IMPROVEMENTS TO THE I3T USER INTERFACE

Bc. Dan Rakušan

Master's thesis
Faculty of Electrical Engineering
Czech Technical University in Prague
Department of Computer Graphics and Interaction
Study program: Open Informatics
Specialisation: Computer Graphics
Supervisor: Ing. Petr Felkel, Ph.D.
May 23, 2025

MASTER‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name:	**Rakušan Dan**
Personal ID number:	**493291**
Faculty / Institute:	**Faculty of Electrical Engineering**
Department / Institute:	**Department of Computer Graphics and Interaction**
Study program:	**Open Informatics**
Specialisation:	**Computer Graphics**

II. Master’s thesis details

Master’s thesis title in English:

Improvements to the I3T user interface

Master’s thesis title in Czech:

Vylepšení uživatelského rozhraní aplikace I3T

Name and workplace of master’s thesis supervisor:

Ing. Petr Felkel, Ph.D.   Department of Computer Graphics and Interaction

Name and workplace of second master’s thesis supervisor or consultant:

Date of master’s thesis assignment: **12.02.2025**   Deadline for master's thesis submission: ____________

Assignment valid until: **20.09.2026**

Head of department’s signature

prof. Mgr. Petr Páta, Ph.D.
Vice-dean´s signature on behalf of the Dean

III. Assignment receipt

The student acknowledges that the master’s thesis is an individual work.
The student must produce his thesis without the assistance of others, with the exception of provided consultations.
Within the master’s thesis, the author must state the names of consultants and include a list of references.

Date of assignment receipt

Student’s signature

MASTER‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name:	**Rakušan Dan**
Personal ID number:	**493291**
Faculty / Institute:	**Faculty of Electrical Engineering**
Department / Institute:	**Department of Computer Graphics and Interaction**
Study program:	**Open Informatics**
Specialisation:	**Computer Graphics**

II. Master’s thesis details

Master’s thesis title in English:

Improvements to the I3T user interface

Master’s thesis title in Czech:

Vylepšení uživatelského rozhraní aplikace I3T

Guidelines:

I3T is an interactive learning tool for transformations developed by students at the Department of Computer Graphics and Interaction FEL CTU. The application is in the later stages of development, but there are still some major issues with the UI implementation of the node-based editor of the application (the "workspace"), which needs to be reviewed and reworked to allow the introduction of new improvements.
1) Familiarize yourself with the current workspace node editor implementation in the I3T application [1, 2].
2) Evaluate functional and visual issues with the current implementation and issues with the codebase and its documentation.
3) Design and implement changes to address issues with the old workspace editor.
4) Use the new editor in collaboration with other students, whilst employing user-centered design, to implement design changes and features to enhance the usability of I3T, such as improving node and pin layout, making connecting nodes easier, invalid matrix value highlighting, addressing DPI scaling issues, or displaying individual nodes in a separate window.
5) As a secondary goal, explore the possibility of better visualizing the projection and viewport transformations and implement any necessary improvements to the 3D scene view [3]. Include a comparison between the OpenGL and Vulkan normalized device coordinate systems.

Bibliography / sources:

[1] Jaroslav Holeček: Adaptive learning in I3T software for education of geometric transformations, Master's thesis, http://hdl.handle.net/10467/107077
[2] Martin Herich: Restructuralization of Interactive Tool I3T for Teaching Transformations and Reimplementation of User Interface Using Dear ImGui Library, Bachelor's thesis, http://hdl.handle.net/10467/96746
[3] Dan Rakušan: Scene view for the I3T application, Bachelor's thesis, http://hdl.handle.net/10467/109656

Citation of this thesis: Rakušan Dan. *Improvements to the I3T user interface.* Master's thesis. Czech Technical University in Prague, Faculty of Electrical Engineering, 2025.

I would like to thank my supervisor, Ing. Petr Felkel, Ph.D., for his valuable comments and numerous pieces of advice. I would also like to thank the other members of the I3T team for all their cooperation throughout the project, especially Barbora for her help with the testing and for various design suggestions.

FAKULTA ELEKTROTECHNICKÁ
FACULTY OF ELECTRICAL ENGINEERING
Technická 2
166 27 Praha 6

DECLARATION

I, the undersigned

Student's surname, given name(s):	Rakušan Dan
Personal number:	493291
Programme name:	Open Informatics

declare that I have elaborated the master's thesis entitled

Improvements to the I3T user interface

independently, and have cited all information sources used in accordance with the Methodological Instruction on the Observance of Ethical Principles in the Preparation of University Theses and with the Framework Rules for the Use of Artificial Intelligence at CTU for Academic and Pedagogical Purposes in Bachelor's and Continuing Master's Programmes.

I declare that I used artificial intelligence tools during the preparation and writing of this thesis. I verified the generated content. I hereby confirm that I am aware of the fact that I am fully responsible for the contents of the thesis.

In Prague on 23.05.2025

Bc. Dan Rakušan
....
student's signature

Abstract

I3T is an interactive tool for teaching 3D transformations. This thesis focuses on improving the user interface of its node-based matrix editor implemented using the Dear ImGui C++ library. The first part of the thesis develops a new node editor component UI library to replace the existing implementation and then uses it to enhance usability and visual design of I3T. The second part of the thesis implements a new animated camera visualization technique, allowing the user to physically view the transformation of model vertices from world space all the way to screen space coordinates, bringing clarity to this often obscured and hard to understand process.

Keywords computer graphics, 3D transformations, I3T, OpenGL, C++, Dear ImGui, perspective projection visualization, Dear ImGui, user interface, node editor component

Abstrakt

I3T je interaktivní nástroj pro výuku 3D transformací. Tato práce se zaměřuje na vylepšení uživatelského rozhraní jeho maticového editoru implementovaného pomocí C++ knihovny Dear ImGui. V první části práce je vyvinuta nová knihovna realizující UI komponentu node editoru, která nahrazuje stávající implementaci, a která je využita ke zlepšení použitelnosti a vizuálního designu I3T. Druhá část práce se věnuje implementaci nové techniky animované vizualizace transformací kamery. Ta uživateli umožňuje názorně sledovat transformaci vrcholů modelu ze souřadnic světového prostoru až do souřadnic prostoru obrazovky, čímž vnáší do tohoto často nepřehledného a špatně pochopeného procesu novou míru jasnosti.

Klíčová slova počítačová grafika, 3D transformace, I3T, OpenGL, C++, Dear ImGui, vizualizace perspektivní projekce, Dear ImGui, uživatelské rozhraní, komponenta node editor

Contents

List of Figures

List of Tables

List of Code listings

Chapter 1
Introduction

The *Interactive Tool for Teaching Transformations*, or I3T, is an educational application developed at the Department of Computer Graphics and Interaction at FEL CTU. It is used to help students learn about matrix transformations in the computer graphics programming class taught at FEL as well as FIT.

It takes the form of a C++ desktop application in which the user can compose together various matrix transformations and view their effects on objects in a 3D world. The application is made up of two key interfaces: the 3D scene view and a 2D node editor (the *workspace*) in which the user can create and connect together nodes representing transformation matrices, models, and cameras. Inside the *workspace*, the user can construct a scene graph of the 3D scene rendered in the viewport. Both of these components are interactive, so changes to the matrices can be immediately observed to help with building intuition for how transformation matrices work.

I3T has been developed for many years now by numerous students of the faculty. It was originally developed in 2015 in a master's thesis by Michal Folta [1], but the development of a newer, more advanced, and user-friendly version started later in 2019 [2]. I joined the project in 2022 to work on my bachelor's thesis [3], which focused on a new viewport implementation that was previously started but unfinished by a previous student [4].

In the new version of I3T, the *Dear ImGui* library [5] was chosen to be used for the user interface. It is generally used for custom tooling for OpenGL/Vulkan applications rather than for more complicated end-user applications, but it is not a bad choice for a simple learning tool and students are often familiar with it. One issue that arose was the need for a *Dear ImGui* implementation of a node editor component, which is a rather complex UI component. There are existing open-source implementations around, but ultimately it was decided to implement it from scratch due to the need of nesting nodes inside other nodes. This was done as part of a master's thesis by Jaroslav Holeček [6, s. 92], in which he created the DIWNE[1] library implementing the node editor component and built upon it the current *workspace* implementation.

Since 2022, the application has progressed to a much more stable and usable state. However, one growing concern is the state of the *workspace*, which still contains many bugs, visual glitches and unusual behaviours that need attention. Perhaps more alarming is the underlying codebase that has been left largely unmaintained since Jaroslav left the team after completing his studies.

The *workspace* implementation was not the main topic of Jaroslav's thesis and was unfortunately developed very hastily to get a working solution, rather than a polished and scalable result. Over time, as other members of the team required new functionality, additional code was mixed in with the existing codebase without a proper understanding of the underlying design and problems, which only worsened the situation and made further development more difficult.

1.1 Goals

The primary goal of this thesis is to review and improve the *workspace* node editor implementation of I3T. Evaluating and eliminating its current issues, which primarily stem from the custom-made DIWNE node editor UI library upon which its built. The DIWNE library should be reviewed and reworked to achieve modular and maintainable code, improving the workspace implementation in the process.

The new codebase and UI implementation will be used to enhance the usability of the I3T application. If possible, this should be done in collaboration with other members of the I3T team, who work on the project concurrently and should include iterative testing in line with the user-centered design methodology.

Another key goal of this thesis is to expand the application's visualization tools to include a descriptive 3D depiction of the *view*, *projection* and *viewport* transformations — expanding the horizons of the application beyond the world space into the intricacies of the camera transformations and fixed-function stages of the OpenGL and Vulkan graphics pipelines. A particular focus will be placed on the perspective projection transformation, which is often difficult to understand and visualize.

[1] *Dear IMGUI Wrapper Node Editor*

1.2 Thesis structure

The thesis is structured into two main chapters. Chapter 2 addresses the first goal of the thesis, focusing on the UI improvements of the application's workspace node editor and the underlying codebase. Chapter 3 builds upon the new UI implementation, but explores the largely unrelated concept of camera transformations in pursuit of the secondary goal. Although this goal is introduced as the *secondary* goal in the assignment, it does in a way carry more importance, and the primary goal could be understood as its means to an end. For that reason, Chapter 3 is in fact longer and more detailed than Chapter 2.

1.3 Application overview

In case the reader is unfamiliar with I3T, this section serves as a brief introduction to it. It is followed by an even shorter cursory overview of the project's codebase.

I3T is a desktop application, developed in C++, consisting of two main components: the workspace and the scene view (formerly called the *viewport* or the *world*).

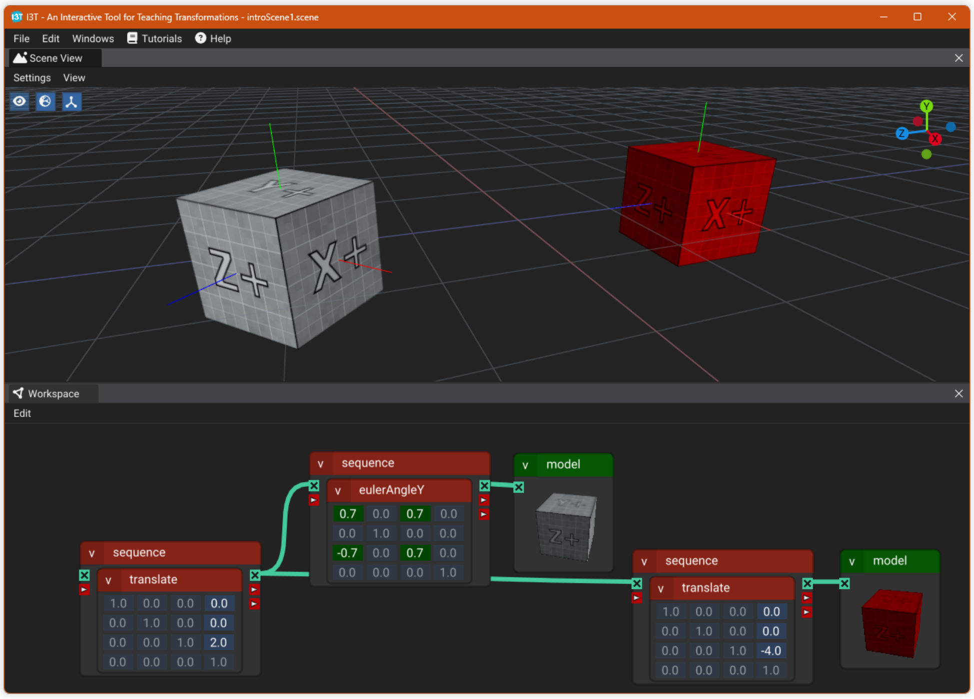
The workspace functions as a node-based editor, where users can create nodes (sometimes referred to as "boxes") which can be connected together using links by connecting their input and output pins. The most important nodes are the transformation nodes, represented as editable matrices, which can be placed into *sequence* nodes in a desired order, allowing users to construct composite transformations whose internal order can be changed easily via drag-and-drop.

Sequences can be chained together using links representing matrix multiplication, and a 3D model can be attached to a sequence to visualize the result of the final transformation matrix on the model's vertices. The scene view displays these models in a 3D world from the perspective of an independent user-controlled camera.

Transformation sequence chains can branch out, forming a structure akin to a scene graph (specifically, a directed acyclic graph). Each sequence computes its general "model" matrix by multiplying its parent's output matrix with its own local transformations.

Figure 1.1 shows an example scene in I3T: Two cubes (grey and red) share a parent transformation that translates by two units along the Z axis. The grey cube is then rotated around the Y axis, and the red cube is further translated by minus four units along Z. This places the red cube at world coordinates $(0, 0, -2)$. If disconnected from its parent, the red cube would instead be at $(0, 0, -4)$.

A camera node can be placed at the beginning of a transformation chain, allowing the user to input projection and view matrices. A connected *screen* node displays the 3D scene from that camera's perspective. Figure 1.2 shows an I3T scene demonstrating a camera using the orthographic projection.


I3T - An Interactive Tool for Teaching Transformations - introScene1.scene
File Edit Windows Tutorials Help
Scene View
Settings View
Workspace
Edit
sequence
translate
sequence
eulerAngleY
model
sequence
translate
model

translate

1.0	0.0	0.0	0.0
0.0	1.0	0.0	0.0
0.0	0.0	1.0	2.0
0.0	0.0	0.0	1.0

eulerAngleY

0.7	0.0	0.7	0.0
0.0	1.0	0.0	0.0
-0.7	0.0	0.7	0.0
0.0	0.0	0.0	1.0

translate

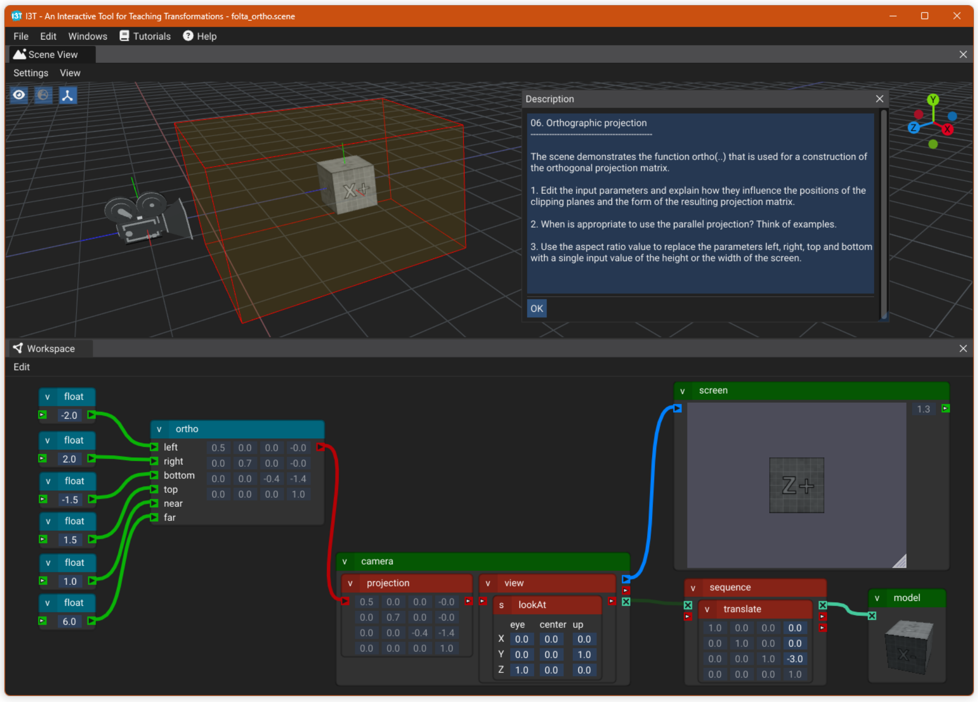
1.0	0.0	0.0	0.0
0.0	1.0	0.0	0.0
0.0	0.0	1.0	-4.0
0.0	0.0	0.0	1.0

Figure 1.1 Screenshot of the I3T application at the beginning of this thesis. The loaded scene contains two transformations that share a third parent transformation, each applied to a cube.

Connections between nodes can also carry data. A sequence can output a matrix computed only from its own local transformations, without including its parent. These data connections are primarily used by the *operator* nodes, which perform numerical operations such as matrix inversion, and allow manipulation of numbers, vectors, matrices, and quaternions. They also handle type conversions between data types. The result of such operations can be plugged back into a sequence using its matrix data input. For example, operators are used in figure 1.2 to assemble an orthographic projection matrix from individual camera frustum plane position parameters.

Unlike other educational tools focused on fixed scenarios, I3T offers a flexible environment capable of illustrating a broad range of computer graphics concepts. The application also contains tutorials that guide the user through various examples.

A similar introduction to I3T has been given in a previous thesis about the application's scene view implementation [3, pp. 6–7]. It contains similar figures to 1.1 and 1.2, shown in the original I3T application by Michal Folta [1].



06. Orthographic projection

The scene demonstrates the function ortho(..) that is used for a construction of the orthogonal projection matrix.

1. Edit the input parameters and explain how they influence the positions of the clipping planes and the form of the resulting projection matrix.

2. When is appropriate to use the parallel projection? Think of examples.

3. Use the aspect ratio value to replace the parameters left, right, top and bottom with a single input value of the height or the width of the screen.

Figure 1.2 An I3T scene demonstrating orthographic projection using operator nodes (wih blue headers) to construct the projection matrix.

1.3.1 Concurrent work

I3T as a project has been around since 2016, during which time many students have collaborated on its development, be it with direct code contributions or with design and usability studies. A comprehensive history of the projects and theses contributing to it can be found in my previous thesis [3, p. 7].

The project has been open-sourced and is actively used in class. Although not much effort has been put into its public propagation yet, as the development still continues.

Currently, four students are working on I3T as part of their master's and bachelor's theses concurrently with this one:

- **Barbora Hálová, bachelor's thesis**
  User research and UX review of the I3T app. Design and prototyping to improve the usability of the I3T app. [7]
- **Martin Herich, master's thesis**
  Working on a new scripting system and automatic GitHub publishing [8].
- **Matvei Korikov, bachelor's thesis**
  Dockable windows, history, logging, language mutations and optimization of the I3T tool.

- **Karolína Zapletalová, bachelor's thesis**
  Optimizing the learning environment in the I3T Tool for different user types.

1.3.2 Codebase

Figure 1.3 presents a package diagram of the application's codebase, which will be referred to occasionally in the coming chapters. Each package represents a directory of the application's main Source directory. The top-level directories represent largely independent modules of the application. Chapter 2 focuses on the improvements to the workspace, focusing on the GUI and DIWNE directories. Chapter 3 introduces new core functionality and makes changes in the Core▸Nodes and Viewport directories as well.

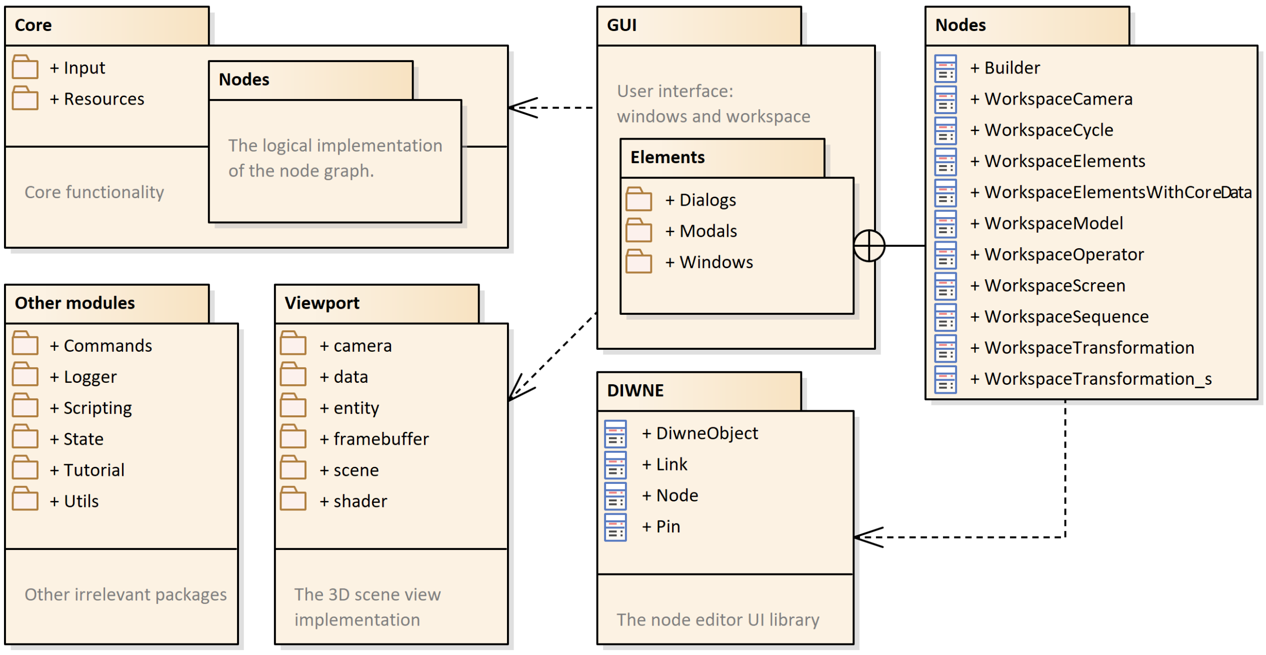


Figure 1.3 Package diagram of the I3T project's relevant codebase.

Chapter 2
The workspace

This chapter focuses on the primary goal of this thesis: improvement of the application's node-based editor (The *workspace*). To do so, the existing user interface solution must first be analysed. It consists of two main parts, the original DIWNE node editor component library, developed by Jaroslav Holeček [6] and its use in the actual I3T node editor implementation.

First, the *Dear ImGui* library is introduced in further detail as it is the library the entire application's UI is built upon and its design heavily influences the design of the *workspace* implementation and the underlying node editor library.

Next, the DIWNE library is introduced and its current issues are described. The DIWNE library and the *workspace* implementation can essentially be considered to be the same codebase, as they are substantially interconnected, which is one of the major issues this thesis aims to resolve.

From the beginning of this thesis, it has been clear that the DIWNE library requires a major rewrite, a sort of DIWNE version 2, which is the focus of the rest of the chapter.

2.1 Dear ImGui library

Dear ImGui is a popular open-source C++ library most commonly used to develop user interfaces in game engines and real-time 3D applications as it offers easy integration and quick iteration in development [5].

It uses the *immediate mode* paradigm[1], a methodology and a style of API that avoids any retained state and promotes direct synchronization of the UI with its data. In contrast, most UI frameworks use a *retained mode* paradigm, which generally uses object-oriented design and explicit data synchronization (using events or callbacks).

[1]More info at https://github.com/ocornut/imgui/wiki/About-the-IMGUI-paradigm

In *Dear ImGui*, the user interface is completely defined by a sequence of commands executed every frame, which always builds the entire UI from the ground up. One frame may display an interface completely different from the previous one. The actual rendering occurs once at the end of the frame, by rendering all the so-called `DrawLists`, containing lists of rendering commands that have been sequentially created by each call to the *Dear ImGui* functions in that frame. Each frame responds dynamically to user actions directly when building the UI. The application queries the state of user input each frame when constructing the UI elements.

Code listing 2.1 illustrates the principle behind *Dear ImGui* with an example of a button. Calling `ImGui::Button()` causes a button to appear at the current position in the window. Inside this method, user input is immediately evaluated and a boolean is returned indicating whether the user just clicked the button. Thus, it is possible to react to the button press in the same place in the code where the button was created. Since this reaction is directly in the code building the UI elements, the reaction can be, for example, the creation of more nested UI elements.

```
if (ImGui::Button("Click me"))
{
    std::cout << "Button was clicked" << std::endl;
}
```

Code 2.1 Example of an Dear ImGui button

2.2 DIWNE library

The purpose of the DIWNE library is to implement a node editor UI component and allow the user to structure regular *Dear ImGui* code into free-floating nodes, essentially tiny windows, that can be dragged around and connected together using pins to create a node graph. This interface allows the user to express a wide range of concepts like operations in regular or matrix algebra in our *workspace*. This interface is often used in more technical applications such as Blender (shader and geometry nodes), Unreal Engine (application logic) or Houdini (visual effects). Node editor of Blender and the existing I3T *workspace* can be seen in Figure 2.1.

As mentioned, there are existing libraries for implementing a node editor using *Dear ImGui*, which usually follow its *immediate mode* paradigm. Code listing 2.2 contains an example of how a node is constructed in the third-party library *imgui-node-editor* [9].

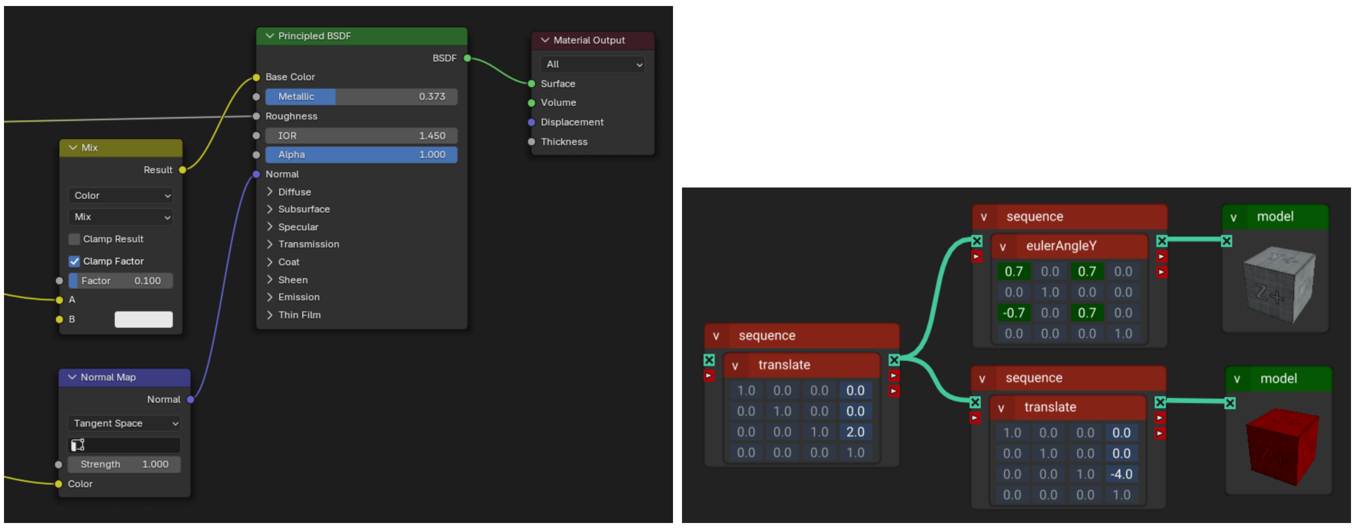


Figure 2.1 Node editor interface examples from Blender (left) and I3T (right).

```cpp
ed::BeginNode(uniqueId++);
    ImGui::Text("Node A");
    ed::BeginPin(uniqueId++, ed::PinKind::Input);
        ImGui::Text("-> In");
    ed::EndPin();
    ImGui::SameLine();
    ed::BeginPin(uniqueId++, ed::PinKind::Output);
        ImGui::Text("Out ->");
    ed::EndPin();
ed::EndNode();
```

Code 2.2 Example usage of the *imgui-node-editor* library [9]

In contrast, DIWNE uses an object-oriented design, much more similar to what conventional *retained mode* GUI frameworks use, like Qt[2] or Java Swing[3]. This was a design choice made by Jaroslav, and it is largely implied in the name „Dear Imgui Wrapper Node Editor“, as in wrapping the *immediate mode* Dear ImGui function calls into methods called by an overarching object-oriented structure.

