Scientific Visualization, 2022, volume 14, number 5, pages 77 - 95, DOI: 10.26583/sv.14.5.06

On Expressiveness of Visualization Systems' Interfaces

Pavel Vasev[1,A], Mikhail Bakhterev[2,A], Dmitry Manakov[3,A], Sergey Porshnev[4,A,B], Majid Forghani[5,A,B]

[A] N.N. Krasovskii Institute of Mathematics and Mechanics of the Ural Branch of the Russian Academy of Sciences,
[B] Ural Federal University named after B.N. Yeltsin

[1] ORCID: 0000-0003-3854-0670, vasev@imm.uran.ru
[2] ORCID: 0000-0001-8016-9946, m.bakhterev@imm.uran.ru
[3] ORCID: 0000-0001-6852-8096, manakov@imm.uran.ru
[4] ORCID: 0000-0001-6884-9033, s.v.porshnev@urfu.ru
[5] ORCID: 0000-0002-9443-3610, forghani@imm.uran.ru

Abstract

The paper discusses elements of logical models of graphical user interfaces used in both universal and specialized scientific visualization systems. Criteria of expressiveness of programming language that are discussed in “Structure and Interpretation of Computer Programs” book are applied to graphical interfaces. It is shown that graphical interfaces allow users to operate on the same digital substance and with the same logical approaches as in textual programming languages. Both give basic elements, allow their combination, and support the procedures of abstraction. Authors suggest considering these aspects when developing graphical interfaces. This perspective is applied to the following presentation of the paper. The idea of modifiers (also known as behaviors or effects) and the idea of extensions (also known as plugins, modules, and applications) are discussed. Some methods of programming of scene dynamics are presented. Also languages and ontologies of scientific visualization are discussed, e.g. models for editing visualization pipeline: adding data to projects, filtering of that data, and methods of description of data representation on screen. Finally, we discuss additional ideas on systems analysis of visualization systems.

Keywords: scientific visualization, visualization system, logical model, user interface.

1. Introduction

Two kinds of visualization systems are applied to achieve the goals of scientific visualization. They are universal visualization systems (general purpose systems) and specialized systems. Universal systems enable assembling (programming) the visualization pipeline from ready-made blocks. This approach has advantages. But universal systems have the following fundamental disadvantages.

Firstly, when solving a particular visualization problem the user’s labor costs may be excessive for some tasks which could be automated. For instance, it would be possible to automatically identify the type of file being opened and to depict a certain *view* (visualization method) selected by that type. But view selection usually requires a lingering walk through a non-trivial dialog window sequence.

Secondly, a universal visualization system may not provide visualization methods needed to depict particular information. And the methods provided could hide the data features important for the given visualization task.

In order to solve these problems, specialized visualization systems are being developed. They boost users’ productivity in solving their certain visualization problems and allow to emphasize exactly those features which are important for the certain task, ensuring visualization correctness and informativity [1, 2].

Note that real visualization systems exhibit features of both universal and specialized systems. The cause of this is that even specialized systems, developed to solve certain tasks, require further development during operation. They need improvements to meet new users' needs. Those improvements are "aftercoded" by an expert in visualization system programming.

However the major part of such an "aftercoding" may be handed over to the users themselves, if they are provided with suitable tooling. Hence the universal visualization systems features are introduced into the specialized ones. In this way, users obtain an ability to construct visualization algorithms, to build more diverse visual images, and to test more hypotheses by themselves. Potentially, that results in that the main goal of visualization – the understanding phenomenon under study – is achieved quicker and with better quality.

This raises questions about the kinds of elements, logical models, and approaches which may be distinguished in existing universal systems and effectively reapplied in the development of new visualization systems. We present our answers in this paper.

2. Visualization programming options

One can program visualization (or, more accurately stated, views, defined in [3]) using programming languages with text-based representation of code (*textual PL*), or with visual tools. Within both approaches, there are universal tools as well as specialized ones for different tasks. We consider these options in more detail.

Programming with universal textual programming languages:

C1. Universal (general-purpose) codes. The program is being developed as a separate project for a certain visualization task. Small visualization problems are often solved in this way. The development using general-purpose PL is facilitated by programming libraries for graphics and visualization. For instance, those are Matplotlib, Kitware trame, Vedo.embl.es for the Python language, and Three.js, d3.js, Babylon.js for the JavaScript language.

C2. Universal codes utilizing programming frameworks. Frameworks, unlike libraries, determine the control flow of a program to a large extent, so they implement the inversion of control principle. Popular frameworks such as A-frame, Unity, and Unreal Engine may be referred to as this kind. Authors have developed several such frameworks, for example Sharpeye [19] and JUJ [20].

C3. Specialized codes. A visualization program is written in some domain-specific visualization language (DSL) such as Vega Grammar or VRML.

The **C1** and **C2** options suit specialists in programming. Option **C3** is also targeted at them, as a tool for labor automation. But the **C3** option could be convenient for the users of visualization too.

Visual programming:

V1. First of all, these are visual programming environments based on graphs. When working in some system of that class, users place on the surface symbols of various entities, establish connections between them and configure the parameters of entities. The system of entities and their connections is being interpreted in some model of computation, for instance in a data flow model. As a result of this interpretation, the process implementing the desired visualization pipeline is formed. Examples of such environments are DataExplorer, Enso, XcluDev, and SciVi [13].

V2. Visual environments with strict models. Examples are well-known programs like Paraview, Visit, and Blender. In such environments, users often have no ability to insert arbitrary codes into the visualization pipeline, but they can customize (program) visualization, utilizing commands of a visual interface (buttons, menus, etc) or of a scripting subsystem within the programming model provided by the environment.

Option **V1** suits both users and specialists in visualization (however, it seems that specialists in programming prefer textual program representations usually). Option **V2** suits more users of visualization rather than specialists.

Let us additionally illustrate the theses on the gradual elaboration of visualization systems, stated in the introduction. The involvement of a specialist programmer in visualization development has the following specificity. The result of such a programmer's work usually is a program (specialized visualization system) solving the given task. But this program is not a programming tool itself. Its users will be able to load data, and the program will show preconstructed views and provide instruments to configure them. But, as a rule, this is what such a program is limited to, it does not go any further than configuration.

For instance, users cannot append a new instance of some visual entity, say, a second coordinate system, because the program assumes displaying only one instance of the coordinate system. If the users need to add quantitatively or qualitatively new views, they ought to address these issues to the specialist, because mechanisms necessary for doing that are not included in the program.

The involvement of specialists in visualization development is a widespread practice. Its advantages are the extension of interdisciplinary dialogue, interaction, and information exchange. Its disadvantages are the dependency of a user on a specialist, the increased cost of development, and time delays.

Those disadvantages might be leveled out by providing the users with tools for independent rapid solution of the new visualization problems emerging during research.

So, our main interest during the present paper is in means enabling users to independently program their visualizations. Our experience with different options of such programming tells us to focus attention on the approaches to visual programming outlined in **V2**. The community has worked out a number of methods of such programming during many years of the development of computer graphic systems. We continue with the analysis of some of these methods.

3. Expressiveness of visual languages

Consider the user workflow from the following perspective. The user working with one or the other program builds a *digital artifact* in most cases. By the term digital artifact, we collectively call information entities in digital form: text, video, presentation, database et cetera. A typical program with a graphical interface offers the user a visual language: the graphical interface itself and its semantic – logical interpretation of the user's actions. In this way, a user working with a visualization system forms the visual images set necessary for him – by constructing a visualization pipeline.

The structure consisting of an interface (representation) and its semantics resembles the traditional programming languages' structure: source code (representation) and its semantics. The necessary conditions of expressiveness are formulated for the latter, cf. [4, 5].

An expressive programming language has to possess the following features.

S1. Primitive expressions, which represent the simplest entities the language is concerned with.

