Grafix – сетевая технология визуализации

П. А. Васёв
Институт математики и механики им. Н. Н. Красовского
Уральского отделения Российской академии наук,
г. Екатеринбург, Россия

Аннотация. В работе предлагается технология управления визуальными образами, такими как пользовательские интерфейсы и сцены визуализации. Технология базируется на сетевом взаимодействии. Это позволяет реализовать программные компоненты, отвечающие за визуализацию, на одних языках и библиотеках, а общую компоновочную и логическую части на других. Благодаря этому обеспечивается запуска проекта в разных графических средах (например веб и настольные приложения), привлечение более широкого круга разработчиков за счёт использования известных языков, переносимость проекта между разными поколениями графических технологий. В работе представлена программная модель, описана её реализация, приведены примеры использования.

***Ключевые слова*:** визуализация, архитектура программного обеспечения, пользовательские интерфейсы

1. Введение и постановка задачи

Технологии графических интерфейсов [1] и библиотек стремительно меняются. Это означает, что проект, написанный 10 лет назад, становится морально устаревшим. Под моральным устареванием понимается то, что появляются технологии с более логически выверенными программными интерфейсами, с лучшей образностью, с поддержкой современного оборудования, и т.п. Например, на смену когда-то впечатляющей технологии QML пришли новые технологии, такие как React или Kotlin Compose Multiplatform.

Но тоже самое произойдёт и с этими новыми технологиями. Важным аспектом многих подобных технологий является то, что они предлагают сквозное решение, которое охватывает все основные слои пользовательской программы. Это означает, что если программу реализовывать с использованием подобных технологий, то она вскоре морально устареет вместе с этими технологиями. Это является проблемой.

Что делать? Человечество решает этот вопрос, постепенно двигаясь к разделению графических программ на слои и далее пытается реализовывать эти слои на разных технологиях. Такое разделение на слои наблюдается следующее:

1. Графический слой – элементы управления, визуальные образы, т. е. компоненты которые непосредственно взаимодействуют с оконной системой и графическими драйверами.
2. Логический слой – абстракции более высокого уровня, в которых программируются прикладные модели. Этот слой опирается на графический слой, использует его в своих алгоритмах.
3. Компоновочный слой — увязывает элементы графических и логических частей в более крупные абстракции, формирует итоговый вид программы.
4. Дополнительные прикладные части и библиотеки.

Например, следующие технологии опираются на подобное разделение: среда X Window System, парадигма model-view-controller, клиент-серверные веб-технологии и их вариации, такие как Ruby on Rails, и уже упоминавшийся QML.

Имеющиеся решения не идеальны. Многие технологии всё ещё являются сквозными, т. е. предполагают реализацию всех слоёв в рамках одной экосистемы. Другие являются достаточно сложными для программирования взаимодействия слоёв, например веб-технологии подразумевают применение низкоуровневых протоколов (Websocket, Server sent events и т. п.).

Одним из кардинальных методов решения обозначенной проблемы является переход на другой уровень абстракции. Например, в среде SciVi [2] предлагается описывать предметную область в форме онтологии, а затем среда генерирует коды для всех вышеупомянутых слоёв и связи между ними. Это означает, что для смены технологий слоёв достаточно обновить адаптеры, которые генерируют их код, и программа останется всё также технологически свежа.

Другим возможным решением является подход, когда слои разделены явно, и взаимодействуют с помощью сетевых технологий. Это позволяет описывать слои и даже части слоёв на разных языках программирования. А это, в свою очередь, позволяет составлять программы из разных языков и что особенно ценно – обновлять слои

независимо друг от друга. При этом ключевое требование – чтобы взаимодействие слоёв было удобным для программирования.

Этот подход применяется, например, в платформе Логос и описан в работе [3].

Настоящая работа также посвящена исследованию этого подхода. Необходимо отметить, что каждая технология подразумевает некоторую программную модель (сущности, их взаимосвязи и взаимодействия). В работе представлена оригинальная модель, которая по мнению автора решает обозначенную проблему.

2. Модель управления визуальными образами

Сущности модели и их взаимодействие следующие**.**

2.1. Сущности модели

- **Компонента** - это логический процесс в смысле Хоара [4]. Компоненты в контексте настоящей работы подразумеваются графические. Например: кнопка, слайдер, линия, график, трёхмерный объект, контейнер. Над компонентами вводится отношение "родитель-дети"[1].
- **Исполнитель** – процесс операционной системы (ОС), работающий на некотором оборудовании. Компоненты выполняются внутри исполнителей. Например, графические компоненты можно выполнять внутри исполнителей связанных с оконной подсистемой ОС.
- **Канал** — сущность для передачи данных. Вводится операция записи сообщений в канал. Вводится операция подписки (реакции) на сообщения в канале.
- Каналы разделяются на **локальные** и **глобальные**. Каждой компоненте сопоставляется набор входных и выходных локальных каналов, посредством которых компонента взаимодействует с внешним миром. Локальные каналы сопоставляются локальному пространству имён, связанным с компонентой. Глобальные каналы сопоставляются пространству имён, которое является общим (глобальным) для всех участников (исполнителей, компонент).
- **Связь** между двумя каналами — специальный процесс, который производит передачу сообщений, записываемых в первый канал, во второй канал. При этом связь может быть как между глобальными, так и между локальными каналами, в том числе и вперемешку.

2.2. Операции взаимодействия

Список операций для взаимодействия в модели следующий:

Открытие глобального канала:

open_channel: channel_id → channel

где *channel_id* — имя канала в общем для всех исполнителей пространстве имён. Возвращает локальный объект для операций с каналом (локальный в смысле доступный для взаимодействия в том исполнителе и в том языке программирования, в котором выполнена операция *open_channel*).

Создание локального канала:

create_local_channel: void → channel

Создаёт локальный канал и возвращает локальный объект для операций с этим каналом.

Запись сообщения в канал:

put: channel, msg → void

где *channel* — локальный объект для операций с каналом, *msg* — сообщение в некоторой форме. Технически в зависимости от реализации операция *put* может поддерживать передачу дополнительных уведомлений о ходе отправки сообщения, например посредством обещаний (promise).

Создание реакции:

react: channel, fn → reation

---

1 Такая иерархия обусловлена тем, что она применяется в графических пользовательских интерфейсах и сценах визуализации. В интерфейсах иерархию используют для расположения объектов в плоскости экрана. Создают т. н. "контейнеры", например "колонка", "ряд", и рекурсивно размещают объекты в таких контейнерах. В трёхмерных сценах визуализации древовидная структура применяется для формирования матриц преобразования положения объектов.

где *channel* — локальный объект для взаимодействия с каналом, *fn* — функция, которая будет вызываться средой при записи кем-либо следующего значения в канал, *reation* — локальный объект для последующего управления реакцией. При записи сообщения в канал его получают все подписавшиеся стороны, то есть вызываются все реакции. В глобальный канал можно записывать сообщения находясь в одном исполнителе, а получать эти сообщения — в этом же или других исполнителях, что обеспечивает сетевое взаимодействие. Название *реакция* выбрано не случайно, оно соответствует термину "реакция" из модели Lingua Franca [5]. Есть и аналогичные термины, например "подписка на сообщения" в более общей модели "издатель-подписчик".