Nodes in DIWNE are represented as `DIWNE::Node` class instances which can use inheritance to create various node types. Pins of the node are separate instances of the `DIWNE::Pin` class which the node can create in its constructor. Similarly, the links connecting them are using the `DIWNE::Link` class. The content of the node is specified by implementing the `content` method, which contains regular *immediate mode* Dear ImGui code. These classes are all subclasses

[2]https://www.qt.io/product/framework
[3]https://en.wikipedia.org/wiki/Swing_(Java)

of a `DiwneObject` class which serves as the root of the inheritance hierarchy, viz. class diagram in Figure 2.2.

This approach mirrors the *retained mode* approach in the general node editor implementation while still benefiting from the *immediate mode* paradigm inside the actual contents of the nodes and other editor elements. The idea is that object-oriented design is generally more intuitive and better suited for complex systems like a node editor. It also offers fine-grained control over the library, as each object can be modified using inheritance.

```cpp
class ExampleNode : public DIWNE::Node {
  BasicNode(NodeEditor& editor) : DIWNE::Node(editor, "Node A") {
    // Constructor intialization code
    Node::addInputPin("-> In");
    Node::addOutputPin("Out ->")
  }
  void content() {
    ImGui::Text( ... ); // Any ImGui code
  }
};
// Initialization called once
editor.createNode<ExampleNode>();
// Drawing every frame
editor.draw();
```

Code 2.3 Example DIWNE pseudocode[4]

This hybrid design obviously breaks the main pillars of the *immediate mode* paradigm. Notably, state, in the form of various `DiwneObject` instances, needs to be stored manually by the user. Code listing 2.3 shows a DIWNE code example equivalent to the previous code listing 2.2. A node is defined in a new separate class, and then an instance must be created and stored in some data structure (in this case handled internally by a node editor instance).

The actual drawing and input handling of the node is handled in an *immediate mode* manner by repeatedly calling a `draw` method of the overarching node editor object, which internally calls `draw` methods for each node. Inside the draw methods, any state synchronization is kept to a minimum as the standard *Dear ImGui* API is used.

Compared to the *immediate* approach, the setup and storage are more complicated, but at any point, we have an object-oriented representation of the editor that can be used as a data model itself. If the node editor is used only to visually represent an existing node graph model, explicit state synchronization of nodes and their connections does become necessary, but the structure of the node editor's graph is expected to be mostly static with only occasional creations of new nodes, pins or links.

[4]This is just illustrative pseudocode of the desired API

On the other hand, if the underlying node graph model is itself object-oriented, as in the case of the core I3T node logic implementation, each UI node can be associated with its logical equivalent, and synchronization becomes rather simple and self-contained.

2.3 Existing issues

The sections above outline what the DIWNE library should be, rather than what it really is. The example code in listing 2.3 is fictional pseudocode that cannot be realized yet, as the library was never properly finished to function in a stand-alone manner. The following sections outline the individual issues with the current implementation's codebase itself, as well as its usage.

2.3.1 Codebase

One of the main issues with the current DIWNE implementation is that the DIWNE library itself was not written independently from the *workspace* implementation of I3T. Instead, much of the core functionality is implemented somewhere down the inheritance hierarchy tree inside specialized *workspace* classes that, ideally, should only be concerned with I3T-related functionality and not the inner workings of the node editor.

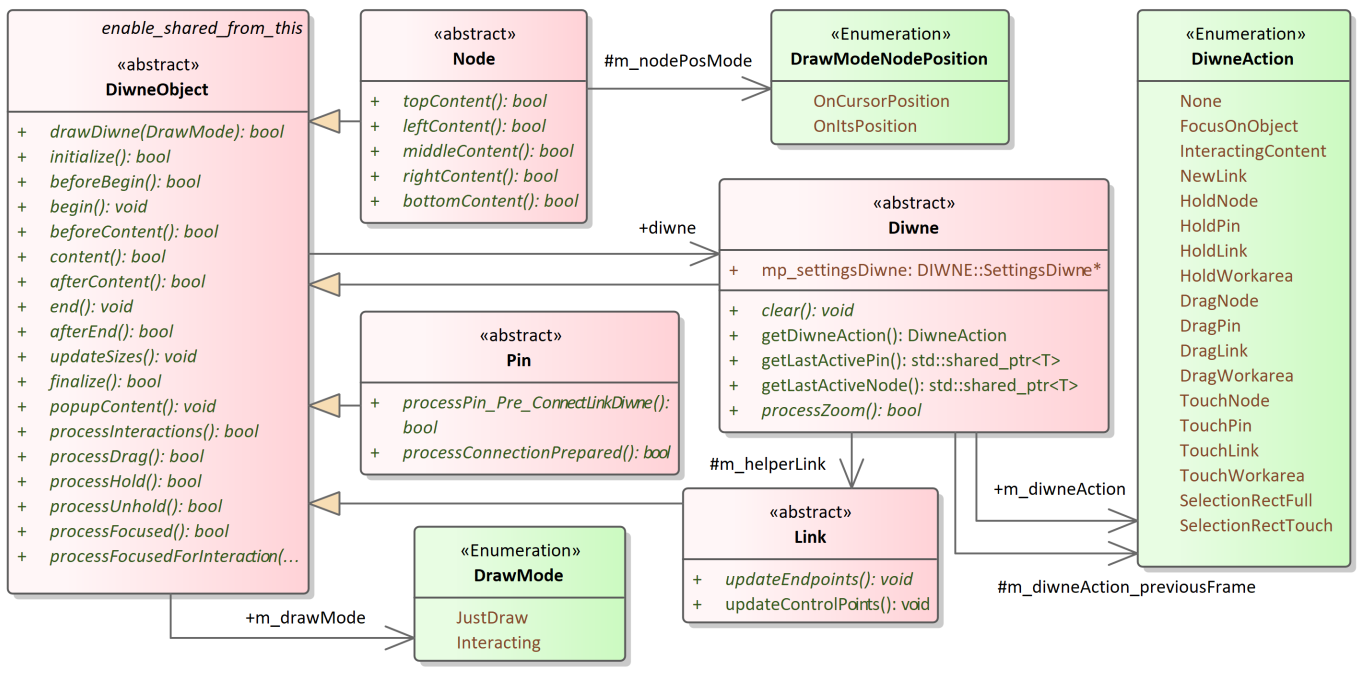


Figure 2.2 Class diagram of the current architecture of the DIWNE library. The library is located in the Source▸DIWNE directory, see Figure 1.3 for the package overview.

Figure 2.2 shows all classes inside of the Source ▸ DIWNE directory which form the library. They contain a small percentage of the fundamental implementation that is otherwise scattered elsewhere. They effectively serve as a mere abstract interface, which would not be a problem if there actually were an independent concrete implementation. But this is not the case.

Workspace implementation

The bulk of the general node editor implementation is contained in various *workspace* classes located in the Source ▸ GUI ▸ Elements ▸ Nodes. Their implementation completes the essential functionality of the DIWNE library, which should be moved back into the library itself.

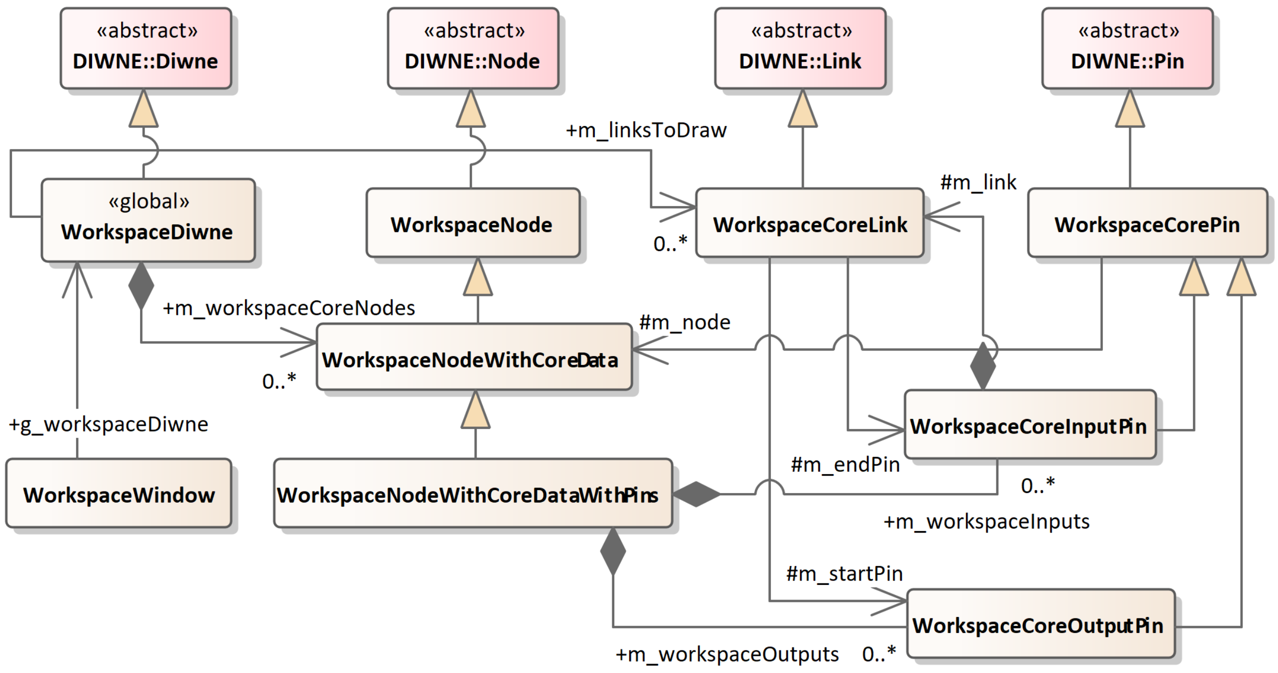


Figure 2.3 Class diagram of the basic *workspace* classes and how they relate to the DIWNE library.

Diagram in Figure 2.3 shows how the four primary DIWNE classes are used in the workspace implementation. Each of them is subclassed to create a workspace variant. This design makes sense if the classes contained in the DIWNE library itself were purely virtual interfaces and the workspace classes served as their concrete implementations. Which is almost true, except that some of the implementation is already contained in the DIWNE classes and the workspace classes also additionally contain I3T specific code[5]. A particularly large offender is the `WorkspaceDiwne` class, which implements essential node editor features while also serving as the main workspace class.

[5]Also if it were true, the DIWNE library wouldn't be much of a library at all.

The `WorkspaceNodeWithCoreData` class acts as the base node for all workspace nodes, wrapping and managing an instance of a *core* logical node from the Source ▸ Core ▸ Nodes package. This is the internal logical representation of the scene graph that the workspace UI layer represents. This is a deliberate design choice that splits the user interface and the node graph data into two separate modules. More information about the core nodes can be found in a previous thesis by Martin Herich [2, p. 30] as well as his more recent thesis about scripting in I3T [8]. A partial class diagram of the core nodes is shown later in Figure 3.13 in Chapter 3.

The remaining classes do mostly implement DIWNE functionality and are not particularly related to the workspace implementation. There are some design choices that are tied to the specific design of I3T that might be worth revisiting.

The new design should extract the essential node editor functionality, making most of the classes from the 2.3 diagram a part of the DIWNE implementation. This has the benefit of making the DIWNE library usable in other applications, making its code more concise, reducing the intermingling of responsibilities, and letting the workspace classes focus on I3T specific functionality.

The existing code contains many duplicated code fragments or functionality that should instead use polymorphism or regular functions. A notable example would be numerous reimplementations of the method responsible for drawing node headers, which workspace nodes use to change their color.

Virtual methods are widely used to allow the user to override essentially any method and functionality of the library. The issue is that sometimes tiny essential code fragments are moved into separate methods that the user would realistically have no need to override. This, combined with often very confusing method names, often makes code highly unreadable, since anyone inspecting it needs to follow a chain of many unnecessary method calls.

The code lacks any substantial Doxygen documentation, and the few regular comments inside the files are sparse and often don't explain much in detail. There is some external documentation that does dwell into the general principles of the library a little further, as well as some additional description of some of the classes, which should have been added to the code as Doxygen comments in the first place.

Crucially, there is no guide describing the usage of the library nor any code examples. The issue of lack of examples is mentioned by the author himself and is complicated by the aforementioned fact that the library cannot really be used as a stand-alone component, due to its dependency on the I3T application code.

2.3.2 Functionality and appearance

The *workspace* is in a fairly usable state as it fulfills all basic requirements and has already been used in practice by students in the Graphics programming class, but there are some major issues with the usability of the interface.

Focus issues

One fundamental issue is behaviour regarding the focus of elements and context-aware restriction of inputs, like keystrokes, to a specific element in the node editor. For instance, editing a text field and pressing Ctrl + A will not only select all the text inside the text field, as expected, but will also select all nodes in the node editor, which is not desirable. This is because the node editor registers a global keystroke and has no knowledge about what is happening within that frame.

The node with the text field does pass along information to the node editor that *something* is occurring in it, but gives no indication of what that *something* is, since it is only passing a single `bool` flag. This flag indicates some kind of „interaction", but that includes any interaction such as a mere mouse hover over the node, which cannot be used to decide whether to block keystrokes to the node editor or not.

Another example of this issue is the inability to pan the node editor when the mouse is hovering over a node, due to the editor not knowing whether the node is being interacted with in such a way that should prevent panning, so it just doesn't allow panning at all. Similarly, dragging of a node is only allowed when the mouse grabs its header, as any elements in the node's body would block the drag operation, even when they themselves do not react to the input.

In other words, the node editor is missing some kind of an input propagation system that would inform subsequent elements whether they shall interact with input or not. The meaning of the passed along `bool` flag could be modified to fit that purpose, but at the same time, we might want to retain information about hovering or any visual changes to an element, or perhaps information about other interactions that don't prevent other elements from receiving input.

This is a very common concept in user interface frameworks called "event bubbling" or "event propagation" [10, p. 463] which is, for example, used in JavaScript, Qt or Java Swing.

Visual inconsistencies

There are issues with the colors and widths of new connection links. The width of links seemingly changes at random, and their color sometimes does not correspond with the starting pin.

There are some problematic cases when links are drawn over existing nodes and then disappear again when a different node is focused. It might be desirable to always draw links behind all nodes to avoid that, as many node editors generally do.

The visual style of nodes is also somewhat unappealing, consisting of only a brightly colored square with sharp corners and an icon within. Not corresponding to the more polished style of nodes with rounded corners and edge borders.

Pin icons have scaling issues when zooming (inconsistent widths, and pins themselves cannot be made bigger to make creating connections easier, as that would break the hardcoded layout system inside the `WorkspaceNodeWithCoreDataWithPins` class. In rare cases, the layout system also encounters an issue at smaller zoom levels causing the pins to „vibrate“ side by side by one pixel.

Lack of UI menus and auxiliary elements

The *workspace* does have basic key shortcuts for selection, duplication and deletion of elements. However, there are no corresponding UI menu bar items, and thus no indication of their existence.

Some useful quality-of-life features are missing, for instance some sort of background grid that would indicate the movement of the editor's infinite plane or a „minimap“ indicating the position of the current view relative to all the nodes present in the editor. Without these improvements, the user can easily get „lost“ in the editor when panning to an area that has no nodes present.

2.4 New DIWNE library

The following sections describe the new implementation of the DIWNE library and are accompanied by Figure 2.4 showing all packages and classes of the new DIWNE library implementation.

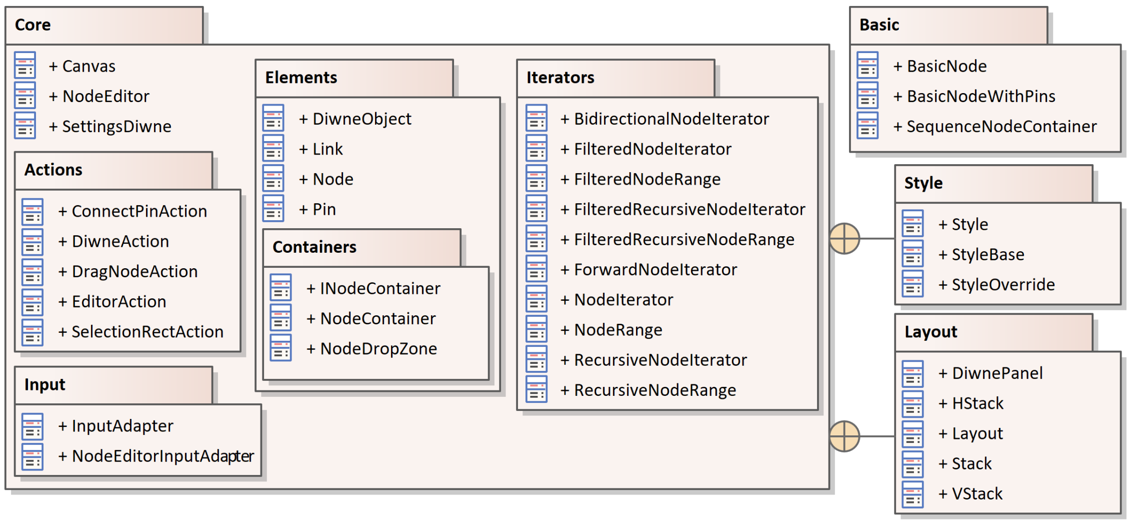


Figure 2.4 Package diagram of the new DIWNE library located in the Source ▸ DIWNE directory.

2.4.1 Refactoring

One of the first steps in the new implementation was to do a major refactor of the existing code. There were many confusingly named methods and variables that used terms that didn't have a well-defined meaning. This was an issue primarily due to the lack of documentation. A noteworthy example would be the use of terms „focus“ and „focus for interaction“, the former meaning the mouse is hovering over an element and the latter meaning the element is actually focused by previously clicking on it.

There were countless issues with the previous implementation regarding code being in places it shouldn't be, violating the single-responsibility principle[6]. The node editor class, formerly named `Diwne`, was a giant class that included a large amount of code related to the drawing of icons and geometric primitives within the node editor's coordinate system. This code was moved into a separate `Canvas` class. As well as all code relating to keeping track of the node editor's viewport position and transformation from screen space into local coordinates of the "infinite canvas". Methods that handle the node editor's zoom function were moved into `Canvas` as well.

The `Diwne` class, now named `NodeEditor`, also contained a large amount of methods that acted as proxies for querying input from *Dear ImGui* (methods like `bypassIsItemClicked0`, `bypassIsMouseDown0` and `bypassIsMouseReleased1`). These were moved into another separate class named `NodeEditorInputAdapter` that inherits from a more general `InputAdapter` object. These provide an interface and default implementation of various input queries, which can be modified by subclassing the adapter to change input bindings, at least at compile time. Input rebinding is not supported by I3T yet, but these classes would serve as a good basis for it.

The GUI ▸ Nodes directory has been moved into a new GUI ▸ Workspace directory that now contains all files related to the workspace node editor implementation. Most workspace nodes have been renamed to shorten their names and the "Workspace" prefix has been replaced with a namespace.

2.4.2 New architecture

As discussed in Section 2.3.1, key node editor functionality has been extracted from the workspace classes and moved into DIWNE, only the `DiwneObject` remains abstract. A new class diagram comparable to the one from Figure 2.3 can be seen in Figure 2.5.

For simplicity, the separate input and output pin classes were removed. The pin now has a boolean parameter, specifying whether it is an input pin or not. Links are no longer managed individually by input pins, but instead are owned by the node editor, just like nodes. Links now get destructed when disconnected, as previously they were only hidden, because every input pin contained a link instance, no matter if it was plugged in or not.

[6]https://en.wikipedia.org/wiki/Single-responsibility_principle

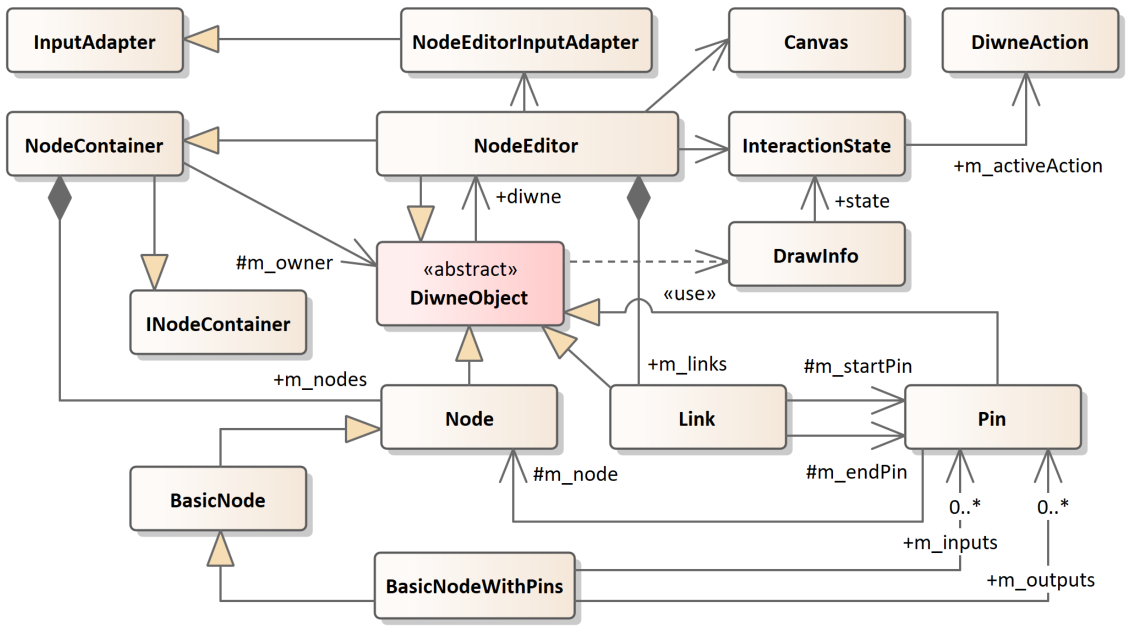


Figure 2.5 Class diagram of the main classes of the new DIWNE library.

To extract the core functionality of the library from the *workspace* implementation whilst retaining a reasonable level of abstraction in the DIWNE base classes, an intermediary „reference“ concrete implementation of the `DIWNE::Node` class was created in a new Basic folder. This implementation extracts the layout and rendering code of I3T nodes, giving a basic implementation of a general node that is not specific to only our *workspace*.

This new class named `DIWNE::BasicNode` separates the rectangular node into a header at the top, and below a content area divided horizontally into a left portion, filled with input pins, an auto-expanding centered content in the middle, and a right portion housing output pins.

The layouting of content inside nodes is a rather difficult topic due to the absence of any standard UI layouting tools in *Dear ImGui*. The previous implementation relied on hardcoded calculations of specific sizes on a case-to-case basis, leading to very inconsistent results and cluttered code. A more general solution is presented later in Section 2.4.8.

The `NodeEditor` is now a subclass of a generic `NodeContainer` class that holds all the nodes within it. This is explained in Section 2.4.6.

2.4.3 Input propagation

To resolve the focus issues mentioned in Section 2.3.2, the draw method of `DiwneObjects` must return multiple `bool` values, flags, that indicate different types of interaction. The draw method is called by the node editor in a frontto-back fashion, meaning the topmost node has its draw method called first, then the nodes below it. This allows us to use these flags as a sort of a filter, once a particular flag is returned as true by an element in the foreground, other nodes can query some „global“ state and adjust their behaviour accordingly. This state then resets each frame.

This is essentially how the old implementation worked, but it only used a single `bool` flag as the return type for each draw method. As can be seen in code listing 2.4. It contains the `DiwneObject::drawMethod` that calls many individual frame "lifecycle" methods that derived objects can override. Each one of these methods returns its own flag value that is then combined with a logical OR operator into a single flag that ultimately represents the whole object.

This approach works well with a single boolean value, but is rather difficult to extend to return multiple boolean values. It requires each method that contains any drawing code to define and return a new boolean value, and any method calling it to merge it with its own boolean flag. If this boolean was replaced with an object (a `struct`), the code might get somewhat confusing, and it would be good to reduce the amount of required boilerplate anyway.

```
bool drawDiwne(DrawMode drawMode) {
  bool flag = false;
  flag |= initializeDiwne();
  if (allowDrawing()) {
    flag |= beforeBeginDiwne();
    begin();
    flag |= beforeContentDiwne();
    flag |= contentDiwne();
    flag |= afterContentDiwne();
    end();
    updateSizes();
    flag |= afterEndDiwne();
    flag |= processInteractionsDiwne();
  }
  flag |= finalizeDiwne();
  return flag;
}
```

Code 2.4 Old DiwneObject drawing code.

```
void drawDiwne(DrawInfo& context,
               DrawMode mode)
{
  initializeDiwne(context);
  if (allowDrawing()) {
    beginDiwne(context);
    content(context);
    endDiwne(context);
    afterDrawDiwne(context);
  }
  finalizeDiwne(context);
}
```

Code 2.5 New DiwneObject drawing stages, its frame "lifecycle".

Instead, the new implementation in code listing 2.5 adds a single argument „context“ of type `DrawInfo` to each drawing method. This context object is passed along by reference and accumulates changes automatically. To avoid any confusion, it provides utility methods with descriptive names to apply changes to itself. The number of methods the `DiwneObject` draw method is split into was also reduced, to improve code readability.

The structure of the `DrawInfo` object can be seen in code listing 2.6. It defines several types of flags. The focus issues are resolved by discerning three different types of interaction: purely visual, logical and input blocking. Additionally, extra information is retained about the state of hover and popup menus, which was previously kept in the global state of the node editor.

```cpp
class DrawInfo {
  unsigned short visualUpdates{0};
  void visualUpdate();
  unsigned short logicalUpdates{0};
  void logicalUpdate(bool isVisualUpdateAsWell = true);
  unsigned short inputConsumed{0};
  inline void consumeInput() { inputConsumed++; }
  unsigned short hoverConsumed{0};
  inline void consumeHover() { hoverConsumed++; }
  unsigned short popupOpened{0};
  inline void popup() { popupOpened++; }
}
```

■ **Code 2.6** The `DrawInfo` `struct`

Since this object is passed to each method by reference, and then modified inside that method, it is not simple to capture information about what specific changes were made inside a particular draw method. If each method returned a new instance of `DrawInfo`, the returned object would contain that information, but the whole merging step would need to be done everywhere. Knowing this immediate change is required, for example, when figuring out whether a particular node encountered a „logical“ interaction causing it to be brought into the foreground. But in most cases, knowing the change is not necessary.

To allow working out the change, each flag is stored as a number (a `short`), and a helper class named `ContextTracker` can be used to store a copy of a `DrawInfo` object before making a draw method call, and then compare the number values of each flag to determine the change, stored in a separate `DrawInfo` object that can be queried. This process can be seen in code listing 2.7 showing an alternative to the `DiwneObject` draw method that does capture and return the immediate change.

```cpp
DrawInfo DiwneObject::drawDiwneEx(DrawInfo& context, DrawMode
    drawMode) {
  ContextTracker tracker(context);
  drawDiwne(context, drawMode);
  return tracker.end(context);
}
```

■ **Code 2.7** Usage of the `ContextTracker` class to track changes of a `DrawInfo` instance passed to a draw method.

