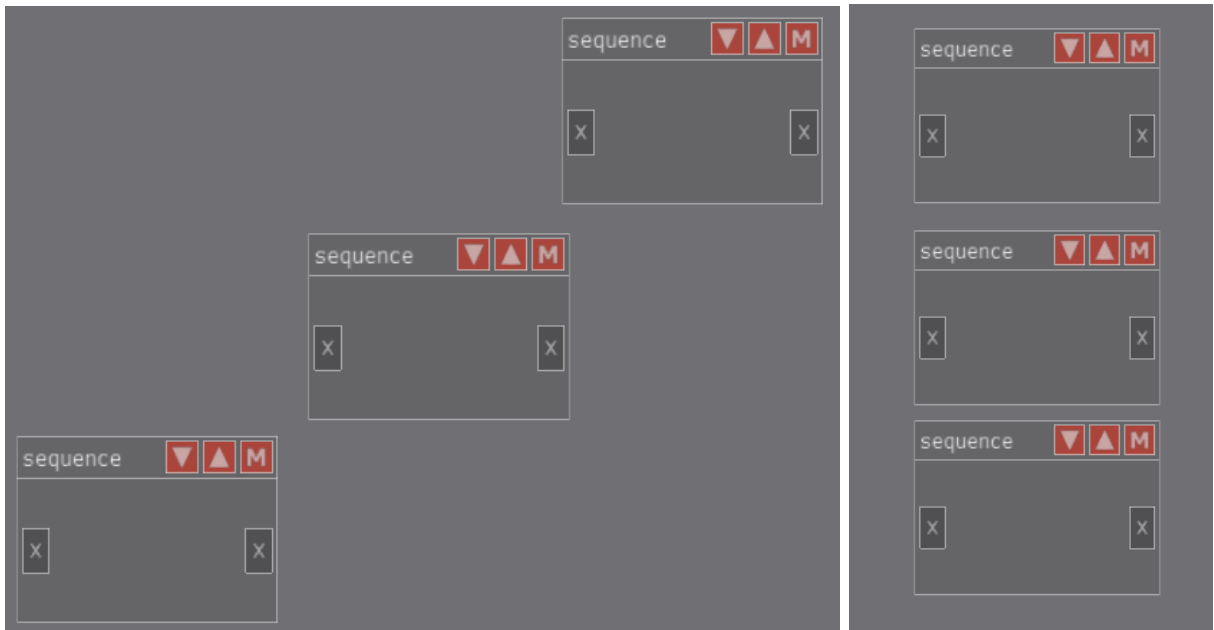


Figure A.4: Example of a usage of vertical alignment



Figure A.5: Example of a usage of vertical distribute

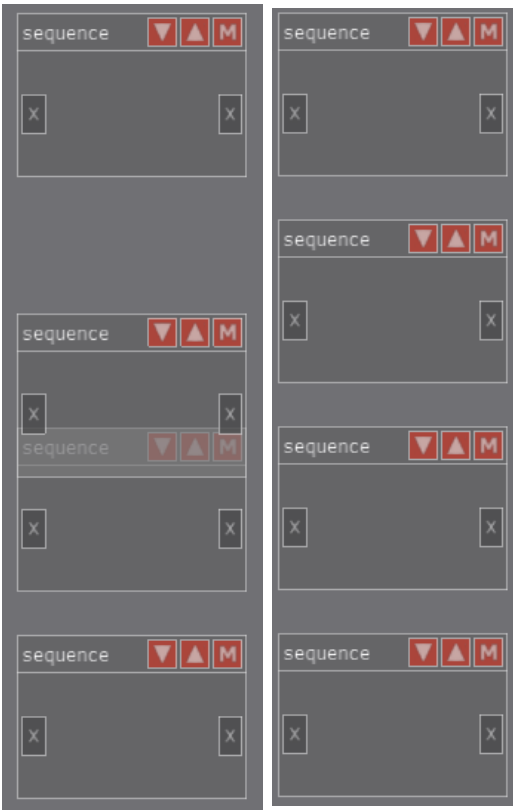


Figure A.6: Example of a usage of even vertical distribute

