Scientific Visualization, 2025, volume 17, number 1, pages 97 - 113, DOI: 10.26583/sv.17.1.08

Grafix – a Networked Visualization Technology

P.A. Vasev[1,A]

Institute of Mathematics and Mechanics named after. N.N. Krasovsky Ural Branch of the Russian Academy of Sciences, Ekaterinburg, Russia

[1] ORCID: 0000-0003-3854-0670, 0000-0003-3854-0670

Abstract

The paper proposes a technology for managing views, such as user interfaces and visualization scenes. The technology is based on network interaction. This allows implementing software components responsible for visualization in some languages and libraries, and the general layout and logical parts in others. Due to this, the project can be launched in different graphical environments (for example, web and desktop applications), attract a wider range of developers due to the use of well-known languages, and ported between different generations of graphical technologies. The paper presents a software model, describes its implementation, and provides examples.

Keywords: visualization, software architecture, user interfaces.

1. Introduction

Graphical interface technologies [1] and libraries are changing rapidly. This means that a project written 10 years ago is becoming obsolete. Obsolescence means that new technologies are emerging, with more logically verified program interfaces, with better imagery, with support for modern equipment, etc. For example, the once impressive QML technology has been replaced by new technologies such as React or Kotlin Compose Multiplatform.

But the same thing will happen with these new technologies. The important aspect of many of these technologies is that they offer an end-to-end solution that covers all the major layers of the user program. This means that if a program is implemented using these technologies, it will soon become obsolete along with these technologies. This is a problem.

What to do? Humanity solves this problem by gradually moving towards dividing graphic programs into layers and then trying to implement these layers on different technologies. For example, the following technologies rely on such a division: the X Window System, the model-view-controller paradigm, client-server web technologies and their variations such as Ruby on Rails, and the already mentioned QML.

The division into layers is observed as follows:

1. Graphical layer – contains user controls, multimedia, so on, i.e. components that directly interact with the windowing system and graphics drivers.
2. Logical layer – higher-level abstractions in which application models are programmed. This layer relies on the graphical layer and uses it in its algorithms.
3. Composition layer – links elements of graphic and logical parts into larger abstractions and forms the final appearance of the program.
4. Additional application parts and libraries.

The existing solutions are not ideal. Many technologies are still cross-layer, i.e. they assume the implementation of all layers within one ecosystem. Others are quite complex for programming the interaction of layers, for example, web technologies imply the use of low-level protocols (Websocket, Server sent events, etc.).

One of the cardinal methods for solving the problem is to move to another level of abstraction. For example, in the SciVi environment [2] it is proposed to describe the subject area in

the form of an ontology, and then the environment generates codes for all the above-mentioned layers and links between them. This means that if one needs to change layers technologies, it is enough to update adapters that generate layers code, and the program will remain technologically fresh.

Another possible solution is an approach where layers are separated explicitly and interact using network technologies. This allows layers and even parts of layers to be programmed in different programming languages. This, in turn, allows programs to be compiled from different languages and, what is especially valuable, to update layers independently of each other. The key requirement is that the interaction of layers should be convenient for programming.

This approach is used, for example, in the Logos platform and is described in [3].

This paper is also devoted to the study of this approach. It should be noted that each technology implies a certain software model (entities, their interrelations and interactions). The paper presents an original model, which, in the author's opinion, solves the problem.

2. The model

This is the model for management of visual entities, which allows to implement visual interfaces and any visualization using networked layered approach described in introduction. The entities of the model and their interaction are as follows**.**

2.1. Model entities

- A **component** is a logical process in the sense of Hoare processes [4]. Components in the context of this work are understood to be graphical. For example: a button, a slider, a line, a graph, a three-dimensional object, a container. The "parent-child" relationship is introduced over the components [1].
- An **executor** is an operating system (OS) process running on some hardware. Components are executed inside executors. For example, graphical components can be executed inside executors associated with the OS window subsystem.
- **Channel** is an entity for data transfer. Following operations are introduced: 1) the operation of writing *message* to a channel, 2) the operation of subscribing (reacting) to messages in a channel (see below).
- The channels are divided into **local** and **global**. Each component is associated with a set of input and output local channels through which the component interacts with the outside world. Local channels are mapped to a local namespace associated with the component. Global channels are mapped to a namespace that is common (global) to all participants (executors, components).
- A **link** between two channels — a special process that transmits messages written to the first channel to the second channel. The connection can be between both global and local channels, including mixed ones.

2.2. Interaction operations

The list of operations for interaction in the model is as follows:

Opening of a global channel

$$open\_channel: channel\_id \rightarrow channel$$

where *channel_id* is the name of a channel in the namespace common to all executors. Returns a local object for operations with the channel (local in the sense of being available for interaction in the executor and in a programming language in which the *open_channel* operation was performed).

---

1 This hierarchy is due to the fact that it is used in graphical user interfaces and visualization scenes. In interfaces hierarchy is used to arrange objects in the plane of the screen. So-called "containers" are created, for example "column", "row", and objects are recursively placed in such containers. In three-dimensional visualization scenes, a tree structure is used to form matrices for transforming the position of objects and managing other properties.

Creation of local channel:

create _local_channel: void → channel

Creates a local channel and returns a local object for operations on this channel.

Sending a message to channel:

put: channel, msg → void

where *channel* is a local object for channel operations, *msg* is a message in some form. Technically, depending on the implementation, the *put* operation may support additional notifications about the progress of sending a message, for example via promises.

Creating a reaction:

react: channel, fn → reaction

where *channel* is a local object for interaction with the channel, *fn* is a function that will be called by the environment when someone writes a message to the channel, *reaction* is a local object for ongoing reaction maangement. When a message is written, all subscribed parties receive it, e.g. functions of all created reactions are called. Messages can be written to a global channel in one executor, and these messages can be received in the same or other executors, which ensures network interaction. The name *reaction* is not chosen by chance, it corresponds to the term "reaction" from the Lingua Franca model [5]. There are similar terms, for example, "message subscription" in the more general "publisher-subscriber" model.