Создание связи:

create_link: source_channel, target_ channel → local_link_object

где *source_channel* это канал-источник, *target_channel* — канал-приёмник, *local_link_object* — локальный объект связи для последующих операций с ней (удаления). После создания связи все записи в канал-источник приводят к копированию этих сообщений в канал-приёмник. Это относится и к глобальным каналам. После создания связи, если на каком-либо исполнителе на каком-либо компьютере будет произведена запись в канал-источник, то это сообщение будет передано по всем связям, созданным с помощью данной операции, в том числе на другие исполнители на других компьютерах.

Дополнительно к перечисленным выше операциям в модели вводятся и обратные операции, которые в данном тексте пропущены для сокращения описания: закрыть канал, отменить подписку на сообщения, удалить связь, удалить реакцию.

Таким образом, предложена модель для коммуникации разных логических процессов, сгруппированных по исполнителям (процессам ОС) и работающих на разных компьютерах. Эта модель разработана ранее в рамках проекта Parallel Programming Kit (PPK) [8,9], опубликованном на сайте github.com/pavelvasev/ppk.

2.3. Операции по управлению графическим интерфейсом

Для управления графическим интерфейсом предлагается следующий набор операций. Подразумевается, что логический слой программы запрашивает выполнение этих операции, а выполняются они в графическом слое.

Создание дерева компонент:

create: descr, parent_id → void

где *descr* — описание создаваемого компонента, parent_*id* – идентификатор компонента, в дерево которого которого следует добавить создаваемый компонент (если это необходимо).

Описание *descr* рекурсивно и определяет компонент и дерево вложенных в него компонент в рамках иерархии "родитель-дети"[2]. Структура описания:

- *type* — идентификатор типа создаваемого компонента. Исполнитель графического слоя должен уметь создавать компоненты этого типа.
- *id* — идентификатор компоненты (необязательный параметр), в пространстве имён идентификаторов компонент исполнителя графического слоя.
- *params* — параметры компоненты (будут переданы в конструктор компоненты).
- *links_in* — список связей между глобальными каналами и локальными каналами компоненты, то есть это входящая в компоненту информация.
- *links_out* — список связей между локальными каналами компоненты и глобальными каналами, то есть это исходящая из компоненты информация.
- *tags* — список меток, которым сопоставляется компонента. Метка это текстовая строка; наличие меток позволяет в дальнейшем применять массовые операции по отношению к компонентам, что существенно упрощает и оптимизирует программирование [3].
- *items* — список описаний вложенных компонент (описания такой же структуры как *descr*).

Удаление дерева компонент:

remove: id → void

где *id* — идентификатор удаляемого компонента. Вместе с компонентом удаляются и все его вложенные компоненты.

---

2 Древовидность описания descr обусловлена 1) необходимостью работать с иерархиями в принципе, см. предыдущую сноску, и 2) оптимизацией, т. к. типовой пользовательский интерфейс содержит сотни и тысячи компонент, и создавать их поштучно было бы расточительно.

Передача сообщения всем компонентам по метке:

put_by_tag: tag, channel_name, msg → void

где *tag* — метка (строка), *channel_name* – имя канала в локальном пространстве имён компонент, *msg* — сообщение. Операция находит все компоненты, соответствующие метке *tag* в памяти графического слоя, для каждого компонента выбирает локальный канал с именем *channel_name*, и передаёт в него сообщение *msg*.

2.4. Схема работы

Порядок работы подразумевается следующий. Любым удобным образом запускаются процессы уровня ОС – исполнители (на локальной машине или на наборе машин, связанных сетью):

1. Исполнители логического слоя. В простом случае это может быть единственный процесс ОС.
2. Исполнители графического слоя, способные создавать компоненты пользовательского интерфейса и (или) сцен визуализации. В простом случае это может быть также единственный процесс ОС, запускаемый например по инициативе логического слоя (1).

Считается, что исполнители слоёв запускаются асинхронно. После запуска очередного графического исполнителя он посылает сообщение в глобальный канал с именем **gui_attached**.

Код логического слоя заранее подписывается на сообщения в канале *gui_attached*, и таким образом получает информацию о запуске исполнителей графического слоя. Далее логический слой инициирует создание необходимых деревьев графических компонент в подключившемся графическом исполнителе, с помощью указанной выше операции *create*.

При необходимости, логический слой подписывается на сообщения в глобальных каналах, связанных с графическими компонентами для получения сигналов от пользователя. Получая эти сигналы, логический слой посылает сообщения в каналы, связанные со входами графических компонент, создаёт новые компоненты, и так далее. Работа программы идёт сообразно алгоритму, описанному в логическом слое, а её визуальное представление реализуется в графическом слое.

Предложенная схема работы позволяет реализовывать различные конфигурации из исполнителей:

1. В простом случае, как отмечено выше, это может быть один исполнитель логического слоя и один исполнитель графического слоя.
2. Исполнителей графического слоя может быть и несколько, если это необходимо, например в многомониторных конфигурациях, системах типа Cave, в многопользовательской среде виртуальной реальности, и т. п.
3. Исполнителей логического слоя также может быть несколько. Например, можно выделить на каждого пользователя по отдельному исполнителю; исполнители могут работать на узлах суперкомпьютера, если подразумевается создания системы типа «виртуальный испытательный стенд» [12], и т.д.
4. Исполнители графического слоя могут отключаться и подключаться по мере необходимости. Возможна и естественным образом поддерживается многопользовательская работа, но это требует специальной проработки со стороны логического слоя (например если необходимы различные уровни прав пользователей).

Отметим, что исполнители могут быть написаны на разных языках и разных программных платформах. Более того, это считается наиболее перспективным направлением, т. к. одни слои удобно и эффективно выражать на одних языках, а другие на других.

Например, можно использовать исполнитель графического слоя на языке QML, а исполнитель логического слоя – на языке Python. Можно затем заменить графический слой QML на слой реализованный на веб-технологиях. При такой замене, что важно, программа логического слоя, в идеале, останется неизменной. Также можно заменить и программу логического слоя, например, переписав её на другом языке. Можно заменить не весь логический слой, а его часть, например, вынести часть в другой исполнитель или язык.

Необходимо особо отметить роль связей между каналами компонент и глобальными каналами. Такое решение со временем показало своё удобство и эффективность. В ранних версиях модели был другой дизайн: все каналы компонент были глобальными. Для этого приходилось генерировать уникальный идентификатор для каждой компоненты, но главное это вносило ненужные накладные расходы, т. к. система тратит ресурсы на обслуживание каждого глобального канала, а в то же время далеко не все из них задействовались.