2.4.4 Object lifecycle and input processing

As shown in code listing 2.5, each `DiwneObject` is drawn using the draw method. This method is responsible for drawing the object as well as reacting to any user input as it is equivalent to the usual Dear ImGui draw methods. The method is divided into several stages which can be overridden in derived objects:

- **`initialize()`**
  First method to be called every frame. Does not handle drawing.

- **`begin()`**
  First method to be called during object drawing. Can be used to initialize drawing code. Dear ImGui's "begin" calls can be placed here.

- **`content()`**
  Draws the main object content.

- **`end()`**
  Used to end content drawing. Dear ImGui's "end" calls can be placed here.

- **`updateLayout()`**
  This method is responsible for keeping track of the objects size. The method is called by DIWNE right after the `end()` method in the internal `endDiwne()` method.

- **`processInteractions()`**
  Method for reacting to user input after the object is fully drawn and its dimensions are known. Called internally right before `afterDraw()`, but after the internal `processInteractionsDiwne` method that performs the usual input processing.

- **`afterDraw()`**
  Called last during drawing. At this point the size of the object should be calculated by the `updateLayout` method and this method should be able to work with it. Because of that the drawing code within shouldn't affect the objects size anymore.

- **`finalize()`**
  The final method to be called, gets called every frame and doesn't do any drawing.

When the `allowDrawing` method returns false, only the `initialize` and `finalize` methods are called. The internal `processInteractionsDiwne` method is also divided into several methods that incrementally check whether a particular interaction is occurring:

- **`processHoverDiwne()`**
  Checks whether the object is currently hovered. This is usually done by simply checking the `m_internalHover` flag that has been previously set in the `end` method by calling `ImGui::IsItemHovered()` on the drawn content. Hovering is often a prerequisite for further interactions. Triggers the `onHover` callback method.

- **`processPressAndReleaseDiwne()`**
  Determines whether the object is pressed or released. Being pressed means that an input is activated over the object while it was hovered. Triggers the `onPressed` and `onReleased` callback methods.

- **`processDragDiwne()`**
  Processes whether a dragging operation with this object as source should begin. The object must be first pressed and then moved by more than the mouse drag threshold while still being pressed (but may be no longer hovered). Only one drag operation can be active at a time. When active, the `onDrag` callback is triggered with the `dragStart` and `dragEnd` parameters. The callback is guaranteed to be called at least twice, with each parameter set to true once.

What input triggers a particular interaction state can be modified by overriding specific "input trigger methods". The objects react to the left mouse button by default, but they can react to any input and even multiple inputs. The input processing methods do not differentiate between multiple allowed inputs. That must be done manually later.

2.4.5 Actions in DIWNE

While the `DrawInfo` object solves issues with input blocking within a single frame, there are also interactions that span multiple frames and require a kind of special „mode“ of operation. These interactions are called *actions* and are, for example, the act of dragging a selection rectangle in the node editor to select multiple nodes, or the dragging of a new link connection from a pin. When these actions are occurring, the elements of the node editor should ignore any irrelevant inputs.

Previously, some kind of an action was always occurring, in the form of a fixed global `DiwneAction` enum that considered even trivial interactions like hovering or holding a key down over an element to be an action. The actions were then further divided into what type of element (node, pin or link) was performing the action, each action being a separate value of the action enum, which can be seen in Figure 2.2.

Elements then queried this global action state in various parts of their code with complicated if statements that checked if a particular enum value was set. This system was further complicated by storing action types for the current and previous frame, often reacting if an action occurred in the current or the previous frame, to avoid any race conditions relating to some value being set from an element drawn before or after an element reacting to it.

The new implementation removed this system and introduces a self-contained `DiwneAction` object, that represents a special mode that other elements can react to. An action contains a reference to a source element that initiated it and is responsible for ending it. This usually corresponds to the beginning and end of a drag operation. The active `DiwneAction` resides in an `InteractionState` member variable of a node editor. This is in principle an extension of the `DrawInfo` context that is persistent across frames.

The interaction state also contains information about dragging, which is initiated when the mouse (by default, can be changed to another keystroke) is pressed above an element and then dragged away while still being held down. This causes the `dragging` flag of the `InteractionState` object to become true and a source element of the dragging operation is also noted down, the `onDrag()` method of the element is then called, which can be used to initiate an action. The `onDrag()` method is then called throughout the dragging operation until it ends, at which point it is also called with an argument notifying that the drag has ended, and any initiated action should be ended.

The `InteractionState` object is always accessible by reference from the `DrawInfo` context object, and contains similar helper methods for querying and initiating actions. For a specific action, the `DiwneAction` object can be subclassed to create a specialized action which can store specific data relating to the action.For instance, the action responsible for connecting pins has a subclass named `ConnectPinAction` that stores a reference to the currently dragged link and the pin it originates from. Other pins can check if the `ConnectPinAction`[7] is currently active and connect themselves to the dragged link if the dragging operation happened to end above them. Example usage is shown in code listing 2.10. The outlines of the `DiwneAction` and `InteractionState` classes can be seen in code listings 2.8 and 2.9.

Specialized action objects can also override an `DiwneAction::onEnd()` method that gets called at the end of the frame they ended in to perform any cleanup (like destroying an unconnected link). The action instance is only cleared at the end of the frame to avoid race conditions where data of an action was already deallocated when it was stopped in an element drawn before another one wanting to react to it.

```cpp
struct DiwneAction {
  std::string name;
  std::weak_ptr<DiwneObject> source;
  bool endActionThisFrame{false};

  virtual void onEnd(){};
};
```

Code 2.8 Outline of the `DiwneAction` object.

```cpp
class InteractionState {
  std::unique_ptr<DiwneAction> action;

  ... action related helper methods ...

  bool dragging{false};
  bool dragEnd{false};
  std::weak_ptr<DiwneObject> dragSource;
};
```

Code 2.9 Outline of the `InteractionState` object.

[7]Each action has an id (name) that can be queried.

```cpp
// DrawInfo& context is passed as an argument, state is an instance of InteractionState
// Start an action
auto connectPinAction = context.state.setAction<Actions::ConnectPinAction>(m_labelDiwne);

// Try to retrieve an action (will be null if not)
auto connectPinAction = context.state.getAction<Actions::ConnectPinAction>();

// End the active action
context.state.clearAction();
```

Code 2.10 Example of initiating, querying and ending the connect pins action.

2.4.6 Node containers

At the heart of DIWNE is the idea of nodes containing other nodes. This was cited as one of the reasons a custom node editor implementation was chosen in the first place [6, p. 92]. When a node is inserted into a sequence, it is removed from the `NodeEditor`'s list of nodes and moved into the internal list of the sequence, which becomes responsible for drawing the node inside of it.

In many cases, it is required to iterate over all nodes in the editor, including inner nodes and potentially the inner nodes of inner nodes[8]. The original DIWNE implementation used hardcoded if statements to specifically check for nested nodes inside cameras and sequences.

The new implementation generalizes this concept to all nodes using "node containers". A node container is a DiwneObject implementing the `INodeContainer` interface that exposes methods to query which nodes reside inside of it. This is truly only an interface and a basic implementation of the storage and management of nodes is provided by the `NodeContainer` class which any `DiwneObject` can inherit from using multiple inheritance and gain the ability to hold child nodes.

A `DiwneObject` can implement the `INodeContainer` interface directly and provide its own node storage, but this is generally not recommended. Or it can delegate the child node queries to an existing `NodeContainer` instance that is held as a member variable. The `NodeEditor` class itself is an instance of `NodeContainer`. The `Sequence` node only implements the `INodeContainer` interface and delegates its calls to an internal `NodeDropZone` object. This object is a UI component into which nodes can be dragged and dropped, extracting the functionality originally implemented only by the `Sequence` node. The sequence now uses a customized `SequenceDropZone` that is modified to only accept transformation nodes. The `NodeDropZone` is a new component of the DIWNE library into which any node can be placed, including ones with another drop zone, allowing infinite nesting of child nodes.

The `NodeDropZone` is a `DiwneObject` that is meant to be used as part of a `Node`, not on its own. For the DIWNE library example, the `SequenceN-`

[8]The camera, for example, contains sequences that contain transformations

odeContainer node was added to the DIWNE‣Basic package to showcase its functionality.

2.4.7 Node iterators

As mentioned in the previous section, it is quite common that for one reason or another, the nodes of the node editor need to be queried from the outside. Often only a particular subset of nodes is needed. Sometimes nested child nodes should be included, sometimes not. Quite a common query is to, for example, fetch all model nodes in the editor when processing scene view selection or finding models that should be tracked (viz. Chapter 3).

In the old implementation, such queries were satisfied by methods that iterated over the editor's node list, allocating and constructing a new temporary list that is then returned and eventually discarded. When child nodes were to be included, special checks were added to detect sequence nodes, and additionally also iterate over their list of child nodes.

These special checks are eliminated with the use of the `INodeContainer` interface from the previous section. Only a single check whether a node is or is not a node container is sufficient, and all nodes within, no matter the derived node type, can be retrieved using the `INodeContainer::getNodes` method.

Question however remains, what should this method return? To avoid the need to constantly create new temporary lists, a better solution is to use the well-established concept of iterators. As it is intended for DIWNE to be eventually released as a Dear ImGui library, it was decided to target an older version of C++ like C++11, which does not support C++20 Iterator Concepts, so the "legacy" iterators are used[9].

They are implemented in the DIWNE‣Iterators package[10] (see Figure 2.4). In order to implement common iterator functionality without resorting to using virtual methods, the *curiously recurring template pattern*[11] is used to achieve compile-time polymorphism. The `INodeContainer::getNodes` method returns a `NodeRange` object. This is a wrapper around a simple `NodeIterator`, which enables iteration over an `std::vector` of nodes. The `RecursiveNodeRange` and iterator can be used to iterate over all nodes, including all nested child nodes. This iterator detects nodes that implement the `INodeContainer` interface, and steps into their child nodes as if it were traversing a tree. Furthermore, the "filtered" variants of these node ranges can be used to filter the results to only return nodes that fulfill a particular predicate. The recursive node range also takes an optional predicate that decides which nodes the iterator will dive into.

[9]See https://en.cppreference.com/w/cpp/iterator.
[10]Not a package, but the diwne_iterators.h file.
[11]See https://en.cppreference.com/w/cpp/language/crtp.

2.4.8 Layouting

One of the major challenges that the new DIWNE library must resolve is the matter of UI layouts. Specifically, the ability to center content in nodes, right-align pins to the right edge, and right-align buttons and icons in the headers of nodes. Due to how Dear ImGui operates, everything is left-aligned by default, and aligning UI elements requires manual calculations of the remaining space. That often requires assumptions about the content.

This is a very inflexible and bug-prone approach. Dear ImGui does offer some capacity to right-align content, but only within ImGui windows, which the DIWNE nodes aren't, as that would incur significant overhead. A glimpse of hope is the "stack layout" implementation from the creator of the *imgui-node-editor* library [11]. But upon further inspection, it does not quite fit the required use case and doesn't work well within the DIWNE canvas environment, which does not guarantee integer coordinates, which is something Dear ImGui struggles with.

In the end, a custom solution has been developed in the form of the `DiwnePanel` class. This is an independent object that represents a rectangular area, in which "springs" can be placed, which occupy a portion of extra unused space.

The functioning principle is quite simple: the total non-overlapping width and height of all "fixed" elements is summed up during UI construction. After that, a layout manager sets the desired width and height of the panel. Then, the fixed dimensions are compared to the adjusted dimensions to calculate extra space in each axis. Next frame, this extra space is divided up proportionately among the springs, which are freely placed between items (but they must not overlap).

Since springs cannot overlap in the same axis, to achieve several rows of right-aligned items, like is the desired layout for pins on the right side of a node, several `DiwnePanel`s can be arranged in a vertical stack. Each panel begins with a spring that takes up 100% of the available space. The `Stack` layout manager has been added to arrange panels in such a way. Its use in the right pin rendering code can be seen in code listing 2.11.

```
m_vStack.begin();
for (auto const& pin : m_rightPins) {
  if (pin->allowDrawing()) {
    DIWNE::DiwnePanel* row = m_vStack.beginRow();
    row->spring(1.0f);
    pin->drawDiwne(context);
    m_vStack.endRow();
  }
}
m_vStack.end();
```

Code 2.11 A code snippet of the right aligned pin layouting code.

Although this system comes with an inherent delay of one frame, it does work quite nicely. To account for inaccuracies in the size measurements as well as general floating point inaccuracies, a damping system is added that prevents the springs from changing size below a certain threshold. Since the dimensions of the fixed portion of the panel are recorded, instead of the spring size, there should never be a feedback loop causing the springs to expand indefinitely. Occasional issues with pixel oscillations have been observed, but much less often than before.

`DiwnePanel`s can be nested within each other without any special handling. The `BasicNode` implementation uses diwne panels for each of its parts: the header, left panel, center panel, right panel and "middle" panel that centers the "center" panel between the left and right panels. The right panel then contains individual pin panels that right-align the pins with the right edge of the node.

2.5 Workspace improvements

The structure of the workspace implementation from Figure 2.3 has largely been retained, but the classes now rely more heavily on the underlying DIWNE classes, narrowing their focus to I3T. As can be seen in the original architecture overview in Figure 1.3, the individual workspace nodes are implemented in specific subclasses of the original `WorkspaceNodeWithCoreDataWithPins` (now renamed to `CoreNodeWithPins`). Figure 2.6 shows a comparison of the old and new sequence and model node designs.

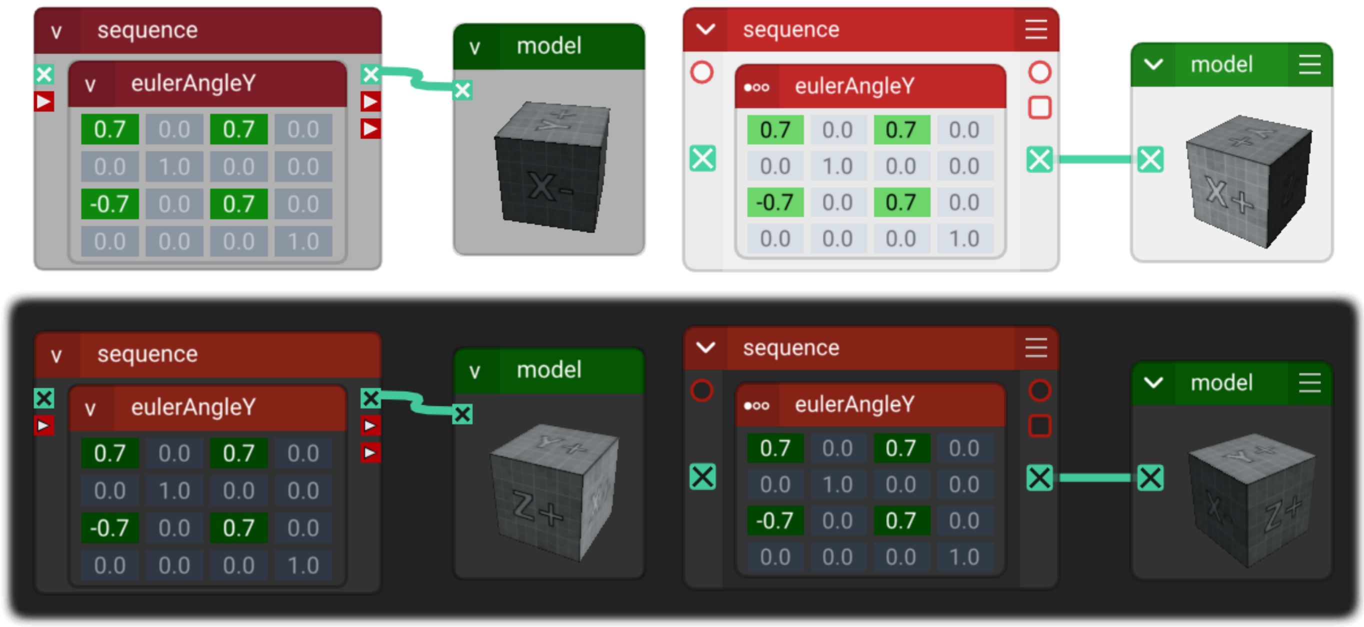
Armed with the new DIWNE library, all of these nodes benefit from the improvements to the input propagation system that has resolved focus issues when interacting with UI elements nested within the nodes (Previously described in Section 2.3.2).

The restriction that nodes could only be dragged by their header was lifted, allowing nodes to be dragged from any part of their body, as long as the drag operation isn't first captured by a number input field, for example. This makes it much easier to drag nodes around or to access their context menu, which can also now be triggered by right-clicking its body, again, assuming a child element doesn't capture the "event" first.

The dragging operation can be disabled for a particular object in a particular area by overriding the additional user specified interaction condition. For dragging, that is the `DiwneObject::allowDragStart` method.

An issue with nodes "shifting" around during the dragging operation has been resolved, as well as various issues stemming from the old "action" system that caused the node editor to freeze when a node was dragged outside of the window.

The node selection system, which now allows shrinking of the selection with the selection rectangle when Ctrl is held. The logic determining which nodes are dragged when multiple are selected has been improved to take child nodes into account, preventing extraction of nodes out of sequences when they are not the directly dragged node.



0.7	0.0	0.7	0.0
0.0	1.0	0.0	0.0
-0.7	0.0	0.7	0.0
0.0	0.0	0.0	1.0

0.7	0.0	0.7	0.0
0.0	1.0	0.0	0.0
-0.7	0.0	0.7	0.0
0.0	0.0	0.0	1.0

0.7	0.0	0.7	0.0
0.0	1.0	0.0	0.0
-0.7	0.0	0.7	0.0
0.0	0.0	0.0	1.0

0.7	0.0	0.7	0.0
0.0	1.0	0.0	0.0
-0.7	0.0	0.7	0.0
0.0	0.0	0.0	1.0

Figure 2.6 Comparison of the old and new sequence and model node design. Both light and dark modes.

2.5.1 Iterative design process

With the DIWNE rework finished and merged back into the development branch of I3T. Work could begin on further improving the existing workspace implementation. As mentioned in Section 1.3.1, concurrently to this thesis, Barbora Hálová has been working on the user research and UX review of the application [7]. Her assignment was to explore and test design changes that would improve the usability of I3T.

This presented a good opportunity for close collaboration between our work, as designs she created could be implemented as part of this thesis, and then tested in the application itself. Avoiding the need for mockup testing. On the other hand, this thesis can focus on the implementation itself and quickly receive feedback.

The first round of testing was performed before the initial new implementation of the DIWNE library was finished and was performed using a paper mockup. The testing focused on the ways transformations are inserted into sequences, context menu layout and connection options for the camera and screen node.

The new designs were then implemented for the second round of testing, which focused on new pin styles and new display mode icons.

The third round of testing further tested changes that were made in response to the second round. But the focus of the test scenes was the new "tracking" functionality which Chapter 3 of this thesis is about. This wasn't a problem as most of the relevant functionality to be tested was not tied to a specific scene.

As there were many individual changes, and some of the major ones are thoroughly described in the thesis by Barbora, including their rationale and detailed descriptions of the testing procedures and results, the following section will mainly summarize some of the key design changes that were implemented, together with a brief overview of their implementation.

2.5.2 Notable design changes

Pin design For a long time, there has been an issue with the pin layout and design. The pins were too small, and there have been issues with their spacing, right alignment and label centering. The new `DiwnePanel` layouts were used to implement a new pin design, ensuring that they are laid out correctly at any pin size.

The visual design has been changed as well, experimenting with four distinct pin styles:

1. Square - The original square pins, but with better hover indicators and rounded corners.
2. Socket - Inspired by the Unity's Shader Graph plugin[12], socket pins are unfilled circles with a shaded background. When hovered or plugged in, a filled inner circle appears to signify the link connection. This style works best with transparent node backgrounds and with the pins fully inside of the node.
3. Square Socket - Same as the socket, but using an unfilled square and a filled rounded inner square.
4. Circle - A very simple style of a filled circle with a dark border. Meant to be placed at the node boundary, sticking out of the node. This is the pin style used by Blender[13].

In the end, it was decided to use the square style for pins representing operations, like the matrix multiplication or cycle node "pulses". While the data pins use the socket pin style. This creates a nice visual distinction between the two, as students often confuse the matrix data pins with matrix multiplication pins.

To further differentiate the matrix multiplication pin, it has been vertically centered in the sequence node. This is done using two springs of the `DiwnePanel` layout class from Section 2.4.8.

The square socket has been used for the special second "model" matrix data output of the sequence node, which outputs a chain of transformations from the scene graph root up to the sequence itself. Tooltips were also added to various pins in order to clarify their function.

[12] https://unity.com/
[13] https://blender.org/

As a new feature, pins support being "offset" outside of the node, placing them over the edge of the node. This required some major adjustments as DIWNE has always assumed that the contents of a `DiwneObject` always remain inside of its rectangle.

During the second round of testing, users were presented with combinations of various styles. Settling on the one described above and shown in Figure 2.6.

Pin drag assist Even though pins were made bigger before the second round of testing to ease the creation of link connections, users were still often struggling. Despite the pins reaching a size that could no longer be reasonably increased. To improve the situation, pins were extended to cover the horizontal gaps between two pins to prevent situations where the node is grabbed instead by this tiny gap.

Furthermore, the spacings around pins were increased, and it was made possible to drag pins not just from the pin icon itself, but from its label as well. This has improved the situation somewhat, but during the third testing round, users still reported that they had occasional issues with dragging the correct pin, especially when moving the mouse quickly.

One user suggested implementing a feature from Blender that allows nodes to be connected by holding Alt and dragging from one node to another. This creates a connection from the first available output to the first available input. This is a nice feature, but it is unlikely to help resolve the issue as it cannot be used to connect specific pins and requires a modifier key to be pressed.

Instead, the *pin drag assist* was introduced. This feature makes every node with pins periodically check the distance of the mouse to its nearest pin. When this distance crosses a threshold, a special flag is set on the pin's `DiwneObject` which forces the `processHoverDiwne` method to trigger the hover state, even though the hover checks fail. This allows a subsequent drag operation to start on the pin, creating circular "drag assist" zones around pins. This makes it much easier to start a connection. Making a connection still requires the mouse to hover directly over the target pin, but that might be implemented in a similar way in the future.

Unfortunately, the pin drag assist has not been tested with users due to time constraints, but it does look very promising.

Pin hover backgrounds and hover indicators During the extension of pins to allow dragging over a larger area that includes the label. Pins were made to render a dark rectangle below them when hovered, to make it easier to recognize when a pin is hovered. This required more adjustments to the pin layout and spacings, to ensure that the object rectangles of pins were truly symmetrical with even spacings.

The pin icon is also made brighter, and the square pin style grows in size a little bit on hover.

Link reconnecting In the previous implementation, links could be unplugged by clicking the input pin, by selecting the link and deleting it with a shortcut, or opening its context menu and selecting "delete". Link selection is problematic as it does not react to the selection rectangle, which is only meant to select nodes. Many node editors in other applications do not support link selection at all, and for simplicity, it has been removed.

Links can now be unplugged by simply dragging the pin to which its end is connected. This unplugs the link and makes its end follow the mouse again. Releasing the mouse deletes the link, but it can also be re-plugged into another pin, which avoids the need to drag the link the entire way from the start pin, which may be far away just to plug it again into a nearby node.

DPI-aware UI scaling The entire application has been made DPI-aware. Meaning the user interface scales itself to be larger on monitors with high pixel density, like laptops. This was a matter of replacing all sizes defined in absolute pixel values with sizes relative to a UI scaling factor itself, or the font size, for example.

The retrieval of the UI scaling factor itself is already handled by Dear ImGui, by querying the state of the GLFW[14] platform windows.

Both I3T and DIWNE use a styling system in which variables can be fetched using global enum keys. Not all of the variables are meant to be scaled by the UI scaling factor, so changes had to be made to include that information. DIWNE already uses a scaling factor in a very similar manner to implement its zooming functionality, so implementing the scaling factor in it was a rather simple matter in principle. Making sure that all parts of the old code take UI scaling into account was the hard part.

Background grid A background grid has been added to aid with navigation inside of the node editor. It can be toggled on and off in the view menu of the top menu bar. A grid of dots is shown by default instead of lines, which is a common modern design used in node editors. A grid of faint dots still provides clues about the relative movement and position of the node editor's viewport, while being less distracting. The inspiration for this design came from Blender and the node editor of Embergen[15].

Node icons and context menu button Using the new layout system, it is easy to right align content in the node header. Allowing the realization of Barbora's design that adds a button to the top-right corner of particular nodes that contains a burger menu icon, opening the context menu on click. This was done to ensure that users are aware that nodes have hidden options in their context menus.

Only some notable nodes have this button, as making every operator and transformation node have it would introduce a lot of visual clutter. Although

[14]GLFW is a C++ utility library for handling windows, OpenGL context and inputs.
[15]Volumetric simulation software https://jangafx.com/software/embergen

some users noted that it is confusing that some nodes lack the button, but still have a context menu. Ultimately, the button serves its purpose of making users aware there are context menus in the first place.

The node header right align is also used to place the matrix validity icon in the top right corner, as well as the tracking and reference frame indicators introduced later in Chapter 3.

The top left corner of all nodes contains a “display mode” button, that usually shows an arrow that can be used to collapse and expand the node. This arrow used to be a letter, but now uses a proper icon font. The icons of nodes with more than two states have been changed for a set of three dots, which was a design proposal by Barbora [7].

Sequence drop zone In order to clearly indicate that transformation nodes can be dragged and placed into sequences. The `DIWNE::NodeDropZone` component has been made to have a slightly visible background, denoting the area into which nodes can be dragged. Since this is an absolutely key feature of I3T, a clearly visible label indicating that transformation nodes are to be placed here is displayed when the sequence is empty.

This was originally implemented as it was the easiest solution for the time being. Alternative designs by Barbora proposed a few kinds of clickable icons instead, but during the first round of paper mockup testing, users agreed that a simple text label is sufficiently clear.

Chapter 3

Tracking

This chapter expands on the established concept of *"tracking"* inside I3T, which is a feature that visualizes the gradual application of individual transformations inside sequences onto a model, or, in the other direction, the progressive changes each transformation applies to the basis vectors of a vector space. Essentially, instead of viewing the final effect of a single composite matrix on a model, the piecewise effects of each transformation are shown, accumulating into an interpolated matrix by multiplying each matrix along the chain from the left, or from the right.

This feature, available in I3T from its very inception [1, p. 40] visualizes the transformation of loaded model vertices[1] into their desired position in world space. But when viewing the world in a 3D application, that is not quite the full story, as what follows are transformations of the "imaginary" camera, which then transform the world space models into their final position on the screen. This entire journey vertices undertake in the graphics pipeline is captured in Figure 3.1 and notably the current tracking implementation only operates within the "model" matrix shown in the figure, that transforms vertices from the model local space to their world space position.

The remaining transformations, that transform world space vertices into a form that is eventually rasterized and displayed on the screen, are all represented in I3T by a camera node. This node is implicitly connected to the left matrix multiplication pin of all sequences that would be otherwise disconnected, placing the camera at the left end of every transformation chain. This connection is however purely visual and cannot be modified by the user, indicating that every model in the scene can be seen by the camera.