Creating a link:

create_link: source_channel, target_channel → local_link_object

where *source_channel* is the source channel, *target_channel* is the destination channel, *local_link_object is* the local link object for subsequent operations with it (e.g. deletion). After creating a link, all writes to the source channel result in copying these messages to the destination channel. This also applies to global channels. After creating a link, if a write is made to the source channel on any executor on any computer, this message will be transmitted to all links created using this operation, including other executors on other computers.

In addition to the operations listed above, the model also introduces reverse operations, which are omitted in this text to shorten the description: close the channel, unsubscribe from messages, delete a connection, delete a reaction.

Thus, a model for communication of different logical processes grouped by executors (OS processes) and running on different computers is proposed. This model was developed earlier within the framework of the Parallel Programming Kit (PPK) project [8,9], published on the website github.com/pavelvasev/ppk.

2.3. Graphical interface management operations

The following set of operations is proposed for managing a graphical interface. It is assumed that a logical layer of the program requests the execution of these operations, and they are executed in a graphical layer.

Creating a component tree:

create: descr, parent_id → void

where *descr* is the description of the component being created, *parent_id* is the identifier of the component to whose tree the component being created should be added (if necessary).

The *descr* description is recursive and defines a component and a tree of nested components within a parent-child hierarchy [2]. The description structure is:

- *type* — identifier of the type of the component being created. The executor of a graphical layer must be able to create components of this type.
- *id* — component identifier (optional parameter), in the namespace of component identifiers of a graphical layer executor.
- *params* — component parameters (will be passed to the component constructor).

---

2 The tree-like nature of the *descr* description is due to 1) the need to work with hierarchies in principle, see the previous footnote, and 2) optimization, since a typical user interface contains hundreds and thousands of components, and creating them individually would be wasteful.

- *links_in* — a list of links between global channels and local channels of the component, that is, this is information incoming into the component.
- *links_out* — a list of links between the component's local channels and global channels, i.e. this is information outgoing from the component.
- *tags* — a list of tags to which the component is associated. A tag is a text string; the presence of tags allows for bulk operations to be applied to components, which significantly simplifies and optimizes programming [3].
- *items* — a list of descriptions of nested components (descriptions of the same structure as *descr*).

Removing the component tree:

$$remove: id \rightarrow void$$

where *id* is the identifier of the component being removed. All of its nested components are removed along with the component.

Passing a message to all components by tag:

$$put\_by\_tag: tag, channel\_name, msg \rightarrow void$$

where *tag* is a label (string), *channel_name* is the name of the channel in the local component namespace, *msg* is the message. The operation finds all components that match the *tag* in the graphical layer memory, selects a local channel named *channel_name* for each component, and passes the *msg* message to it.

2.4. Workflow

The order of execution is as follows. OS-level processes - executors (on a local machine or on a set of machines connected by a network) are launched in any convenient way:

1. Logical layer executors. In a simple case, this may be a single OS process.
2. Graphical layer executors capable of creating user interface components and/or visualization scenes. In a simple case, this can also be a single OS process, launched, for example, by the logical layer (1).

Layer executors are considered to be started asynchronously. After the next graphical executor is started, it sends a message to the global channel named **gui_attached**.

A logical layer code pre-subscribes to messages in the *gui_attached* channel, and thus receives information about launching the graphical layer executors. Next, the logical layer initiates the creation of the necessary graphical component trees in the connected graphical executor, using the *create* operation specified above.

If necessary, the logical layer subscribes to messages in global channels associated with graphical components to receive signals from the user. Receiving these signals, the logical layer sends messages to channels associated with the inputs of graphical components, creates new components, and so on. The program operates according to the algorithm described in the logical layer, and its visual representation is implemented in the graphical layer.

The proposed workflow allows to implement various configurations of executors:

1. In a simple case, as noted above, this could be one logical layer executor and one graphical layer executor.
2. There may be several graphical layer executors if necessary, for example in multi-monitor configurations, Cave-type systems, in a multi-user virtual reality environment in helmets, etc.
3. There may also be several executors of the logical layer. For example, one executor can be allocated for each user; executors can work on supercomputer nodes if the intention is to create a system of the “virtual test stand” class [12], etc.
4. Graphical layer executors can be disconnected and connected as needed. Multi-user work is possible and naturally supported, but this requires special consideration on the part of the logical layer (for example, if different levels of user rights are required).

Note that executors can be written in different languages and different software platforms. Moreover, this is considered the most promising direction, since some layers are conveniently and effectively expressed in some languages, and others in others.

For example, one can use a QML graphical layer executor, and a Python logical layer executor. One can then replace the QML graphical layer with a layer implemented using web technologies. With such a replacement, what is important is that the logical layer program, ideally, will remain unchanged. One can then replace the logical layer program, for example, by rewriting it in another language. One can replace not the entire logical layer, but a part of it, for example, by moving a part to another executor or language.

It is necessary to note the role of links between component's local channels and global channels. This solution has proven its convenience and efficiency over time. In contrast, in earlier versions of the model, component's channels were global. It was necessary to assign unique id's for all components to support this, and also introduces unnecessary overhead, because system spends resources for every global channels while most of them are inactive.

The concept of a connection between channels in the presented model is a higher- level abstraction, the same as a channel and a component. The program is a directed graph of processes implementing components, the edges of which are the links between the channels of these processes (implemented through links with global channels).

The highly efficient implementation of local links is possible. For example, let a pair of components be created on some executor, and their two channels are linked via a global channel. This chain of links can be technically implemented in such a way that writing to the channel of the first component will lead to a direct call of the reaction functions on the channel of the second component, without network interaction (for more details, see [8,9]). This allows describing sets of components and links between them externally, and their execution will be as efficient as the usual interaction of codes inside the OS process.

3. Grafix library for Python language

A Python library named Grafix implements the model proposed in this paper. The source codes are published on the Internet at hub.mos.ru/vasev/grafix.