## A.2 Semi-autonomous layout technique

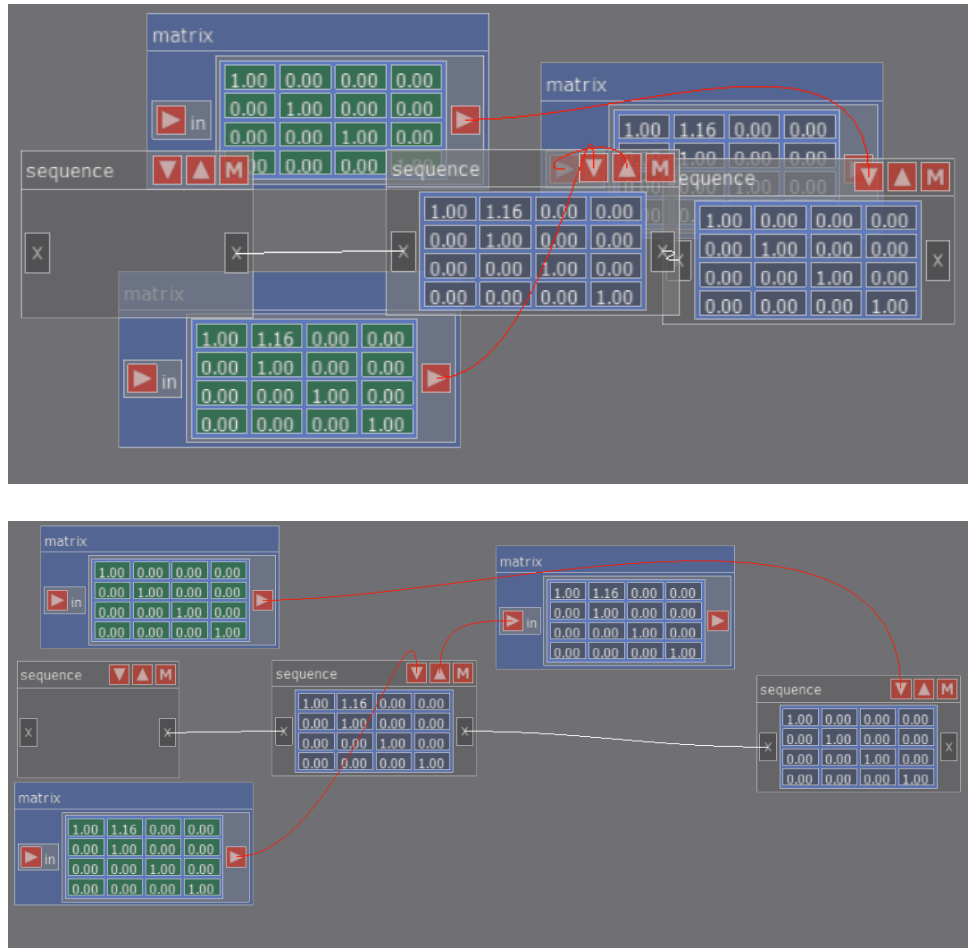


Figure A.7: Example of a usage of distribute

### A.3 Autonomous layout technique

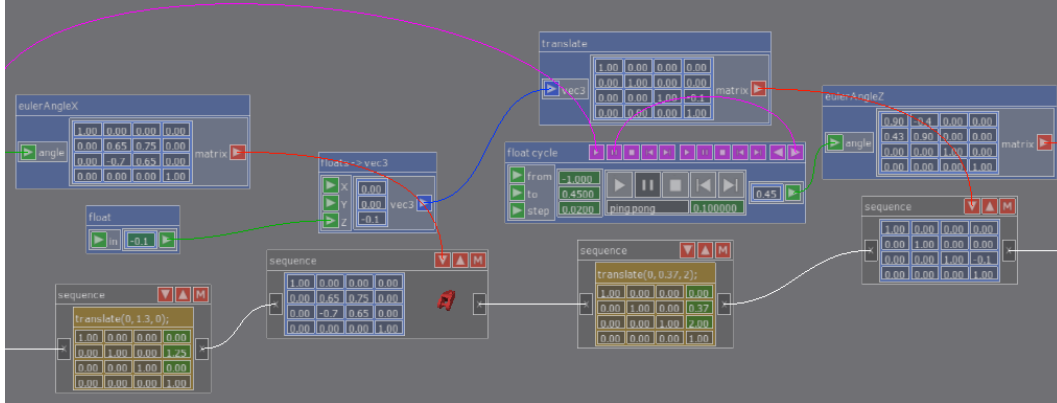


Figure A.8: First version of the layout technique on a part of a scene called armAnimated