Понятие связи между каналами в представленной модели это абстракция высшего уровня, такая же как канал и компонента. Программа является направленным графом из процессов, реализующих работу компонент, ребра в котором — это связи между каналами этих процессов (реализованные посредством связей с глобальными каналами).

Возможна высокоэффективная реализация связей между локальными каналами. Покажем это на следующем примере. Пусть создаётся пара компонент на некотором исполнителе, и два их канала связываются посредством глобального канала. Эту цепочку связей можно технически реализовать таким образом, что запись в канал первой компоненты будет приводить к прямому вызову функций реакции на канале второй компоненты, без сетевого взаимодействия (подробнее см. [8,9]). Это позволяет описывать наборы компонент и связей между ними внешним образом, а их выполнение при этом будет столь же эффективным, как обычное взаимодействие кодов внутри процесса ОС.

3. Библиотека Grafix для языка Python

Предложена библиотека Grafix для языка Python, которая реализует модель, описанную в настоящей работе. Исходные коды опубликованы в сети Интернет по адресу hub.mos.ru/vasev/grafix.

Для операций взаимодействия (раздел 2.2) в Grafix используется библиотека Parallel Programming Kit. Она обеспечивает сетевую коммуникацию программных компонент. Пример работы со средой показан на рис. 1.

```
import ppk
import asyncio

async def main(rapi): # точка входа в программу
    print("started, PPK url is",rapi.url)
    def fn1(msg): # код реакции
       print("see message",msg)
    ch = rapi.channel("channel1") # открываем глобальный канал
    ch.react( fn1 ) # создаём реакцию
    ch.put( "hello world") # отправляем сообщение
    await asyncio.Future() # переход в асинхронный режим

ppk.start( main, url=None ) # запуск среды обмена сообщениями PPK
```

Рис. 1. Пример программы в среде Parallel Programming Kit. Программа запускается, запускает среду, получает управление в функции main, создаёт реакцию на сообщения в канале с именем *channel1*, и отправляет сообщение в этот канал со значением "*hello world*". В результате будет напечатано: "*see message hello world*". Это простой пример, но он интересен тем, что далее любая другая программа может подключиться к этой программе (используя адрес подключения *rapi.url*) и взаимодействовать с ней.

Библиотека Grafix состоит из двух основных частей:

1. Подсистема запуска графического слоя.
2. Набор компонент графического слоя.

На текущий момент предлагается один вид графического слоя, на основе веб-интерфейса. Также проведены эксперименты с графическим слоем в форме Qt/QML, но этот вариант пока не включён в состав библиотеки.

Рассмотрим эти части подробнее.

3.1. Графический слой на основе веб-интерфейса

Алгоритм взаимодействия с графическим слоем следующий. Пользовательская программа вызывает функцию запуска веб-интерфейса:

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

где *rapi* — указатель на API PPK, *plugins_list* — набор плагинов для слоя (см. далее).

Эта функция:

1. Запускает веб-сервер на найденном свободном ip-порту;
2. Генерирует веб-страницу графического интерфейса программы (см. далее);
3. Передаёт ОС команду на открытие веб-страницы (2), что может влечь запуск веб-браузера или использование текущего запущенного веб-браузера.

Далее пользовательская программа ожидает от веб-страницы (2) сообщения в канале *gui_attached* (согласно схеме работы раздела 2.4), и в ответ на это сообщение посылает в графический слой описание интерфейса. Пример вызова функции *grafix.web.start* и каркас приложения, использующего Grafix, показан на рис. 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"] # идентификатор канала управления графическим слоем
       d = { … } # описание интерфейса
       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() # переход в асинхронный режим

