SWE621 – HW2

Group:

Venkata Krishna Chaitanya Chirravuri **(G01336659)**

Sai Prashanth Reddy Kethiri **(G01322333)**

Reference System – VisTrails

Additional Systems – Visualization Tool Kit (VTK), matplotlib

1. Querying

VisTrails

The first **abstraction** that we considered is querying. This abstraction **represents** a search bar in the user interface. Another **representation** would be an interface which mirrors the Pipeline view, allowing you to construct a (partial) workflow to serve as the search criteria. The main **purpose** of this abstraction is to quickly search through the collection of workflows.

A **link** to the top-level source file implementing the abstraction in the codebase - https://github.com/VisTrails/VisTrails/tree/9b42ca9b3550b599467b0c7dfa56f57bbd97a4f2/vistrails/gui. The classes and methods written in query_view.py are imported and implemented in vistrail_view.py, hiding the internal functionality from the users.

VisTrails consists of different states while querying. There are two methods to query in VisTrails. First one being Query by Example interface which allows you to build query workflows and search for those with similar structures and parameters.

Key state for this method would be workflow graphs (Version Tree). In the Query by Example interface, translating workflows into a text-based syntax is much more complex and we might lose the structure.

Key operation in this method is to construct the relationship in the workflow that makes use of two modules in sequence. This is done by dragging modules from the list on the left side of the window, connections are added by clicking and dragging from a port on one module to a corresponding port on another module. Setting module parameters in this view will narrow the search to matching modules whose parameters fall within the specified range of values.

Second method for querying in VisTrails is a textual interface with straightforward syntax.

Key state of this method is a simple text box for input. For querying specific information there is syntax that asks for username, annotation, tag, version date. In Textual Queries,

Key operations include searching the query by *entering the text query*, VisTrails will attempt to match the logical categories. If the query is not more specific then, VisTrails has *special syntax to markup the query*. Syntax is described in this link.

Key information stored by both of these methods is the workflow graph (version tree). By making use of these workflow graphs, query would parse through the tree and provide the user with the required data for analysis.

The most **common scenario** where this abstraction is used is, to compare the workflows in VisTrails that make use of the same modules and to see the progress. Using this abstraction, makes it simpler for the developers to establish relationships between the workflows. This provides a flexible and intuitive query interface through which one can explore and re-use provenance information, which would make it **simpler** for the developers to view workflows and for the user to get the data. From the user perspective, we can formulate simple keyword-based and selection queries as well as structured queries to retrieve the data.

VTK

The **closest abstraction** that we found in our additional reference system (VTK) compared to our main system is querying.

A **link** to the top-level source file implementing the abstraction in the codebase - https://github.com/Kitware/VTK/tree/master/GUISupport/QtSQL. In QtSQL, under vtkQtSQLQuery.h all the methods for executing the query are declared, which are

imported into vtkQtSQLQuery.cxx and defined here. This file also imports the required database header file (vtkQtSQLDatabase.h).

The **difference** in abstraction compared to VisTrails is, VisTrails had two methods for querying the data whereas VTK takes a SQL query as an input for vtkSQLQuery class. So, the **state** for the abstraction implemented in this system is a SQL Query.

Key Operations that are **different** from VisTrails are, VTK performs operations like:

- *Execute()* - to execute a query which implements vtkRowQuery
- *GetNumberOfFields()* - returns number of fields in query results
- *GetFieldName()* - The name of the field at an index.
- *GetFieldType()* - The data type of the field at an index.
- *NextRow()* - Advances the query results by one row, and returns whether there are more rows left in the query.
- *DataValue()* - Extract a single data value from the current row.
- *Begin/Rollback/CommitTransaction()* - These methods are optional but recommended if the database supports transactions.

Advantages of this abstraction in VTK is that they are efficient, portable, multi-data views. I see there are **no cons** of having this abstraction, since it provides required data in an efficient way and does the work as the user specified. By this abstraction, in complex scenarios like examining 3D data would be **easier** by querying multiple data and comparing them. Consider a project with multiple files and getting required data from those files to extract the data for analysis is much more **complex**, because writing a query would be troublesome.

Matplotlib

The **closest abstraction** Matplotlib and the pyplot interface has is querying, updating and modifying the attributes of the plots and figures using in addition to python code. Matplotlib uses many underlying abstractions to retrieve, update and create the object properties, and as well as to do introspection on the object. In particular matplotlib.artist supports various line-styles and graph-plot styles for the user to create new sub-plots, retrieve the attribute properties and add styling to the plots. The matplotlib.artist uses the Artist as the object and typically the Artist handles all the high level constructs like representing and laying out the figure, text and lines.

A link to the top-level source file implementing the abstraction in the codebase- https://github.com/matplotlib/matplotlib/tree/main/lib/matplotlib . The artist.py contains all the high level constructs for plotting and the studies also suggest that an user spends 95% of an average time using the Artist.