S2. Means of combination, by which compound elements are built from simpler ones.

S3. Means of abstraction, by which compound elements can be named and manipulated as units.

The fuller PL implements features in the list **S1-3** the more expressive it is. We may project these features on the users' workflows in a program with graphical interfaces. That might lead to understanding the features which should be provided by the visual language (interface) of the program. The fuller these features comply with **S1-3** the more expressive the program will be, hence the more results the users will be able to get working with this program.

As an example, consider the workflow in PowerPoint. The user may enter different kinds of basic elementary visual objects: texts, pictures, geometric figures, animations. This is the **S1** feature. That objects can be combined into slides and can be joined into groups. This is the **S2** feature. The **S3** feature, abstracting, is not fully implemented in PowerPoint but still is presented. It is possible to select and copy groups of objects. One could think that those are ab-

stracting operations: many objects in a selection could be conceived as a whole. Note that changes in original constructions will not be mirrored in its copies. We believe it means that the feature of abstracting is not completely implemented on this level in PowerPoint: the processes of user interactions are made not abstractable.

The abstracting is implemented to the more complete extent on the presentations level. For example, a user can create a presentation design template and use it for different projects. Changes in that template will cause changes in presentations utilizing it. We might assume that several visual languages are implemented in PowerPoint for workflows on different levels: for visual objects, for slides, for presentations, and for templates. In each of those languages PowerPoint implements features **S1-3** to a different extent.

Drawing parallels between textual PL and visual languages, among which we include graphical interfaces too, we may draw a conclusion that user building digital artifacts is engaged in programming. Digital artifacts consist of entities similar to the entities described with codes in textual PL: basic elements and their combinations, abstractions, and their applications. We may confirm this conclusion with the fact that the same result can often be obtained in both ways – with programming in textual PL and with working through visual language as well.

The **S1-3** features can also be projected on graphical interfaces of programs for scientific visualization. Those features could be hardly found in simple visualization systems performing visualization within the limits of a strictly predefined view. More "advanced" systems implement some of the **S1-3** features. Universal visualization systems most fully manifest them.

4. Modifiers

Semantic of an advanced textual PL usually defines different means of combining entities: making data structures from separate values, passing arguments to subroutines calls, assembling functions to reify interfaces, collecting classes and interfaces to build new classes, putting computation in lazy or asynchronous (async/await) contexts et cetera. So different means of combining entities are also found in visual languages (particularly, in the languages of graphical interface). One of such means is the *modifiers* mechanism. The other names for this concept are components, effects, behaviors, styles.

A modifier is an additional entity attached to the main entity. A modifier can affect the behavior of the main entity. That effect can be either cosmetic or substantial. Some examples of modifiers:

1. The "follow the object" modifier parametrized by *the target object* controls the position of its main object in such a way that the main object would stay adjacent to the target one as the target one moves. There are different methods to compute the position of the main object: it may firmly maintain a fixed distance to the target object, or that distance may be elastic. Such a modifier can be attached to different objects, for instance, to the camera object.
2. The "section" modifier modifies the depiction of its main visual object in a way to display only the part of it, for instance, the part in a given semispace. The modifier can add a new visual object to the scene; for the given semispace example, the plane defines the section. Those objects may simplify the modifier parameters configuration via *direct manipulation*.
3. The "terrain" modifier (in the Unity development environment [6], cf. fig. 1) is attached to some plane object in the scene and provides an interface to form a hilly surface (its shape and texture) from that plane and to place the instances of plant models on that surface. Direct manipulation is applied: a user forms hills and places plants working directly with the visual image of the surface in the Unity editor's main window.

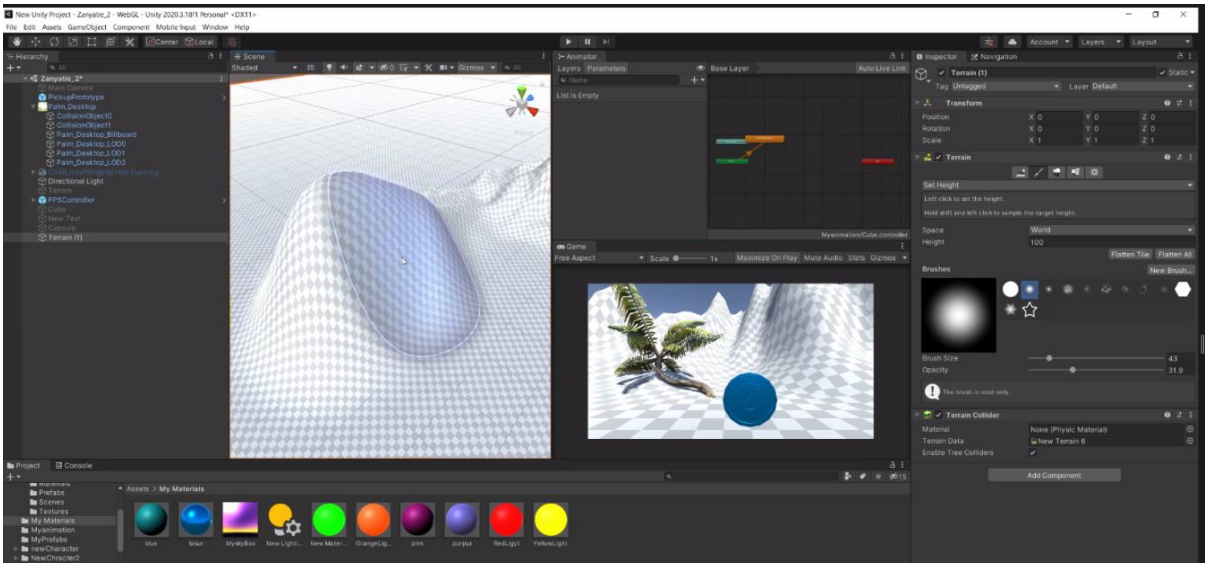

Figure 1 - Application of the "Terrain" modifier in the Unity environment. The graphical modifier interface is to the right, the "Terrain" tab. Interfaces of other "Transform" and "Terrain Collider" modifiers are visible nearby. To the bottom right there is the "Add Component" button to attach new modifiers. Image courtesy of A. E. Krokhaleva.

The "Terrain" example demonstrates that a modifier may have an independent complex interface and rich capabilities despite its subordinate to the main object role.

Technically, a modifier is a complete software component interacting with its main object and with other modifiers attached to that object via certain interfaces.

As an analogue of modifiers in textual PL one could consider mixins from the object-oriented programming paradigm, which are equipped with parameters and being applied not only to classes but also to objects (instances of classes). Modifiers in the Compose Multiplatform (JetBrains) and SwiftUI (Apple) frameworks are another analogue. Programming with modifiers is an implementation of the "composition over inheritance" principle.

In the ultimate case of programming with modifiers, the role of main objects narrows to the modifier containers with no other functionality. For instance, the Entity-Component-System model [7, 8] is arranged this way. The objects are empty in that model, modifiers serve as labels and carriers for values, and the algorithmical layer is implemented within systems. Along with that, a user forms an independent set of active systems.

A modifier may create other modifiers and attach them to the objects in order to implement its behavior. For instance, the A-Frame project relies on modifiers of that kind. The project names them *higher-order components* (similar to higher-order functions), cf. [9].

Besides, we believe that modifiers are the method to implement uniformity of the elements of the user interface visual language. A modifier that is uniformly represented in a graphical interface for different types of objects indicates similar semantics for the user. By the way, same-looking modifiers may have different implementations for different types of objects.

Many visual programming systems have an interesting ability to apply modifiers not only to individual objects but also to sets of objects. For instance, this may be implemented with the model of the user's workflow in which: at first, an object from a library is placed on the project's panel, on that panel modifiers are attached to the object and its properties are configured; after that, individual instances of that construction produced on the project's panel are created in the scene. Systems allow to visually form "proxy-classes" and create their instances in that way.