For interaction operations (Section 2.2) Grafix uses the Parallel Programming Kit (PPK) library. It provides network communication of software components. An example of working with the PPK environment is shown in Fig. 1.

```
import ppk
import asyncio

async def main(rapi): # entry point to the program
print("started, PPK url is",rapi.url)
def fn1 ( msg ): # reaction code
       print("see message",msg)
ch = rapi. channel ("channel1") # open global channel
ch. react ( fn1 ) # create a reaction
ch. put ( "hello world") # send message
await asyncio.Future() # transition to asynchronous mode

ppk.start( main, url=None ) # start the PPK messaging environment
```

Fig. 1. An example program in the Parallel Programming Kit environment. The program starts, launches the environment, receives control in the main function, creates a *reaction* to messages in the channel named *channel1,* and sends a message to this channel with the value "*hello world* ". The result will be printed: "*see message hello world*". This is a simple example, but it is interesting because any other program can then connect to this program (using the connection address *rapi.url*) and interact with it.

The Grafix library consists of two main parts:

1. Graphical layer and launch subsystem.
2. A set of graphical layer components.

Currently, one type of graphical layer is offered, based on the web interface. Also experiments have been conducted with a graphical layer in the form of Qt/QML, but this option is not yet included in the library.

Let's look at these parts in more detail.

3.1. Web-based graphical layer

The algorithm for interaction with the graphical layer is as follows. The user program calls the web interface launch function:

grafix.web.start(rapi, plugins_list) → bool

where *rapi* is a pointer to the PPK API, *plugins_list* is a set of plugins for the layer (see below).

This function:

1. Launches a web server on a found free IP port;
2. Generates a web page of the program's graphical interface (see below);
3. Sends a command to the OS to open a web page (2), which may involve launching a web browser or using the currently running web browser.

Next, the user program waits for a message in the *gui_attached* channel from the web page (2) (according to the workflow in section 2.4), and in response to this message sends a description of a interface to a graphical layer. An example of calling the *grafix.web.start function* and the skeleton of an application using Grafix are shown in Fig. 2.

```
import ppk
import grafix
import grafix.web
import grafix.dom
import grafix.threejs
import asyncio

async def main(rapi):
def gui_attached( msg ):
    gui_channel_id = msg["id"] # graphical layer control channel identifier
    d = { … } # interface description
    rapi.channel ( gui_channel_id ).put({"cmd":" create ", "descr": d, "parent_id": "root"})
rapi.channel ("gui_attached"). react (gui_attached)
await grafix.web.start (rapi,[grafix.dom,grafix.threejs])
await asyncio.Future() # transition to asynchronous mode

# launching the PPK messaging environment
ppk.start(main)
```

Fig. 2. An example of a program using Grafix library. The program starts, starts the PPK environment, starts the graphical layer (using the *grafix.web.start* call), receives a gui_attached message from graphical layer, and sends it a description of the interface in JSON format. The created interface will be connected to the top level of the window, since the parent element parent_id with the value "root" is specified.

3.2. Main web page

This web page is the entry point for a browser and displays a program interface. To implement its functionality, the web page:

1. Loads CSS styles and Javascript libraries required for the operation of graphical components. Scripts may be loaded using both script tags and import maps.
2. Calls plugin initialization functions.
3. Transfers control to the main code of the web page.

The main code of the main web page:

1. Connects to the PPK environment. The PPK connection address is embedded in the page code when it was generated.
2. Generates a unique random channel identifier through which the web page will receive Grafix commands, and sends this identifier to the global *gui_attached* channel.
3. Opens a channel and waits for commands with operations to control the graphical interface (according to section 2.3). Upon their receipt, executes them.

Let's consider how the *create* operation is implemented to create components. When a request for this operation is received, the main code of the web page finds and executes the component creation function, which creates the component of the requested type. Information about the correspondence of type identifiers and component creation functions is stored in a special dictionary. This dictionary is filled with plugins at the initialization stage.

The component creation function receives all the information from the *descr* description as input. In addition to creating the component object, this function is also responsible for all other processing of the description (setting parameters, connections, etc.). If necessary (at the developer's discretion), this function:

1. Sets the value of the parameters from the *params* field of the description.
2. Links the component's local channels to global channels (*links_in* and *links_out* fields).
3. Creates nested components.

After the component creation function is completed, the main web page code receives the Javascript object of the created component. Using this object, the application code replenishes the dictionaries of correspondences between tags (the tags field) and created components, and the dictionary of component identifiers. This information is then used to delete components and for bulk operations with them.

3.3. Plugin system

A **plugin** is a program code that is embedded in a system and changes its behavior. The difference between plugins and libraries is that a library is passive: an external code determines when to call the library functions. A plugin, on the contrary, receives control at a specified time and changes the system's behavior according to specified protocols. In this sense, it is active: it determined in the context of a plugin how it affects the system, and as a result, at what points in time the system's behavior changes.

The Grafix library provides plugins with *system control points* that affect the behavior of the graphical layer. In the web-based graphical layer, these are:

- *add_head* – the ability to supplement the contents of the head section in the html code of the web page of the graphical interface. This allows the plugin to add the necessary style and script tags for loading libraries.
- *add_import_map* – the ability to supplement the composition of library import maps. This allows the plugin to connect javascript libraries to the application in the so-called “ES6 imports” style.
- *add_script* – the ability to supplement the application initialization code. In it, the plugin can, for example, transfer control to itself at the Javascript level or perform some other actions.
- *add_routes* – the ability to add routes to the Grafix web server route map, which allows the plugin to provide web access to its directory on disk, or to add its own web request handler at specific URLs, etc.

The list of plugins is passed to the Grafix system when the *grafix.web.start* function is called (see section 3.1). Among other things, this function calls the plugin initialization functions.

The typical workflow in the system from the point of view of plugins is as follows:

1. The program starts and at a certain point the plugin initialization functions are called.

2. In the plugin initialization function, the plugin embeds the required functionality into the system using system control points.
3. The user opens the Grafix application web page.
4. During web page initialization, the libraries requested by the plugin are loaded.
5. The web server provides access to these library files, which it is able to do because the plugin provided the route information via the *add_routes* control point in step (2).
6. During further initialization of the web page, the plugin receives control (if it has requested it earlier via the *add_script* control point), during which it can inform the web application (i.e. the Grafix graphical layer) about the types of components and the functions for creating them.
7. Next, the graphical layer application waits for commands from the logical layer, including *create* operation, and executes them as described in section 3.2.