The camera contains two child sequences inside of it, which are used to specify the matrices of the view matrix and projection matrix transformations. These matrices are used to determine its position in the world and to also display the camera's camera frustum. Its primary output is a screen pin, which can be connected to a screen node, that will display what the scene looks like,

[1]The local space of the model

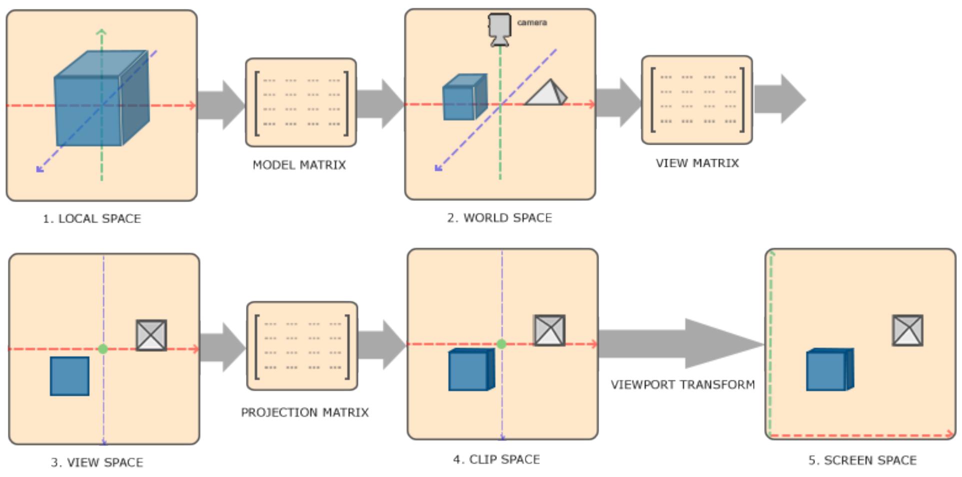


Figure 3.1 Sequence of OpenGL coordinate systems that model vertices go through before they can be rasterized on the screen. Image sourced from LearnOpenGL [12].

if rendered using the camera's current view and projection matrices. A basic scene containing a camera and displaying what that camera can see in a screen node can be seen in Figure 3.2.

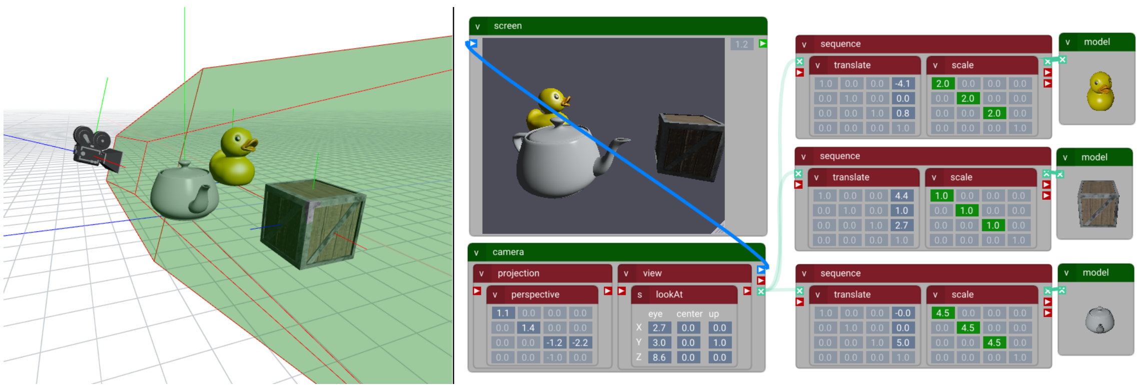
Tracking can begin in any regular sequence, but despite the camera being connected to every sequence, the sequences inside of it cannot be tracked, which prompts the entire premise of this chapter, and the realization of the second goal of this thesis: **extending the concept of tracking to include the camera**, bridging the existing gap between world and screen space coordinates and using the existing tracking mechanism to progressively visualize the effects of the *view*, *projection* and *viewport* transformations.

This can prove quite useful as beginning with the *projection* transformation, what truly happens is often perceived by students as a somewhat abstract process, mainly due to the nature of the perspective projection transformation, which is not an affine transformation like most others, and uses the arguably non-intuitive[2] homogeneous coordinates to achieve a non-linear mapping of space.

A lot of these key concepts, like the perspective divide, primitive clipping as well as *viewport* mapping are performed opaquely as part of OpenGL's fixed function vertex post-processing pipeline [13, p. 441], which makes learning or imagining these concepts more difficult.

The goal of I3T has largely been to visualize complex and hard to imagine concepts in a visual and interactive manner, yet these very important concepts of the camera are still hidden and performed under the hood inside of the camera node. The need for such visualization has even been noted in the discussion of the original thesis by Michal Folta in which I3T was created [1, p. 60].

[2]read — four dimensional



perspective

1.1	0.0	0.0	0.0
0.0	1.4	0.0	0.0
0.0	0.0	-1.2	-2.2
0.0	0.0	-1.0	0.0

lookAt

	eye	center	up
X	2.7	0.0	0.0
Y	3.0	0.0	1.0
Z	8.6	0.0	0.0

translate

1.0	0.0	0.0	-4.1
0.0	1.0	0.0	0.0
0.0	0.0	1.0	0.8
0.0	0.0	0.0	1.0

scale

2.0	0.0	0.0	0.0
0.0	2.0	0.0	0.0
0.0	0.0	2.0	0.0
0.0	0.0	0.0	1.0

translate

1.0	0.0	0.0	4.4
0.0	1.0	0.0	1.0
0.0	0.0	1.0	2.7
0.0	0.0	0.0	1.0

scale

1.0	0.0	0.0	0.0
0.0	1.0	0.0	0.0
0.0	0.0	1.0	0.0
0.0	0.0	0.0	1.0

translate

1.0	0.0	0.0	-0.0
0.0	1.0	0.0	0.0
0.0	0.0	1.0	5.0
0.0	0.0	0.0	1.0

scale

4.5	0.0	0.0	0.0
0.0	4.5	0.0	0.0
0.0	0.0	4.5	0.0
0.0	0.0	0.0	1.0

Figure 3.2 Camera viewing a simple scene inside of old I3T.

3.1 Analysis

The following sections will analyse how this could be achieved, starting with a more detailed description of the current tracking mechanism, as it needs to be reviewed and prepared for further extension. This has largely been made easier now that the I3T *workspace* user interface has been reworked in Chapter 2.

3.1.1 Standard tracking

As stated, tracking can be used to view the effects of each individual matrix along a chain of transformations. This is a very clear visualization of what role each transformation fulfills. The entire tracking operation spans a chain of transformations from some beginning sequence up to the root of the connected sequence graph, ending at the first sequence with an unplugged matrix multiplication input. The root sequence can have more than one matrix multiplication output connection, serving as a root for many different sequences, but tracking always works with a linear chain, a path of sequences from the root sequence to the one that tracking was started from (the *begin* sequence).

To indicate tracking progress along the path, a *cursor* is shown in the *workspace* as a thick yellow vertical line that moves horizontally across transformation nodes inside sequences on the chain. As the chain is always linear, there is always just one cursor.

The cursor splits the chain into the active and inactive sections. In right-to-left tracking, transformations to the left of the cursor are ignored. Transformations to the right of the cursor are multiplied together (accumulated) and then multiplied from the left with the transformation the cursor is currently inside of. But before doing so, this transformation is first interpolated depending on the progress of the cursor within it. This is done with simple element-wise linear interpolation between an identity matrix and the matrix of the transformation. Or, in case it is known the transformation represents a rotation,

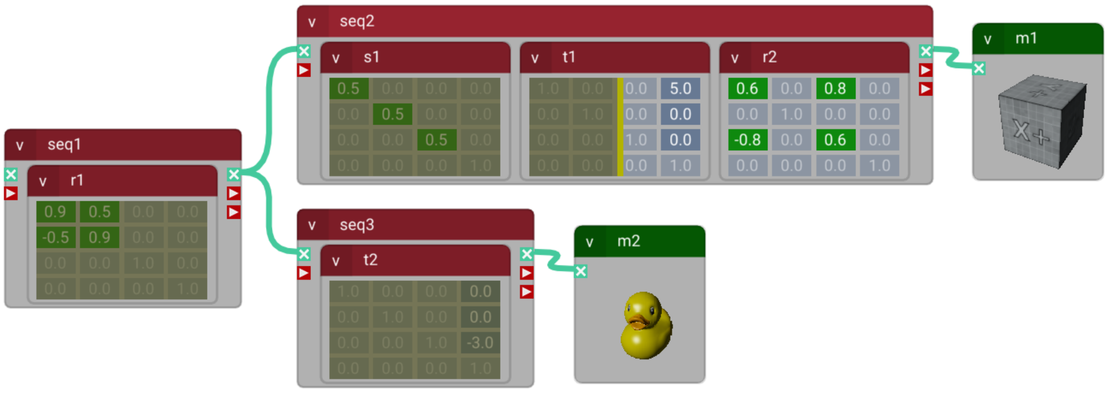


Figure 3.3 Example of tracking across multiple sequences in the old I3T. It was started from the right of sequence *seq2* and creates a chain of matrices from that sequence up to the root sequence *seq1*, displaying the effects on model *m1*.

the top left rotation matrix is converted to a quaternion and interpolated using spherical linear interpolation, to ensure the basis vectors remain orthonormal and interpolation looks natural[3].

As an example, consider the situation shown in Figure 3.3. Right-to-left tracking has been started on sequence *seq2* and the cursor is located halfway inside the *t1* translate transformation. As tracking began in *seq2*, only model *m1* is being tracked. The entire transformation chain consists of 4 matrices:

$$\mathbf{R_1 S_1 T_1 R_2} \tag{3.1}$$

which together form the model matrix of the model *m1*. During tracking, a copy of the model is created and only the $\mathbf{R_2}$ and $\mathbf{T_1}$ matrices are applied to it. Furthermore, since the cursor is only halfway through $\mathbf{T_1}$, the final interpolated matrix applied to the tracked model is $\frac{1}{2}\mathbf{T_1 R_2}$.

Tracking progress can be controlled by the user by using arrow keys to move the cursor freely to either end of the transformation chain and in any direction. In this case, the active part of the chain will always be to the right of the cursor.

Tracking can also be started “from the left” (left-to-right), which would change the situation in Figure 3.3 so that the resulting interpolated matrix would be $\mathbf{R_1 S_1} \frac{1}{2}\mathbf{T_1}$. The difference being that the matrices along the chain are accumulated from the left instead, which changes the interpretation of how the transformations are progressively combined together to achieve the final transformation, which in either case is the same. It can be thought of as two ways to insert parentheses around matrix multiplications in equation 3.1, applying

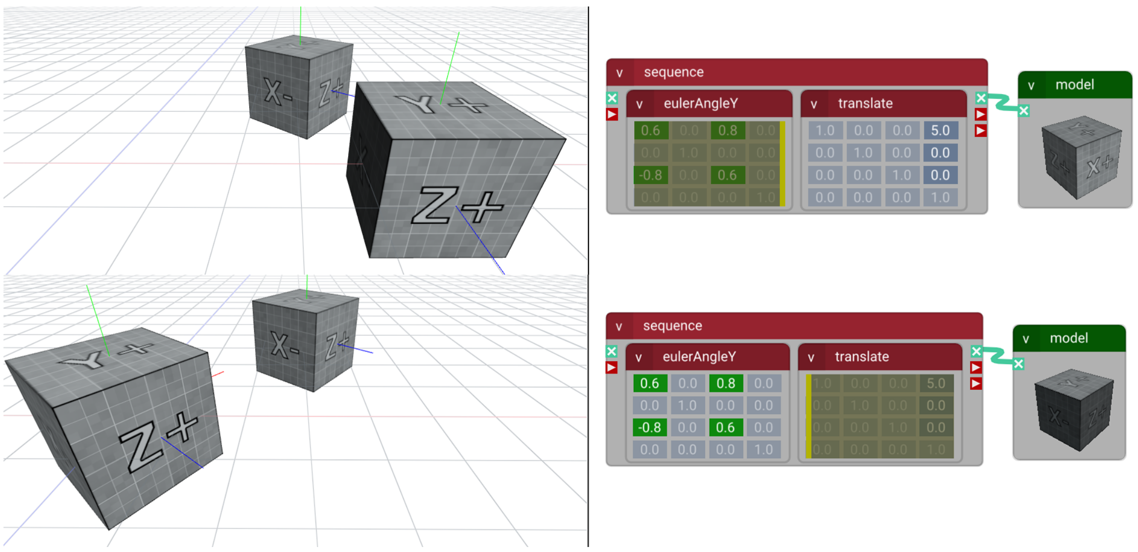
[3]In this case, the translation part of the matrix is interpolated separately using linear interpolation, and the last row is ignored.

multiplication always from the left or right, giving different intermediary results but the same final matrix, as can be seen in equation 3.2:

$$(((\mathbf{R}_1\mathbf{S}_1)\mathbf{T}_1)\mathbf{R}_2) = (\mathbf{R}_1(\mathbf{S}_1(\mathbf{T}_1\mathbf{R}_2))) \tag{3.2}$$

Figure 3.4 compares the two modes: in the first, the cube is translated and then rotated around the world origin. In the other, the rotation is applied first, and then the entire vector space is translated along its local x axis. Each mode reaches the same destination, but the imaginary path the model takes to get there is different.

Note that the direction the cursor is moving in during the actual tracking operation just changes progress along this path, right-to-left or left-to-right tracking does not refer to the movement of the cursor, but rather to the side of the chain which is being accumulated.



eulerAngleY

0.6	0.0	0.8	0.0
0.0	1.0	0.0	0.0
-0.8	0.0	0.6	0.0
0.0	0.0	0.0	1.0

translate

1.0	0.0	0.0	5.0
0.0	1.0	0.0	0.0
0.0	0.0	1.0	0.0
0.0	0.0	0.0	1.0

Figure 3.4 Example of tracking a pair of translation and rotation transformations from the left and from the right in the old I3T.

3.1.2 Camera tracking

Extending tracking beyond world space is, in principle, a simple task. What occurs inside of a camera with a standard "look at" transformation as the view matrix, and an orthographic projection can easily be visualized using the existing tracking mechanism of I3T, by adding these transformations not to the camera node, but into a standard sequence like any other world space transformation and connecting it to an existing transformation chain from the left.

Taking the transformation chain from equation 3.1 (denoted as $\mathbf{M}$), we can expand it with a view matrix $\mathbf{V}$ and projection matrix $\mathbf{P}$ to get:

$$\mathbf{PV} \cdot \mathbf{R}_1\mathbf{S}_1\mathbf{T}_1\mathbf{R}_2 = \mathbf{PVM} \tag{3.3}$$

This allows us to track through the entire **PVM**[4] matrix representing transformation from local model space all the way to clip space, which is equivalent to normalized device coordinates when using a simple orthographic projection, placing the model vertices right before the viewport transformation in Figure 3.1.

Putting all of this together, a custom scene can be created in I3T to demonstrate this concept first using orthographic projection. The view matrix can be interpolated like any other affine transformation, placing models within the bounds of the viewing volume, positioning the camera at the world origin, and the orthographic projection is affine as well and can be trivially interpolated[5].

Such a scene can be seen in Figure 3.5, where a single model of a larger scene is tracked to the normalized device coordinates, transforming the cuboid orthographic frustum into a unit cube around the world origin.

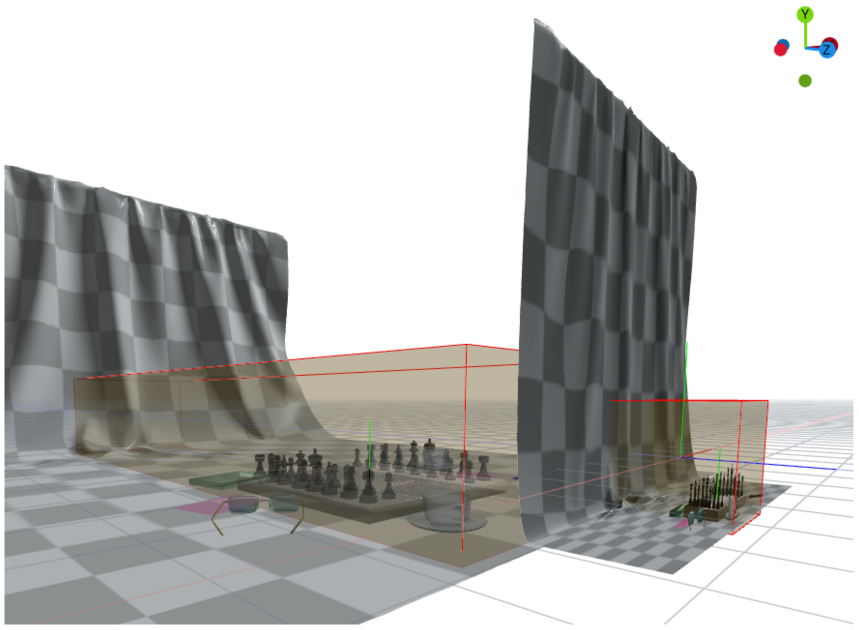


Figure 3.5 Proof of concept visualization of orthographic projection inside of the old I3T using standard tracking. The elongated rectangular cuboid viewing volume is transformed into the NDC "cube", note that depth values are transformed linearly, preserving the repeating checker pattern.

This proof of concept has some issues. The camera frustums visible in the figure are manually placed in their respective positions, as only the camera node is able to visualize them. For the same reason, the frustum itself is not animated.

[4]The well known projection-view-model matrix commonly passed to or computed in OpenGL shaders.

[5]Using element-wise linear interpolation, as it does not contain a rotation.

Another major issue is that the real OpenGL orthographic projection (formulated in Figure 3.6b) transforms points into a left-handed coordinate system that has the Z axis inverted, but the I3T world is right-handed and cameras view it along the negative Z axis. This means that if the orthographic matrix is left unmodified (i.e. the orthographic transformation node is used), the transformation physically flips the entire frustum along the Z axis during its gradual tracking and the resulting NDC space is inverted on the X axis. The world X axis also does not correspond to the NDC space anymore, as the world the NDC is displayed in is still right-handed.

This means that special handling is required if the visualization is to be clear and concise. It is a good idea to also modify the orthographic matrix, preventing the Z flip as it distracts from the main concept of reshaping the frustum into the NDC "cube". A simple solution is to apply a "negation" matrix from the left that flips the sign of the third row of the orthographic matrix, and add another stage of interpolation that performs the Z flip.

Despite those problems, the concept works, and the entire transformation can be viewed as an animation. The viewport transformation can further be added to the left of this chain by manually constructing a matrix emulating the transformation OpenGL does as part of its fixed function pipeline [13, p. 441]. Such a matrix can be seen in Figure 3.6a.

$$\begin{pmatrix} \frac{w}{2} & 0 & 0 & x+\frac{w}{2} \\ 0 & \frac{h}{2} & 0 & y+\frac{h}{2} \\ 0 & 0 & \frac{f-n}{2} & \frac{f+n}{2} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

(a) Viewport

$$\begin{pmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & \frac{-2}{f-n} & -\frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

(b) Orthographic

$$\begin{pmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

(c) Perspective

Figure 3.6 (a) A transformation matrix equivalent to the default OpenGL viewport transformation [13, p. 458]. **(b)** The usual OpenGL orthographic projection matrix [14, p. 153]. **(c)** The usual OpenGL perspective projection matrix [14, p. 143].[6]

The process described above clearly calls for a solution that handles these issues automatically, without the need for a specialized node setup in a custom scene. The camera node is the perfect fit for this as it is already separated into two sequences for the view and projection transformation and has control over them. It can even be expanded with a third sequence for the viewport transformation, which could be populated with a new viewport transformation node,

[6]The projection matrices are equivalent to what is generated by the OpenGL Mathematics (GLM) library in its default configuration. These matrices can have variations as they are not explicitly defined by the OpenGL standard.

that can use the established “set values” mode to accept the same parameters as the OpenGL’s `glViewport` API calls.

Tracking can be expanded to be aware of which type of transformation it is currently interpolating over and thus the coordinate system the user is viewing. Using this knowledge, the 3D scene view can take measures to display the particular coordinate system correctly, physically switching into another coordinate system or merely emulating a different coordinate system using additional visual indicators.

3.1.3 Perspective projection visualization

One key aspect is still not addressed: how to visualize the perspective projection. Various educational materials describing the perspective projection are often accompanied by some sort of visualization, which typically consists of a static image displaying the shape of the projection frustum[7] prior to applying the transformation in view space, or the bounds of the resulting normalized device coordinates space[8]. Rarely are the effects of the transformation displayed on non-trivial geometry within the frustum itself. It is even harder to find an animation showing such an effect.

Displaying the perspective transformation in an interactive manner in a scene consisting of regular textured models is a good visualization of the non-linear effect it has on the depth values of the geometry, which is a concept that many students struggle to understand. Such a visualization offers a more intuitive insight, compared to, for example, showing the depth value relationship in a graph.

An example of such a visualization that this chapter is aiming to achieve can be seen in a video by Josh’s channel on YouTube [16], which served as a major inspiration for exploring the entire topic of camera tracking within I3T. A screenshot from the video can be seen in Figure 3.7, which shows the rendering output of a camera viewing a room in a 3D game on the left[9]. On the right, a visualization of the NDC space that produced such a rendering can be seen, displaying the heavy distortion of world geometry that produces the perspective effect.

The right side of Figure 3.7 is exactly what has been done above with orthographic projection in Figure 3.5. Achieving this effect in I3T would effectively allow the user to “zoom out” of a resulting perspective image and view the NDC space with a second independent camera, which is exactly what I3T does in the example above.

The user-controlled camera which displays the scene view in I3T does in reality add two more matrices to the left end of any transformation chain inside of the workspace, like the one presented in equation 3.3. This is a somewhat obvious fact that is not worth mentioning when working with “world space”

[7] A quadrilateral truncated pyramid

[8] As can for example be seen at page 148 and 151 of [14] and page 101 of [15].

[9] It is the Pillager Outpost tower from Minecraft.

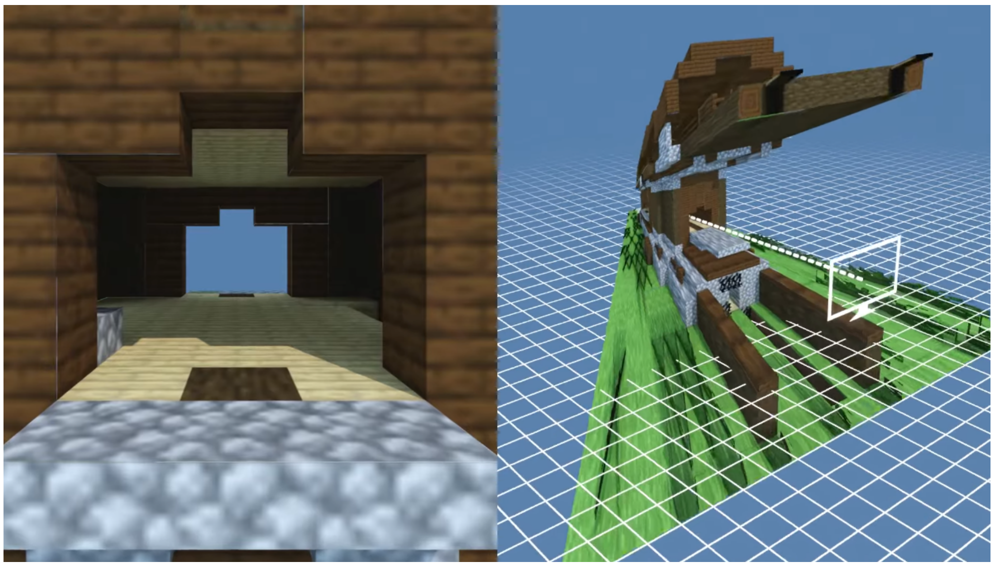


Figure 3.7 Screenshot of the YouTube video by Josh's channel [16] that inspired camera tracking. The effect is equivalent to the one shown in Figure 3.8.

coordinates, that always have their 4th w equal to 1. Although points transformed by an affine view and orthographic projection matrix are not in theory in world space anymore, for the purposes of displaying them in I3T, they can still be considered to be in the world space of the scene view.

However, when replacing the projection matrix $\mathbf{P}$ with a perspective projection matrix, the situation changes and it is worth looking at the full picture. Adding the independent scene view camera's own view ($\mathbf{V_s}$) and perspective projection matrices ($\mathbf{P_s}$), equation 3.3 becomes:

$$\mathbf{P_sV_s} \cdot \mathbf{PV} \cdot \mathbf{R_1S_1T_1R_2} = \mathbf{P_sV_sPVM} \tag{3.4}$$

This is a somewhat unusual situation where a second set of view and projection matrices is applied to previously projected points that have their depth already encoded in their w component by the first projection matrix. This is not discussed much in computer graphics literature as it is usually assumed that projection is performed only once, so it is not immediately clear that the mechanism of homogeneous coordinates continues to function.

It turns out that it does and this is not an obstacle at all, as the projection can in fact be applied any number of times and does not require an immediate application of the perspective divide. This stems from the properties of homogeneous coordinates [14, p. 152] and is at least partially proven in Appendix A that shows that applying the perspective divide immediately after each application of the perspective projection matrix yields the same results as applying the perspective divide once at the end.

This means that no special handling of the perspective divide is needed, and it can be done just once as part of the usual OpenGL fixed function pipeline, a fact that was not initially obvious[10].

Unlike the orthographic projection, perspective is not an affine transformation, and if the perspective divide is considered a part of it, it is not even a *linear* transformation. That would perhaps indicate that special handling is at least needed to interpolate it, but that also turns out to be false.

As can be seen in Figure 3.6c, ignoring the fourth row, the perspective matrix simply consists of non-uniform scaling on the diagonal, and a translation on the Z axis, which can both be interpolated with element-wise linear interpolation. The third component of the fourth row, that moves the z value to the w component of the resulting column vector, can be interpolated in the same way, strengthening or weakening the effect of perspective.

Since no special handling is needed, the desired perspective projection visualization can be achieved in the exact same way as in the orthographic projection example. Result of which is shown in Figure 3.8.

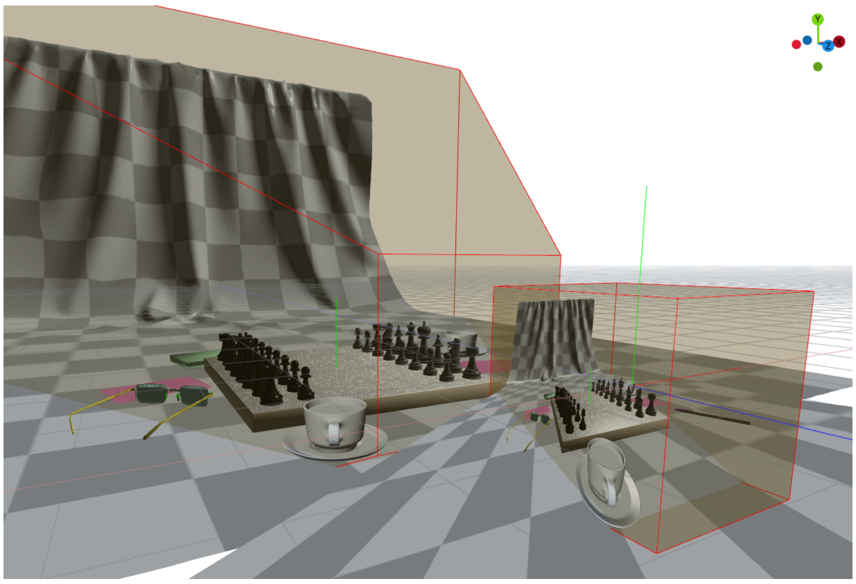


Figure 3.8 Proof of concept visualization of perspective projection inside of the old I3T using standard tracking. The volume of the camera frustum is gradually transformed into the NDC “cube” at world origin. The complex scene geometry is distorted accordingly, showing the non-linear effect perspective has on the depth values.

It should be noted that this method does not visualize the intermediary clip space, which is somewhat difficult to interpret as it is a four-dimensional space containing points with varying w components. This is not addressed further in this text.

Another notable detail is that the tracking of the camera is always considered to be a right-to-left tracking operation. Left-to-right tracking inside the camera

[10]Not to me at least.

shall be forbidden, as applying the projection transformation directly to local model coordinates does not convey much meaning, nor does it result in a clear visual. The model vertices will rarely be initially positioned within the viewing volume, making the visualization display largely unrecognizable results until the model is placed into an appropriate position in world space.