Visual modifiers in interfaces are split into categories sometimes. For instance, in the GDevelop environment, there are "behaviors" and "visual effects".

We believe that the use of modifiers amplifies the expressivity of the languages (not only visual) designed for digital artifacts creation. Modifiers facilitate user programming of individual components of artifact, since generic elementary behaviors (modifications) of objects

may be taken out to the special layers of the code or of the visual workspace and may be applied to different kinds of objects by the means of graphical or text-based user interfaces.

5. Programming of the scene dynamics

An interface for the construction of complex digital artifacts, which include scenes of visualizations, should unfold in semantically different directions. The **S1-3** features should be implemented in each of those directions to the necessary extent.

In this section of the paper, we consider problems of programming the dynamics in visualization scenes, that is, description of processes of the scene state changing: for instance, animation of depicted physical phenomenon evolution in time.

We have assumed it was enough to implement components for the scene dynamics in the objects having been added to the scene or in their modifiers. Many game programming systems use such an approach. They provide a user-predefined set of behavioral modifiers. Examples of behavioral modifiers are the aforementioned “follow the object” modifier, the “object of collision model” (known as collider) et cetera. Attaching such modifiers to objects one may append predefined kinds of behaviors to those objects. However, it has turned out that this approach was not suitable enough in practice to describe all necessary forms of dynamics in scenes.

There are other approaches also. Dynamic behavior may be defined with functional dependence of visual objects’s *properties* (parameters, variables) on “time”. For instance, in Unity, such a dependency is specified by selecting key points on the time axis and by defining properties values corresponding to those points. *Animation clips* are created with this process. When switching to the “scene playback” mode those animations are enabled and being replayed in the cycle. The ability to abstract animations is also implemented in Unity. Clips may be grouped and merged into separate entities for further application to different objects.

That is a convenient mechanism, but it does not allow to program all necessary animations. So Unity implements the *animation controllers* mechanism. It is based on finite automatons which describe switching between behaviors of an object under control, including switching between animation clips when changing states.

Descriptions on the level of physical processes control are also used to program scene dynamics. For instance, cf. spacecraft control model [10] and the SimInTech environment [11, 12].

The Entity-Component-System model [7, 8] is often used in games and simulators. That model assumes the binding “entities” to “components” and the implementation of dynamic processes in the form of “systems”.

The data flow model is also used frequently. In such a model, the data flow defines the dynamics of the system (see the SciVi programming system is an example of that approach [13]).

The hybrid approach may be also found among visual programming tools. In this approach elements of dynamics may be defined both for individual objects and for systems consisting of them. So, the GDevelop system [14] is based on *“events”* (“GDevelop events”). The user configures different elements of scene behavior by programming event handling. Programming in graphical interface reduces to the creation of records in the event handling table, fig. 2.

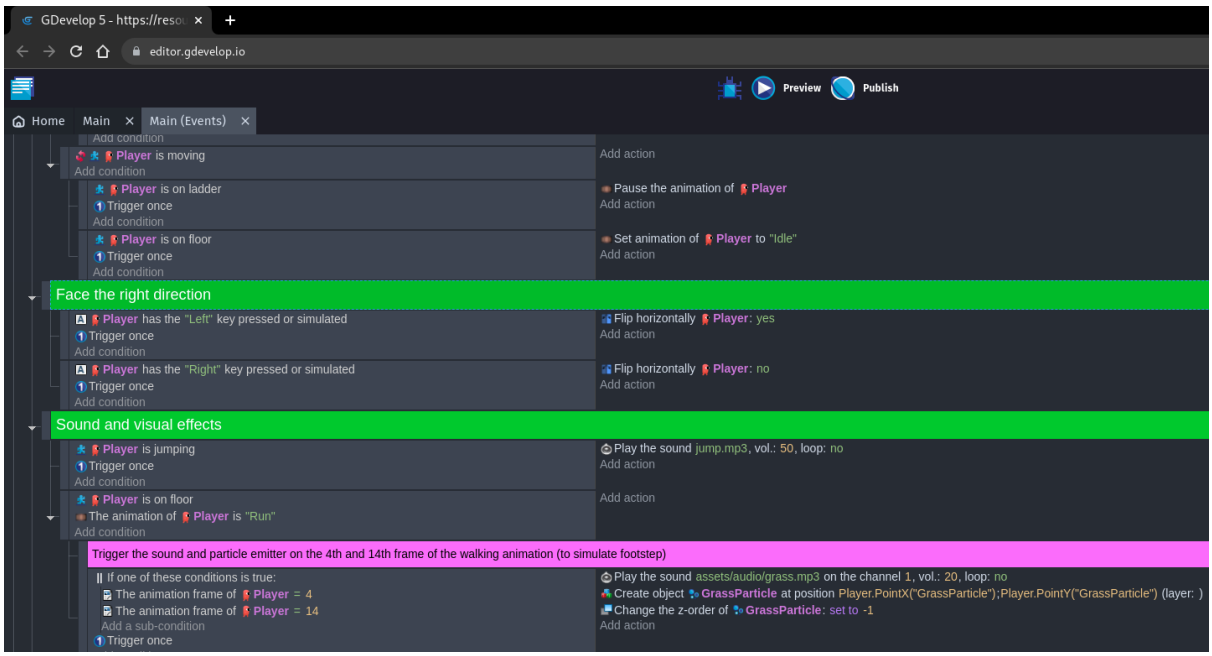

Figure 2 – the event table appearance in the GDevelop environment.

The table consists of two columns – event and reaction. Events can be both elementary: "the program starts", "the button is pushed" et cetera, and emerging from complex conditions: "the position of character matches the position of a coin". Reactions are defined with lists of imperative constructions, such as "increase the value of the *score* variable by 10" or "remove the coin from the scene". Unconditional actions can be placed into the table, they may be coded in the JavaScript language. The created table is processed from top to bottom row by row 60 times per second.

Especially the following GDevelop features may be noted. **1)** Reactions and conditions can have parameters described by expressions, which may contain calls of complex functions. The special wizards help to compose such expressions. **2)** It is possible to create extensions (cf. Section 6). **3)** Descriptions of behaviors can be attached not only to individual objects but to the groups of objects, defined by some condition: for instance, all objects of a given proxy-class (cf. Section 4).

So, we see a spectrum of approaches to scene dynamics description. These approaches despite diverse ontology give the user similar capabilities. We consider this as a sign of the presence of the more fundamental model for scene dynamics programming, which could be one of the variants of process calculus: communicating sequential processes [15] or pi-calculus [16]. It is also possible that the ideas of *automata-based programming* widely researched in the Russian scientific community may be of great use to solve the problems of scene dynamics programming.

6. Extensions

Extensions (also called add-ons, plugins, applications, modules, and packages) allow placing additional program functionality into external codes that a user can attach when it is required. Extensions idea is often used in programs, including visualization systems, see for example QGIS plugins [17] and GDevelop extensions [18].