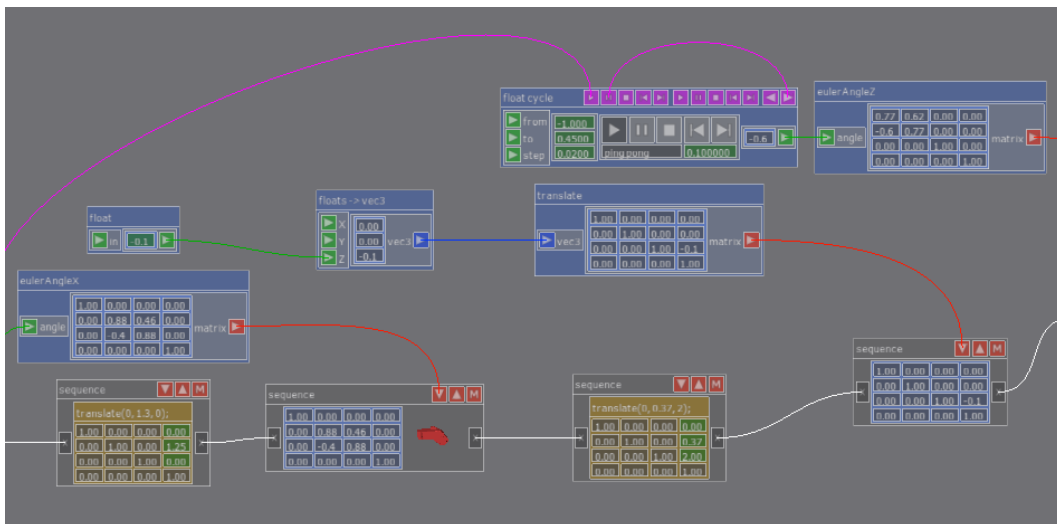


Figure A.9: Final version of the layout technique on a part of a scene called armAnimated

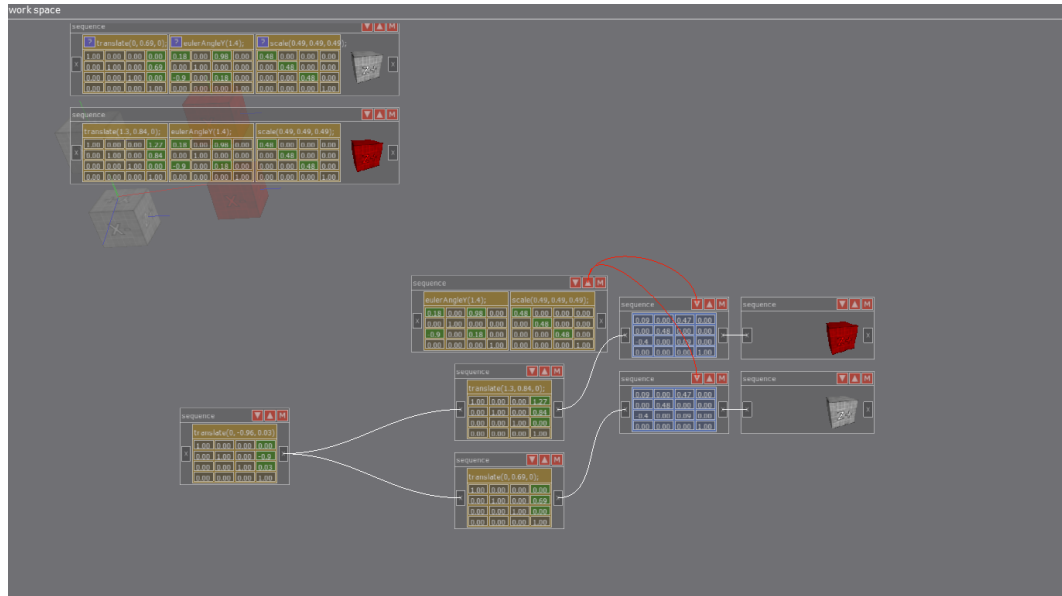


Figure A.10: First version of the layout technique on a scene called 01\_modelTransformation-Graph