# запуск среды обмена сообщениями PPK
ppk.start( main )
```

Рис. 2. Пример программы, использующей Grafix. Программа запускается, запускает среду PPK, запускает графический слой (функцией *grafix.web.start*), получает от него сообщение о запуске и наполняет его описанием интерфейса в формате JSON. Созданный интерфейс будет подключён на верхний уровень окна, поскольку указан родительский элемент *parent_id* со значением "*root*".

3.2. Веб-страница графического интерфейса программы

Эта веб-страница является точкой входа для браузера и отображает интерфейс программы. Для реализации своего функционала веб-страница:

1. Загружает стили CSS и библиотеки скриптов Javascript, необходимые для работы графических компонент. Загрузка скриптов производится как посредством тегов script, так и посредством карт импорта (import maps).
2. Вызывает функции инициализации плагинов.
3. Передаёт управление основному коду веб-страницы.

Основной код веб-страницы:

1. Подключается к среде PPK. Адрес подключения к PPK размещается в коде веб-страницы в момент её генерирования.
2. Генерирует уникальный случайный идентификатор канала, по которому веб-страница будет получать команды Grafix, и посылает этот идентификатор в глобальный канал *gui_attached*.
3. Открывает канал и ожидает команд с операциями по управлению графическим интерфейсом (согласно разделу 2.3). По их поступлению — выполняет их.

Рассмотрим, как реализуется операция *create* по созданию компонент. При поступлении запроса на выполнение этой операции основной код веб-страницы находит и выполняет функцию создания компоненты, которая и создаёт компонент запрошенного типа. Информация о соответствии идентификаторов типов и функций создания компонент хранится в специальном словаре. Этот словарь наполняется плагинами на этапе инициализации.

Функция создания компоненты получает на вход всю информацию из описания *descr*. Кроме создания объекта компоненты, эта функция ответственна и за всю остальную обработку описания (установку параметров, связи и т.п). При необходимости (по решению разработчика), эта функция:

1. Устанавливает значение параметров из поля *params* описания.
2. Связывает локальные каналы компоненты с глобальными каналами (поля *links_in* и *links_out*).
3. Создаёт вложенные компоненты.

После завершения функции создания компоненты основной код веб-страницы получает объект Javascript созданной компоненты. Используя этот объект, код приложения пополняет словари соответствий меток (поле *tags*) и созданных компонент, и словарь идентификаторов компонент. В дальнейшем эта информация используется для удаления компонент и для массовых операций с ними.

3.3. Система плагинов

Под **плагином** подразумевается программный код, который внедряется в систему и меняет её поведение. Отличие плагинов от библиотек заключается в том, что библиотека пассивна: во внешнем коде определяется, когда вызывать функции библиотеки. Плагин напротив, в оговоренный момент времени получает управление, и по оговоренным протоколам меняет поведение системы. В этом смысле он активен: именно в контексте плагина определяется, как он влияет на систему, и как следствие — в какие моменты времени по почину плагина меняется поведение системы.

В библиотеке Grafix для плагинов предоставлены точки управления системой, которые влияют на поведение графического слоя. В графическом слое на основе веб-технологий эти точки следующие:

- *add_head* – возможность дополнить содержимое секции head в html-коде веб-страницы графического интерфейса. Это позволяет плагину, например, добавить необходимые style и script тэги для загрузки библиотек.
- *add_import_map* – возможность дополнить состав карт импорта библиотек (import maps). Это позволяет плагину подключать к приложению библиотеки javascript в т. н. стиле «ES6 imports».
- *add_script* – возможность дополнить код инициализации приложения. В нём плагин например может передать себе управление на уровне Javascript или выполнить какие-либо другие действия.
- *add_routes* – возможность добавить маршруты (routes) в карту маршрутов веб-сервера Grafix, что позволяет плагину обеспечить веб-доступ к своему каталогу на диске при запросах по определённым url-адресам, или добавить свой обработчик веб-запросов по определённым url-адресам, и т. п.

Список плагинов передаётся системе Grafix при вызове функции *grafix.web.start* (см. раздел 3.1). Среди прочих действий, эта функция вызывает функции инициализации плагинов.

Типовой ход работы в системе с точки зрения плагинов получается следующий:

1. Программа запускается и в определённый момент вызываются функции инициализации плагинов.
2. В функции инициализации плагина, плагин встраивает в систему необходимый функционал, используя точки управления системой.
3. Пользователь открывает веб-страницу приложения Grafix.
4. В ходе инициализации веб-страницы загружаются библиотеки, запрошенные плагином через точки управления системой *add_head* и *add_import_map*.
5. Веб-сервер обеспечивает выгрузку файлов этих библиотек, которую он способен реализовать, поскольку плагин на шаге (2) предоставил информацию о маршрутах через точку *add_routes*.
6. При дальнейшей инициализации веб-страницы плагин получает управление (если он запросил его ранее через точку *add_script*), в ходе которого имеет возможность сообщить веб-приложению (т. е. графическому слою Grafix) информацию о типах компонент и функциях их создания.
7. Далее приложение графического слоя ожидает команд от логического слоя, в том числе на создание тех или иных компонент, и выполняет их как описано в разделе 3.2.

3.4. Список встроенных плагинов

Важное достоинство библиотеки Grafix заключается в том, что она предоставляет возможность встраивать любые необходимые типы компонент графического слоя посредством плагинов. Считается, что пользователь библиотеки может сформировать необходимый набор своих плагинов и пользоваться ими в разных проектах.

Для упрощения начала работы с Grafix в её состав включён ряд плагинов, которые пользователь может сразу использовать в своих проектах.

- **grafix.dom –** предоставляет набор типов компонент для реализации пользовательского интерфейса. Среди компонент: text, button, checkbox, slider, numfield, filefield, textfield, combobox, и другие. Также предложены контейнеры для организации пользовательского интерфейса – row, column и grid, которые реализованы на базе технологии CSS Flexbox. Интересной особенностью плагина является метод модификаторов, который используется для управления визуальными аспектами элементов управления[3]. Модификаторы напоминают CSS, а в более явной форме встречаются например в Kotlin Compose и Yandex Divkit. Компонент-модификатор заданного типа ассоциируется с множеством целевых компонент, на которые он влияет, и меняет визуальные аспекты отображения этих компонент согласно своему типу. Таким образом компонентам элементов управления нет необходимости предлагать

3 Классический метод управления визуальными аспектами компонент заключается в установке их свойств (цвет текста, цвет фона, цвет рамки и т. п.). Этот подход неудобный, негибкий, получаются громоздкие компоненты. Например, HTML элементы или элементы QML содержат сотни подобный свойств.

множество свойств по управлению визуальным отображением. Эти свойства по существу «вынесены» в модификаторы, которые пользователь подключает к компоненту по мере необходимости.

- **grafix.dom_screenshot** – реализует возможность получения снимка экрана приложения. Создание такого плагина обусловлено тем, что получение снимков экрана в веб-технологиях в настоящее время это нетривиальная задача, которая требует подключения специальных библиотек.
- **grafix.threejs** – трёхмерная графика на базе библиотеки Three.js (WebGL). Основные компоненты: view (сцена и связанная с ней графическая область вывода, встраивается в контейнеры плагина dom), camera (управление камерой), *lines*, *points*, *mesh* – компоненты вывода графических примитивов. Отображение графики формируется как поддерево с компонентой *view* в корне и компонентами вывода примитивов в листьях дерева. Компоненты вывода примитивов предоставляют каналы для задания координат вершин, цветов и т. д. Логический слой имеет возможность посылать в эти каналы большие массивы данных. Кроме того, введены специальные каналы для частичного дополнения отображаемых данных.
- **grafix.echarts** – рисование графиков на базе библиотеки Apache Echarts. Основные компоненты: *chart* — область вывода графика, *chart_line* — добавляет в chart вывод одного графика, изображаемого линией. В chart можно добавить несколько графиков. Выбор библиотеки Apache Echarts обусловлен тем, что она поддерживает изображение большого количества значений, интерактивно и без задержек. Проверен вывод 1 миллиона значений.
- **grafix.chatbot** – графическая компонента, изображающая чат. Программа может сообщать пользователю полезную информацию в чате в форме сообщений, аналогично как это сделано в чатботах и мессенджерах. Пользователь может посылать текстовые сообщения в этот чат, которые передаются приложению для разбора и реагирования.

4. Результаты экспериментов

Было реализовано несколько прикладных программ, использующих технологию Grafix. Рассмотрим результаты их разработки.

4.1. Специализированная ГИС

В Институте Математики и Механики УрО РАН разрабатываются различные программные алгоритмы по обработке геопространственных данных, для отладки которых создаются специальные графические интерфейсы. Одну из таких работ было решено провести с использованием технологии Grafix.

Для логического слоя был выбран язык Python по причине его распространённости. Ожидалось, что к проекту на Python будет относительно легко подключать дополнительных разработчиков.

Графический слой был реализован на QML и Qt: объекты имевшегося технологического задела были обёрнуты в оболочки и представлены как компоненты модели Grafix. Сущности и операции модели были реализованы в форме объектов C++ с мостами в QML посредством способов Qt (слоты, сигналы). Сетевая часть на C++ была реализована на базе библиотеки Mongoose [4] [5].

Примеры созданных компонент графического слоя (для реализации пользовательского интерфейса) на Qt:

- Ряд (*row*) — компонент в роли контейнера, размещает вложенные компоненты горизонтально.
- Колонка (*column*) — компонент в роли контейнера, размещает вложенные компоненты вертикально.
- Окно (*dialog*) – компонент в роли контейнера, отображает вложенные компоненты в отдельном окне.
- Кнопка (*button*), текстовое поле (*field*), текстовая надпись (*text*), и т. п. – компоненты, реализующие элементы графического интерфейса пользователя.
- Набор картографических слоёв (*map_group*) — компонент для управления отображением картографической информации, взаимодействует с мышкой и клавиатурой пользователя и показывает содержимое вложенных компонент.
- Отображение растровых и векторных карт (*show_image*) — компонент картографического слоя, применяется как вложенный в *map_group*.
- Отображение векторных данных (*show_vector*) — компонент картографического слоя, применяется как вложенный в *map_group*.

---

4 Этот метод реализации оказался очень громоздким. Его мотивация была получить "заодно" реализацию и для C++. Этого удалось достичь, но в совокупности с "мостами" в Qt/QML получился внушительный объем кода. Если делать проект заново, то автор предпочёл бы провести всю реализацию внутри QML, например на базе библиотеки QML Websocket.

5 Библиотека Mongoose оказалась неэффективной. Например, в ней принимаемые и передаваемые данные копируются несколько раз в промежуточных буферах, прежде чем быть переданными программе или ОС.

Логический слой, как отмечено, был реализован на языке Python 3. Слой определяет состав и группировку компонент графического слоя, посылает данные в каналы связанные с этими компонентами, реагирует на сообщения в их выходных каналах (например, движение мышки, клики по картографическим слоям, нажатие на кнопки, и т. п.).

Пример использования кода Python показан на рис. 3, результат работы — рис. 4.

В дальнейшем программа была переключена на вариант графического слоя с применением веб-технологий, и для отображения карт использована библиотека OpenLayers.