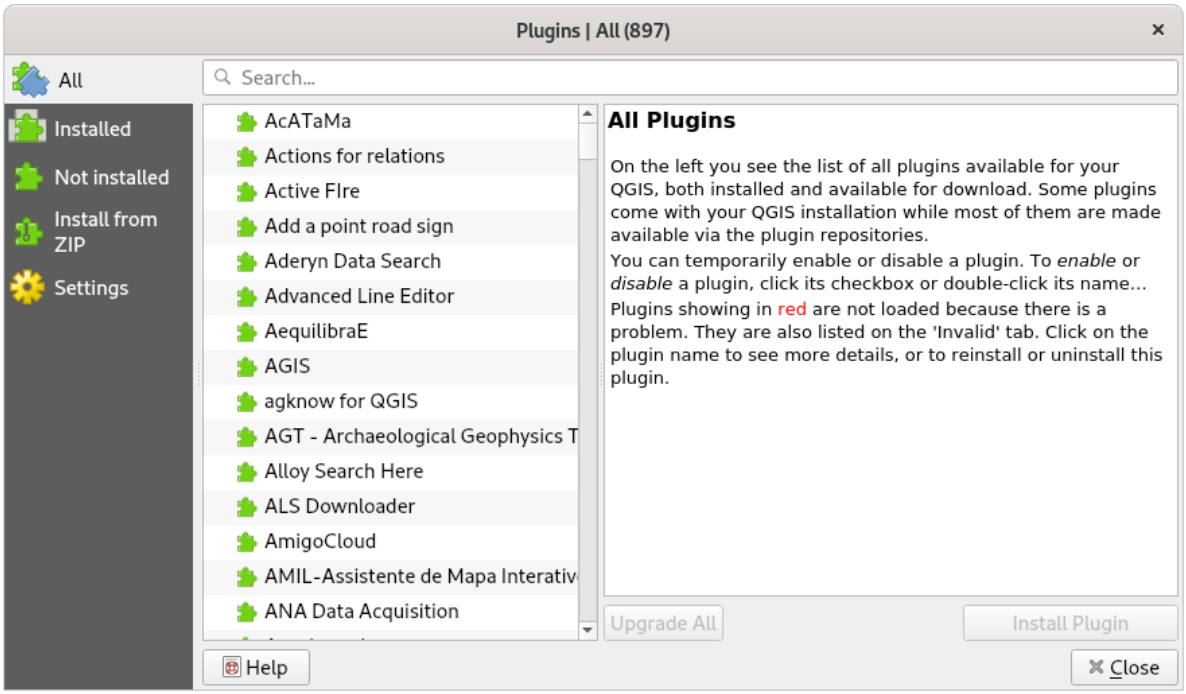

Figure 3. Attaching an extension in the QGIS geospatial visualization environment. The attachment is made in the context of the program, i.e. "activated" extensions are valid in all projects of a user. QGIS offers to install an extension from its list (called repository). Additionally, a user can add other repositories, as well as install an extension from an archive (Install from ZIP menu).

The decision to attach an extension is made by a user, see the example in fig. 3. The attachment process for a user takes place in some context: for example, in the context of the entire program, or in the context of a particular project, or some part of a project. When a user has "attached" an extension, further interaction with it is carried out according to the following model.

The program typically passes control to an extension by providing some kind of Application Programming Interface (API). When attached, an extension, having received control and an entry point to the API, can add entities to the program - for example, add new types of modifiers available to a user. When a user accesses entities implemented in the attached extension, it gains control and interacts with a user and his digital objects in a program project with some level of freedom.

Sometimes the API allows for extension to change the graphical user interface (GUI) of a program. Using this, an extension can, for example, directly add menu items to the GUI.

As a result of attaching an extension, a program becomes more “powerful". Additional types of objects, modifiers, elements of scene dynamics, etc. appear in it. Thus, the principle of additivity described in [5] is implemented.

Usually, to attach an extension, a user selects it from the list. These lists can be formed in various ways. The list can be compiled by the software product vendor. In some cases, the vendors can form this list by implementing an "extension store". In this case, the list is fulfilled by initiatives from extension manufacturers. For example, the Unity Asset Store, which distributes assets of various types, joined in packages, which actually are extensions.

Sometimes the user is given the opportunity to maintain their list of extensions (e.g. “favorite” extensions) in the context of a user profile (perhaps in the cloud). This approach is implemented, for example, in the Google Tag Manager environment.

A user can influence this list through directories in a local file system, or by utilizing a configuration file, or through settings in a GUI. For example, in QGIS, the user can place the program code of an extension in a special directory, and then such an extension appears in the GUI in the list of extensions which are available for attachment. QGIS also has the ability to

"import" extensions from a zip archive using GUI. When imported, the extension archive is actually unpacked into that special directory.

A program's developer has the opportunity to attach extensions at the level of the source code of a program or project. Then the user does not need to initiate the attachment of the extension - the decision to attach it has already been made by the developer and all the necessary actions for attachment are performed at the level of program codes. Based on this observation, we can assume that in the ideal design of a program with extensions, all types of program entities: visual objects, modifiers, etc., are implemented in extensions, some of which are attached at the level of the program source codes, and some during program usage at the discretion of a user.

Let's note the following interesting approach. The user interface of some programs displays the functionality of unattached extensions. When accessing it, the user receives a message about the extension that implements this functionality and an offer to attach it. For example, in the GDevelop environment, the available modifiers are "advertised" in the user interface in such a way.

We believe that extensions are analogues of modifiers on a different semantic level. Modifiers are applied to scene objects, and extensions are applied to projects or to the user's working environment, to the program. Modifiers affect the behavior of objects in the scene, and extensions affect the behavior of the project or the program as a whole, providing new features or affecting those already present in the program.

Like modifiers, extensions are sometimes grouped according to purpose. Attachment processes for different groups may have different user interfaces.

7. Data sources, filters, plots

Visualization systems, as a rule, visualize "data" (represented in the model of files, data streams, processes, network resources, etc.). They often allow configuring data visualization processes by adding *data sources*, *filters*, and *visualization methods* (plots), and establishing relations between them. This configuration is performed in virtual spaces which are often named *projects*.

The user adds data by specifying the path to them in the local file system, or by dragging (drag-and-drop) in a graphical shell of an operating system, or by specifying a URL from the cloud or from another network resource, or, in general, by specifying an arbitrary pre-formed data set with some predefined protocol for accessing them.

Some systems allow you to group sets of files by file name templates. Some sections of names described by such templates can be interpreted by the system as the values of parameters of loaded objects (for example, time) and used in the construction of the scene and its dynamics (for example, to offer animation options).

After adding data in one way or another, a data artifact appears in the project along with its connection to the source of this data. In the simplest case, the entire data is copied and becomes an integral part of the project. Data can be of various types – for example, triangular meshes, tabular data, scalar and vector fields, and others. If not one artifact is loaded, but a set, then somewhere in the user interface there is an opportunity to select the current element of the set.

When adding a visualization method, the program can take into account which data it can be applied to. For example, in the Visit system, there is a concept of current data – this is the currently selected data artifact from the list of previously added artifacts to the project. During the addition of a visualization method, only methods compatible with the current data type are displayed in the list of available methods. Once the user confirms the addition of a visualization method, the relationship of this method with the current artifact in the dataset is also added.