3.1.4 Reference frame

The presented camera tracking proof of concepts have one last major issue. Tracking was always conceptually understood to operate within the scene view world, changing model transformations to move objects through this single world's space. Each tracking operation also originally only affected a single model, moving it through the world while everything else remained stationary.

With the addition of camera tracking, this model-centric concept of tracking becomes somewhat problematic as there can always be models that are unaffected, which can become confusing.

For example, consider the view transformation of a camera being applied to a set of models during camera tracking. This can be understood as transforming the models into a different coordinate space of the camera, but since there are other objects in the world that remain unaffected by the tracking operation, what is seen is essentially a combination of both the view space and the original world space. Continuing the tracking operation into NDC, the tracked set of models is deformed and positioned near the origin of the world, while other models are still in their original position and can be quite large by comparison, often obscuring the tracked models.

Figure 3.9 shows the tracking operation within a scene with three models, the duck is transformed into the NDC space of a nearby camera and is shrunk down significantly and deformed by the perspective. As cameras are understood to always "see" the whole world, it would be much better if the tracking operation affected the whole scene in this case, avoiding the issues mentioned above.

The solution is to make the 3D scene view aware of the *reference frame* or *space* it operates in. Nothing would change when tracking transformations in regular sequences, but when tracking the camera, the model transformations of tracked models would not be modified anymore, instead, the interpolated matrices of the camera would be used to construct a special *reference frame matrix*, which the 3D scene view would apply globally to every model in the scene.

The scene view could also be made aware of what kind of space it is meant to represent, making it possible to add custom visual indicators to the scene view interface, like a second set of axis indicators[11], displaying the "original" world space axes, which can no longer be parallel with the formerly world space grid, or even orthogonal to each other.

[11]The existing indicator can be seen in the top right corner of Figure 3.8. Implemented previously as part of [3].

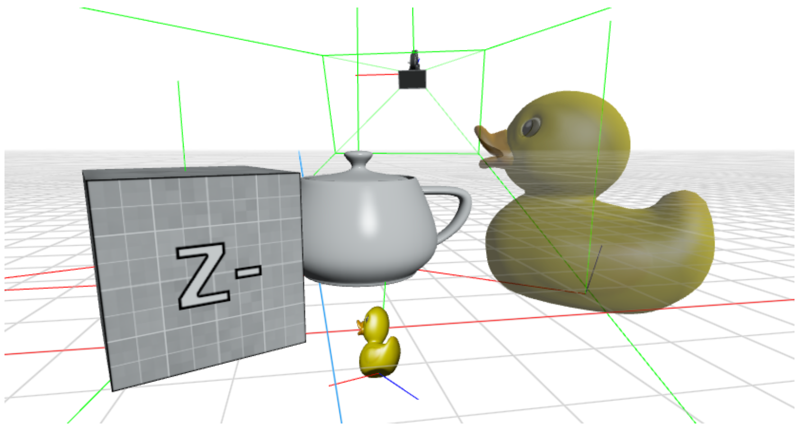


Figure 3.9 A duck model being tracked into the NDC space of a camera, leaving other models stationary in world space.

The infinite world grid[12], would no longer represent the world axes, but rather the local axes of the reference frame. It is, of course, still the same grid, but what it represents changes. The original conceptual world grid, could also be shown, visualizing how the reference frame is positioned in relation to the world space. This would require extension of the infinite grid implementation to allow for rendering of a generalized non axis-aligned grid [3, p. 58].

This new feature of a scene view reference frame could prove to be very useful outside of the camera tracking itself. The reference frame matrix can be set to anything, like a model matrix output of a particular sequence, allowing the user to view the world from the reference frame of any chain of transformations, potentially giving additional insight to how regular world space transformations operate.

An example of such use can be seen in Figure 3.10 which shows a 2D world with a cube model, transformed by a composition of translation and rotation. The unit cube is translated along the red X axis by 5 units (grid squares) and then rotated by 30 degrees counter-clockwise, placing it at a world position that can be seen on the left side of the figure.

On the right side, the world is viewed from the reference frame of the sequence containing this transformation, the original world grid and the cube are transformed by the inverse of the cube's model transformation, placing it at the centre of the coordinate system, which is signified by the axis-aligned local space grid in yellow.

From an implementation standpoint, the 3D scene view must be expanded to insert the additional reference frame matrix $\mathbf{R}$ into the transformation chain of every model. In the case of camera tracking examined in equation 3.4, the reference frame matrix replaces the camera's projection ($\mathbf{P}$) and view ($\mathbf{V}$) matrices, applying them to the entire world so the transformation chain becomes:

[12]Introduced and described in [3].

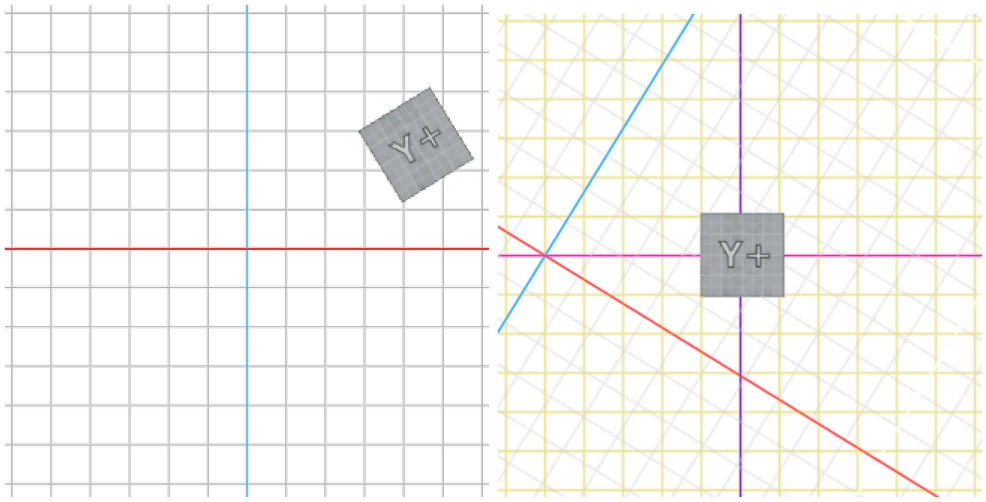


Figure 3.10 A translated and rotated cube viewed in world space (left) and viewed from the reference frame of the inverse of its transformation.

$$\mathbf{P_s V_s} \cdot \mathbf{R} \cdot \mathbf{R_1 S_1 T_1 R_2} = \mathbf{P_s V_s R M} \tag{3.5}$$

3.2 Current implementation

Now that tracking has been introduced, and the concept of camera tracking explored, this section will give a quick overview of the current tracking implementation. The next section will address how this implementation can be expanded to make camera tracking a reality. As tracking is at the heart of its functionality, some issues with its current implementation must be resolved first.

3.2.1 Functionality and user interface

The tracking operation can be started from the context menu of every sequence node in the workspace. It contains the options to “start tracking from the right” and “start tracking from the left”. Despite it being available for every sequence, the tracking operation will not start upon activating either of these options unless the sequence’s matrix multiplication pin is **directly** connected to a model. If multiple models are connected, they will all get tracked, as long as there isn’t another sequence along the connection.

Activating the tracking operation from a different sequence (including the sequences in the camera) will fail. This is one of the major issues with the current implementation as it is impossible to track multiple objects that are connected to the sequence indirectly, meaning the sequence can be connected to a model through another sequence connected to its output matrix multiplication pin. So ideally, the entire subtree, rooted in the sequence from which tracking was activated, should be considered to be a single composite model instead.

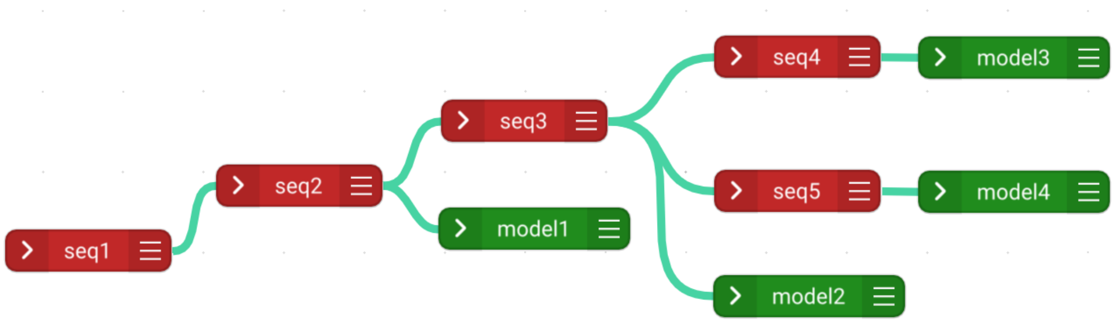


Figure 3.11 Example of branching sequences connected to multiple models (the new node design with collapsed nodes from Chapter 2 is shown)

Once activated, the tracking operation then constructs a transformation chain by traversing the tree of connected sequences, starting from the *begin*[13] sequence, up to the root of the graph. In Figure 3.11, starting tracking from sequence four ("seq3") would construct a chain containing sequences one, two and three. The current tracking implementation would only track model two, but the entire subtree rooted in sequence three could be tracked instead, so that models three and four are also part of the tracking operation, as if transformations inside sequences four and five were part of their local model coordinates.

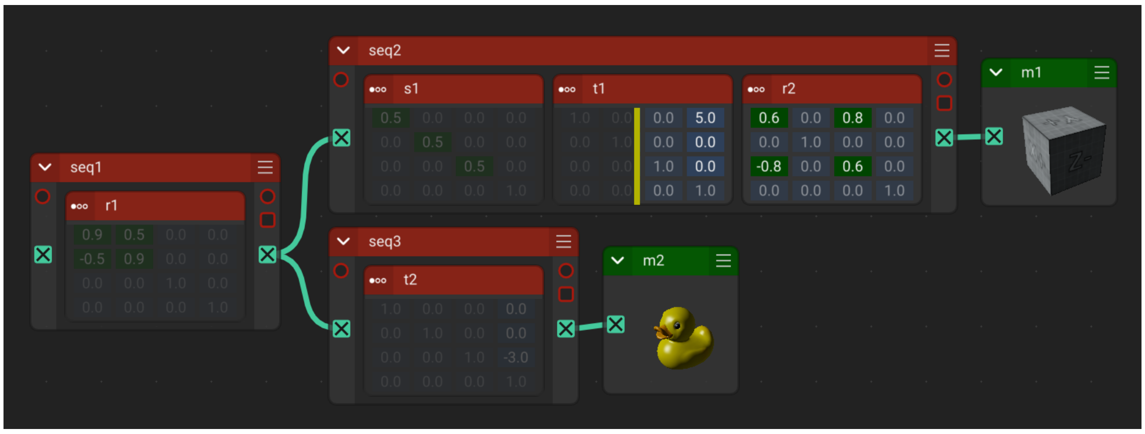
In this context, in this text, sequences one, two and three are considered to be the *tracked chain*, and the entire subtree of sequence three will be referred to as the *model subtree*.

An alternative approach would be to expand the tracked transformation chain to the whole subtree connected to sequence three, but that would turn a linear path into a tree, requiring the tracking cursor to "split" into multiple cursors when passing from sequence three to sequences four and five. This would significantly complicate the design and its implementation, for little additional benefit as the transformations of a model in the subtree can be tracked in their entirety by starting the tracking operation directly from the sequence connected to it.

There is also room for improvement in the user interface and visual indicators of tracking. Figure 3.12 shows the tracking example shown previously in Figure 3.3, but using the updated workspace design. Right-to-left tracking has been started on sequence two. The only visual indicator is the yellow cursor shown in transformation *t1* and an overlay is drawn over every transformation that is not part of the interpolated matrix. There is no other indication of tracking and no way to control it with the mouse as it can only be controlled by the arrow keys on the keyboard.

[13]The sequence the tracking operation was started from.

In the scene view, a copy of model *m1* is shown, with the interpolated transformation applied to it, but it is visually indistinguishable from the original model.



r1

0.9	0.5	0.0	0.0
-0.5	0.9	0.0	0.0
0.0	0.0	1.0	0.0
0.0	0.0	0.0	1.0

s1

0.5	0.0	0.0	0.0
0.0	0.5	0.0	0.0
0.0	0.0	0.5	0.0
0.0	0.0	0.0	1.0

t1

1.0	0.0	0.0	5.0
0.0	1.0	0.0	0.0
0.0	0.0	1.0	0.0
0.0	0.0	0.0	1.0

r2

0.6	0.0	0.8	0.0
0.0	1.0	0.0	0.0
-0.8	0.0	0.6	0.0
0.0	0.0	0.0	1.0

t2

1.0	0.0	0.0	0.0
0.0	1.0	0.0	0.0
0.0	0.0	1.0	-3.0
0.0	0.0	0.0	1.0

Figure 3.12 The tracking example from Figure 3.3 shown after the workspace redesign in chapter 2 (using dark mode as that was the only one available at the time).

To summarize, from a visual and functionality aspect, the most notable issues are as follows:

1. No visual indicator of the tracked nodes making up the chain as all nodes in the workspace are dimmed with an overlay, not just the ones forming the tracked transformation chain.
2. Aside from dimming all transformations and showing the tracking cursor, there is no proper indication that tracking is active. The progress and direction of tracking can also be only determined by inspecting each node along the chain, which again, is not highlighted in any way.
3. Tracking can only be controlled with the arrow keys, there is no way to control it with the mouse and no indication that arrow keys must be used.
4. Tracked models must be directly connected to the *begin* sequence. Other models connected indirectly will not be tracked, meaning the tracking operation often cannot be started from the "root" or "inner" sequences, but only from the "leafs", which are usually connected to models directly.

3.2.2 Codebase

From a codebase perspective, there is also much left to be desired, the bulk of the tracking logic is implemented in the Core ▸ Nodes ▸ Tracking.h[14] file that contains the main `MatrixTracker` class. It also contains some other auxiliary classes,

[14]See architecture of the original I3T in Figure 1.3.

as can be seen in Figure 3.13 displaying a class diagram of the relevant parts of the Core▸Nodes directory. Note that tracking operates with the underlying logical *core* node graph implementation, which is distinct from the UI *workspace* or DIWNE nodes.

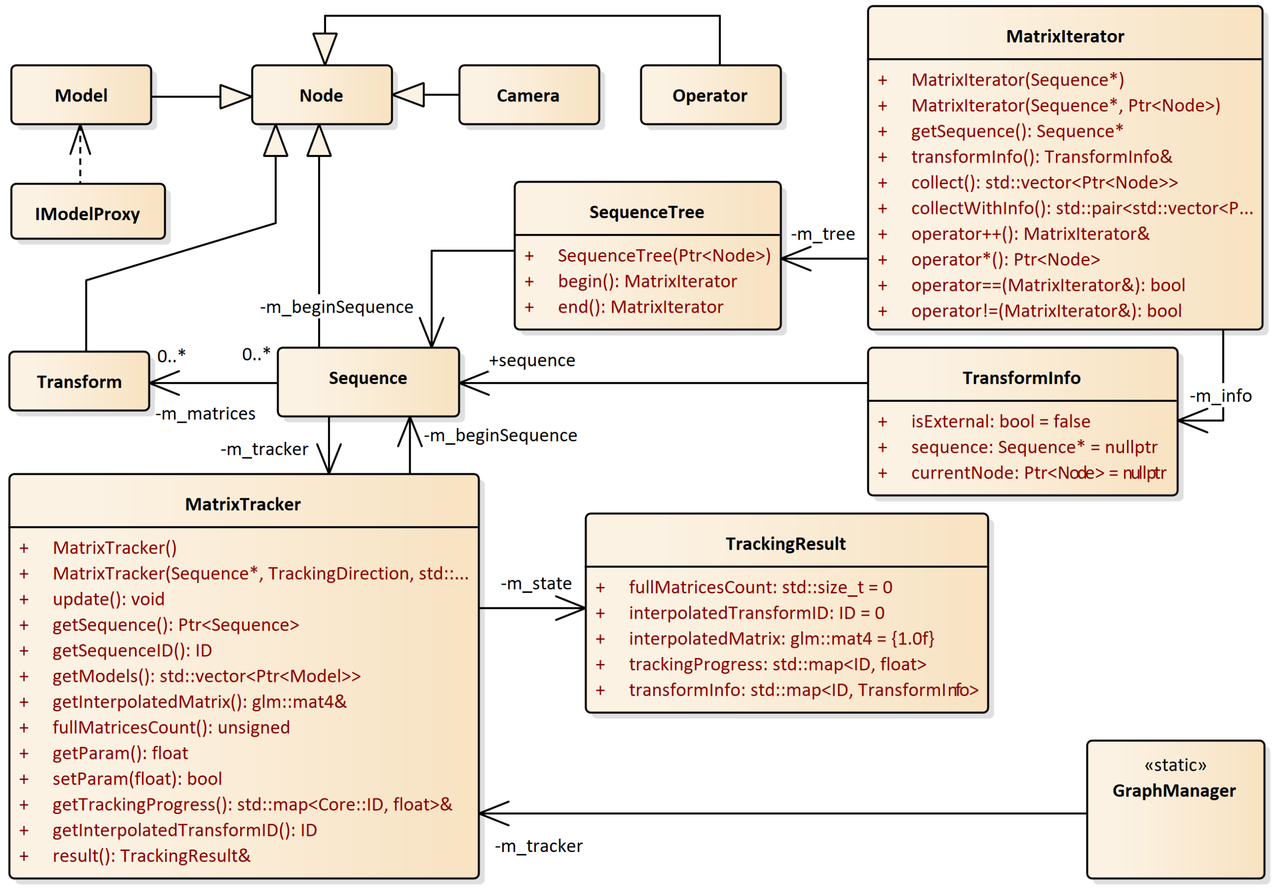

Model
Node
Camera
Operator
IModelProxy
MatrixIterator
+ MatrixIterator(Sequence*)
+ MatrixIterator(Sequence*, Ptr<Node>)
+ getSequence(): Sequence*
+ transformInfo(): TransformInfo&
+ collect(): std::vector<Ptr<Node>>
+ collectWithInfo(): std::pair<std::vector<P...
+ operator++(): MatrixIterator&
+ operator*(): Ptr<Node>
+ operator==(MatrixIterator&): bool
+ operator!=(MatrixIterator&): bool
SequenceTree
+ SequenceTree(Ptr<Node>)
+ begin(): MatrixIterator
+ end(): MatrixIterator
-m_tree
-m_beginSequence
Transform
0..*
0..*
Sequence
+sequence
-m_matrices
-m_tracker
-m_beginSequence
TransformInfo
+ isExternal: bool = false
+ sequence: Sequence* = nullptr
+ currentNode: Ptr<Node> = nullptr
-m_info
MatrixTracker
+ MatrixTracker()
+ MatrixTracker(Sequence*, TrackingDirection, std::...
+ update(): void
+ getSequence(): Ptr<Sequence>
+ getSequenceID(): ID
+ getModels(): std::vector<Ptr<Model>>
+ getInterpolatedMatrix(): glm::mat4&
+ fullMatricesCount(): unsigned
+ getParam(): float
+ setParam(float): bool
+ getTrackingProgress(): std::map<Core::ID, float>&
+ getInterpolatedTransformID(): ID
+ result(): TrackingResult&
-m_state
TrackingResult
+ fullMatricesCount: std::size_t = 0
+ interpolatedTransformID: ID = 0
+ interpolatedMatrix: glm::mat4 = {1.0f}
+ trackingProgress: std::map<ID, float>
+ transformInfo: std::map<ID, TransformInfo>
«static»
GraphManager
-m_tracker

Figure 3.13 Partial class diagram of the Core▸Nodes directory containing core node graph logic with focus on the original tracking implementation.

Tracking is started from sequences, which is likely why the `Sequence` objects contain a pointer to a global `MatrixTracker` instance that is stored in a singleton instance of the global `GraphManager` class. Each sequence receives a pointer to this `MatrixTracker` instance upon its construction, which is performed in the static `GraphManager::createSequence()` method. For the time being, it might be simpler to remove this pointer from sequences, and access the matrix tracker directly through static calls to `GraphManager`, as it has been designed to be used. Avoiding the need to manage many separate pointers to the `MatrixTracker` object in each sequence.

When tracking begins, the `MatrixTracker` instance inside `GraphManager` is reassigned to a new instance constructed inside the `Sequence::startTracking` method. Replacing this with static calls to `GraphManager` might also be preferable. All nodes of the workspace are then iterated in order to find models, whose parent is the begin sequence. This step needs to be switched to a tree traversal algorithm in order to include indirectly connected models. This step is also done only once at the beginning, making it impossible to change what models are being tracked without restarting the tracking operation.

For each model, an `IModelProxy` object is created and stored in the `MatrixTracker` instance. This is an abstract interface that is implemented in the user interface Source▸GUI▸Nodes directory as the `WorkspaceModelProxy`.

This is an issue as it means that new models can only be tracked from within the *GUI* classes that can instantiate the derived `WorkspaceModelProxy` object, as this object cannot be created in the *Core* classes (like `Tracking.h`), since that package is meant to be independent from the user interface layer. For this reason, the tracking implementation cannot easily update the tracked model list on its own.

Furthermore, in order to create a copy of the model, that the interpolated transformation is applied to, a whole new hidden `WorkspaceModel` UI node instance is created, that always remains hidden. This means the original model node is more or less unaware of its tracked variant. This system should be scrapped and instead, the original model node should be merely made aware of the tracking operation, letting it create and manage a new 3D model entity directly in the scene view's Source▸Viewport implementation.

Next, the actual transformation chain is constructed, this is done using the `SequenceTree` container that exposes a `MatrixIterator` instance, which is able to traverse the graph towards the root, collecting information about each sequence and transformation in a dedicated `TransformInfo` data object. This is performed in the `MatrixIterator::collectWithInfo()` method. Each `TransformInfo` object represents one node containing a single matrix in the chain and contains a reference to its sequence. A sequence containing two transformations will be represented with two `TransformInfo` objects with the `currentNode` pointer set to each `Transform` node and the `sequence` pointer set to that sequence.

There are some special cases that are considered, every sequence has a matrix data input pin; when this pin is connected, the contents of the sequence are replaced with the passed data until the pin is disconnected. This is indicated with the `isExternal` flag and the `currentNode` being set to the connected source node (an operator or another sequence). Empty sequences are completely ignored.

The collected `TransformInfo` is then processed in the `MatrixTracker::update` method, where it is potentially reversed depending on the tracking direction (`MatrixIterator` always traverses upwards towards the root, e.g. right-to-left). Based on the tracking progress, matrices are accumulated in the respective direction up to the transformation the cursor is located in, which is interpolated, and the final resulting matrix is saved in the internal `TrackingResult` data object. The progress within each transformation is saved to a hash map, keyed with node IDs, and a similar map stores each `TransformInfo` object for each node. Finally, the transforms of tracked models are updated with the resulting matrix. This entire process is repeated every frame, from beginning to the end, which is not ideal.

In order to display tracking indicators in the user interface, the workspace UI layer nodes must first obtain a reference to the `MatrixTracker` object, and then find their respective entries in the provided result hash maps. Which only contain entries for the nodes that provide matrix data, hence not including entries for sequences (unless connected externally), making it difficult to visualize the entire tracking operation in the UI.

3.3 New implementation

Based on the conceptual analysis of the standard and camera tracking, and the overview and analysis of the current implementation, this section summarizes the requirements of the new tracking implementation into the following table of functional requirements, followed by a brief description of each one:

Table 3.1 Functional requirements of the new tracking implementation.

No.	Name	Category
TF01	Tracking of multiple models	Standard tracking
TF02	Better visual indicators	Standard tracking
TF03	Mouse control	Standard tracking
TF04	Camera tracking	Camera tracking
TF05	Camera reference frame	Reference frame
TF06	Multiple scene view windows	Reference frame
TF07	User-defined reference frame	Reference frame

***TF01* - Tracking of multiple models** Activation of tracking in a sequence should track all models that are part of its subtree. Meaning models that are connected directly to the sequence, as well as models that are connected indirectly, preserving their relative transform set by any sequences along the path.

***TF02* - Better visual indicators** The entire tracked chain should be highlighted in the workspace, including model nodes and all sequences. The workspace model node should also highlight tracked models in the scene view, having control over the tracked and original 3D model at the same time.

***TF03* - Mouse control** It should be possible to control the progress of the tracking operation with the mouse. This can most easily be achieved by displaying a slider in the workspace.

***TF04* - Camera tracking** The camera can be connected to sequences by the user, which allows any world space tracking operations to continue into the camera. It can also be tracked on its own (but always beginning with the view transformation, and only right-to-left).

***TF05* - Camera reference frame** Discussed in Section 3.1.4, the camera tracking should employ the scene view reference frames, together with space indicators and a second non-axis-aligned grid keeping track of the previous reference frame.

***TF06* - Multiple scene view windows** Since the scene view should be able to display the world from a specific reference frame, it would be useful to be also able to see the original world in another dockable window.

***TF07* - User-defined reference frame** The scene view reference frame functionality used for camera tracking could be used to view the scene from the reference frame of any transformation. An option can be added to sequence nodes that sets the reference frame of the scene view to their model matrix output, allowing the user to visualize local spaces of individual sequences.

3.3.1 New architecture

The following sections present the new tracking implementation and are accompanied by the class diagram of its architecture shown in Figure 3.14.

The new tracking implementation removes the old `TrackingResult` class in favor of a more comprehensive set of `TrackedNode` objects, which are the tracking representations of existing nodes, that are managed entirely by the `MatrixTracker` object to keep a detailed internal representation of the entire tracking operation at all times.

To allow easy and direct access to tracking information, each core `Node` instance now contains a pointer to a `TrackedNodeData` object, which is managed by the corresponding internal `TrackedNode` instance using RAII[15]. When the `TrackedNode` object is created by the `MatrixTracker`, the tracking data pointer for that particular core node is set. When the tracking operation ends, this pointer is reset back to `nullptr` to indicate that no tracking data is present.

This way, any node can access a set of parameters detailing the tracking status of that node in an active tracking operation; this includes parameters like the tracking progress and whether it is an active part of the chain or a part of the model subtree, etc.

[15]A C++ idiom, see [17] section E.6

Nodes that directly carry transformation data, like transformations and sequences, are represented by the `TrackedTransform` class. It is a derived type of `TrackedNode`, containing methods to access the underlying matrix data. Other nodes involved in tracking, such as the models and, going forward, the cameras, do not use this specialization. The `TrackedNodeData` of models carries an additional `TrackedModelData` object, which contains the final interpolated matrix data.

The `IModelProxy` and `WorkspaceModelProxy` classes have been removed entirely. The workspace Model node implementation can now access the `TrackedModelData` object through its core node reference. When tracking data is available, it adds and manages a new entity in the scene view, removing it when its tracking data is reset back to `nullptr`.