```
import ppk
import grafix
import grafix.web
import grafix.dom
import asyncio
async def main(rapi):
    await grafix.web.start(rapi,[grafix.dom])
    ############ формирование описания интерфейса - таблица из 10 полей и кнопки
    lst1 = [ grafix.node("h3",value="Укажите параметры")]
    for x in range(0,10):
        d = grafix.node("row", items=[
                grafix.node( "text",value="Поле "+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="ПОЕХАЛИ!",links_out={"click":["go1","q2"]}) )
    lst1.append( grafix.node("css",value="align-items: center"))
    d = grafix.node("column",items=lst1)
    ### создание интерфейса на графическом исполнителе при его подключении
    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 )
    ############## реакция на нажатие кнопки
    def go(msg):
        print("ЕДЕМ!")
    rapi.channel("go1").react( go )
    await asyncio.Future() # переход в асинхронный режим
# запуск среды обмена сообщениями PPK
ppk.start( main )
```

Рис. 3. Пример кода на языке Python. Код формирует описание графических компонент заголовка, 10 полей и 1 кнопки, выполняет операцию по созданию компонент согласно этому описанию, и добавляет реакцию к выходному каналу нажатия кнопки.

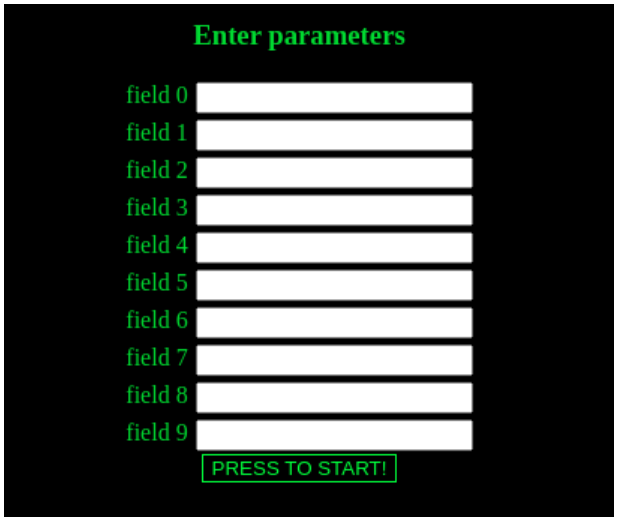

Рис. 4. Результат выполнения кода, приведённого на рис. 3.

4.2. Визуализация параллельного вычисления

Разрабатывалась параллельная версия одного алгоритма обучения с подкреплением. На начальном этапе разработки оказалось, что версия имеет низкий коэффициент распараллеливания. Чтобы понять причины этого, было принято решение визуализировать работу программы. Был разработан вид отображения (см. рис. 5):

- каждая задача (в смысле графа задач, см. [6]) отображается отрезком вдоль оси OX. Положение отрезка на оси X соответствует времени начала и конца выполнения задачи, положение на оси Z соответствует номеру параллельного исполнителя;
- дополнительно изображаются передачи данных между процессами, поскольку передача может занимать значительное время, и это хотелось бы видеть. Параллельная версия использует схему мастер-рабочий, и коммуникация осуществляется только между мастером и рабочими. Поэтому чтобы не захламлять изображение, было решено показывать передачу вертикальными отрезками от плоскости рабочих Y=1 к плоскости Y=0, что означает пересылку данных на вход рабочему от мастера и в обратном направлении.

Графический слой реализован на веб-технологиях, с применением плагинов *grafix.dom* и *grafix.threejs*, представленных в разделе 3.4.

Логический слой был реализован на языке Python 3. Это было обусловлено тем, что обучение (однопоточная и параллельная версии) были разработаны именно на этом языке. В уже имеющуюся параллельную программу были добавлены операции представленной модели для управления трёхмерной сценой.

Вычисления выполняются в рабочих, реализованных в форме исполнителей настоящей модели. При начале вычисления задачи исполнитель замеряет текущее время. По завершению он вычисляет время, затраченное на задачу. Затем исполнитель посылает сообщение в канал «решённые задачи». Специальный модуль логического слоя («рисователь») обрабатывает эти сообщения и формирует координаты вершин для сцены. Затем он пересылает эти координаты посредством сообщений в каналы «модификация данных» компонентам графического слоя, ответственным за рендеринг.

Программа запускается, запускает сетевую среду выполнения, переходит к вычислениям и показывает веб-ссылку для запуска визуализации. Пользователь может в любой момент открыть по этой ссылке визуализацию в браузере, программа по ходу вычислений обновляет графическое представление. Таким образом, была получена визуализация работы параллельного вычисления в режиме онлайн [7], т. е. по ходу вычисления.

Пример полученной визуализации показан на рис. 5. Эта визуализация была использована для понимания работы и последующей оптимизации параллельного алгоритма.

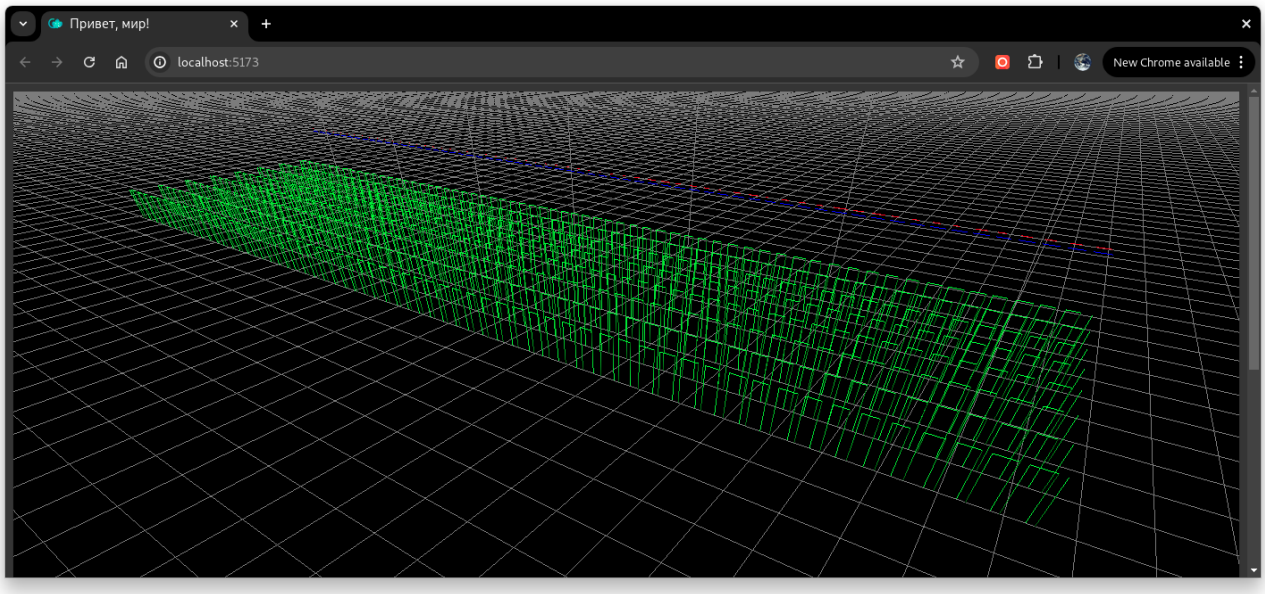

Рис. 5. Пример визуализации, полученной с помощью представленной технологии. Показана работа 8 рабочих процессов. Время идёт слева направо. Рабочие расположены каждый на своей оси Z (ближе-дальше к экрану). Горизонтальная линия на оси рабочего — выполнение задачи. Вертикальная — получение рабочим входных данных от мастера и высылка ему результата. На дальнем плане синие отрезки — ожидание мастером всех рабочих, красные — выполнение мастером оценки текущей ситуации.

4.3. Интерактивная 3D визуализация