3.4. List of built-in plugins

An important advantage of the Grafix library is that it provides the ability to embed any necessary types of graphical layer components using plugins. It is believed that the library user can form the necessary set of his own plugins and use them in different projects.

To make it easier to get started with Grafix, it includes a number of plugins that the user can immediately use in their projects.

- **grafix.dom –** provides a set of component types for implementing the user interface. Among the components: *text, button, checkbox, slider, numfield, filefield, textfield, combobox,* and others. Containers for organizing the user interface are also offered – *row*, *column*, and *grid*, which are implemented using CSS Flexbox technology. An interesting feature of the plugin is the modifier approach, which is used to manage the visual aspects of controls [3]. Modifiers resemble CSS, and in a more explicit form are found, for example, in Kotlin Compose and Yandex Divkit. A modifier component of a given type is associated with a set of target components that it affects, and the modifier changes the visual aspects of the display of these components according to its type. Thus, user interface components do not need to offer many properties for managing the visual display. These properties are essentially "moved out" to modifiers, which the user connects to the component as needed.
- **grafix.dom_screenshot** – implements the ability to obtain a screenshot of the application. Obtaining screenshots in web technologies is currently a non-trivial task that requires special libraries.
- **grafix.threejs** – three-dimensional graphics based on the Three.js library (e.g. WebGL). Main components: *view* (scene and related graphic output area, built into dom plugin containers), *camera* (camera control), *lines*, *points*, *mesh* – graphic primitive output components. Graphics display is formed as a tree hierarchy with the *view* component at the root and primitive output components at leaves. Primitive output components provide channels for specifying vertex coordinates, colors, etc. A logical layer has the ability to send large data arrays to these channels. In addition, special channels have been introduced for modification of a displayed data.
- **grafix.echarts** – drawing charts based on the Apache Echarts library. Main components: *chart* — chart output area, *chart_line* — adds one chart output to the chart, depicted by a line. User can add several chart lines to the chart. The *Apache Echarts* library was chosen because it supports the display of a large number of values, interactively and without delays. The output of 1 million values has been tested.
- **grafix.chatbot** – a graphical component depicting a chat. The program can communicate useful information to the user in the chat in the form of multimedia messages, simi-

3 The classic method of managing visual aspects of components is to set their properties (text color, background color, border color, etc.). This approach is inconvenient, inflexible, and results in bulky components. For example, HTML elements or QML elements contain hundreds of such properties.

lar to how it is done in chatbots and messengers. The user can send text messages to this chat, which are transmitted to the application for parsing and response.

4. Experimental results

Several applications using Grafix technology have been implemented. Let's look at the results of their development.

4.1. Specialized GIS

N. N. Krasovsky Institute of Mathematics and Mechanics is developing various software algorithms for processing geospatial data. For their debugging, special graphical interfaces are created, a kind of specialized geographic information systems (GIS). It was decided to conduct one of such works using Grafix technology.

Python was chosen for the logical layer because of its widespread use. It was expected that it would be relatively easy to add additional developers to a Python project.

The graphical layer was implemented in QML and Qt: objects of the existing technological backlog were wrapped in shells and presented as Grafix model components. Entities and operations of the model were implemented as C++ objects with bridges in QML using Qt methods (slots, signals). The network part in C++ was implemented based on the Mongoose library. [4] [5].

Examples of created components of the graphical layer (for implementing the user interface) on Qt:

- *row* – a component that acts as a container, placing nested components horizontally.

column – a component that acts as a container and places nested components vertically.

- *dialog* – a component acting as a container, displays nested components in a separate window.
- *button*, *textfield*, *slider*, etc. are components that implement elements of the graphical user interface.
- *map_group* - container component for holding nested map layers components, interacts with the user's mouse and keyboard and displays the contents of nested components one over another.
- *show_image* – component for displaying raster and vector maps, used as a nested component of *map_group*.
- *show_vector* – component for displaying vector data, e.g. features, used as a nested component of *map_group*.

The logical layer, as noted, was implemented in Python 3. The layer defines the composition and grouping of the graphical layer components, sends data to the channels associated with these components, and responds to messages in their output channels (e.g., mouse movements, clicks on map layers, button presses, etc.).

An example of using Python code is shown in Fig. 3, the result of the work is shown in Fig. 4.

Later, the program was switched to a web-based graphical layer; the OpenLayers library was used to display maps.

---

4 This implementation method turned out to be very cumbersome. His motivation was to get an implementation for C++ "at the same time". This was achieved, but together with the "bridges" in Qt/QML, the code volume was impressive. If the project were to be done again, the author would prefer to do the entire implementation inside QML, for example, based on the QML Websocket library.

5 The Mongoose library has proven to be inefficient. For example, it copies incoming and outgoing data multiple times into intermediate buffers before passing it to the program or OS.

```
import ppk
import grafix
import grafix.web
import grafix.dom
import asyncio
async def main(rapi):
    await grafix.web.start(rapi,[grafix.dom])
    ############# forming the interface description – a table of 10 fields and buttons
    lst1 = [ grafix.node("h3",value="Specify parameters")]
    for x in range(0,10):
        d = grafix.node("row", items=[
               grafix.node( "text", value="Field "+str(x)),
               grafix.node( "numfield",id="field_"+str(x)),
               grafix.node("gap", value="5px"),
               grafix.node( "margin", value="2px")
        ])
    lst1.append(d)
    lst1.append( grafix.node("button",value="Let's GO!",links_out={"click":["go1","q2"]}) )
    lst1.append( grafix.node("css",value="align-items: center"))
    d = grafix.node("column",items=lst1)
    ### creating an interface on a graphical executor when it is connected
    def gui_attached(msg):
        gui_channel_id = msg["id"]
        rapi.channel( gui_channel_id ).put({"cmd":"create", "descr": d, "parent_id":"root"} )
    rapi.channel("gui_attached").react( gui_attached )
    ############## reaction to pressing a button
    def go(msg):
        print("LET'S GO!")
    rapi.channel("go1").react( go )
    await asyncio.Future() # transition to asynchronous mode
   # launching the PPK messaging environment
ppk.start(main)
```

Fig. 3. Example of code in Python. The code forms a description of the graphical components of the header, 10 fields and a button, performs an operation to create components according to this description, and adds a reaction to the button press output channel.