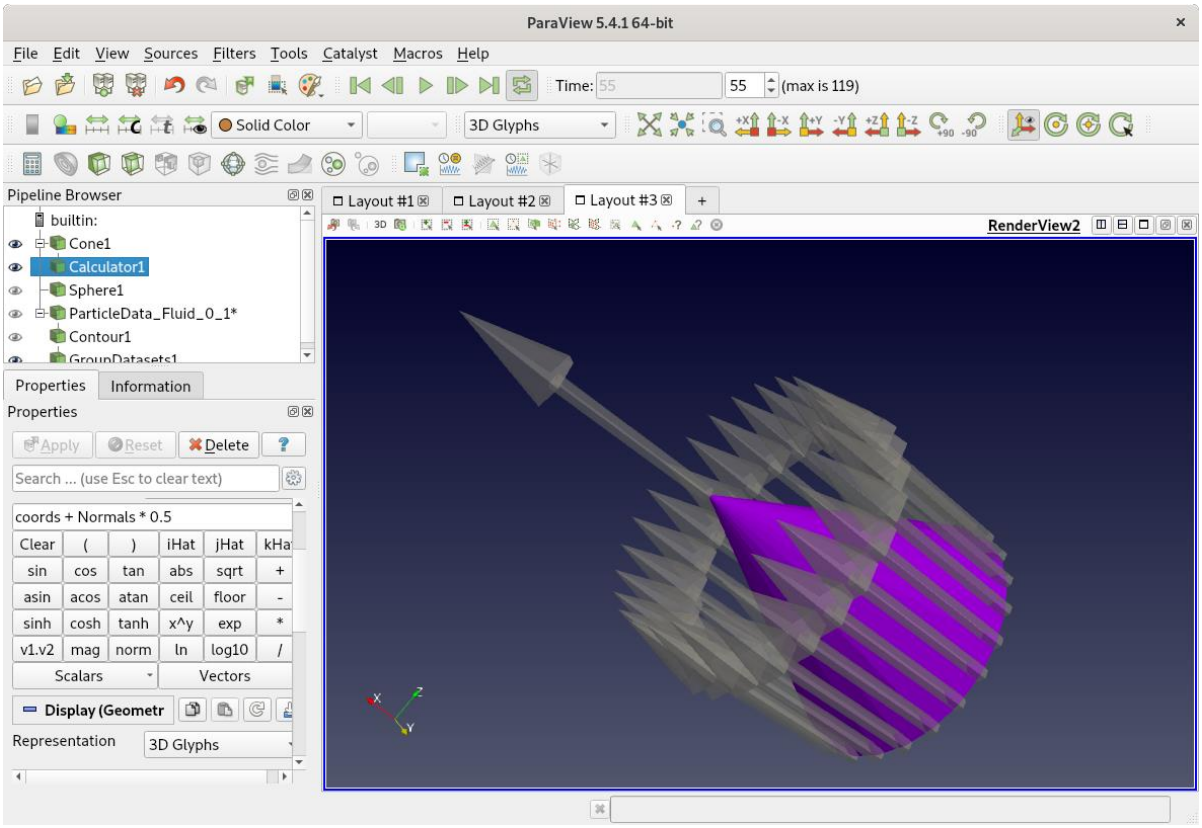

Figure 4. Paraview project window with a cone added and a calculation block of the Calculator type added. The block generates vectors based on the control points of this cone. The "built-in" vector visualization method in the block has been activated.

A dual approach is also possible. For example, the system Kepler.gl allows you to add an arbitrary visualization method to the project (from the list), and then establish its relationship with the data elements added to the project. When establishing such a connection, users can select only data elements that are compatible with the visualization method being configured.

Some systems allow performing calculations on the data added to the project. Calculations can be performed in command mode: upon completion of such calculation, its result is saved in the project as a new data element. Calculations can also be carried out in *process* mode. In this case, the object describing the calculation block is saved in the project and is available for further adjustment and subsequent recalculation. The result of such calculations is available in the form of a special data artifact — the *output* parameter of this calculation block. Such blocks are sometimes called *filters*.

Block output parameters can be used as data sources for other blocks - as input parameters. By joining blocks by inputs and outputs, it is possible to form a pipeline from transformation blocks. Pipelines can react to changing parameters, including parameters of intermediate links, and automatically recalculate with new versions of data.

Hybrid calculation models are also implemented when the user is given the opportunity both to carry out a one-time calculation in command mode and to build a pipeline saved in the project from configured blocks, which is suitable for repetitive calculations. Such a hybrid model is offered, for example, by the QGIS processing toolbox.

As a result of calculations, new data is generated that needs to be visualized. Some systems take into account such new data as well as those added by users to the project — they allow linking this data with the selected visualization method. In other systems, calculation blocks are equipped with built-in visualization tools, and then the result of the block is both data and visual images that can be "turned on", see for example fig. 4.

The application of a series of computational blocks and the subsequent connection of visualization methods with them is, in our opinion, an essential part of user programming in universal visualization systems.

8. Systems analysis of visualization systems

The origins of systems analysis and its methodological concepts lie in those disciplines that deal with decision-making problems: operations research and control theory. To work with the visualization system the user must control it. For example, if the work with the visualization system is considered as visual programming, then the user solves the problem of computational steering.

Let's consider such an important aspect of visualization as the computational steering of supercomputers, firstly related to online visualization and secondly related to computational steering [21]. Here is a list of typical tasks in this area:

1. Debug of supercomputer programs. The program needs debugging at different levels, for example, mathematical models, algorithms, and their implementations, as well as at level of performance debugging. In time debugging programs it is useful to be able to get information about the launch of the programand to interact with it.

2. Evaluation of the progress of the computation and its current state. It allows estimation of how far the calculation has progressed, to make a decision on the expediency of its continuation, to conduct a visual analysis of the correctness of both the algorithm run and the computational system components.

3. Online adjusting the program launch parameters. During the evaluation of the program, there may be a need to make adjustments to the data and parameters of computational algorithms in the program execution continuation mode. This can be useful both for debugging programs and for conducting parametric research in a computational experiment.

4. Interactive computational experiment in the mode "what-if?". The researcher interactively sets the initial data of the task, receives the result, analyzes it, and again sets the initial data. This differs from the batch mode of the computational experiment precisely in the interactive mode. Here the decision on the options for the initial data and the directions of the calculations is made by the researcher in the online mode.

5. A computer program that simulates the environment, called a virtual test stand. In it, researchers can interactively perform various experiments that are more structurally complex than experiments in the “what-if” mode. Also, virtual test stands can be used as simulators to develop people's skills [22].

6. Automated control or the control of the program runs from another program, for example, solving an inverse problem by enumerating its initial data and parameters [23].

7. Interaction with other supercomputer programs. Supercomputer computing can be built as an ensemble of interacting programs. Each program uses its own supercomputer programming technologies. Programs are configured in such a way that during the program runs they exchange current results. For example, each program can calculate its own side of a physical phenomenon and an exchange takes place between them in order to enrich information about the simulated environment [24]. Also, this direction allows you to simulate the operation of control systems when the effect of the control action is immediately calculated and is closed by means of "virtual sensors" back to the control system [25].

8. Participation in a natural experiment. In modern practice, the following situation began to arise. A series of full-scale experiments is being carried out at the facility and in order to make a decision on the next step of the experiment, it is necessary to quickly process the current results [26]. The necessary processing speed can be provided by supercomputer technologies, whereas the program running in such a situation should not be run in batch mode, however it should be run online in order to respond and process data quickly. In practice, it even happens that the facility is located in one part of the planet but the supercomputer is located in another [26].

An analysis of these examples leads to the idea that the visual pipeline as part of a computer modeling cycle [27] can be considered not only from the point of view of operations research but also from the point of view of a dynamical system.

A computer modeling cycle can be thought of as a data mapping sequence. For example, the data flow model can be represented not as a static graph, but as a dynamic one, and the routing of flows is not scalar, but vector, that is, vector-based flow routing. At any given time, a dynamical system has a state representing a point in an appropriate state space. A phase space is the set of all possible configurations of a system. Thus, a dynamical system is characterized by its initial state and the law by which the system passes from the initial state to another. Therefore, the interactive process, which is displayed on the screen, creates a trajectory in a phase space.

The authors actively use the concept of a trajectory when creating animations that are considered a continuous map in terms of perception. In this case, the trajectory is a linear approximation between the target states of the program. Target states define a set of those states that should be visited during the animation.

Let's apply one of the systems analysis approaches, called abstracting, related to the visual pipeline being the part of a computer modeling cycle. Data abstraction models are generalizations that allow you to abstract from the source and ontology of data during visual analysis. Regarding a computer modeling cycle, different levels of abstraction can be distinguished. For example, the mathematical level of data abstraction includes mathematical, algorithmic, software, visual, and visual analytics models. The authors propose to consider a structural or semantic unit of visual analysis, also called a unit, as a continuous map of a class of data subsets onto a logical space. Since the mapping of data to a visual representation is not an isomorphism, it is necessary to take into account the mathematical level of data abstraction [28].