Была поставлена задача визуализировать временные сечения максимальных стабильных мостов в линейных дифференциальных играх с фиксированным моментом окончания и выпуклым терминальным целевым множеством [10,11], см. рис. 6. Разработка видов отображения и построение геометрических моделей были реализованы авторами — С. С. Кумковым и А. В. Михайловым. Технология Grafix использована для представления графики и взаимодействия с пользователем.

С технической стороны задача выглядит следующим образом. Данные представлены в форме каталога с подкаталогами. Каждый подкаталог соответствует некоторому моменту времени. В каждом подкаталоге размещён набор файлов 3d-образов в формате PLY, которые соответствуют частям сцены. Необходимо:

1. Иметь возможность задавать момент времени из доступных.
2. Отображать все PLY-файлы, соответствующие этому моменту времени.

Программа визуализации реализована на языке Python, в качестве интерфейса используется веб-браузер. Её исходный код доступен по адресу hub.mos.ru/vasev/2025-bridges.

Общий алгоритм программы следующий:

1. Сканируется каталог входных данных и строится массив времён и соответствующие временам списки файлов PLY.
2. Ожидается сообщение **gui_attached**.
3. По получению этого сообщения формируется графический интерфейс, который включает элементы управления (слайдер, галочка анимации и т. п.). Кроме этого, создаётся сцена трёхмерной графики и для каждого файла PLY из первого момента времени создаётся графический объект Grafix типа *mesh* (набор треугольников), который подключается в сцену. К каждому объекту *mesh* прикрепляется входящая ссылка из глобального канала, что позволит управлять их содержимым в последующем.
4. К выходному каналу слайдера времени прикрепляется исходящая ссылка, ведущая в глобальный канал **current_t**. При смене значения слайдера пользователем, в этот канал посылается сообщение.
5. В основной программе добавляется реакция на сообщения в канале **current_t**, которые определяют выбранный пользователем момент времени. В коде реакции считываются файлы PLY, соответствующие этому моменту времени. Координаты вершин из этих файлов направляются в глобальные каналы (индивидуальные для каждого файла), интерфейс получает уведомления, и визуальные образы принимают соответствующую форму.

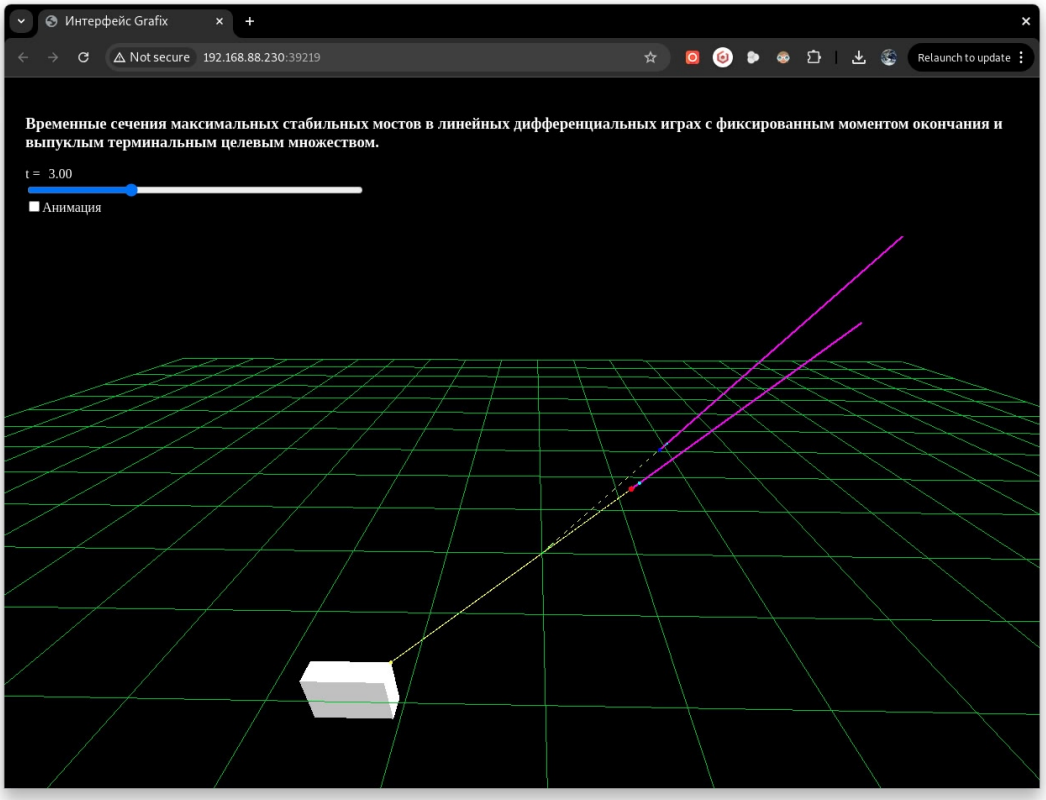

Рис. 6. Пример интерактивной трёхмерной визуализации, полученный с помощью технологии Grafix. Показаны временные сечения максимальных стабильных мостов [10,11]. Пользователь может управлять положением камеры с помощью мышки, и задавать текущий момент времени с помощью слайдера.

5. Обсуждение результатов

5.1. Сравнение с другими системами

Отличие предложенной модели от модели X Window System следующее. В последней компоненты графического слоя как правило выполняются в основном процессе программы, и они отправляют графические примитивы в программы, работающие на графическом оборудовании (на «сервер» в терминологии X Window). В обратном направлении высылаются команды пользователя — сигналы от мыши и клавиатуры. В предложенной модели посылаются не графические примитивы, а высокоуровневые сообщения для компонент графического слоя, которые работают в процессах на графическом оборудовании. В обратном направлении также движутся высокоуровневые сообщения («нажата кнопка», «изменено значение бегунка» и т. п.). Другими словами, модель X Window System выносит в графический слой низкоуровневые графические компоненты.

Отличие предложенной модели от модели Yandex Divkit следующее. В Divkit предложена богатая модель встроенной динамики компонент графического слоя. Это сделано потому, что подразумевается большая автономность этого слоя от логического слоя, подход «загрузи и работай». В модели настоящей работы подразумевается более тесное, постоянное взаимодействие компонент графического и логического слоёв.

В модели настоящей работы заимствован ряд идей из модели Логос [3], среди которых:

– В описании компонент, передаваемом в графический слой, содержится информация о связях локальных каналов компонент с глобальными каналами. Графический слой ответственен за реализацию этих связей. Такой подход несёт ряд преимуществ. Например, он позволяет программировать динамику взаимодействия графических компонент сразу на уровне описания, посредством передачи сообщений между компонентами (опосредовано через глобальные каналы).

– Метки, которые позволяют включать каждую компоненту в несколько множеств. Поддерживаются групповые операции над элементами таких множеств. Например, операция «установить значение *X* в локальный канал *value*» для всех компонент, отмеченных заданной меткой.