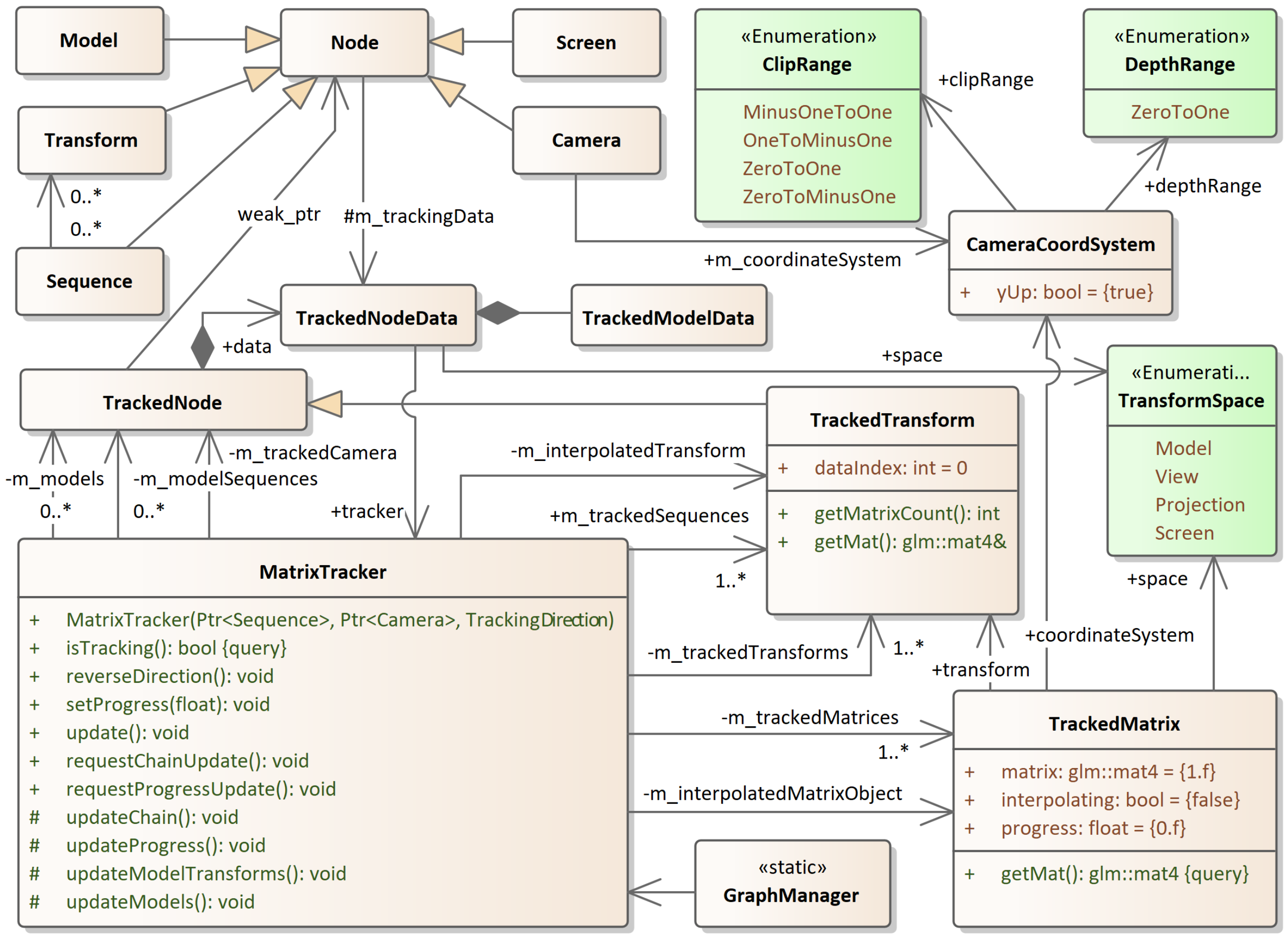


Figure 3.14 Class diagram of the main parts of the new tracking implementation.

Although `TrackedTransform` objects represent individual transformation nodes, which contain a single matrix, the `TrackedTransform` can itself contain multiple matrices in special cases. The camera tracking implementation, which will be described in detail in a later section, requires the ability to split the matrix of a projection transformation node into multiple separate matrices, in order to achieve a more concise visualization. This was previously mentioned in Section 3.1.2.

For this reason, an additional list of `TrackedMatrix` objects is maintained. During standard tracking, there is a one-to-one mapping between the `TrackedMatrix` and `TrackedTransform` objects, and each `TrackedMatrix` contains a reference to the corresponding `TrackedTransform` object, to which the acquisition of matrix data is delegated.

Other changes introduced due to camera tracking and shown in Figure 3.14 are the introduction of the `TransformSpace` enumeration and the `CameraCoordSystem` object. These objects are used in the camera tracking implementation to retain information about the coordinate system of the camera and individual `TrackedMatrix` objects and will also be addressed later.

3.3.2 Traversal iterators

The old `SequenceTree` object has been reworked to represent the model subtree of the tracking operation. It no longer uses a `MatrixIterator`, but a new `SequenceTreeIterator`. The new iterator is a forward iterator[16] that performs a pre-order tree traversal on the subtree of a given sequence, traversing across all matrix multiplication links. Encountered models are added to the `MatrixTracker::m_models` list, and encountered sequences are added to the `m_modelSequences` list, as it should be possible to distinguish them visually. This way all directly or indirectly connected models will be tracked (***TF01***).

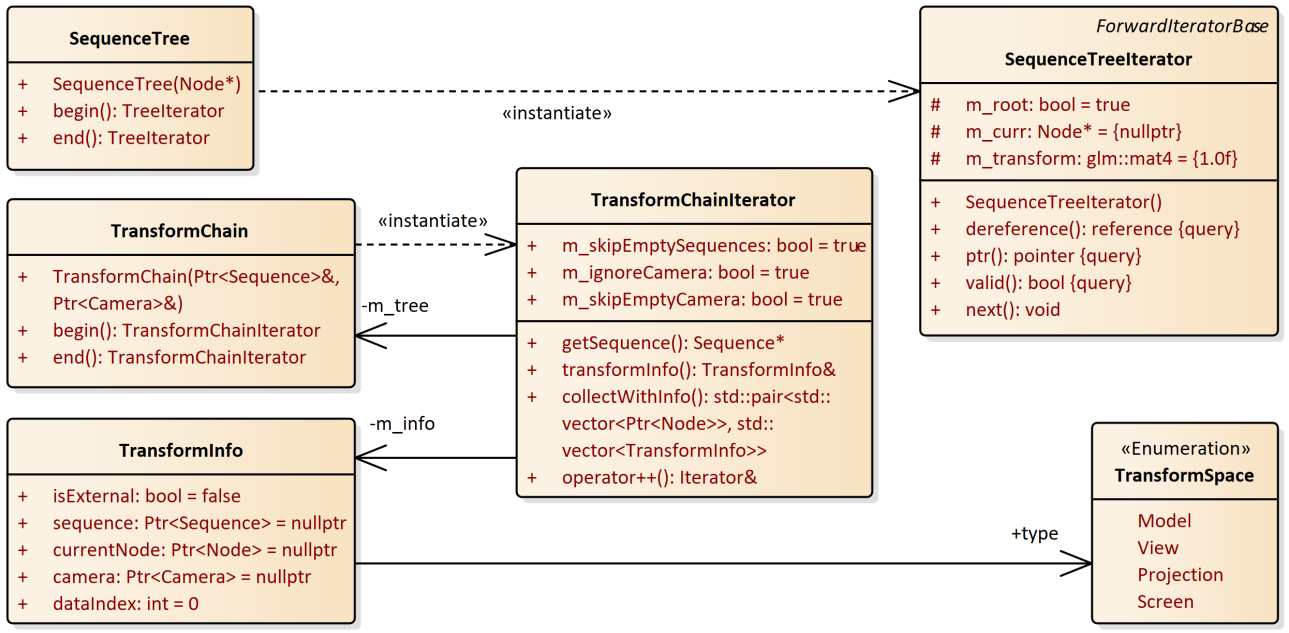
The `MatrixIterator` has been renamed to `TransformChainIterator`, serving as an iterator for a `TransformChain` object (that replaced the old `SequenceTree`). In regards to standard tracking, the only change has been that it now handles empty sequences properly, including them in the resulting list of `TransformInfo` objects, as empty these sequences should also have a corresponding `TrackedNode` representation. The iterator has otherwise been prepared to support camera tracking, by expanding its logic to take Camera nodes into account, with proper handling of empty cameras, and an option to ignore cameras entirely.

The `TransformInfo` object can thus also contain a camera, and a `dataIndex` variable now indicates which data storage of the underlying node should be accessed to retrieve its matrix. This is an improvement over the previous version, which always assumed that the first data storage should be accessed, which corresponds to the data of the first output pin of the node [2, p. 31]. This is used to retrieve the local transformations of a sequence which has its matrix data input plugged in. Previously, the connected node was set as the `TransformInfo::currentNode`, but it is preferable to set it to the sequence node itself[17], which contains that same data in its second data storage.

[16]Built upon the same forward iterator base class as described in Section 2.4.7.

[17]Because the connected node is not really a part of the transformation chain itself.

`TransformInfo` also contains an identifier of the type of “space” the transformation operates in. This is part of the new camera tracking and will be explained in a later section.



SequenceTree
+ SequenceTree(Node*)
+ begin(): TreeIterator
+ end(): TreeIterator
«instantiate»
ForwardIteratorBase
SequenceTreeIterator
# m_root: bool = true
# m_curr: Node* = {nullptr}
# m_transform: glm::mat4 = {1.0f}
+ SequenceTreeIterator()
+ dereference(): reference {query}
+ ptr(): pointer {query}
+ valid(): bool {query}
+ next(): void
TransformChain
+ TransformChain(Ptr<Sequence>&, Ptr<Camera>&)
+ begin(): TransformChainIterator
+ end(): TransformChainIterator
«instantiate»
-m_tree
TransformChainIterator
+ m_skipEmptySequences: bool = true
+ m_ignoreCamera: bool = true
+ m_skipEmptyCamera: bool = true
+ getSequence(): Sequence*
+ transformInfo(): TransformInfo&
+ collectWithInfo(): std::pair<std::vector<Ptr<Node>>, std::vector<TransformInfo>>
+ operator++(): Iterator&
-m_info
TransformInfo
+ isExternal: bool = false
+ sequence: Ptr<Sequence> = nullptr
+ currentNode: Ptr<Node> = nullptr
+ camera: Ptr<Camera> = nullptr
+ dataIndex: int = 0
+type
«Enumeration»
TransformSpace
Model
View
Projection
Screen

Figure 3.15 Class diagram of the new graph traversal iterators implemented in `Core/Nodes/Iterators.h` and used in the new tracking implementation. The `SequenceTree` is used to traverse the model subtree, to find all tracked models, and the `TransformChain` builds the tracked transformation chain.

3.3.3 MatrixTracker **object**

When tracking is started, the `MatrixTracker` object constructs all the various `TrackedNode` objects, based on the traversal results of the `TransformChain` started from the begin sequence, reversing the order if left-to-right tracking is enabled.

Instances of `TrackedNodeData` contain details about the structure of the entire tracking operation, which is reflected in the `childIdxStart` and `end` fields, as illustrated in Figure 3.16. Tracking data instances linked to sequences contain indices of contained transformation nodes (referred to as just *transforms*). Transform nodes carry this information as well, as individual transformations can actually consist of multiple matrices themselves, which is a topic discussed later in the camera tracking section.

The `SequenceTree` is used to find all models and sequences connected to the begin sequence. Tracking data is stored for all of those nodes with the `modelSubtree` flag enabled.

The constructed chain is then stepped through to update progress infor-

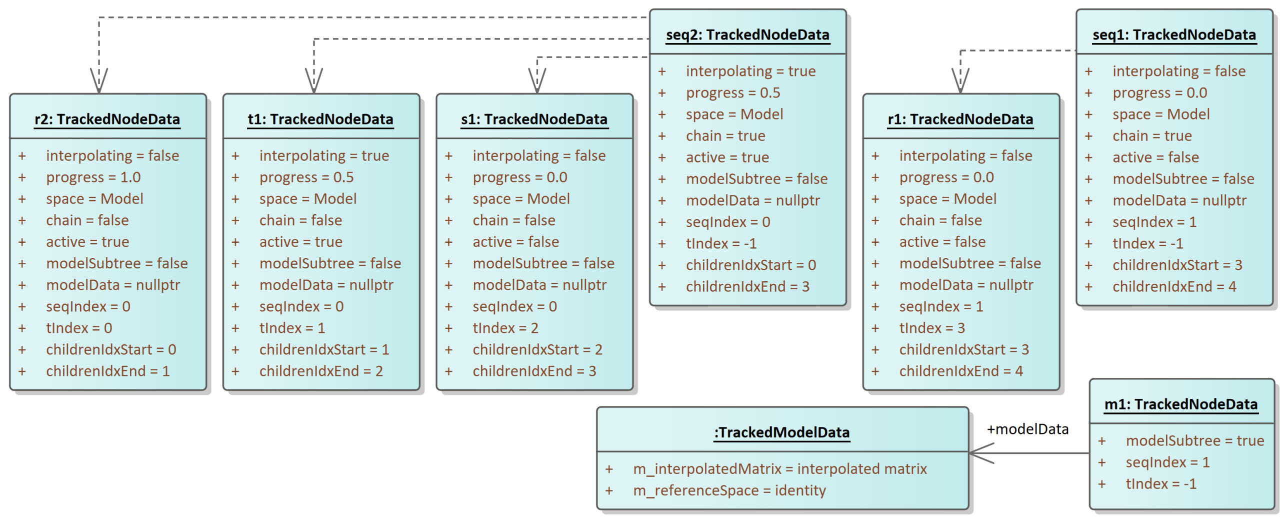


Figure 3.16 Object diagram of the `TrackedNodeData` instances managed by the `MatrixTracker` in the tracking example from Figure 3.3. Every node involved in the tracking operation has a corresponding `TrackedNodeData` instance and the model has an extra `TrackedModelData` instance.

mation and compute the final interpolated matrix, just like before. Progress is computed for overarching sequences as well.

In the end, the `TrackedModelData` of each model is updated with the resulting matrix. As models can be connected indirectly to the begin sequence, their model matrix is not set to the tracked matrix directly, but it is first multiplied by the inverse of the untracked begin sequence transformation, preserving its local transformation within the model subtree.

Unlike before, these individual steps are not performed together every frame anymore. Instead, they are separated into individual update methods, each called only when necessary. This is achieved by leveraging the existing core node update callbacks for value and structural changes. The methods are as follows:

1. `updateChain()` - Constructs the tracked chain by traversing the node graph. Called on structural changes to the tracked chain.

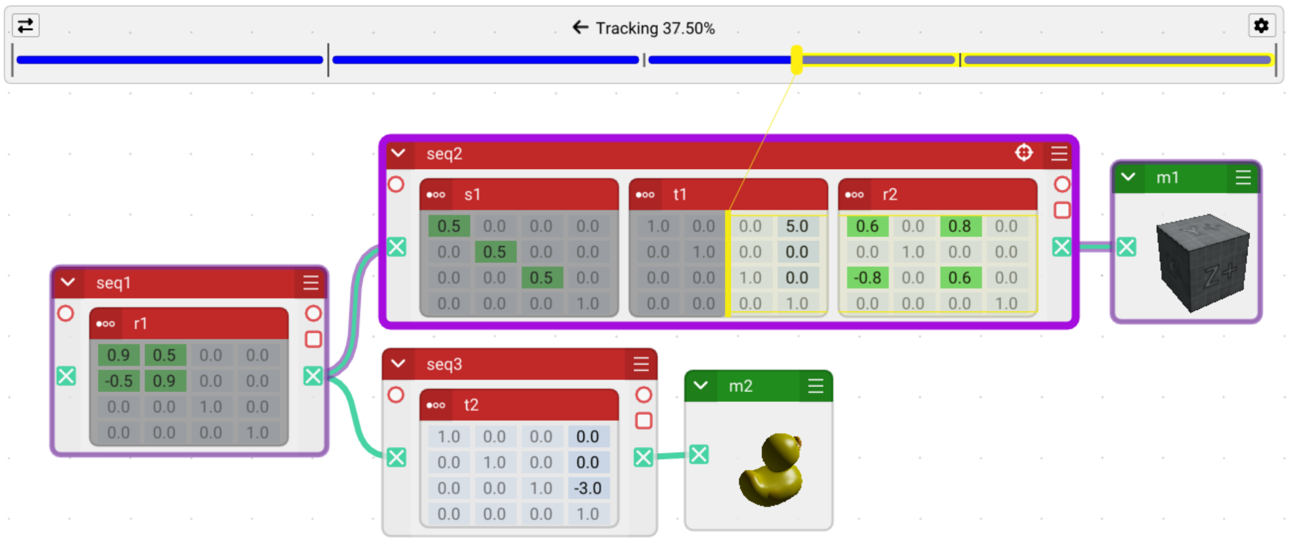
2. `updateProgress()` - Cycles over the chain and computes the final tracking matrix whilst updating progress values. Called on value and structural changes to the tracked chain or when the tracking progress changes.

3. `updateModels()` - Constructs the begin sequence model subtree. Called on value and structural changes to the model subtree.

3.3.4 User interface

Finally, using the detailed information about tracking each core node holds. The UI layer workspace nodes can easily draw visual indicators of the entire chain

(***TF02***). The new visuals can be seen in Figure 3.17. The entire tracked chain is highlighted with purple borders. The colors are, of course, subject to change, as part of the I3T UI theming system. The dark mode version of the new design can be seen in Figure 3.18, which is the default I3T theme the solution is tuned for[18].



s1

0.5	0.0	0.0	0.0
0.0	0.5	0.0	0.0
0.0	0.0	0.5	0.0
0.0	0.0	0.0	1.0

t1

1.0	0.0	0.0	5.0
0.0	1.0	0.0	0.0
0.0	0.0	1.0	0.0
0.0	0.0	0.0	1.0

r2

0.6	0.0	0.8	0.0
0.0	1.0	0.0	0.0
-0.8	0.0	0.6	0.0
0.0	0.0	0.0	1.0

r1

0.9	0.5	0.0	0.0
-0.5	0.9	0.0	0.0
0.0	0.0	1.0	0.0
0.0	0.0	0.0	1.0

t2

1.0	0.0	0.0	0.0
0.0	1.0	0.0	0.0
0.0	0.0	1.0	-3.0
0.0	0.0	0.0	1.0

Figure 3.17 Tracking example from Figure 3.3 shown in the new tracking implementation. All nodes involved in the tracking operation are highlighted, including the respective links. A tracking indicator is shown at the top of workspace, containing additional settings and an interactive tracking slider, that also mimics the structure of the tracked chain.

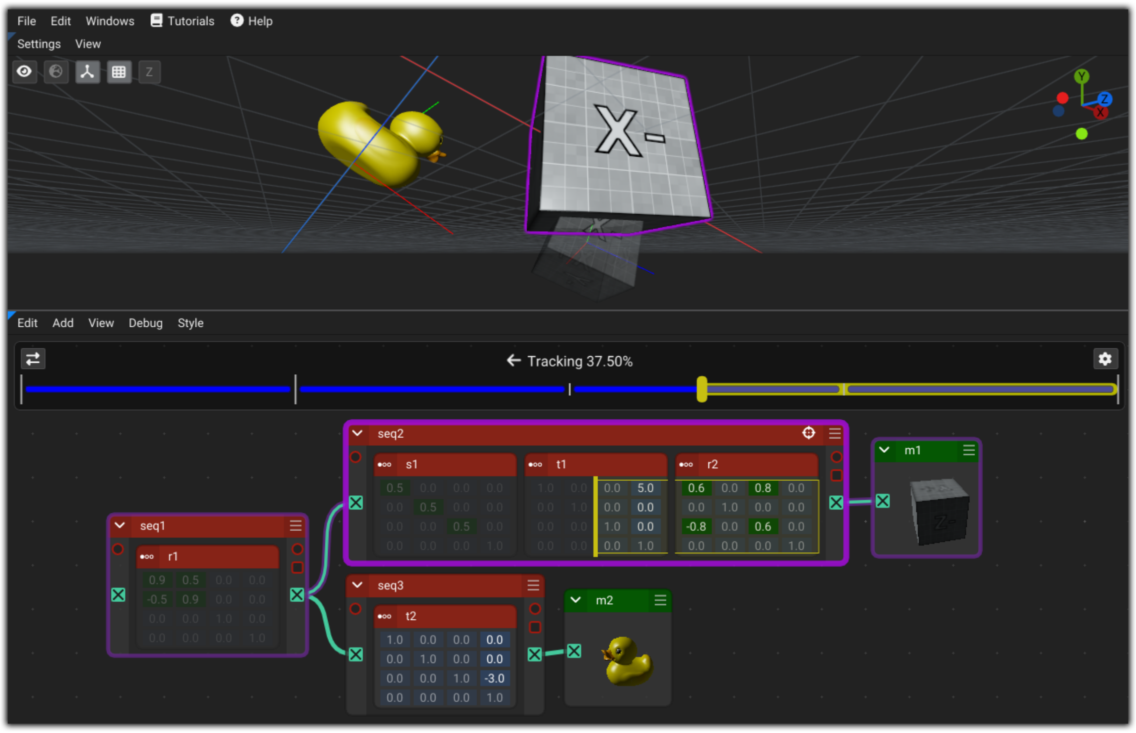
Since the UI layer is now aware which exact nodes are part of the tracking operation, only the tracked transformations can be dimmed, making it easier to see the transformations that are part of the interpolated matrix. The "active" part of the chain, before the cursor, has also been made more visible.

Functional requirement ***TF03*** is realized by adding a custom Dear ImGui slider to the top of the workspace when tracking is active. This slider controls the tracking progress, making it possible to control tracking using the mouse. The slider also visualizes the structure of the tracking operation, which is always visible, no matter the workspace viewport position. The floating window the slider is displayed in is also used to show tracking progress, its direction and a button to switch tracking direction in the top left, and a button with various settings in the top right.

Each transform is shown as a colored segment of the tracking slider. These segments are separated by vertical ticks, which indicate how transforms are structured into sequences. A short tick indicates a border between two transforms within a sequence, while long ticks indicate a jump between two distinct

[18]The less polished light mode is mostly shown in this thesis due to contrast and print quality.

sequences. The color of each transform represents the reference frame it operates in. This is a prelude to the camera tracking implementation that will be discussed shortly. Blue, in this case, signifies world space, which is the conceptual space standard tracking operates in.



r1

0.9	0.5	0.0	0.0
-0.5	0.9	0.0	0.0
0.0	0.0	1.0	0.0
0.0	0.0	0.0	1.0

s1

0.5	0.0	0.0	0.0
0.0	0.5	0.0	0.0
0.0	0.0	0.5	0.0
0.0	0.0	0.0	1.0

t1

1.0	0.0	0.0	5.0
0.0	1.0	0.0	0.0
0.0	0.0	1.0	0.0
0.0	0.0	0.0	1.0

r2

0.6	0.0	0.8	0.0
0.0	1.0	0.0	0.0
-0.8	0.0	0.6	0.0
0.0	0.0	0.0	1.0

t2

1.0	0.0	0.0	0.0
0.0	1.0	0.0	0.0
0.0	0.0	1.0	-3.0
0.0	0.0	0.0	1.0

Figure 3.18 Dark mode variant of Figure 3.17, showing the scene view as well, in which the tracked model is highlighted in purple and shown transparently in its original position.

When interacting with or hovering over the slider, a faint yellow line is drawn, connecting the slider with the current cursor position in the workspace. This is done to reinforce the correlation between the cursor and the slider, as well as to make the cursor easier to locate in the workspace.

Figure 3.18 also shows the tracking visuals inside the scene view. Tracked models are highlighted in purple, using the existing scene view highlighting functionality used for selection [3, p. 59]. The original model is made transparent for the duration of the tracking operation.

3.4 Camera tracking implementation

The matrix multiplication of the camera can now be connected to any sequence by the user. This link, much like before, does not have any effect on the models in the scene view. But starting tracking from a sequence connected to a camera will add the camera's projection and view transformation to the tracked chain, which can be seen in Figure 3.19. These transformations are special in that their `TrackedNodeData`'s `space` parameter is set to `TransformSpace::Projection` and `View` respectively, identifying them as camera transformations, which is shown by the tracking slider by coloring the corresponding segments red (projection) and green (view). Transformations outside of the camera are colored blue (world space, or).

Describing the individual stages as world, view and projection (or NDC) space is perhaps slightly inaccurate, as each segment is an interpolation from one space to another. For example, the "real" view space is shown when the tracking slider is at the left end of the green segment, in other words when the entire view transformation[19] is accumulated into the final tracking matrix.

The perspective frustum transformation serving as the camera's projection transformation is automatically split into two matrices and is displayed in the tracking slider as two segments separated by a small dot. The frustum matrix is modified to prevent the Z axis from flipping[20] and the second one performs the Z axis flip as a separate transformation, ensuring the final result is equivalent. This is the reason why the `TrackedMatrix` objects are needed in the `MatrixTracker` implementation from Section 3.3.3. Visually, the tracking cursor still passes through the single frustum node, but slower.

Previously the camera was implicitly connected to all sequences with an unplugged matrix multiplication output to signify that the camera is positioned at the left end of each transformation chain. This is not a bad concept, but in practice, displaying many of these links leads to visual clutter when many sequences are present in the scene. For this reason, this idea has been abandoned. To make it explicitly clear that the camera transformations are applied to the left end of each sequence, the particular connections can be created manually to make a point, but will be omitted in cases when they're not relevant.

Tracking can be started from the camera itself, in which case the tracking operation begins at the border of the world and view space[21] in the right-to-left direction[22].

3.4.1 Reference frame and view space

In the simplest form of camera tracking, which can be called "world space tracking", the tracked camera transformations are applied to the tracked models just like any other transformation. Leading to issues noted in Section 3.1.4, which introduced the notion of a "reference frame" and a corresponding *reference frame matrix*.

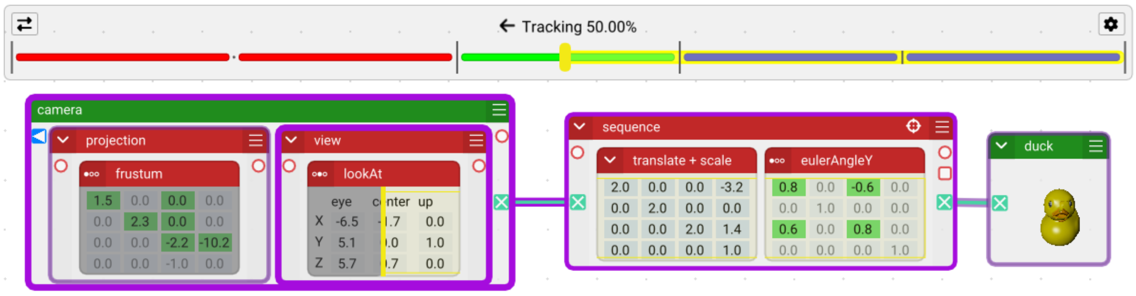
During camera tracking, the reference frame matrix is set to the result of

[19] Or multiple view transformations, as the camera's view sequence can contain multiple transformations.

[20] See Section 3.1.3.

[21] The "uninterpolated" view space, e.g. the right side of the green segment in Figure 3.19.

[22] Which is the only direction camera tracking is allowed in viz. Section 3.1.3.



frustum

1.5	0.0	0.0	0.0
0.0	2.3	0.0	0.0
0.0	0.0	-2.2	-10.2
0.0	0.0	-1.0	0.0

lookAt

	eye	center	up
X	-6.5	-1.7	0.0
Y	5.1	.0	1.0
Z	5.7	.7	0.0

translate + scale

2.0	0.0	0.0	-3.2
0.0	2.0	0.0	0.0
0.0	0.0	2.0	1.4
0.0	0.0	0.0	1.0

eulerAngleY

0.8	0.0	-0.6	0.0
0.0	1.0	0.0	0.0
0.6	0.0	0.8	0.0
0.0	0.0	0.0	1.0

Figure 3.19 Tracking of a sequence connected to a camera.

multiplying the interpolated view and projection transformations. These transformations are handled in a special manner by the `MatrixTracker`, as they are not added to the resulting interpolated matrix, but instead individually stored in special variables `m_iViewMatrix` and `m_iProjMatrix`. The scene view reference frame implementation then multiplies *all* models in the scene with the interpolated camera transformations, which is why they are omitted from the standard tracked matrix, as the camera transformations would be applied twice.