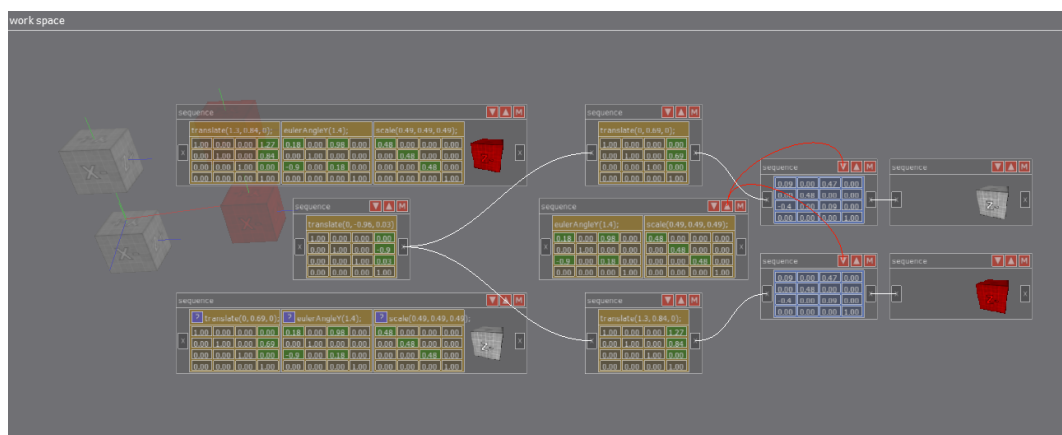


Figure A.11: Final version of the layout technique on a scene called 01\_modelTransformation-Graph