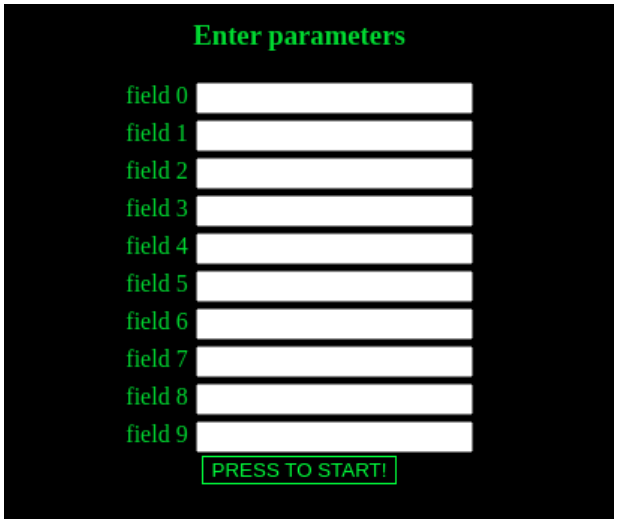

Fig. 4. The result of executing the code shown in Fig. 3.

4.2. Online visualization of parallel computing

A parallel version of a reinforcement learning algorithm is being developed. At the initial stage of development, it turned out that the version has a low parallelization coefficient. In order to understand the reasons for this, it was decided to visualize its operation. The following view was developed (see Fig. 5):

- each task (in the sense of the tasks of the task graph [6]) is displayed as a segment along the OX axis. The position of the segment on the X axis corresponds to the start and end time of the task execution, the position on the Z axis corresponds to the number of the parallel executor;

- additionally, data transfers between processes are shown, since the transfer can take a significant amount of time, and we would like to see this. The parallel version uses a master-worker scheme, and communication is carried out only between the master and the workers. Therefore, in order not to clutter the image, it was decided to show the transfer in vertical segments from the plane of the workers Y=1 to the plane Y=0, which means sending data to the input of the worker from the master and in the opposite direction.

The graphical layer is implemented using web technologies, using the *grafix.dom* and *grafix.threejs* plugins presented in section 3.4.

The logical layer was implemented in Python 3. This was due to the fact that training (single-thread and parallel versions) were developed in this language. The operations of the presented model for managing the three-dimensional scene were added to the existing program.

Specifically, the implementation is as follows. When the task calculation starts, the executor measures the current time. When it is finished, it calculates the time spent on the task. Then a message is sent to the "solved tasks" channel. A special module of the logical layer ("painter") processes these messages and forms the coordinates of the vertices for the scene. Then it sends these coordinates via messages to the "data modification" channels to the components of the graphical layer responsible for rendering.

The program starts, launches the PPK execution environment, proceeds to calculations and displays a web link to start the visualization. The user can open the visualization in the browser at any time using this link; the program updates the graphical representation as it works. Thus, a visualization of the parallel calculation was obtained in online mode [7], as the calculation progressed.

An example of the resulting visualization is shown in Figure 5. This visualization was used to understand the operation and subsequent optimization of the parallel algorithm.

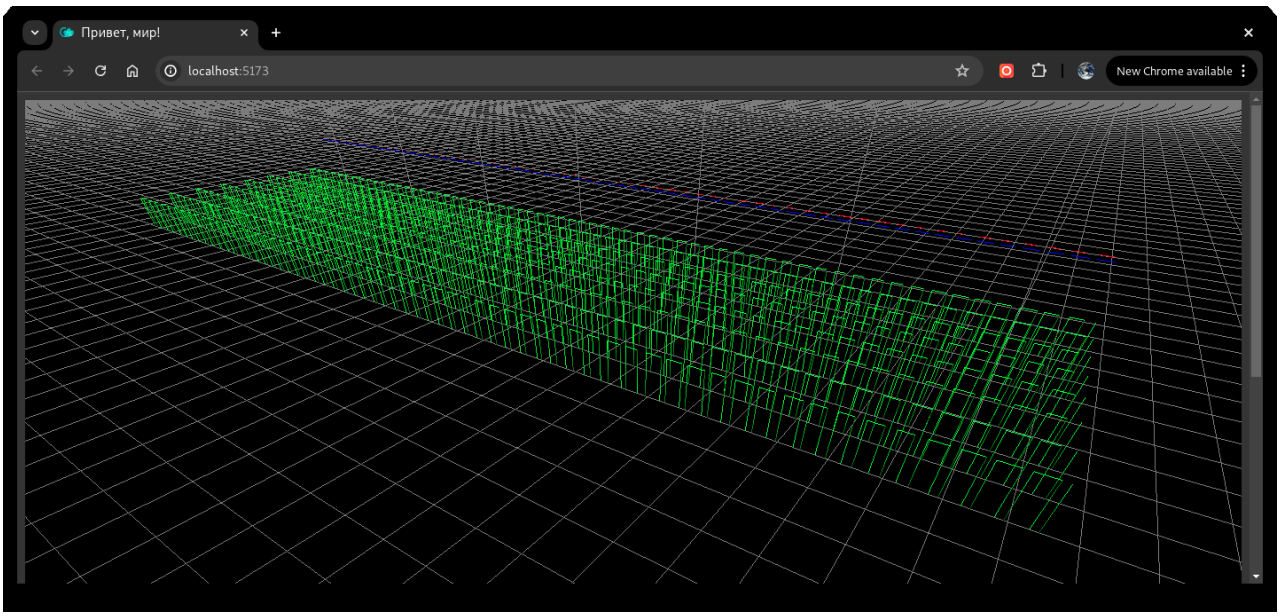

Fig. 5. An example of visualization obtained using Grafix technology. The work of 8 work processes is shown. Time goes from left to right. Workers are located each on their own Z axis (closer to and further from the screen). The horizontal line on the worker axis is the task execution. The vertical line is the worker receiving input data from the master and sending him the result. In the background, the blue segments are the master waiting for all the workers, the red ones are the master estimating the current situation.