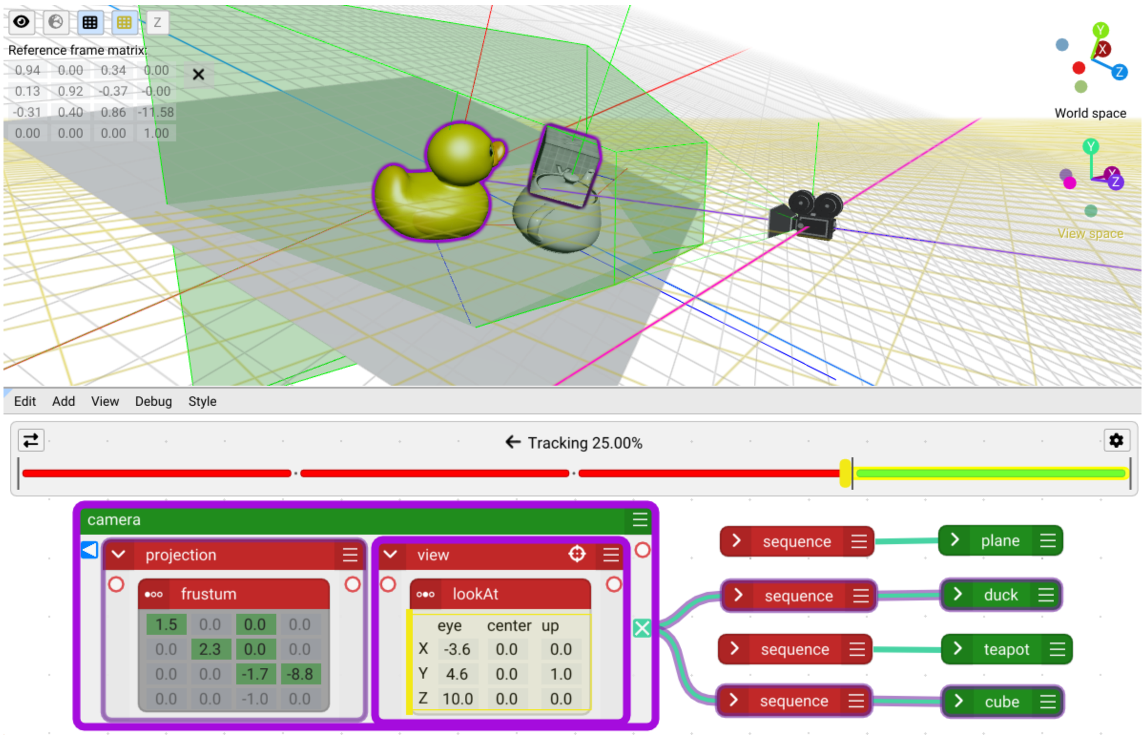
Reference frame is implemented in the scene view by extending the existing scene view rendering method `Viewport::drawViewport` with an additional reference frame matrix parameter. Note that it is also internally referred to as the "implicit model matrix" or just "model" matrix[23].

The scene view's rendering method is called every frame and reacts to its parameters dynamically, so when tracking is not active, an identity matrix can be passed instead. The full signature of the main scene view rendering method becomes: `drawViewport(renderTarget, width, height,` **`glm::mat4& referenceSpace`**`, renderOptions, displayOptions)` with the new parameter in bold.

Internally, the scene view passes the reference matrix to the `Scene` object, which performs the actual rendering of all entities in it. And then to the rendering method of each entity (`Entity::render`), which pre-multiplies the entity's model matrix with the reference matrix. Entities can opt out of this process with a special `m_ignoreReferenceSpace` flag or by overriding the rendering method implementation.

Setting the reference frame matrix to the currently tracked view allows the scene view to display the view space as a whole, positioning the camera at the centre of the local view space world. In order to keep track of the "original" world, a second grid transformed by the reference frame matrix is shown, indicating the relative orientation of the reference frame. The grey grid now represents the original axis-aligned world grid, which has been recolored yellow.

[23] Also note that the scene view `Viewport` class has nothing to do with the viewport transformation, but refers to a generic "3D viewport window". Which is admittedly a little confusing.



Reference frame matrix:

0.94	0.00	0.34	0.00
0.13	0.92	-0.37	-0.00
-0.31	0.40	0.86	-11.58
0.00	0.00	0.00	1.00

frustum

1.5	0.0	0.0	0.0
0.0	2.3	0.0	0.0
0.0	0.0	-1.7	-8.8
0.0	0.0	-1.0	0.0

lookAt

	eye	center	up
X	-3.6	0.0	0.0
Y	4.6	0.0	1.0
Z	10.0	0.0	0.0

Figure 3.20 Tracking of the view transformation, showing the scene using the view space reference frame. The reference frame matrix is shown in the top left. World space and view space axes indicators in the top right. The the original world grid is drawn in grey, and the local view space grid in yellow.

The achieved effect can be seen in Figure 3.20, which shows the view space of a camera looking down upon a scene of three objects sitting on a plane. The camera is positioned at $(0, 0, 0)$, looking along the $-Z$ axis, and the scene now appears rotated. The tracking progress can, of course, be animated, showing the original world grid shift away and the reference frame grid slowly fades into view.

Even though only the duck and cube models are connected to the camera, every model is transformed by the camera's "lookAt" transformation, preserving a unified look into the camera's view space. The original *world space tracking* behaviour that does not use the reference frame functionality can be restored by enabling the "track in world space" option in the tracking settings.

Models that are not connected to the camera could be hidden instead, but that would mean they disappear from the scene view when tracking is activated, which is odd, because the scene view acts as an independent view of the entire scene. It would make sense if unconnected models were hidden in the camera's screen display output, but that has not been implemented.

The secondary non-axis aligned grid is implemented by modifying the original infinite grid shader implementation to transform the ray segments representing each pixel by the reference frame matrix, prior to performing axis-aligned intersection tests with the ground plane. This is a very simple modification that preserves the original grid rendering logic that does not support generic plane orientation. Transformation of the unprojected near and far ray points is performed in the vertex shader, so the performance impact is minimal. A single matrix multiplication is still performed in the fragment shader in order to determine the final fragment depth, which leaves some room for improvement. See [3, p. 58] for more details about the infinite grid implementation.

The reference frame matrix is displayed in the scene view window in the top left corner[24]. The world axis indicator in the top right corner has been expanded to show a second set of axes. The original axes show the axes of the now transformed world space orientation, and a second set of axes below it now shows the local space. Each set of axes is labelled based on the current context.

3.4.2 Multiple scene views

In order to satisfy requirement ***TF06***, the scene view implementation has been expanded to allow multiple scene view windows to be opened at the same time. A new `ViewportModule` class has been added, which extracts common scene view functionality from `ViewportWindow`. Each `ViewportWindow` has been given its own scene view camera object to render the scene with, and changes were made in the `Viewport` class to accept such cameras as parameters. Each window now must also serialize its own set of camera parameters. Tracking currently always sets the reference space of the first scene view window. An unlimited amount of other windows can be created from the Windows › Scene view window menu.

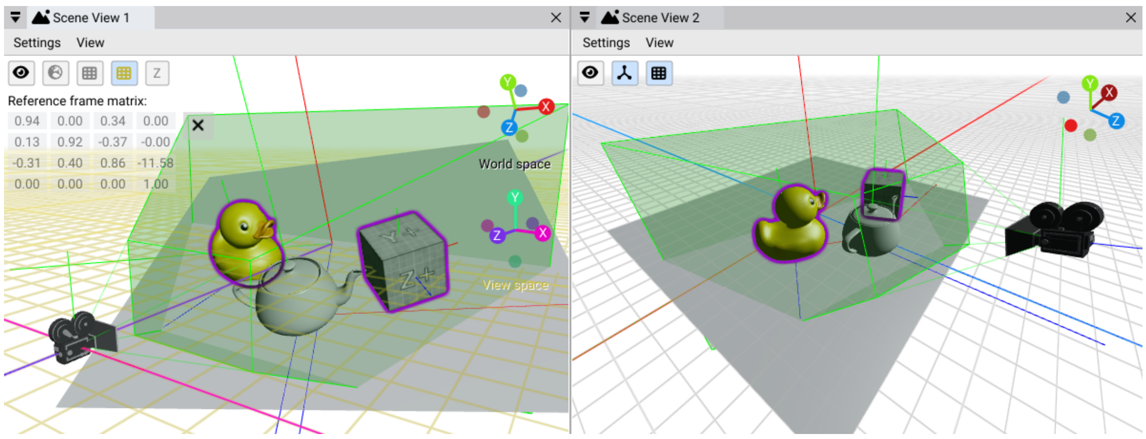
The example from Figure 3.20 can be viewed in the primary scene view, and a secondary scene view can show the original scene in world space. Giving crucial context about the original orientation and position of the camera. This can be seen in Figure 3.21.

3.4.3 Camera projection

The projection transformation is tracked in the same way as the view transformation, but it does have its own specifics. As mentioned previously in Sections 3.1.2 and 3.3.1, the projection transformation is split into multiple matrices when tracked. For simplicity, this is only done when the camera's projection sequence contains a single transformation, which is usually the case. This process is referred to as "decomposition" and can be toggled on and off in the tracking settings.

It is implemented in the `MatrixTracker`'s `handleProjectionTransform` method which takes a `TrackedTransform` object and splits it into multiple `TrackedMa-`

[24]Unless it is set to an identity matrix.



Reference frame matrix:

```
 0.94  0.00  0.34   0.00
 0.13  0.92 -0.37  -0.00
-0.31  0.40  0.86 -11.58
 0.00  0.00  0.00   1.00
```

Figure 3.21 View space visualization using two separate scene view windows. The view space of the camera is shown on the left, while the scene in world space is shown on the right.

trix objects when appropriate. The most basic decomposition was already presented before, splitting the projection into two matrices so that the projection itself has its Z axis flipped. This is done so that the world does not physically flip during the interpolation. Interpolation of such a decomposed perspective projection is shown in Figure 3.22.

The OpenGL normalized device coordinates are a left-handed coordinate system. The scene view and I3T in general uses a right-handed coordinate system, which is largely ingrained in its internal design. For that reason, the left-handedness of the OpenGL NDC space is only emulated by the visible axis indicators. This can be done by omitting the second Z flip matrix entirely and only explicitly flipping the Z axis indicator in the top right corner. This would have to be done either instantly at the beginning or end of the interpolation, or gradually during its progress.

To make it very clear that the NDC coordinate system is different, the second flip matrix is retained, but it also flips the X-axis. Resulting in an interpolation that flips the entire space, during which the local axis indicator's X axis is gradually flipped, adding the words "Left handed" below it when the X axis flips. This is shown in Figure 3.23.

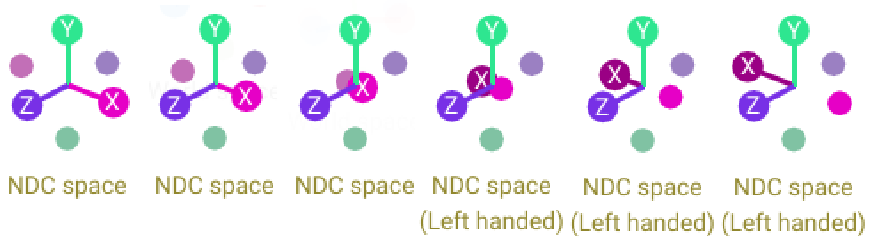


Figure 3.23 Gradual flip of the X axis of the local axes indicator, which is performed as a separate step after the projection interpolation.

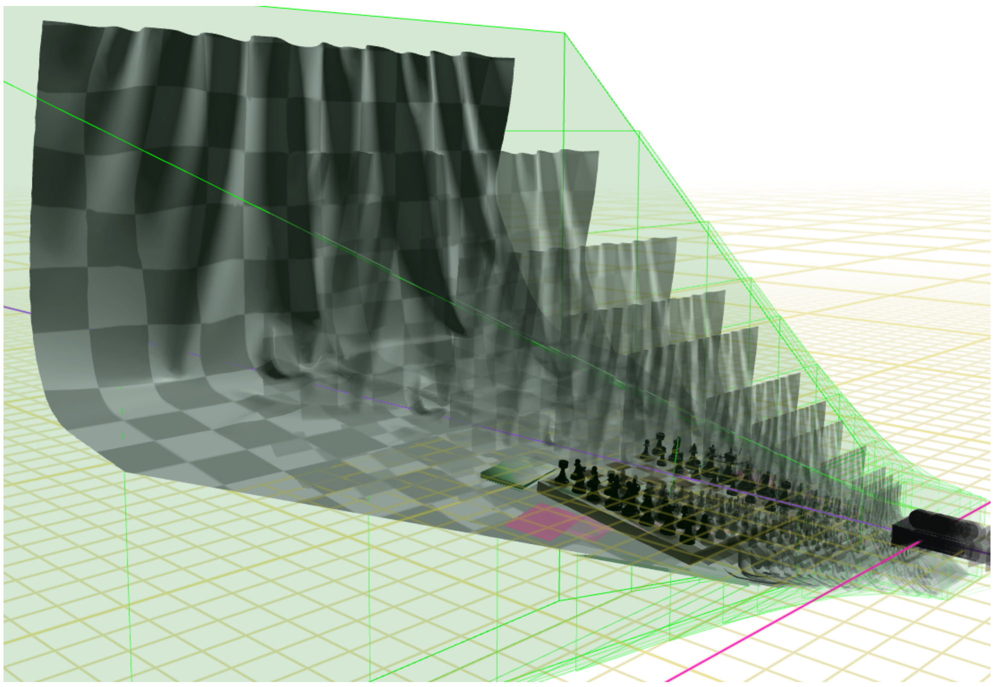


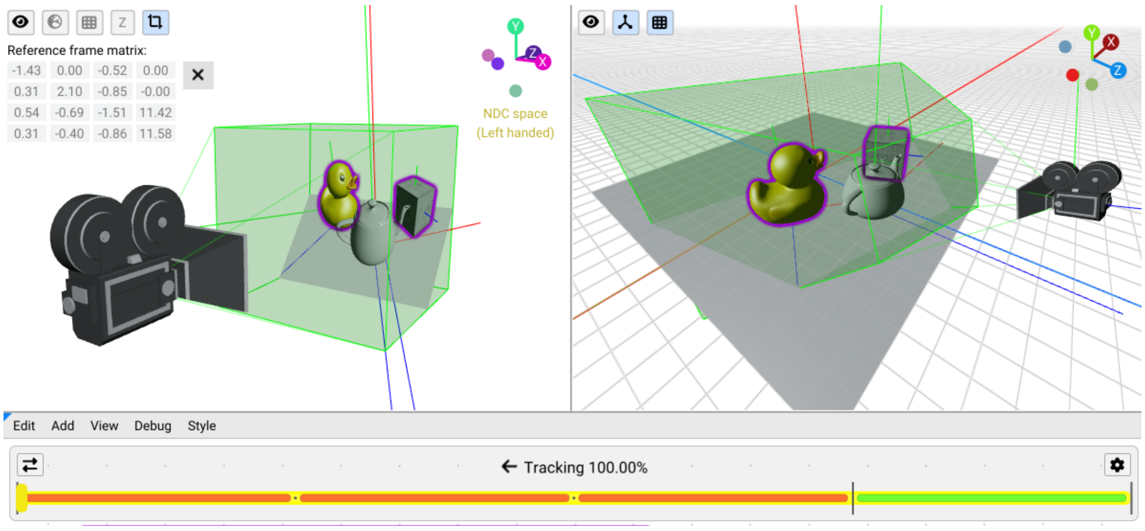
Figure 3.22 Composite image showing the interpolation of the perspective transformation. The frustum is reshaped into the NDC “cube”. The Z axis of the resulting NDC space is flipped.

This successfully emulates a left-handed NDC space, but the ZX flip does often introduce lighting artifacts. An alternative is to only flip the Z axis of the indicator, whilst interpolating over an identity matrix, but that might be confusing as it could seem that nothing is happening.

The final NDC space visualization is displayed in Figure 3.24 showing the same scene as Figure 3.21. During projection interpolation, the scene view camera model is manually moved to an arbitrary position outside of the NDC viewing volume. This is simply done to avoid visual obstruction and also to retain an indicator of the near frustum plane. Four faint lines with the same color as the camera frustum outline also connect the near plane corners with the camera model origin. This is an additional new near plane indicator that is always visible.

Shirley decomposition

A perspective projection matrix can further be decomposed into its orthographic and projective components. Such a decomposition is described by Shirley et al. in Chapter 7 of *Fundamentals of Computer Graphics* [14, p. 150] which separates a perspective projection matrix into an *orthographic* ($\mathbf{M}_{\mathbf{orth}}$) and *perspective* ma-


Reference frame matrix:

-1.43	0.00	-0.52	0.00
0.31	2.10	-0.85	-0.00
0.54	-0.69	-1.51	11.42
0.31	-0.40	-0.86	11.58

Figure 3.24 Visualization of the NDC space of a camera. Corresponds to the scene from Figure 3.21.

trix (**P**) like so:

$$\mathbf{M}_{\mathbf{orth}}\mathbf{P} = \begin{pmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & \frac{2}{n-f} & -\frac{n+f}{n-f} \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & n+f & -fn \\ 0 & 0 & 1 & 0 \end{pmatrix} \tag{3.6}$$

Splitting the transformation this way simplifies the matrix that is responsible for the perspective distortion effect. It also makes the overall perspective transformation interpolation much clearer, as the interpolation over the *perspective* matrix leaves the depth values of the near and far planes constant. In fact, points on the near plane are not modified at all, and the far plane shrinks along the XY axes until the entire frustum turns into a rectangular cuboid. This cuboid is scaled and translated into the final NDC "cube" shape. Figure 3.25 shows the chess scene before and after applying the perspective matrix.

Brown decomposition

Another kind of decomposition that further splits the orthographic and perspective matrices into two more matrices each can be found at the *Learn WebGL* website by Dr. Wayne Brown [18]. This approach simplifies the matrices further, but loses the nice visualization properties compared to the decomposition above.

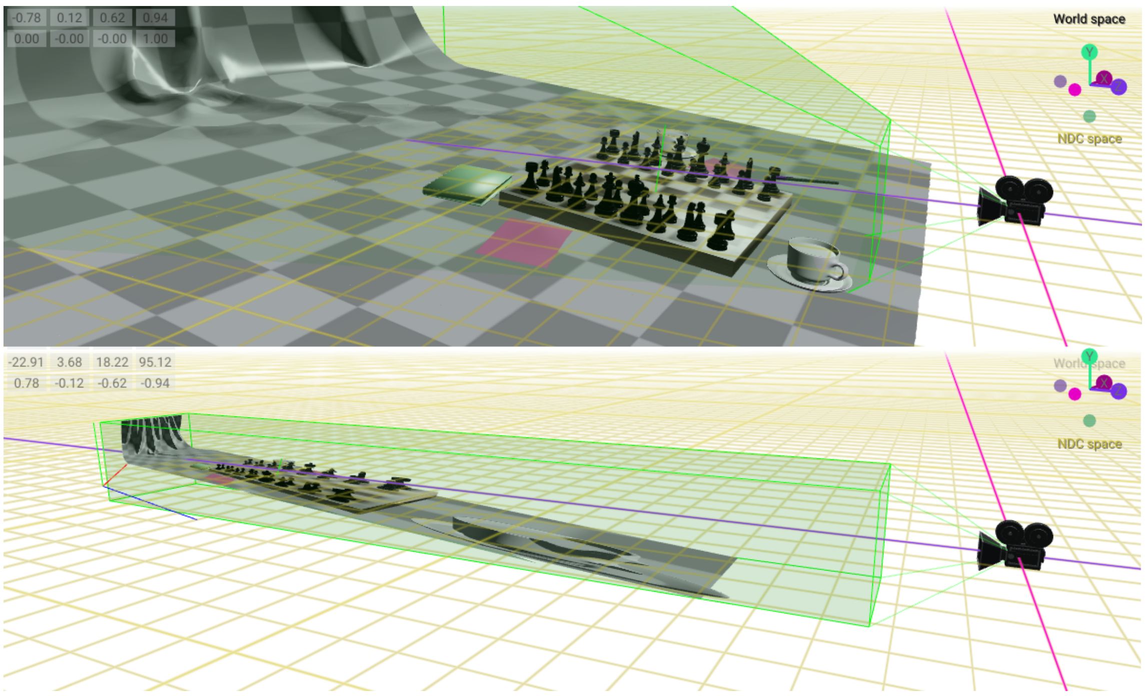


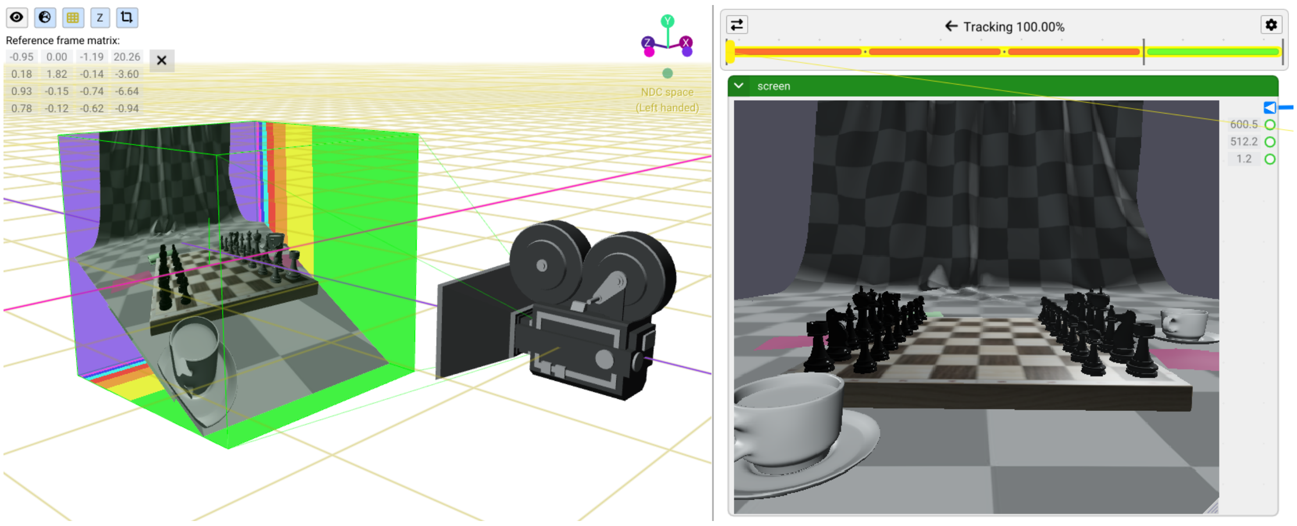
Figure 3.25 Perspective decomposition by Shirley et al. visualized with camera tracking. Notice that the near plane of the frustum does not move.

Nevertheless, both decompositions are implemented and can be enabled by the respective options in the tracking settings. The "Shirley" decomposition is the default.

Depth visualization and clipping

Figure 3.26 shows the resulting NDC space of the scene from Figure 3.25. The display output of the camera is shown on the right in a screen node. To emphasize the non-linear effect the perspective transformation has on depth values, depth visualization mode can be enabled using one of the buttons in the top left corner of the scene view. This mode renders eight color bands at the far side of the camera frustum. The color bands are spaced evenly in world space, but in NDC space, the first color band is disproportionately large, showing how depth precision is "wasted" close to the near plane.

One of the buttons in the top left corner can also be used to toggle clipping of the viewing volume. This feature enables custom clipping planes that are used to clip vertices outside of the tracked camera's viewing volume, visualizing the clipping step of the fixed function graphics pipeline. It also makes the visualization clearer by eliminating often heavily distorted models outside of the viewing volume. Clipping is implemented using OpenGL user-defined clip planes [13, p. 454], which are controlled in shaders by setting the `gl_ClipDistance` vertex shader output array. This means the clipping is performed in hardware by OpenGL itself, staying true to what usually happens out of the view.



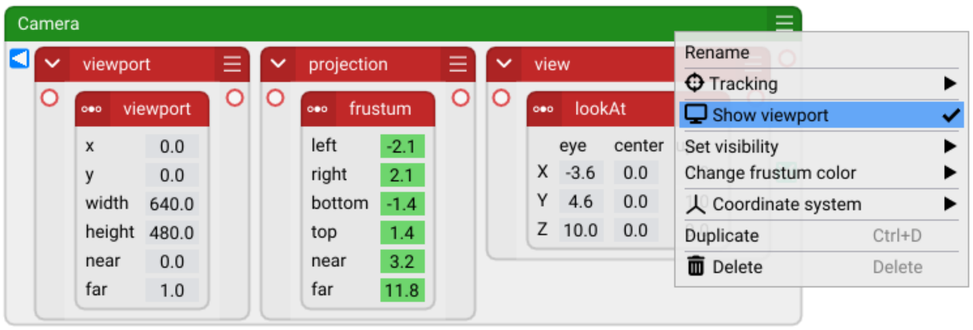
Reference frame matrix:

-0.95	0.00	-1.19	20.26
0.18	1.82	-0.14	-3.60
0.93	-0.15	-0.74	-6.64
0.78	-0.12	-0.62	-0.94

Figure 3.26 Visualization of NDC space depth values and a comparison with the rendered image.

3.4.4 Viewport transformation

The last step of camera tracking is the viewport transformation which scales and translates the normalized device coordinates into pixel coordinates (the screen space). As the viewport transformation was missing from I3T entirely, a new sequence for it has been added to the camera node. The viewport sequence is optional and hidden by default. It can be enabled from the camera's context menu, as can be seen in Figure 3.27.



viewport

x	0.0
y	0.0
width	640.0
height	480.0
near	0.0
far	1.0

frustum

left	-2.1
right	2.1
bottom	-1.4
top	1.4
near	3.2
far	11.8

lookAt

	eye	center
X	-3.6	0.0
Y	4.6	0.0
Z	10.0	0.0

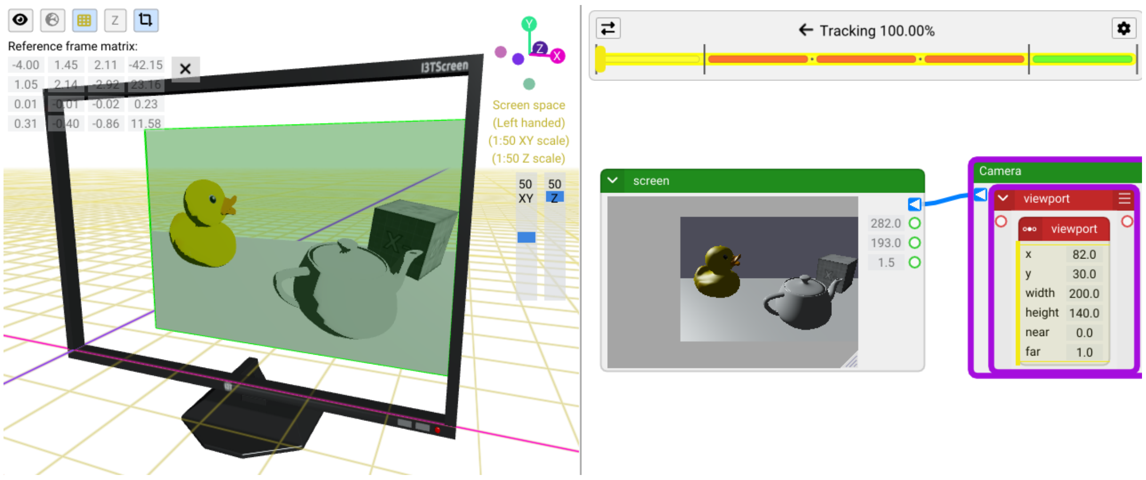
Figure 3.27 The camera node with the “Show viewport” option enabled.

When disabled, the camera functions just like before. It outputs the view and projection matrices from its blue *display* output pin, which can be plugged into a screen node. The matrices are used to render the screen's image. The viewport transformation that is used to render the image is determined automatically based on the current size of the screen node.