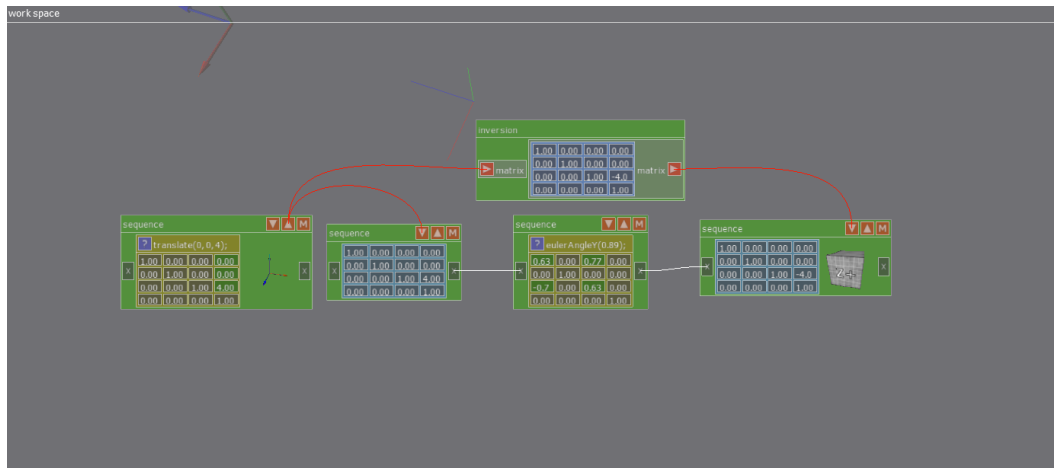


Figure A.12: Laid out scene called 03\_rotateAroundPoint

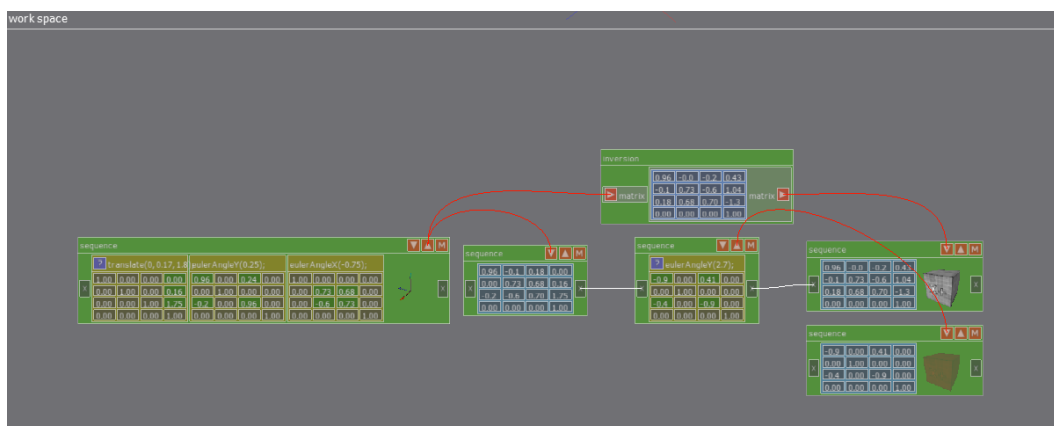


Figure A.13: Laid out scene called 03\_rotateAroundPoint-1

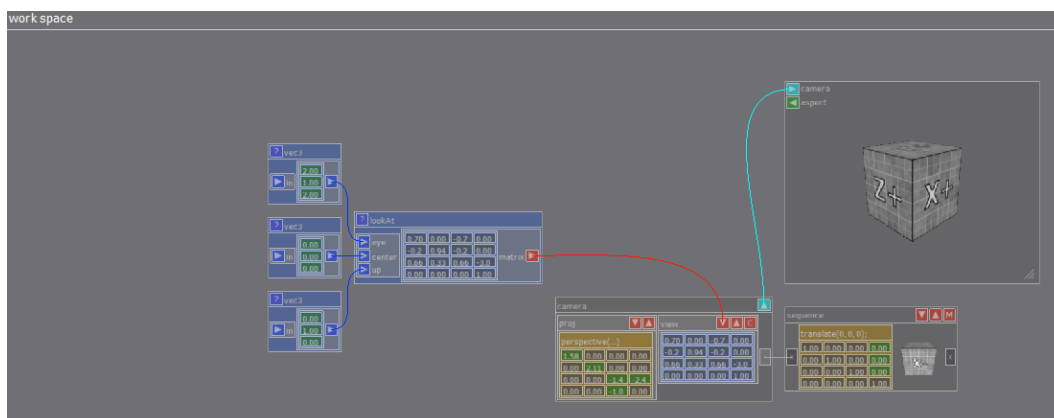


Figure A.14: Laid out scene called 05\_lookAt



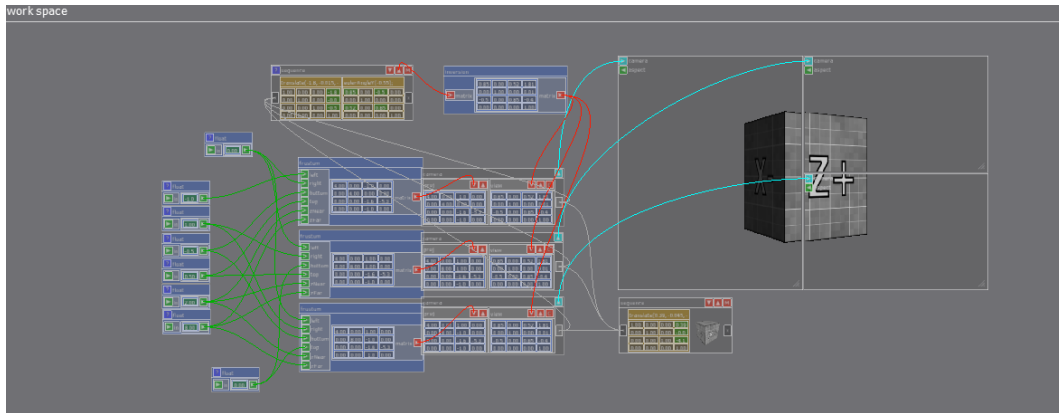


Figure A.15: Scene 09\_frustumMultiMonitor laid out by a user

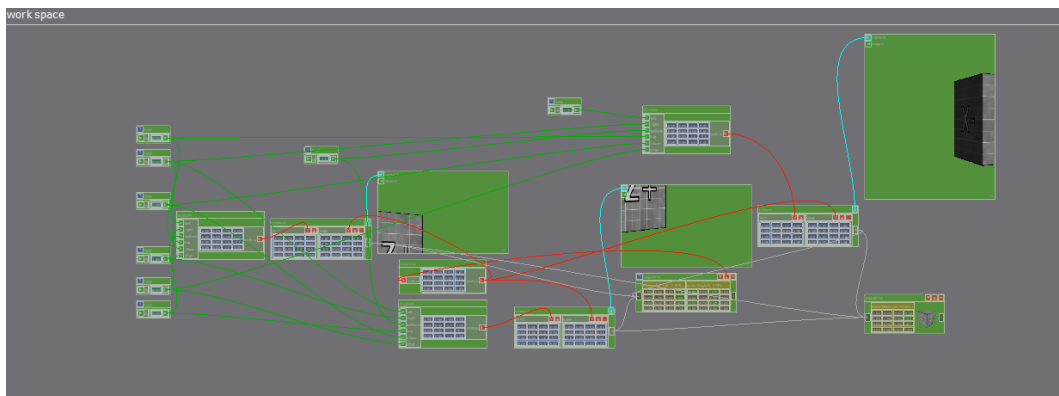