The structural unit of visual analysis is a visually perceived image, interpreted as a truth answer, in the original source [29] an unambiguous answer to one of the intermediate questions. In terms of system analysis, the structural unit of visual analysis is a controlled system S with feedback R(t), see fig. 5.

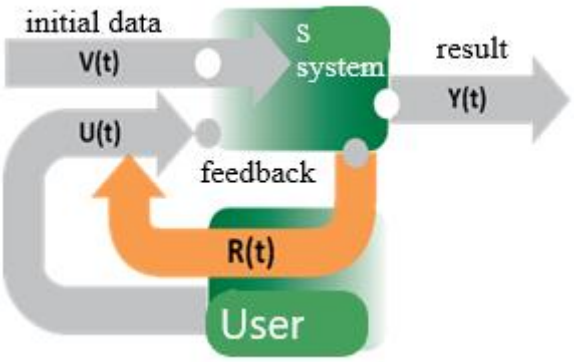

Figure 5. Structural unit of visual analysis [29].

Since visualization becomes the medium of an automated analytical process, areas related to self-organization are of interest to visual analytics: dissipative systems, autonomous computing, and synergetic. For example, it is possible to clarify the concept of the structural unit of visual analysis in fig. 5 from the standpoint of dissipative systems.

A dissipative system is a quasi-stationary open system, a characteristic feature of which is the process of self-organization, which occurs as a result of the action of a negative vector, for example, the friction force. Let us explain the difference between a dissipative system and an optimal control problem in the case of linear systems:

$$\dot{y}=Ay-Bu,$$

where u is the optimal control to be found.

Since the dissipative system is an open system, the control *V(t)* comes from outside, it is initially set. The introduction of a negative vector or negative feedback *R(t)* narrows the range of problems to models with saturation. The emergence of feedback should be considered not only as a result of purposeful but also chaotic, random user actions, for example, similar to Brownian motion, since the self-organization process occurs much faster when there are external and internal noises in the system. Such control is called stochastic closed-loop control [30]. User control *U(t)* is desirable to be optimal *u=u(U(t),V(t))*, at least it must exist. For example, you can offer the user y_i - a monotonically convergent sequence of solutions [31].

The paper [32] provides a justification for the application and considers the benefits of the parameters control of the visual program, as well as of the supercomputer program. If the mathematical model is taken into account during abstracting, then, for example, a parametric model of a white-noise random process [33] or tensor expansions [34] can be applied to this approach. From a programming point of view, parameter's control can be called parameter's abstracting. (Recall that abstracting is considered in the context of data abstraction models). Visual abstract parameters are defined as a special case of abstract data types whose function range is the dynamic visual image [35]. One of the examples considered in this work is an abstraction called a modifier. Including, the modifier from the point of view of the programming language is a component, and from the point of view of visualization is a structural or semantic unit of visual analysis.

As stated in [36], there are certain connections between dissipative systems and Anokhin's theory of functional systems, one of the areas of general systems theory, at least in relation to visual perception. When studying such factors as structure, composition, state, and environment in systems theory, large-scale studies of elements organization of the lower structural-hierarchical levels, that is the system infrastructure, are acceptable. In this case, each element is considered as a relatively heteronomous, but also relatively autonomous system to the structure, environment, composition, and state of which the principles of system decomposition are equally applied. The authors introduce the concept of an autonomous algorithm, which is a composition, for example, of the modifier, calculation, and view. For example, autonomous algorithms are online parallel computing services. The decomposition of the data flow into autonomous algorithms allows us to consider the dynamic system as a stationary one in the neighborhood of the autonomous algorithm.

One of the system methods is verification. The authors come to the conclusion that formal approaches should be applied to applications and visualization systems simultaneously with visual verification. First of all, of interest are parametric models related to stochastic control, sensitive analysis, data filtering, and visual calibration of model parameters. Formalization is not an end in itself, it is important that there is a certain pragmatism, that is, the application of modeling to solve specific visualization applications. For instance, designing a programming language in order to develop specialized visual systems or considering the rendering equation as a diffusion process.

9. Thoughts on visualization pipeline

The concept of computer visualization implies the construction of visual images and their display with the help of one or another output equipment. To describe the process of building visual images, a model called the **visualization pipeline** [37,38] is often used. The visualization pipeline describes the (step-wise) process of creating visual representations of data [39].

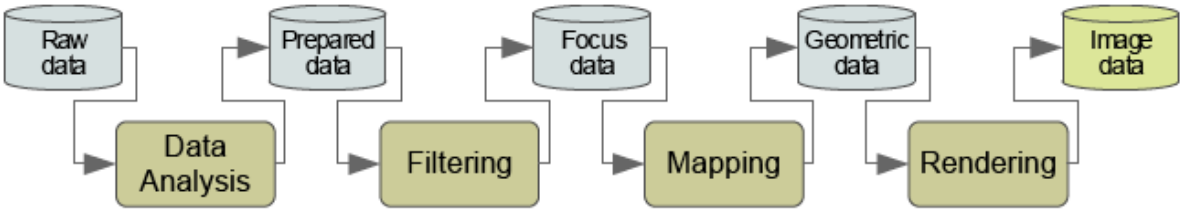

Figure 6. A classical view of the visualization pipeline [39].

1. **Data Analysis**: data is prepared for visualization (e.g., by applying a smoothing filter, interpolating missing values, or correcting erroneous measurements).
2. **Filtering**: selection of data portions to be visualized.
3. **Mapping**: focus data is mapped to geometric primitives (e.g., points, lines) and their attributes (e.g., color, position, size); the most critical step for achieving expressiveness and effectiveness.
4. **Rendering**: geometric data is transformed to image data.

The visualization pipeline is the only one point of view onto the process of visualization. We would like to annotate it with the following.

Actually, the visualization pipeline is a **networked graph**, not a straightforward pipeline. It contains a lot of interconnected nodes, see figure 7. There are a lot of reasons for that, one of which is the following.

The "classical" view of the visualization pipeline does not directly express the generation of **additional data** artifacts. However, in practice, such a step exists. For example, one may desire to see a vector field of normals of the body under study (as in fig. 4). In that case, the normals will be computed and displayed. This adds "branching" to the pipeline and invokes additional computations. That is, data transforms into new data. Thus, a visualization pipeline may be thought of as a view of the part of the visualization process (e.g. one visualization contains a lot of pipelines, maybe growing from one root).

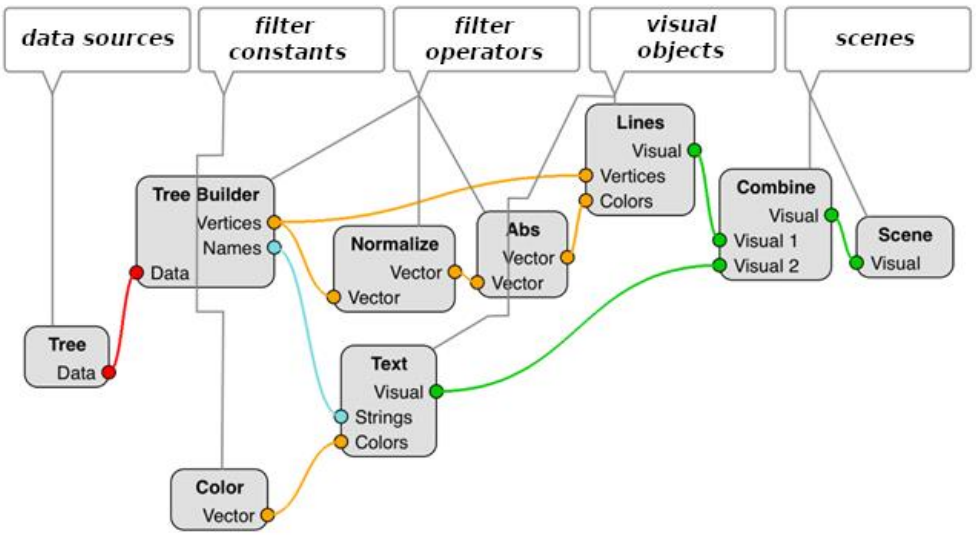

Figure 7. A real life of the visualization pipeline. The networked nature of the pipeline is shown. Interestingly, this figure is a screenshot from a real user interface of the Scivi [13] visualization system, which is based on direct manipulation of the visualization pipeline.