4.3. Interactive 3D visualization

The task was to visualize the time sections of maximal stable bridges in linear differential games with a fixed termination time and a convex terminal target set [10,11], see Fig. 6. The development of views and the construction of geometric models were implemented by the au-

thors of mathematical methods – S. S. Kumkov and A. V. Mikhailov. In this case, Grafix technology used to interactive graphics display.

From a technical point of view, the task looks like the following. The data is presented in the form of a directory with subdirectories. Each subdirectory corresponds to a certain point in time. Each subdirectory contains a set of 3D image files in PLY format, which correspond to parts of the scene. It is necessary:

1. Be able to set a time point from those available.
2. Display all PLY files corresponding to this point in time.

The visualization program is implemented in Python, using a web browser as an interface. Its source code is available at hub.mos.ru/vasev/2025-bridges.

The general algorithm of the program is as follows:

1. Data directory is scanned and an array of times and lists of PLY files corresponding to the times are built.
2. Program waits for *gui_attached* message.
3. Upon receiving this message, a graphical interface is formed, which includes control elements (slider, animation checkbox, etc.). In addition, a 3D graphics scene is created and for each PLY file from the first moment of time, a Grafix *mesh* (a set of triangles) component is created, which is connected to the scene. An incoming link from the corresponding global channel is attached to each mesh object, which will allow their content to be managed later.
4. Global channel **current_t** is attached to the output channel of time slider. When the user changes the slider value, a message is sent to this channel.
5. The main program adds a reaction to messages in the **current_t** channel, which contains messages with time moment selected by user. The reaction code reads the PLY files corresponding to selected time moment. The vertex coordinates from these files are sent to global channels (individual for each file), the graphical layer receives them and routes to local channels of mesh components, and the view updates.

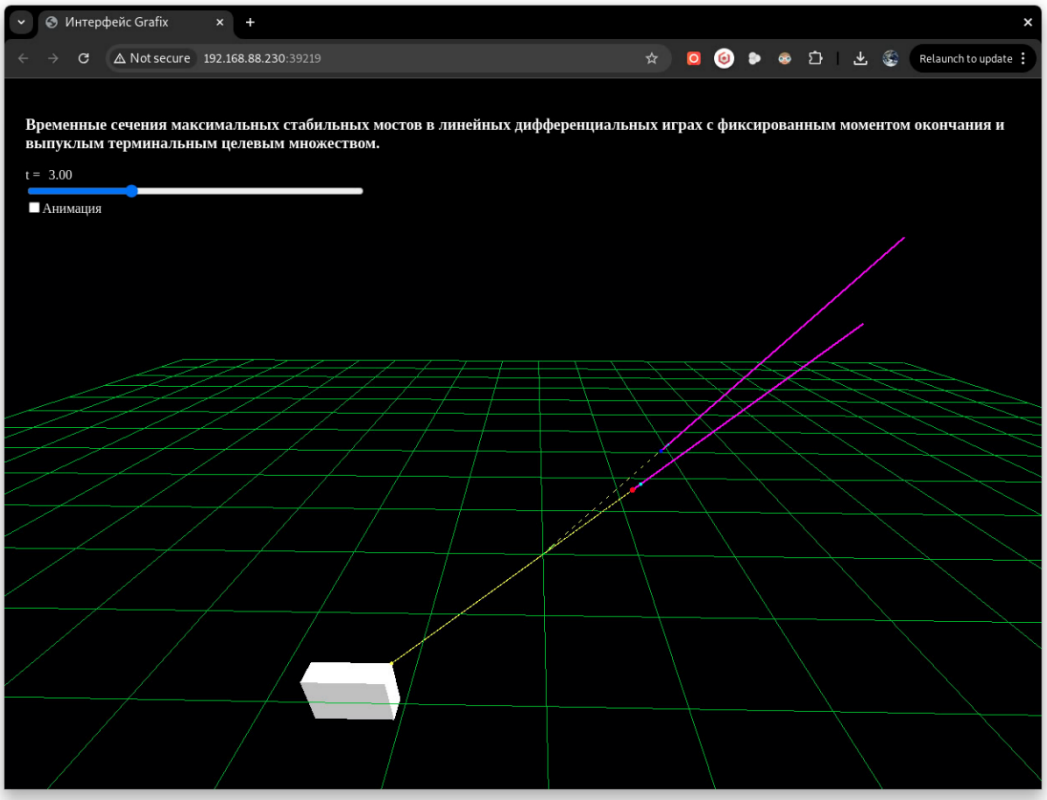

Fig. 6. An example of interactive 3D visualization obtained using Grafix technology. Time sections of maximum stable bridges are shown [10,11].

5. Discussion

5.1. Comparison with other approaches

The difference between the proposed model and the X Window System model is as follows. In the latter, the components of the graphical layer are executed in the main process of the program, and they send graphic primitives to programs running on the graphic equipment (to the "server" in X Window terminology). In the opposite direction, user commands are sent - signals from the mouse and keyboard. In the proposed model, it is not graphic primitives that are sent, but high-level messages for the components of the graphical layer that work in processes on the graphic equipment. High-level messages also move in the opposite direction ("button pressed", "slider value changed", etc.). In other words, X Window System uses very low-level components in a graphical layer, closer to rendering primitives.

Yandex Divkit offers a rich model of built-in dynamics of the graphical layer components. This is done because it implies greater autonomy of this layer from the logical layer, the "load and work" approach. In other hand, the Grafix model implies a closer, constant interaction of the graphic and logical layers.

The Grafix model borrows a number of ideas from the Logos model [3], including:

– When creating components, links between local component channels and global channels are placed into the description. This approach allows, among other things, programming the dynamics of interaction of graphical components immediately at the description level, by transmitting messages between components indirectly through global channels. The directions of links can be specified in both directions – from local to global and vice versa.

– Tags that allow each component to be included in several sets, and bulk operations on elements of such sets are supported. For example, the command “set the value X to the channel C” for all components marked with a given tag is possible.