Figure A.16: Scene 09\_frustumMultiMonitor laid out by the layout technique

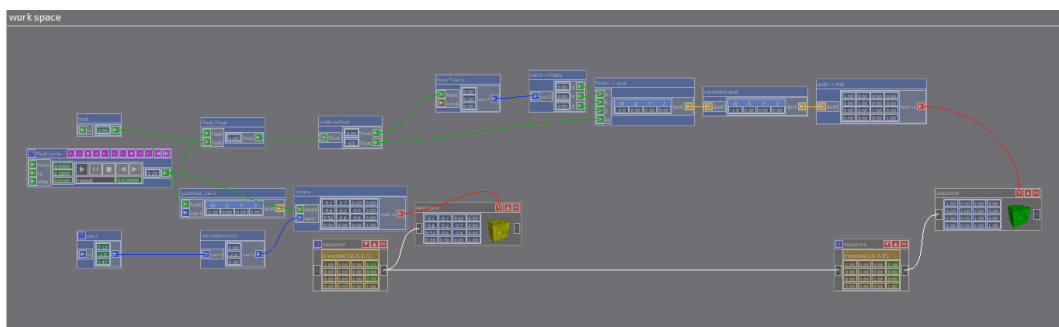


Figure A.17: Laid out scene called 10\_quaternionMatrixComparison

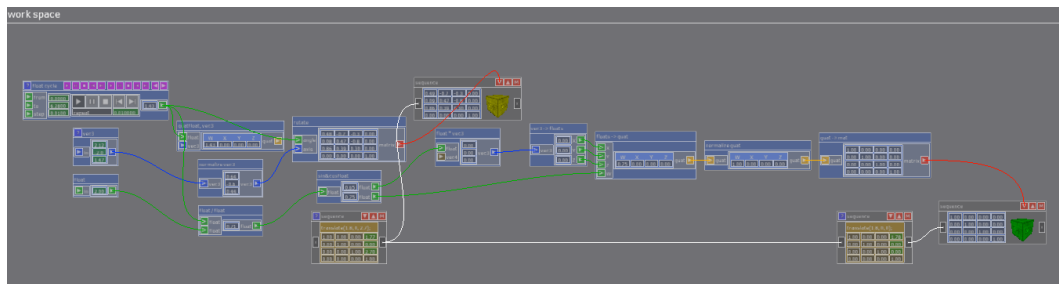


Figure A.18: First version of the layout technique on a scene called 10\_quaternionMatrixComparison

## Appendix B

### contents of the enclosed CD

```
CD
├── SOURCE.....Folder containing source files of I3T
└── document.pdf.....Pdf containing this document
```