For the viewport information to be readily available to the screen node, the data passed along the display output link has been changed from a pair of matrices to a dedicated `ScreenData` object. This object contains the view and projection matrices, an additional viewport transformation matrix, and a flag indicating whether the viewport transformation is enabled.

When it is, the screen extracts the x, y, *width* and *height* parameters from the viewport matrix and emulates their effect during UI rendering by translating and scaling the resulting image. This is done for simplicity as the scene view implementation currently does not support custom viewport settings. And flexibility, as special handling is required to emulate Vulkan viewports inside OpenGL.

An explicitly set viewport transformation is shown in Figure 3.28. Its result is shown in the scene view as well as in the image of the screen node. The viewport transformation is displayed in the tracking slider as an additional yellow segment.


Reference frame matrix:

-4.00	1.45	2.11	-42.15
1.05	2.14	-2.92	23.16
0.01	-0.01	-0.02	0.23
0.31	-0.40	-0.86	11.58

Screen space (Left handed) (1:50 XY scale) (1:50 Z scale)

Tracking 100.00%

screen: 282.0, 193.0, 1.5

Camera — viewport — viewport

x	82.0
y	30.0
width	200.0
height	140.0
near	0.0
far	1.0

Figure 3.28 Visualization of camera screen space using an explicit viewport transformation. Corresponds to the scene from Figures 3.24 and 3.21. The screen node in workspace is represented with a 3D model of a screen into which the NDC space is transformed into.

During interpolation, the camera is faded out and a 3D model of a monitor is faded in. This computer monitor has the same size and aspect ratio as the screen node's image. The screen size is now fixed and directly controllable by the drag handle in its bottom right corner. The screen's width and height in pixels are now accessible using float output pins. These can be used to construct a custom viewport transformation by using the new *viewport* operator node.

As the viewport transformation scales the viewing volume to a size specified in pixel units, the relative size difference between the viewing volume in NDC space and screen space is quite large. For that reason, the viewport matrix is internally modified, similarly to the projection transformation, to reduce the massive change in size and keep the viewing volume visible through its interpolation, without the need to adjust the scene view camera. There is a separate scaling factor for the XY and Z axes, and both are adjustable with sliders below the axes indicators.

3.4.5 Vulkan support

I3T has always been developed with OpenGL in mind, as that is the main graphics API that students at CTU encounter and work with. There are a few other major graphics APIs, one of them being Vulkan, a successor to OpenGL developed by The Khronos Group [19, p. 14]. Vulkan is different in many ways, but the basic principles of its graphics pipeline remain similar, and the new system of camera tracking can be used to visualize it as well.

There are, however, some differences in the used coordinate systems between the two, which, although minor in principle, have a major impact on the resulting image. This section briefly describes how the camera tracking process can be modified to faithfully visualize the same process when using the Vulkan API instead of OpenGL.

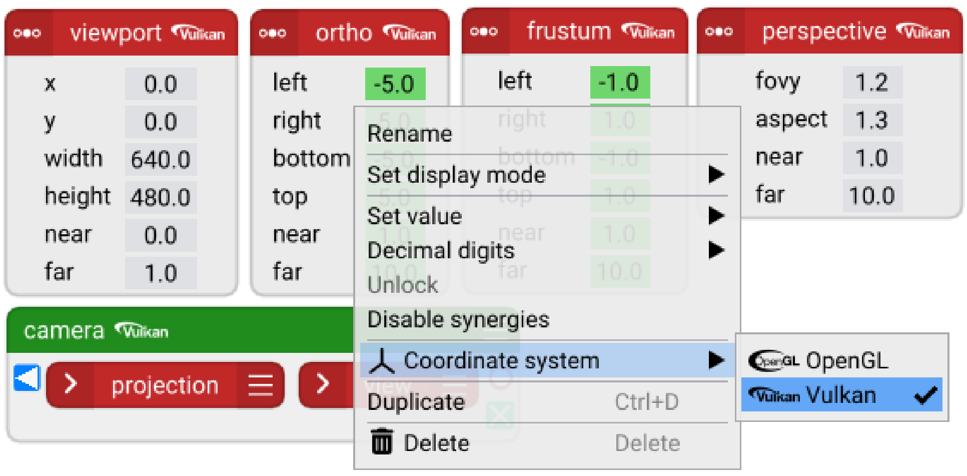
The key differences are the following:

1. Vulkan NDC space is a **right-handed** coordinate system with the Y axis pointing "downwards". The depth range is changed from $[-1, 1]$ to $[0, 1]$.
2. The same is true for the Vulkan screen space. The screen origin is in the **top-left corner**, as opposed to the bottom-left corner in OpenGL.

Note that these are differences between the default or "usual" Vulkan and OpenGL coordinate systems. Both Vulkan and OpenGL offer ways to change their coordinate systems to emulate each other. For example, in OpenGL, the NDC space depth range can be controlled with `glClipControl` and the screen space depth range with `glDepthRange`. Furthermore, the depth test comparison function can be changed with `glDepthFunc` and the winding order of polygons with `glFrontFace`. Meaning the graphics APIs can be made to use a wide range of coordinate systems, but this thesis will focus on and compare the most commonly used configurations in Vulkan and OpenGL.

The camera node, the new viewport transformation node and all of the projection nodes (*ortho*, *perspective* and *frustum*) have been given an option to switch their coordinate system to Vulkan. The new node variants are seen in Figure 3.29.

For the transformation nodes, this changes how their matrix is constructed from their respective parameters to account for the change in depth range. The camera changes its assumptions about the viewing volume to ensure that the camera frustum is visualized correctly. Screen nodes connected to the camera also react by changing their assumptions about the screen space, placing the



viewport Vulkan

x	0.0
y	0.0
width	640.0
height	480.0
near	0.0
far	1.0

ortho Vulkan

left	-5.0
right	
bottom	
top	
near	
far	

frustum Vulkan

left	-1.0

perspective Vulkan

fovy	1.2
aspect	1.3
near	1.0
far	10.0

Figure 3.29 New Vulkan variants of the camera, projection transformations and viewport transformation.

3D computer monitor in the scene such that its top-left corner is placed in the world origin. Tracking detects the change in the camera's coordinate system and adjusts the NDC and screen space axes indicators, so that the Y axis is visually flipped during interpolation.

The comparison between Vulkan and OpenGL normalized device coordinates of a camera with the same projection parameters can be seen in Figure 3.30.

3.4.6 User-defined reference frame

The reference frame feature from Section 3.4.1 can be set to use any user-defined matrix, for example, the output of a sequence node, realizing ***TF07***. Any of the multiple scene view windows can be set a reference of any sequence from its context menu: Reference frame » Set as reference frame. The reference frame is then set to the inverse of the sequence's model matrix $\mathbf{S_M}$:

$$\mathbf{R} = \mathbf{S_M^{-1}}$$

An example of a scene view with the reference frame set to a sequence that defines the local coordinates of Jupiter in a Solar System scene can be seen in Figure 3.31.

3.5 Testing

One testing session focused on the new tracking functionality was performed with five students by Barbora [7]. Overall, the reception was positive. The most notable complaint was that the matrices of the perspective decomposition

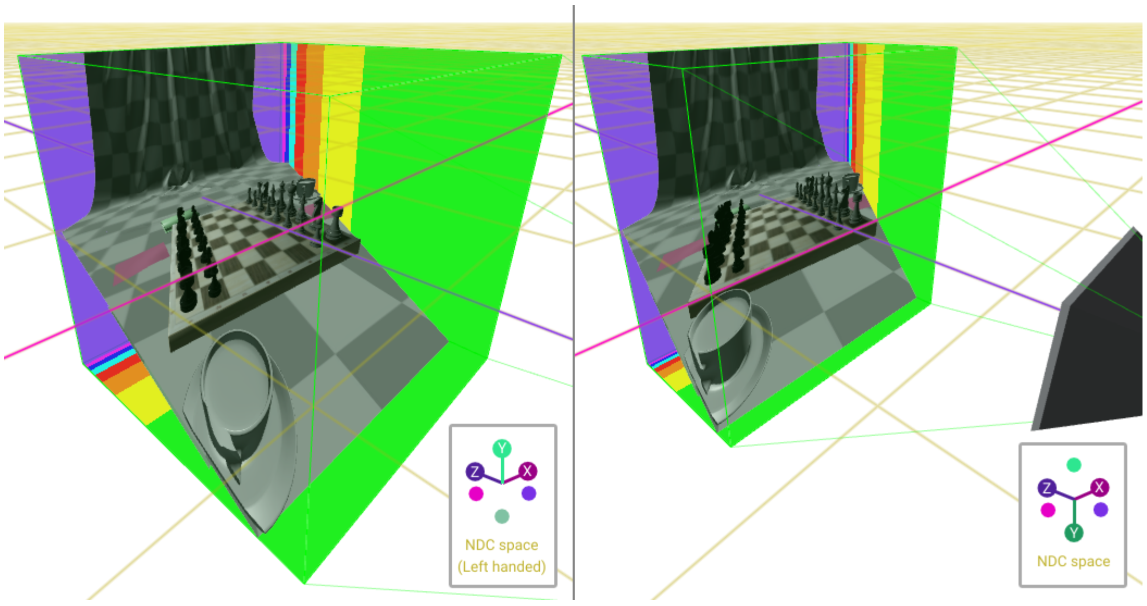
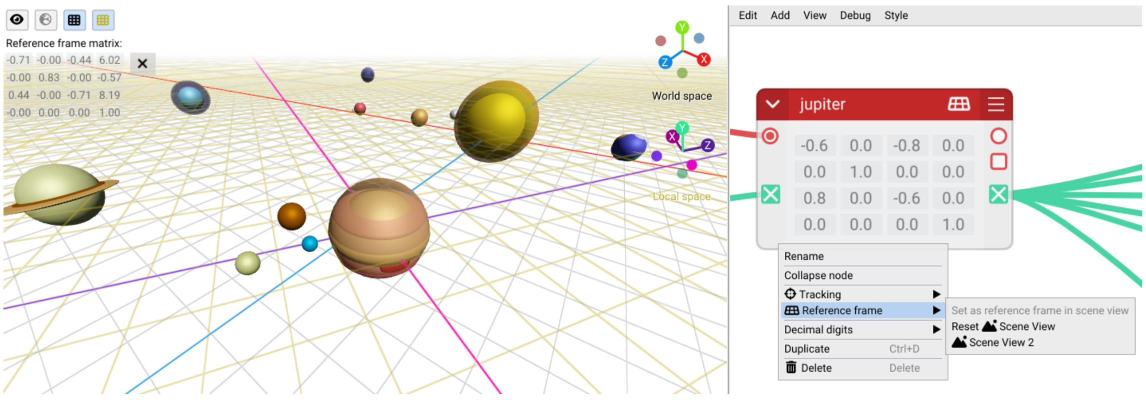


Figure 3.30 Comparison between Vulkan and OpenGL normalized device coordinates.

are not displayed anywhere, and so it is unclear what exactly the individual matrix segments represent without prior knowledge about the decomposition.

Expanding transforms into multiple nodes inside a sequence is not a good idea. Showing the matrices as part of the tracking slider is also potentially very cluttered. A good solution would be to add a floating “popup node” above the camera’s projection sequence, which could “expand” the projection transformation in order to view the individual matrices making up the decomposition, with additional information available on hover. This popup would only be visible for the duration of the tracking operation progressing through the projection sequence.



Reference frame matrix:

-0.71	-0.00	-0.44	6.02
-0.00	0.83	-0.00	-0.57
0.44	-0.00	-0.71	8.19
-0.00	0.00	0.00	1.00

jupiter

-0.6	0.0	-0.8	0.0
0.0	1.0	0.0	0.0
0.8	0.0	-0.6	0.0
0.0	0.0	0.0	1.0

Figure 3.31 User-defined reference frame of Jupiter in a scene of the Solar System.

Chapter 4
Discussion

DIWNE canvas zooming The zoom feature of the node editor could be achieved by transforming resulting `DrawList` vertex data, instead of scaling Dear ImGui size variables. This approach probably should have been explored in the beginning, but the old approach was already ingrained in the previous DIWNE implementation, and is also used by a few older existing node editor implementations. So the concept was not investigated much deeper.

But such an approach could significantly help with some of the current layout issues and so is probably worth exploring in the future. The original zoom behaviour can be retained, and both methods could be implemented and switched between for comparison. The current zoom behaviour can be disabled by fixing the zoom level to one. The compiler would likely optimize any zoom factor calculation overhead if this switch was made at compile-time. The DIWNE `Canvas` class could be abstracted and contain two implementations.

Better text rendering Text tends to become unreadable with lower zoom levels. There is some room for improvement within the implementation, but unfortunately, this is largely a Dear ImGui issue, as its default implementation of text rendering doesn't support auto-hinting or sub-pixel rendering. There is an experimental branch of Dear ImGui that could perhaps improve the current rendering.

Clip space visualization This topic has not been explored in this thesis but might be interesting. There are some attempts at visualizing the 4D homogeneous coordinates, but they are all somewhat confusing just due to the nature of 4D.

Different types of perspective matrices Notably, ones with the reversed NDC depth range of 1.0 to 0.0, which optimizes the floating point accuracy of the depth buffer. Other example would be a perspective frustum with a far plane at infinity.

Frames and group nodes With generic support for nested nodes and the fact that the node editor is itself a `DiwneObject`, nested node editors could be supported. However, as this Blender dev article mentions (code.blender.org /2012/01/improving-node-group-interface-editing/), inline group nodes with visible inner graphs are not exactly the best approach, so a simple group node that opens a second editor in another window or just as a window overlay would also be possible, and likely not that difficult, as it is simply a matter of creating and displaying a second `NodeEditor` instance.

Invalid matrix value highlighting Although invalid matrix value highlighting was mentioned in the assignment, it was given as an example and other parts of the implementation took priority. The new DIWNE layout system does improve the placement of the invalid matrix icon, but does not address improvements like specific descriptions of the error, or indicators showing where the error has occurred. Such functionality requires larger modifications of the *core* node graph code, which was not the main focus of the workspace improvements.

Chapter 5
Conclusion

The primary goal of the thesis has been largely fulfilled. The workspace user interface implementation of I3T has been reworked and major issues with the underlying DIWNE library have been addressed. Major improvements have been made to the user interface. Giving the workspace a significant visual upgrade that also majorly improved its usability and ease of interaction.

The work in progress implementation was consulted with and tested on multiple occasions by a fellow student Barbora Hálová, who was concurrently working on a usability review of the application. This collaborative effort helped gather crucial user feedback, whilst letting this thesis focus primarily on the many aspects of the new implementation.

The DIWNE library was reworked from the ground up and predominantly decoupled from the existing *workspace* UI implementation that deals with the specifics of I3T. The original ambition to completely separate the library did not fully come into fruition due to time constraints, as the library is still kept as a direct part of the application source code, but key parts of it do function as an independent library, as it can be compiled and run as an independent application. With little remaining effort, the last portion of I3T code responsible for the new pin design can be moved into DIWNE and the library will be ready for a stand-alone open-source release, expanding the Dear ImGui library ecosystem.

The new codebase is accompanied by Doxygen documentation comments and additional comments explaining basic principles, although for the codebase to be released to the public, the documentation still needs some attention and likely a separate set of documents explaining its higher-level concepts. For which this thesis can serve as a basis.

Fundamental issues with input propagation and interaction modes have been conceptually resolved in the form of the `DrawInfo` context object and an overarching object-oriented `DiwneAction` state.

The secondary goal of the thesis was thoroughly fulfilled, implementing a fully interactive and animated visualization of all the camera transformations within the context of OpenGL and Vulkan. The visualization was seamlessly integrated into the existing transformation tracking functionality, which in itself has been significantly improved using the new workspace UI implementation.

The new camera tracking fills the conceptual gap between world and screen space coordinates that could not have been previously easily explored within I3T. Allowing the user to physically view the journey model vertices take through the often opaque and hard-to-understand mechanism of the perspective camera, all the way to the final screen space coordinates, which get rasterized onto the screen.

With this functionality, essential concepts like the “lookAt” transformation, viewport transformation, or the non-linear effects of the perspective divide can be explored. With only the notable omission of the clip space coordinates, which have been mostly omitted, as they cannot be easily visualized in a 3D world.

As a side effect of the camera tracking implementation, the reference frame feature has been added to all sequences in the workspace, allowing the user to view the 3D scene from the point of view of any transformation, while displaying the relative orientation of the original world using a second rotated infinite grid. Hopefully further deepening the understanding of regular model transformations I3T focused on from its inception. Or at least making experimenting with them more interesting.

To add context, the 3D scene can now be viewed from multiple independent cameras at the same time, possibly each with a different reference frame, which was previously not possible.

Appendix A

Multiple perspective projections

Comparison of transforming a 3D point by two distinct OpenGL perspective projection matrices with the perspective divide being performed once at the end, and twice after each matrix multiplication. This serves as a basic proof that the perspective division operation can truly be performed just once, despite applying multiple distinct projection matrices. The proof certainly isn't definitive, as an intermediary view transformation between the projections is omitted. For simplicity, a symmetrical perspective projection matrix is used with $r = t = 1$ and $l = b = -1$. Although the result is the same for any valid values of those parameters.

▶ **Proposition A.1.** *Given two distinct projection matrices P_s and P_w and a 3D point x, are the two following sequences of operations equivalent?*

$$P_w x \xRightarrow[\textit{divide}]{\textit{Persp.}} x' \to P_s x' \xRightarrow[\textit{divide}]{\textit{Persp.}} r_1$$
$$P_s P_w x \xRightarrow[\textit{divide}]{\textit{Persp.}} r_2$$
$$r_1 \overset{?}{=} r_2$$

Proof.

$$P_w x = \begin{bmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & \frac{-f-n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} nx \\ ny \\ -\frac{2fn+fz+nz}{f-n} \\ -z \end{bmatrix} \xRightarrow[\textit{divide}]{\textit{Persp.}} \begin{bmatrix} -\frac{nx}{z} \\ -\frac{ny}{z} \\ \frac{2fn+fz+nz}{fz-nz} \\ 1 \end{bmatrix} = x' \tag{A.1}$$

$$P_s x' = \begin{bmatrix} n_s & 0 & 0 & 0 \\ 0 & n_s & 0 & 0 \\ 0 & 0 & \frac{-f_s-n_s}{f_s-n_s} & -\frac{2f_sn_s}{f_s-n_s} \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} -\frac{nx}{z} \\ -\frac{ny}{z} \\ \frac{2fn+fz+nz}{fz-nz} \\ 1 \end{bmatrix} = \begin{bmatrix} -\dfrac{nxn_s}{z} \\ -\dfrac{nyn_s}{z} \\ -\dfrac{\dfrac{(f_s+n_s)(2fn+fz+nz)}{z(f-n)}+2f_sn_s}{f_s-n_s} \\ -\frac{2fn+fz+nz}{fz-nz} \end{bmatrix} \xrightarrow[divide]{Persp.}$$

$$\xrightarrow[divide]{Persp.} \begin{bmatrix} \dfrac{nxn_s(f-n)}{2fn+fz+nz} \\ \dfrac{nyn_s(f-n)}{2fn+fz+nz} \\ \dfrac{n_s(2fn+fz+nz)+f_s(2z(f-n)n_s+2fn+fz+nz)}{(f_s-n_s)(2fn+fz+nz)} \\ 1 \end{bmatrix} = r_1 \quad \text{(A.2)}$$

$$P_sP_wx = \begin{bmatrix} n_s & 0 & 0 & 0 \\ 0 & n_s & 0 & 0 \\ 0 & 0 & \frac{-f_s-n_s}{f_s-n_s} & -\frac{2f_sn_s}{f_s-n_s} \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & \frac{-f-n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} =$$

$$= \begin{bmatrix} nn_s & 0 & 0 & 0 \\ 0 & nn_s & 0 & 0 \\ 0 & 0 & \dfrac{(f+n)n_s+f_s(f+n+2(f-n)n_s)}{(f-n)(f_s-n_s)} & \dfrac{2fn(f_s+n_s)}{(f-n)(f_s-n_s)} \\ 0 & 0 & \frac{f+n}{f-n} & \frac{2fn}{f-n} \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} =$$

$$= \begin{bmatrix} nxn_s \\ nyn_s \\ \dfrac{n_s(2fn+fz+nz)+f_s(2z(f-n)n_s+2fn+fz+nz)}{(f-n)(f_s-n_s)} \\ \dfrac{2fn+fz+nz}{f-n} \end{bmatrix} \xrightarrow[divide]{Persp.}$$

$$\xrightarrow[divide]{Persp.} \begin{bmatrix} \dfrac{nxn_s(f-n)}{2fn+fz+nz} \\ \dfrac{nyn_s(f-n)}{2fn+fz+nz} \\ \dfrac{n_s(2fn+fz+nz)+f_s(2z(f-n)n_s+2fn+fz+nz)}{(f_s-n_s)(2fn+fz+nz)} \\ 1 \end{bmatrix} = r_2 \quad \text{(A.3)}$$

$$r_1 = r_2$$

□

List of acronyms

API	Application Programming Interface
CTU	Czech Technical University in Prague
DIWNE	Dear ImGui Wrapper Node Editor
DPI	Dots Per Inch
FEL	Faculty of Electrical Engineering
FIT	Faculty of Information Technology
GLFW	Graphics Library Framework
GLM	OpenGL Mathematics
GUI	Graphical User Interface
I3T	Interactive Tool for Teaching Transformations
NDC	Normalized Device Coordinates
RAII	Resource Acquisition Is Initialization
UI	User Interface

Glossary

affine transformation

A type of geometric transformation that preserves points, straight lines, and planes. It includes operations such as translation, scaling, rotation, and shearing. Affine transformations maintain parallel lines but not necessarily distances and angles. 34, 38, 42

camera frustum

The 3D volume captured by the camera. Sometimes referred to as the viewing frustum or viewing volume. The term frustum refers to the shape of the captured volume which is a quadrilateral truncated pyramid when using the perspective projection. However at least in this text, the viewing volume is still referred to as a frustum even in cases when it takes shape of a simple rectangular cuboid, like when using the simpler orthographic projection for example. 4, 33

clip space

The coordinate system projection transformations map into from view space before performing the perspective divide. The divide doesn't need to be performed when not using the perspective projection, in which case the clip space and normalized device coordinates are equivalent. 38, 42

mockup testing

Mockup testing involves testing a mockup, which is an early prototype or a simplified representation of a product, to evaluate its design, usability, and performance. 27, 31

node editor

A user interface component that is made out of draggable nodes on an infinite canvas that can be connected together to form a graph. 1

normalized device coordinates

The coordinate system projection transformations map into from the view space. If the w component of a transformed vector is not equal to one, like it is the case after applying a perspective projection, a perspective division must be performed to reach NDC. The space before applying the perpsective divide (or homogeneous division [14, p. 169]) is called *clip space* [14, p. 141]. Normalized device coordinates are usually followed by a fixed function viewport transformation done in hardware that maps vertices into the final screen space coordinates, on which rasterization is performed. 38, 40, 62, 66, 81

projection matrix

A transformation matrix used to map 3D coordinates from view space to normalized device coordinates. 33

view matrix

A view matrix is a transformation that converts world coordinates into view space coordinates (determining the observer's position and setting the viewing direction. [20, p. 307] 33, 38

Bibliography

1. FOLTA, Michal. *Teaching of Transformations*. 2016. Available also from: http://hdl.handle.net/10467/64836. Master's thesis. FEL CTU.
2. HERICH, Martin. *Restructuralization of Interactive Tool I3T for Teaching Transformations and Reimplementation of User Interface Using Dear ImGui Library*. 2021. Available also from: http://hdl.handle.net/10467/96746. Bachelor´s thesis. FEL CTU.
3. RAKUŠAN, Dan. *Scene view for the I3T application*. 2023. Available also from: http://hdl.handle.net/10467/109656. Bachelor´s thesis. FIT CTU.
4. GRUNCL, Daniel. *Transformation manipulators and scripting in I3T*. 2021. Available also from: http://hdl.handle.net/10467/94657. Bachelor´s thesis. FEL CTU.
5. CORNUT, Omar. *Dear ImGui Bloat free Graphical User interface for C++* [online]. 2014. [visited on 2025-01-19]. Available from: https://github.com/ocornut/imgui.
6. HOLEČEK, Jaroslav. *Adaptive learning in I3T software for education of geometric transformations*. 2023. Available also from: http://hdl.handle.net/10467/107077. Master's thesis. FEL CTU.
7. HÁLOVÁ, Barbora. *User research and UX review of the I3T app. Design and prototyping to improve the usability of the I3T app*. 2025. Bachelor´s thesis. FEL CTU.
8. HERICH, Martin. *Scripting in I3T and automating GitHub publishing*. 2025. Available also from: http://hdl.handle.net/10467/120310. Master's thesis. FEL CTU.
9. CICHOŃ, Michał. *Node Editor in ImGui* [online]. 2016. [visited on 2025-01-19]. Available from: https://github.com/thedmd/imgui-node-editor.
10. FLANAGAN, David. *JavaScript: the definitive guide*. " O'Reilly Media, Inc.", 2011.

11. CICHOŃ, Michał. *ImGui Stack Layout* [online]. 2025. [visited on 2025-03-23]. Available from: https://github.com/thedmd/imgui/tree/feature/layout-external.

13. SEGAL, Mark; AKELEY, Kurt. The OpenGL® Graphics System: A Specification (Version 4.6 (Core Profile)-May 5, 2022). *The Khronos Group.– 2006-2022 (https://registry. khronos. org/OpenGL/specs/gl/glspec46. core. pdf)*. 2022.

14. SHIRLEY, Peter; ASHIKHMIN, Michael; MARSCHNER, Steve. *Fundamentals of computer graphics*. AK Peters/CRC Press, 2009.

15. DE VRIES, Joey. Learn opengl. *Licensed Under CC BY*. 2015, vol. 4.

16. JOSH'S CHANNEL. *In Video Games, The Player Never Moves*. 2022. Available also from: https://www.youtube.com/watch?v=wiYTxjJjfxs.

17. STROUSTRUP, Bjarne; SUTTER, Herb, et al. *C++ Core Guidelines* [online]. 2025. [visited on 2025-05-16]. Available from: https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines.

18. BROWN, Wayne. *LearnWebGL - Perspective Projections* [online]. 2025. [visited on 2025-04-30]. Available from: https://learnwebgl.brown37.net/08_projections/projections_perspective.html#building-the-prospective-projection-transform.

19. KHRONOS GROUP, The Khronos® Vulkan Working Group. *Vulkan® 1.4.315 - A Specification (with all registered extensions)* [online]. 2025. [visited on 2025-05-09]. Available from: https://registry.khronos.org/vulkan/specs/latest/pdf/vkspec.pdf.

20. ŽÁRA, Jiří; BENEŠ, Bedřich; SOCHOR, Jiří; FELKEL, Petr. *Modern Computer Graphics (2nd edition)*. Computer press, 2005. ISBN 80-251-0454-0.

Image sources

12. DE VRIES, Joey. *OpenGL coordinate systems* [online]. 2017. [visited on 2025-05-16]. Available from: https://learnopengl.com/Getting-started/Coordinate-Systems. Licensed under CC BY-SA 4.0 (http://creativecommons.org/licenses/by/4.0/).

Contents of the attachment

The attached medium contains relevant parts of the source code of the application. An executable version of the application for Windows is located in a compressed archive in the `bin` directory.

The source code does not contain all necessary files to build the application. The application is, however, open source and published at the following GitHub repository:

https://github.com/i3t-tool/i3t

A more recent version of the application is usually available at the internal CTU FEL GitLab repository:

https://gitlab.fel.cvut.cz/i3t-diplomky/i3t-bunny

- readme.txt ....
- bin ....
  - I3T-binary.zip .... Archive with the Windows executable
- scenes .... Additional camera tracking scenes
- src .... Source code of the application
  - Source .... Main C++ source code directory
  - Test .... GTest test cases
  - Data
    - Shaders. .... Used GLSL shaders