Additionally, it can be noted that in Logos:

– the ability to upload component descriptions in JSON language placed in files has been provided;

– both versions of the graphical layer for web and desktop platforms are supported, with a large number of component types implemented;

– the work is actively used in practice, but, unfortunately, it is not available to third-party developers.

Other projects that use similar models include Plotly (Dash), which allows to build various complex display types from basic 2D and 3D display types (see Fig. 7).

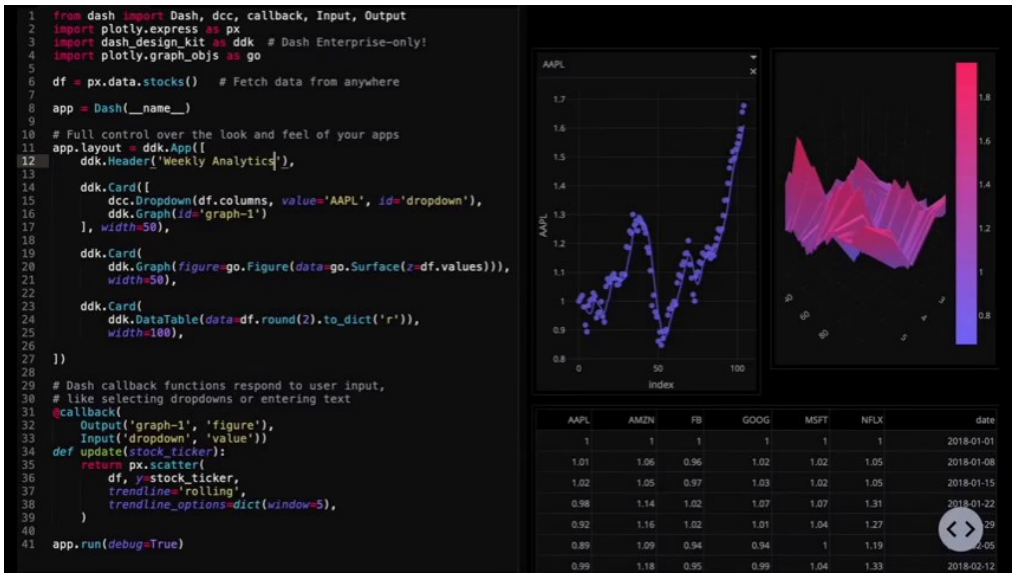

Fig. 7. An example of visualization using Plotly technology. The source code of the application is shown on the left, and the result of executing this code is shown on the right.

This project, like the Grafix library, uses a combination of Python and a graphical interface in a web environment. A user describes a display type in Python and then transfers this information to the web interface via Plotly's internal protocols. The differences from Grafix are that:

– Plotly offers a rich set of basic display types (graphs, charts, surfaces, etc.), i.e. components of visualization level. At the moment, there are not many visualization level components implemented in the Grafix, with hope that they will be created later in plugins.

– Plotly is more focused on data visualization and less on building user interfaces from controls (however such controls are implemented in Plotly). Grafix is more versatile in this sense, not focused on any particular kind of graphics or visualization.

5.2. Development Prospects

Among the prospects for the development of the model proposed in this paper, there are the following.

Visualization and analysis of links. When working with the proposed model, the developer may not understand the picture of all global channel links. This is unfortunately and apparently inevitable, since the model allows calling the operation of creating a link in arbitrary places in any programming languages. The global graph of channels and links between them is described in different files, in different places and at different times. On the one hand, this provides full flexibility of the solution, on the other hand, the overall picture is lost.

The correct solution has not been found at the moment. Among the known solutions:

1. Visualization of the picture of global channels and links obtained during execution. This option does not seem suitable for calm static analysis of programs.
2. Use of comments with machine-readable information. In practice, it turns out that not all links are marked, and thus the picture becomes incomplete.
3. There are approaches where connections are described in a programming language so explicitly that their visualization becomes possible. For example, the Lingua Franca language [5], which allows static analysis of component connections and has a built-in visualization tool, see Fig. 8.

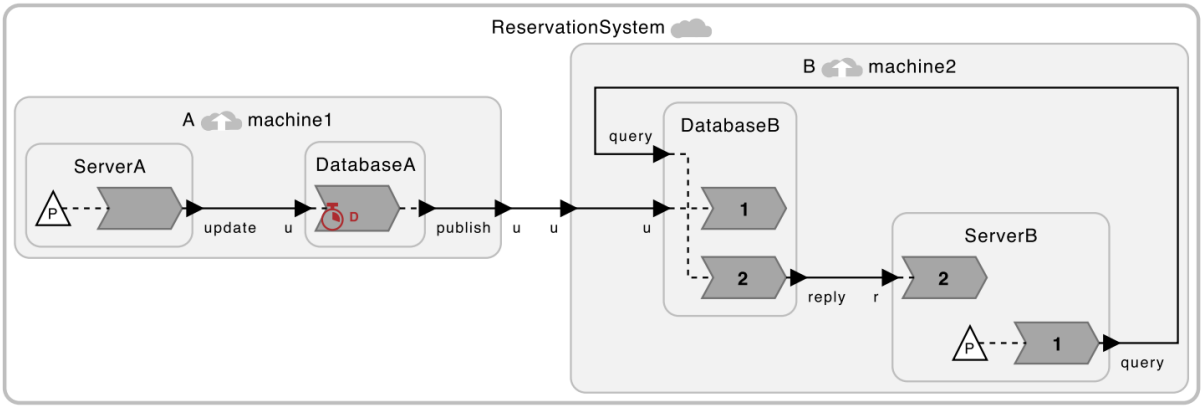

Fig. 8. An example of visualization of connections between components in the Lingua Franca language [5]. The images are built automatically using a tool that analyzes the source code of the program.

Another example is that in the Logos project, interfaces are described in JSON format, which includes links between component local channels and global channels. This means that it is possible to automatically analyze and visualize such links, similar to Lingua Franca.