Отмечая отличия между моделями и проектами, можно отметить что в Логос:

– предоставлена возможность загружать описание компонент в форме текста на языке JSON из файлов;

– поддерживаются версии графического слоя для веб и настольных платформ, которые реализуют большое количество компонент;

– проект активно применяется на практике, но, к сожалению, он недоступен для сторонних разработчиков.

Среди других проектов, использующих аналогичные модели, можно отметить проект Plotly (Dash), который позволяет строить комплексные виды отображений из базовых 2D и 3D видов отображения (см. рис. 7).

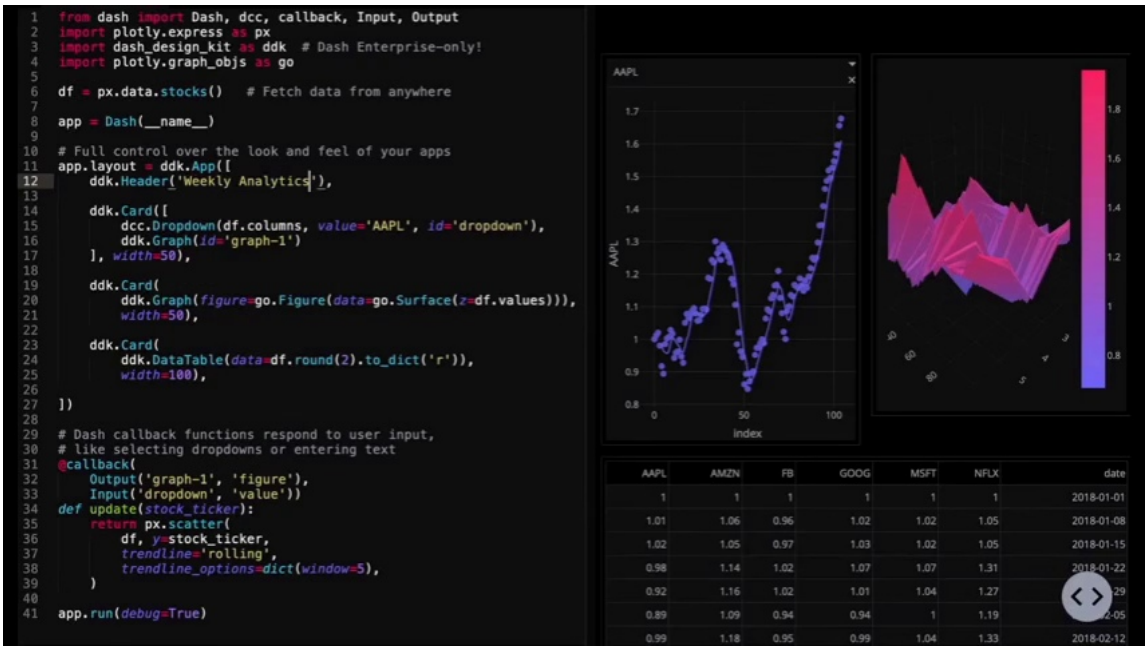

Рис. 7. Пример визуализации с помощью технологии Plotly. Слева показан исходный код приложения, справа — результат выполнения этого кода.

Этот проект также, как и библиотека Grafix, использует связку языка Python и графического интерфейса в веб-среде. Пользователь описывает вид отображения на Python и далее по внутренним протоколам Plotly передаёт эту информацию в веб-интерфейс. Отличия от Grafix заключается в том, что:

– Plotly предлагает богатый набор компонент для визуализации данных (графики, диаграммы, поверхности и т. п.). В составе Grafix существенно меньше встроенных компонент визуализации.

– Plotly больше ориентирован на визуализацию с помощью образов и в меньшей мере на построение интерфейсов из элементов управления. Grafix в этом смысле более универсален, не содержит ярко-выраженной направленности на какие-либо специальные области визуализации.

5.2. Перспективы развития

Среди перспектив развития модели, предложенной в настоящей работе, можно выделить следующие.

Визуализация и анализ связей. При работе с предложенной моделью у разработчика со временем может возникать непонимание картины всех связей глобальных каналов. Это вызвано тем, что модель позволяет вызывать операцию создания связи в произвольных местах на любых языках программирования. Глобальный граф каналов и связей между ними описывается в разных файлах, в разных местах и в разные моменты времени. С одной стороны это даёт полноценную гибкость решения, с другой стороны — теряется общая картина.

Решения этой проблемы на текущий момент не найдено. Среди известных решений:

1. Визуализация картины глобальных каналов и связей, получаемая во время исполнения. Этот вариант, как кажется, не подходит для статического анализа программ.
2. Использование комментариев с машиночитаемой информацией. На практике оказывается, что иногда разработчик забывает внести разметку, и картина при визуализации становится неполной.
3. Известны подходы, когда связи описываются в языке программирования настолько явным образом, что становится возможна их визуализация. Например, язык Lingua Franca [5] позволяет статически анализировать связи компонент и имеет встроенный инструмент визуализации, см. рис. 8.

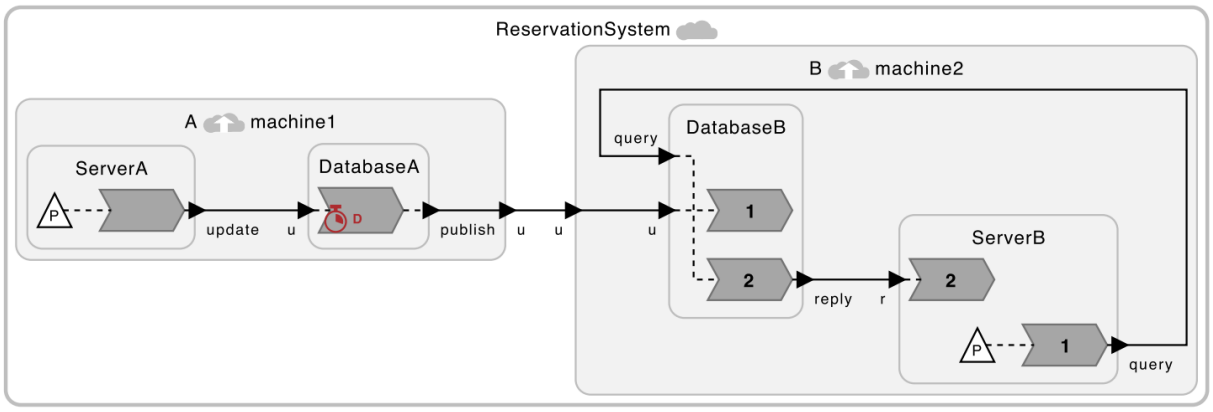

Рис. 8. Пример визуализации связей между компонентами в языке Lingua Franca [5]. Изображения строятся автоматически с помощью инструмента, который анализирует исходный код программы.

Другой пример: в проекте Логос интерфейсы описываются в файлах в формате JSON, что включает и описание связей между локальными каналами компонент и глобальными каналами. Это означает, что возможно автоматически анализировать и визуализировать связи, аналогично Lingua Franca.

Облачная публикация интерфейсов приложений. В проекте Gradio.app содержится следующая интересная идея. Пользователь запускает приложение на своём компьютере. Приложение (а точнее логический слой) при старте связывается с облачным сервисом и публикует графический слой в форме веб-страницы по специальной ссылке. Пользователь получает эту ссылку и может опубликовать её, передать другим пользователям. Эти пользователи подключатся по ссылке и через браузер загружают графический слой, который при этом оказывается связан с приложением, запущенным на компьютере пользователя. Идея выглядит привлекательной, в том числе потому что облачный сервис можно реализовать и в закрытом контуре предприятия. Повышается разнообразие вариантов взаимодействия пользователей, отладки, появляются другие варианты. Например приложение можно запустить на суперкомпьютере, и через облачный сервис получить доступ к его графическому интерфейсу, что упрощает развёртывание онлайн-визуализации [7].

Многопользовательский режим. Предложенная модель позволяет работать с логическим слоем программы нескольким пользователям одновременно. При этом необходимо:

1. Выбрать, на каком уровне синхронизируются экземпляры графических интерфейсов. Это может быть полная синхронизация, включая все элементы управления. В этом случае пользователи обладают общим видом отображения в едином состоянии. Также это может быть частичная синхронизация, когда в общем состоянии содержатся только сущности логического слоя, а состояние вида отображения различаются. Развивая эту мысль, можно прийти к каким-то другим вариантам, например когда пользователи находятся в общей виртуальной среде, ведут себя независимо и наблюдают мир независимо, но могут взаимодействовать, делиться найденными артефактами вычислений, и т.п.

2. При необходимости реализовать идентификацию пользователей. Это потребует ввести в модель логического слоя понятие пользователя, и реализовать какую-либо модель предоставления прав на действия, или ведение истории действий с их привязкой к пользователю. В итоге можно построить полноценную многопользовательскую среду.

6. Выводы и заключение

В настоящей работе предложено вынести интерфейс взаимодействия графического и логического слоёв программ на сетевой уровень. Это позволяет:

1. Не зависеть от развития и устаревания технологий. При устаревании технологии, на которой написан слой, его можно переписать. В первую очередь это касается графического слоя. Также можно переписывать и логический слой, в том числе постепенно. Предложенная технология такова, что обеспечивает возможность работы частей слоя одновременно в разных процессах ОС и на разных языках программирования.

2. Удобно переключать реализации графического слоя, если это ценно для задачи. Например, менять "веб" на "настольную" версию, или варианты с OpenGL на Vulkan.
3. Применять наиболее удобные и подходящие в контексте задачи языки для реализации слоёв. Например, графический слой можно реализовать на веб-технологиях, а логический на Python, что реализовано в представленной библиотеке и использовано в примерах.
4. Запускать графические программы в многомашинной среде "естественным" образом, благодаря сетевому протоколу взаимодействия. Например, графический слой может выполняться на исполнителях на машинах с мониторами, а логический на исполнителях на других машинах, в том числе на узлах суперкомпьютера.

Технология протестирована на ряде прикладных задач, и показала свою работоспособность. Вместе с тем её можно развивать, некоторые варианты развития показаны в разделе "Перспективы развития".

Исходные коды технологии Grafix в форме библиотеки для языка Python опубликованы в сети Интернет по адресу hub.mos.ru/vasev/grafix.

Автор благодарит многоуважаемых коллег, сотрудников Академии наук, Корпорации Росатом, Уральского федерального университета и других организаций – за поддержку, плодотворное обсуждение и синтез идей.

Список литературы

1. В. Л. Авербух, Развитие человеко-компьютерного взаимодействия // Научная визуализация 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. Губайдулина Е. А. и др., Инструмент разработки специализированных пользовательских графических интерфейсов для проведения комплексного математического моделирования средствами пакета программ ЛОГОС // Тезисы XIX Международной конференции «Супервычисления и математическое моделирование», 20 - 24 мая 2024г., г. Саров, стр. 68. URL: www.cv.imm.uran.ru/e/3241932.
4. C. A. R. 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: 10.1145/3448128.
6. Васёв П.А., Визуализация работы алгоритма планирования параллельных задач // Труды 33-ей Международной конференция по компьютерной графике и машинному зрению ГрафиКон 2023, 19-21 сентября 2023 г., город Москва. С. 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. Васёв П. А., Среда параллельного программирования Parallel Programming Kit // Тезисы докладов Национального суперкомпьютерного форума (НСКФ-2024), 26-29 ноября 2024, Институт программных систем им. А.К. Айламазяна РАН, г. Переславль-Залесский. RR-9450, Inria Bordeaux - Sud Ouest. 2022, pp.30. Hal-03547334. URL: www.cv.imm.uran.ru/e/3241946.
10. A. 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. A. V. Mikhailov, S. S. Kumkov, Geometric Procedure for Solving Linear Differential Games with High-Dimensional State Vector // Динамические системы: устойчивость, управление, дифференциальные игры (SCDG2024) : Международная конференция, посвященная 100-летию со дня рождения академика Н.Н. Красовского, 9–13 сентября 2024, Екатеринбург, Россия : труды. Екатеринбург, 2024. С. 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.