We would like to note that each part of the visualization pipeline is a computational process. This also includes rendering. Thus, the visualization pipeline defines computation.

As it was mentioned above in section 3, when a user works with a scientific visualization system, he actually constructs and manages the visualization pipeline. One may think that a

user just refines the pipeline, for example by changing parameters of its nodes; however, we claim that in common cases the user controls the pipeline structure. A user may load data and visualize it through some pre-defined views, but at the moment he desires to look in a new way, for example, to switch the view, he begins changing the pipeline.

Users may do this employing visual tools, for example, using traditional user interfaces, as in fig. 4; using graph-based visual interfaces, as in fig. 7; or probably using some other visual models. He may also achieve the same effects using general-purpose programming languages, which is discussed in section 2.

We conclude that at least within visualization systems, both the use of graphical user interfaces and general-purpose programming languages are essentially the same processes. Because in both cases a user configures the computational process, in the case of visualization systems this computational process is the visualization pipeline.

This leads us to the idea that graphical interfaces may be enriched with ideas from general-purpose textual languages and vice versa. As it is shown in the paper, this idea is widely used in practice. Modifiers, used in graphical interfaces (section 4), are counterparts of mixins from general-purpose languages and implement the idea of composition over inheritance. Extensions are a kind of program libraries: one may be attached to a project and then entities declared in it might be used. Extensions even may influence the program on their own behalf, just by the fact that they are attached because they receive control during initialization. Abstractions and combinations, as a general idea of constructing things, are also widely used.

More ideas to come. For example, in the programming world, the idea of copy-paste development is widely used nowadays. One may find a working solution somewhere on the Internet, and copy its code to his own project. This is a kind of distributed abstraction, where abstractions are shared and used on the responsibility of participants. As well as there are more organized processes for the same purpose, like package management systems. Same things might happen for GUI. They already actually happen in the form of asset stores, asset and experience interchange forums, and “how-to” articles. In the latter case, the “code” might not be copy-pasted but might be reproduced.

It is interesting that visual interfaces have some distinctive features from general-purpose programming languages. For example, we noted the following aspect. In textual PL, programmers feel comfortable when using the same syntax on any level of abstraction. For example, it is considered normal that the code of the sinus function and the code of some complex formula function are expressed in the same language.

In contrast, in graphical interfaces users feel uncomfortable and lost in case when the visual structure of the interface repeats. It seems that is why we have to use menus, tabs, buttons, modal activities, and dialogs, which again contain menus, tabs, and so on. We need to visually underline the context and abstraction level where the current user focus resides. That is, programs should change their visual appearance when the context changes. If this was not true, all visual programs might have an interface like regedit (registry editor) in Windows. A lot of interesting ideas of human-computer visual interaction and their evolution are described in [40].

Sometimes textual PLs also use the idea of visual differentiation of various context levels. For example, in Python, a colon is used to depict if statements, cycles, function definitions, and other special forms. However, that is not necessary, e.g. Lisp language uses its ordinal syntax for these operations. Another example is Python’s function decorators. Although they are just functions, a special syntax for calling them is introduced (by prefixing a decorator name with @ symbol).

In some visualization tasks, it is not convenient to manage the visual pipeline directly. Graphical interfaces already hide such management, by introducing various interface abstractions and logical models on top of them. However, one might go even deeper, and introduce drastically different models for pipeline management. For example, in [41] the following visualization pipeline alternatives are introduced: functional field model, MapReduce model, and domain-specific languages.

We think this is a commonly used approach when a particular task is suggested to be solved with appropriate tools, probably with substantially different interactions and logical models. For example in mathematics, a differential form of a function is sometimes found more convenient to define the function, instead of using its traditional formula.

Sometimes it is even found convenient to suggest more than one method of doing the same thing, and users might choose a comfortable one according to their situation. For example, in the XCluDev platform, a user might manage links between objects of the visualization pipeline in two modes: 1) by managing individual links using a visual interface, and 2) by editing all links in the project in a textual form (and even copy-paste them).

10. Conclusions

The paper provides an overview of some elements of the logical models of user interfaces aiming at scientific visualization systems. We consider these elements to be effective, providing a wide range of possibilities for the formation of visual images. This is indirectly confirmed by the fact that they are implemented in many visualization systems. Their application is possible both in new universal visualization systems and, as needed, in specialized systems. We are experimenting with these models by developing a programming library for scientific visualization systems [42]. Its codes are published on the Internet at https://github.com/viewzavr address.

Interacting with the visualization system, the user acts in the logical model offered to him. Its clarity, expressiveness and efficiency are the key to success.

11. Acknowledgements

The authors thank their colleagues – mathematicians, physicists, specialists in the field of visualization, as well as reviewers – whose interaction led to the appearance of this work.

This study was funded by the Russian Foundation for Basic Research (RFBR), project number 19-31-60025.

12. References

[1] V.L. Averbukh, N.V. Averbukh, D.V. Semenischev. Activity approach in design of specialized visualization systems (2019). Scientific Visualization 11.3: 1 - 16, DOI: 10.26583/sv.11.3.01

[2] Averbukh, V. L. . Sources of Computer Metaphors for Visualization and Human-Computer Interaction. In: Silvera-Roig, M. , Azcárate, A. L. , editors. Cognitive and Intermedial Semiotics [Internet]. London: IntechOpen; 2019. DOI: 10.5772/intechopen.89973

[3] V. L. Averbukh, "Towards the conceptions of visualization language and visualization metaphor," Proceedings IEEE Symposia on Human-Centric Computing Languages and Environments (Cat. No.01TH8587), 2001, pp. 390-391, doi: 10.1109/HCC.2001.995296. URL: https://www.cv.imm.uran.ru/e/3536 (accessed 11.11.2022)

[4] Abelson, H.; Sussman, G. J. & Julie Sussman (1996), Structure and Interpretation of Computer Programs, MIT Press/McGraw-Hill, Cambridge.

[5] Abelson, H.; Sussman, G. J. & Julie Sussman, et al, Structure and Interpretation of Computer Programs, JavaScript Edition. URL: https://sourceacademy.org/sicpjs/1.1 (accessed 15.10.2022)

[6] Unity development environment. URL: https://unity.com (accessed 30.06.2022).

[7] Martin, Adam. "Entity Systems are the Future of MMOG Development". URL: http://t-machine.org/index.php/2007/09/03/entity-systems-are-the-future-of-mmog-development-part-1/ (accessed 30.06.2022).

[8] Entity-component-system, URL: https://en.wikipedia.org/wiki/Entity_component_system (accessed 30.06.2022).

[9] High-order components in A-frame project: https://aframe.io/docs/1.3.0/introduction/entity-component-system.html#higher-order-components (accessed 30.06.2022).

[10] A.A. Tugashev, An approach to ensuring the fault tolerance of spacecraft based on the automation of the design of intelligent onboard software // Reliability and quality of complex systems. – 2016. – № 4 (16). – PP. 106–112. In Russian. DOI 10.21685/2307-4205-2016-4-15.