Cloud publishing of application interfaces. The Gradio.app project contains an interesting idea. The user launches the application on his computer. The application (e.g. the logical layer) at startup connects to the cloud service, which publishes application's graphical layer in the form of a web page acessible via a special URL. The user receives this URL and

can publish it, e.g. transfer it to other users. These users connect via the URL and run graphical layer through the browser, which is actually linked via cloud to the application running on the user's computer. The idea looks attractive, including because the cloud service can be implemented in a closed enterprise circuit. This, in particular, increases the diversity of user interaction options, debugging, and adds other options. For example, the application can be launched on a supercomputer, and through the cloud service, access to its graphical interface is possible, which simplifies the online visualization HPC mode [7].

Multi-user mode. The proposed model allows several users to work with the logical layer of a program simultaneously. In this case, it is necessary:

1. Select the level at which instances of graphical interfaces are synchronized. This can be full synchronization, including all controls. In this case, users have a common display view in a single state. It can also be partial synchronization, when the common state contains only entities of the logical layer, and the state of the display view differs. Developing this idea, we can come to some other options, for example, when users are in a common virtual environment acting independently, and also may interact with each other, for example by sharing found computational artifacts.
2. If necessary, implement user identification. This will require introducing the concept of a user into the logical layer model, and it is possible to implement some model for granting access rights to actions, or maintaining a history of actions with their binding to the user. As a result, it is possible to build a full-fledged multi-user environment.

6. Conclusion

In this paper, a Grafix technology is introduced. It uses approach, when interaction between graphical and logical layers of programs is performed on a network level. This allows:

1. Not to depend on the development and obsolescence of technologies. When the technology on which the layer is written becomes obsolete, the layer can be rewritten. This primarily concerns the graphical layer. It is also possible to rewrite the logical layer, including gradually. The proposed technology is such that it provides the ability to work simultaneously in different OS processes and in different programming languages.
2. It is convenient to switch the implementation of the graphics layer if it is valuable for the task. For example, change the "web" to the "desktop" version, or the options from OpenGL to Vulkan.
3. Use the most convenient and appropriate languages for the task context to implement layers. For example, a graphical layer can be implemented using web technologies, and a logical layer can be implemented using Python, which is used in the examples presented.
4. Run graphical programs in a multi-machine environment in a "natural" way, thanks to a network protocol for interaction. For example, the graphical layer can be executed on executors on machines with monitors (e.g. *terminals*), and the logical layer on executors on other machines, including supercomputer nodes.

The technology has been tested on a number of applied tasks and has shown its efficiency. At the same time, it can be developed, some options are shown in the "Discussion" section. The source codes of the technology for the Python language are published on the Internet at hub.mos.ru/vasev/grafix.

The author thanks his esteemed colleagues from the Russian Academy of Sciences, Rosatom Corporation, Ural Federal University and other organizations for their support, fruitful discussion and synthesis of ideas.

References

1. V. L. Averbukh, Evolution of Human Computer Interaction // Scientific visualization 12.5: 130 - 164, DOI: 10.26583/sv.12.5.11.

2. Ryabinin K., Chuprina S. Ontology-Driven Edge Computing // Lecture Notes in Computer Science. – Springer, 2020. – Vol. 12143. – P. 312–325. DOI: 10.1007/978-3-030-50436-6_23.
3. Gubaidulina E. A. et al., A tool for developing specialized user graphical interfaces for complex mathematical modeling using the LOGOS software package // Abstracts of the XIX International Conference "Supercomputing and Mathematical Modeling", May 20-24, 2024, Sarov, p. 68. URL: https://lomonosov-msu.ru/archive/Lomonosov_2023/data/7815/151643_uid799493_report.pdf (in Russian).
4. CAR Hoare, Communicating Sequential Processes, December 4, 2022.
5. Marten Lohstroh, Christian Menard, Soroush Bateni, and Edward A. Lee. 2021. Toward a Lingua Franca for Deterministic Concurrent Systems // ACM Trans. Embed. Comput. Syst. 20, 4, Article 36 (July 2021), 27 pages. DOI: doi.org/10.1145/3448128.
6. Vasev P.A., Visualization of the parallel task scheduling algorithm // Proceedings of the 33rd International Conference on Computer Graphics and Machine Vision GraphiCon 2023, September 19-21, 2023, Moscow. P. 341-353. DOI: 10.20948/graphicon-2023-341-353.
7. Pavel Vasev, Analyzing an Ideas Used in Modern HPC Computation Steering // 2020 Ural Symposium on Biomedical Engineering, Radioelectronics and Information Technology (USBEREIT), Yekaterinburg, Russia, 2020, pp. 1-4, DOI: 10.1109/USBEREIT48449.2020.9117685.
8. Pavel Vasev, A Computational Model for Interactive Visualization of High-Performance Computations // In: Voevodin, V., Sobolev, S., Yakobovskiy, M., Shagaliev, R. (eds) Supercomputing. RuSCDays 2023. Lecture Notes in Computer Science, vol 14389. Springer, Cham. DOI: 10.1007/978-3-031-49435-2_9.
9. P. Vasev, Parallel Programming Kit programming environment // Abstracts of the National Supercomputer Forum (NSCF-2024), November 26-29, 2024, A. K. Ailamazyan Institute of Program Systems of the Russian Academy of Sciences, Pereslavl-Zalessky. RR-9450, Inria Bordeaux - Sud Ouest. 2022, pp.30. Hal-03547334 (in Russian).
10. V. Mikhailov, S. S. Kumkov, Linear Differential Games with Multi-Dimensional Terminal Target Set: Geometric Approach // EPiC Series in Computing. 2024. Vol. 104. P. 221-242.
11. V. Mikhailov, S. S. Kumkov, Geometric Procedure for Solving Linear Differential Games with High-Dimensional State Vector // Dynamic Systems: Stability, Control, Differential Games (SCDG2024): International Conference dedicated to the 100th Anniversary of the Birth of Academician N. N. Krasovsky, September 9–13, 2024, Ekaterinburg, Russia: Proceedings. Ekaterinburg, 2024. P. 478–480.
12. V. L. Averbukh, N. V. Averbukh, P. Vasev, I. Gajniyarov and I. Starodubtsev, The Tasks of Designing and Developing Virtual Test Stands // 2020 Global Smart Industry Conference (GloSIC), Chelyabinsk, Russia, 2020, pp. 49-54, DOI: 10.1109/GloSIC50886.2020.9267835.