# Bibliography

- [1] Web page of I3T[online].[cited 24.March 2019]. Available at:<http://i3t-tool.org/>
- [2] Web page of Blender[online].[cited 24.March 2019]. Available at:<https://www.blender.org/>
- [3] Web page of Unreal engine[online].[cited 24.March 2019]. Available at:<https://www.unrealengine.com/en-US/what-is-unreal-engine-4>
- [4] Web page of Lucidchart[online].[cited 24.March 2019]. Available at:<https://www.lucidchart.com/>
- [5] Web page of Public Implementation of a Graph Algorithm Library and Editor[online].[cited 21.June 2019]. Available at:<http://pigale.sourceforge.net/>
- [6] Web page of Graphviz[online].[cited 21.June 2019]. Available at:<http://www.graphviz.org/>
- [7] Documentation of the drawing technique used in graphViz [online]. [cited 24.May 2019]. Available at:<http://www.graphviz.org/Documentation/TSE93.pdf>
- [8] MAZETTI, Viktor a Hannes SÖRENSON. Visualisation of state machines using the Sugiyama framework. Göteborg, Sweden, 2012. Master of Science Thesis. Chalmers University of Technology[online]. Available at:<http://publications.lib.chalmers.se/records/fulltext/161388.pdf>
- [9] Giuseppe Di Battista, Peter Eades, Roberto Tamassia, Ioannis G Tollis, Algorithms for drawing graphs: an annotated bibliography, Computational Geometry [online]. 1994, vol. 4, num. 3[cit. 3.2.2019]. Available at: <http://www.sciencedirect.com/science/article/pii/092577219400014X>
- [10] Kazuo Misue and Peter Eades and Wei Lai and Kozo Sugiyama, Layout Adjustment and the Mental Map, Journal of Visual Languages & Computing[online]. 1995. vol. 6, num. 2 [cit. 29.1.2019]. Available at: <http://www.sciencedirect.com/science/article/pii/S1045926X85710105>
- [11] H. Gibson, J. Faith, P. Vickers, A Survey of Two-Dimensional Graph Layout Techniques for Information Visualisation, Information Visualization[online].2012, vol. 12, p. 324 - 357[cit. 6.2.2019]. Available at:<https://journals.sagepub.com/doi/abs/10.1177/1473871612455749>
- [12] Di Battista G. et al. (1997) Drawing directed acyclic graphs: An experimental study. In: North S. (eds) Graph Drawing. GD 1996. Lecture Notes in Computer Science, vol 1190. Springer, Berlin, Heidelberg

- [13] Kozo Sugiyama, Methods for Visual Understanding of Hierarchical System Structures, IEEE Transactions on Systems, Man, and Cybernetics, 1981, vol. 11. num. 2, p.109 - 125. ISSN 0018-9472
- [14] R. Tamassia, Handbook of Graph Drawing and Visualization. Discrete Mathematics And Its Applications, Taylor and Francis, 2010. ISBN 9781138034242
- [15] B. Berger and P. W. Shor, Approximation algorithms for the maximum acyclic subgraph problem, Information Processing Letters. 1994, vol. 51, num. 3, p. 133 -140. ISSN 0020-0190
- [16] P. Eades and N. C. Wormald, Edge crossings in drawings of bipartite graphs, Algorithmica[online]. 1994, vol. 11, pp. 379 – 403 [cit. 20.5.2019]. Available at:[https://www.researchgate.net/publication/220223342\\_Edge\\_Crossings\\_in\\_Drawings\\_of\\_Bipartite\\_Graphs](https://www.researchgate.net/publication/220223342_Edge_Crossings_in_Drawings_of_Bipartite_Graphs)
- [17] K. Sugiyama, Graph Drawing and Applications for Software and Knowledge Engineers. Series on Software Engineering and Knowledge Engineering, World Scientific, 2002. ISBN 9810248792
- [18] Thomas Gschwind, A linear time layout algorithm for business process models, Journal of Visual Languages & Computing[online]. 2014, vol. 25, num. 2, p. 117 - 132[cit. 3.2.2019]. Available at:<http://www.sciencedirect.com/science/article/pii/S1045926X13000797>
- [19] Emden Gansner, A Technique for Drawing Directed Graphs, Software Engineering, IEEE Transactions on[online]. 1993, vol. 19, p. 214 - 230[cit. 7.2.2019]. Available at:[https://www.researchgate.net/publication/3187542\\_A\\_Technique\\_for\\_Drawing\\_Directed\\_Graphs](https://www.researchgate.net/publication/3187542_A_Technique_for_Drawing_Directed_Graphs)
- [20] R. M. Tarawneh, A general introduction to graph visualization techniques, OpenAccess Series in Informatics[online]. 2012, vol. 27, p. 151 - 164[cit. 3.2.2019]. Available at:[https://www.researchgate.net/publication/286342713\\_A\\_general\\_introduction\\_to\\_graph\\_visualization\\_techniques](https://www.researchgate.net/publication/286342713_A_general_introduction_to_graph_visualization_techniques)