[11] Myznikova V.A., Ustimenko V.V., Chubar A.V., Solopko I.V. Mathematical modeling of an unmanned object motion control system in SimInTech environment // Spacecrafts & Technologies, 2022, vol. 6, no. 1, pp. 45-54. doi: 10.26732/j.st.2022.1.06

[12] Media and educational materials of SimInTech, URL: https://simintech.ru/science/publications/# (accessed 14.11.2022)

[13] Chuprina S., Ryabinin K., Koznov D., Matkin K. Ontology-Driven Visual Analytics Software Development // Programming and Computer Software. – Pleiades Publishing, Ltd., 2022. – Vol. 48, No. 3. – PP. 208–214. DOI: https://doi.org/10.1134/S0361768822030033.

[14] GDevelop development environment URL: https://gdevelop.io (accessed 30.06.2022).

[15] C. A. R. Hoare, Communicating Sequential Processes, May 18, 2015 URL: http://www.usingcsp.com/cspbook.pdf (accessed 30.06.2022).

[16] Robin Milner, Communicating and Mobile Systems: The Pi Calculus, Cambridge University Press, ISBN 0521643201. 1999. URL: https://en.wikipedia.org/wiki/%CE%A0-calculus (accessed 30.06.2022).

[17] Qgis plugin model, URL: https://plugins.qgis.org/ (accessed 30.06.2022).

[18] Gdevelop plugin model, URL: https://github.com/GDevelopApp/GDevelop-extensions (accessed 30.06.2022).

[19] Vasev P.A., Kumkov S.S., Shmakov E.Yu., On undo-redo subsystem of SharpEye scientific visualization system // In proceedings of 23 International Conference Graphicon'2013, 16–20 sep, 2013. Vladivostok, Russia. PP. 174-177. In Russian. URL: https://m.cv.imm.uran.ru/e/3040965 (accessed 30.06.2022).

[20] Bakhterev M.O, Vasev P.A., One method of visualization of supercomputer modeling results // Proceedings of III National Conference "Supercomputing technologies" (SKT-2014), 29 sep - 4 oct 2014. Taganrog, NII MVS UFU. pp. 50-55. In Russian. URL: https://www.cv.imm.uran.ru/e/3241501 (accessed 30.06.2022).

[21] Brooke J. M., Coveney P. V., Harting J., Jha S., Pickles S. M., Pinning R. L., Porter A. R., "Computational Steering in RealityGrid" // Proceedings of the UK e-Science All Hands Meeting, 2003.

[22] 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), 2020, pp. 49-54, doi: 10.1109/GloSIC50886.2020.9267835. URL: https://www.cv.imm.uran.ru/e/3241753

[23] A.K. Alekseev, A.E. Bondarev , Yu.S. Pyatakova. On the Visualization of Multidimensional Functions using Canonical Decomposition (2022). Scientific Visualization 14.3: 73 - 91, DOI: 10.26583/sv.14.3.06

[24] A.G. Naduev, A.D. Cherevan, A.S. Lebedeva, O.V. Lemyaseva. Universal connection adapter "Logos-Platform" // Abstracts of XVIII International conference "Supercomputing and mathematical modeling", 23-26 may 2022, Sarov, Russia, p. 85. URL: http://book.sarov.ru/product/supercomputing-abstracts-18/ (in Russian) (accessed 14.11.2022).

[25] V.A. Bolnov, A.V. Davydov, M.V. Zotova, R.V. Koslov, A.G. Ezekov. Technology of integration of "Logos", "Bort-T", "Simintech" for digital twins of energy systems // Abstracts of XVIII International conference "Supercomputing and mathematical modeling", 23-26 may 2022, Sarov, Russia, pp. 27-28. URL: http://book.sarov.ru/product/supercomputing-abstracts-18/ (in Russian) (accessed 14.11.2022).

[26] R. Kube, RM Churchill, J. Choi, et al., SciPY 2020 Conference Proceedings, “Leading magnetic fusion energy science into the big-and-fast data lane,” https://doi.org/10.25080/Majora-342d178e-013 (accessed 14.11.2022).

[27] Samarskiy A.A. Mathematical modeling and computational experiment // Bulletin of the Academy of Sciences of the USSR 1979, N 5. Pp. 38--49. (In Russian) URL: http://samarskii.ru/articles/1979/1979-002ocr.pdf (accessed 14.11.2022).

[28] D. Manakov, Data abstraction models: Sampling (parallel coordinates), filtering, clustering (2019). Scientific Visualization 11.1: 139 - 176. In Russian. DOI: 10.26583/sv.11.1.11

[29] A.A. Zakharova, E.V. Vekhter, A.V. Shklyar. Methods of solving problems of data analysis using analytical visual models. Scientific Visualization. 2017. Quarter 4. Volume 9. Number 4. Pp. 78-88. In Russian. DOI: 10.26583/sv.9.4.08

[30] Øksendal, Bernt K. (2003). Stochastic Differential Equations: An Introduction with Applications. Berlin: Springer. ISBN 3-540-04758-1.

[31] Manakov, V. Averbukh. Verification of visualization. Scientific Visualization. 2016. Quarter 1. Volume 8. Number 1. Pp. 58-94 (In Russian). URL: http://sv-journal.org/2016-1/04/ (accessed 14.11.2022)

[32] D. Manakov, V. Averbukh, P. Vasev, Visual text as truth subset of the universal space, Scientific Visualization (in Russian) 8 (2016) 38–49. URL: http://sv-journal.org//2016-4/04.php (accessed 14.11.2022)

[33] Kuznetsov, Numerical modeling of stochastic differential equations and stochastic integrals / Saint-Petersburg, 1999. – 459 p. – ISBN 5-02-024905-X. In Russian. URL: http://physics.gov.az/book_Ch/Kuznetsov.pdf (accessed 14.11.2022)

[34] A.K. Alekseev, A.E. Bondarev , Yu.S. Pyatakova. On the Visualization of Multidimensional Functions using Canonical Decomposition (2022). Scientific Visualization 14.3: 73 - 91, DOI: 10.26583/sv.14.3.06

[35] Pavel Vasev, Sergey Porshnev, Majid Forghani, Dmitry Manakov, Mikhail Bakhterev, Ilya Starodubtsev. An Experience of Using Cinemascience Format for 3D Scientific Visualization (2021). Scientific Visualization 13.4: 127 - 143, DOI: 10.26583/sv.13.4.10

[36] D.V. Manakov. The visual information perception models. International Conference Graphicon 2017, Perm, PSU, Russia. Pp. 129-132. In Russian. URL: https://www.cv.imm.uran.ru/e/3241836 (accessed 15.11.2022).

[37] R. B. Haber and D. A. McNabb. “Visualization ldioms: A Conceptual Model for Scientilic Visualization Systems”. In: Visualization in Scientific Computing (1990), pp. 74–93 (cited on page 12).

[38] Bruder, Valentin. Performance Quantification of Visualization Systems. Dissertation. Visualisierungsinstitut der Universität Stuttgart. 2022. DOI: 10.18419/opus-12005

[39] Visualization pipeline description at URL: https://infovis-wiki.net/wiki/Visualization_Pipeline (accessed 15.11.2022)

[40] V.L. Averbukh. Evolution of Human Computer Interaction (2020). Scientific Visualization 12.5: 130 - 164, DOI: 10.26583/sv.12.5.11

[41] Moreland, Kenneth. A Survey of Visualization Pipelines. IEEE transactions on visualization and computer graphics. 2012. 19. 10.1109/TVCG.2012.133.

[42] M.O. Bakhterev, P.A. Vasev, A builder of specialized visualization system builders // Abstracts of XVIII International conference “Supercomputing and mathematical modeling”, 23-26 may 2022, Sarov, Russia, pp. 22-23. URL: https://www.cv.imm.uran.ru/e/3241824 (in Russian) (accessed 30.06.2022).