The **key operations** of this abstraction are ***primitives*** and ***containers***. The standard use is to create a Figure instance, use the figure to create one or more Subplot instances, and use the Axes instance helper functions to create the primitives

The **key states** for primitives are standard graphical objects we want to paint onto the canvas like Line2D, Rectangle, Text etc where the helper methods/functions take the data in numpy arrays and strings. The containers are the places to place them like Axis, Axes and Figure where the data would be list of array of the axes dimensions like width, height, left, bottom

Though the main intuition of this abstraction querying of underlying data lies common among our main system i.e.VisTrails and our other additional reference system Visualization Tool Kit (VTK) , they **differ** in the implementation and the level of implementations. In Matplotlib it mainly focuses on the creation, updation and retrieving of the object parameters and minute details of the graphs and subplots, where in VisTrails we can implement these for the entire workflows and their comparisons as well.

Pros for this abstraction in Matplotlib is that it gives enough emphasis on the figures and plots subject to the given object parameters for the user to analyze. And the major **drawback** is that , VisTrails offer more user friendly ways of creating ,comparing the methods as using Matplotlib requires minimum prior knowledge of coding in python.

2. Packages

VisTrails

VisTrails provides a plugin infrastructure to integrate user-defined functions and libraries. Specifically, users can incorporate their own visualization and simulation code into pipelines by defining custom modules (or wrappers). These modules are bundled in what we call **packages**. A VisTrails package is simply a collection of Python classes stored in one or more files, respecting some conventions that will be described shortly. Each of these classes will **represent** a new module. The main **purpose** of a package is to allow users to include their own or third-party modules in VisTrails workflows.

A **link** to the top-level source file implementing the abstraction in the codebase- https://github.com/VisTrails/VisTrails/tree/v2.2/vistrails/packages .

The most important piece of metadata **(Key Information)**, however, is the package *identifier*, stored in the variable called identifier. This is a string that must be globally unique across all packages, not only in your system, but in any possible system. The

reason for this unique id is, they should not result in any kind of conflicts while importing the packages.

The **key operations** here are defined as per the user requirements as it gives the user enough flexibility to *customize their functionalities*. And use of other packages like *VTK, matplotlib, sklearn, mongodb, sql* are also made possible per the requirement for scientific computation and visualization.

Using matplotlib package enables rich plotting styles, using sklearn gives them using the statistical models for their computations, sql and mongodb extends the usability of connecting to the database.

The ***key states*** here are solely dependent on the *key operation*s defined in the above step and would be object initiated as per the usage and requirements.

(Scenarios making complex work simpler) We find the usage of this abstraction very often in designing the workflows, computing the scientific visualizations and the area of using recommender systems where VisTrails suggest the users based on their previous workflows.

VTK

VTK also provides similar abstraction of packaging culture and usage of wrappers of classes into other languages. This also enables VTK to extend and upgrade their versions accordingly.

A link to the top-level source file implementing the abstraction in the codebase-https://github.com/Kitware/VTK/tree/master/Wrapping/Tools

The difference in abstraction that VTK provides is that it enables the wrapper classes bundled to be used in other languages like Java and Python as well. The wrapper tools consist of executables that pull information from C++ header files, and produce wrapper code that allows the C++ interfaces to be used from other programming languages like Java and Python.

We are stating few of the key operations of the wrappers:

- *Vtkparser* - The header vtkParse.h provides a C API for the C++ parser that wrappers use to read the VTK header files.
- *vtkParsePreprocess* - This is a preprocessor that can run independently of the parser.

- *vtkParseString* - This provides low-level string handling routines that are used by the parser and the preprocessor. Most importantly, it contains a C++ tokenizer. It also contains a cache for storing strings (type names, etc.) that are encountered during the parse.
- *vtkParseExtras* - This file provides routines for managing certain abstractions of the data that is produced by the parser. Most specifically, it provides facilities for expanding typedefs and for instantiating templates
- *vtkParseMerge* - This provides methods for dealing with method resolution order. It defines a data structure for managing a class along with all the classes it derives from. It is needed for managing tricky details relating to inheritance, such as "using" declarations, overrides, virtual methods, etc.
- *vtkParseHierarchy* - A hierarchy file is a text file that lists information about all the types defined in a VTK module. The wrappers use these files to look up types from names. Through the use of vtkParseHierarchy, the wrappers can get detailed information about a type even if the header file only contains a forward reference, as long as the type is defined somewhere in another header.
- *vtkWrapJava* - produces C++ wrapper code that uses the JNI
- *vtkParseJava* - produces Java code that sits on top of the C++ code
- *vtkWrapPythonClass* - creates type objects for vtkObjectBase classes
- *vtkWrapPythonType* - creates type objects for other wrapped classes
- *vtkWrapPythonMethod* - for calling C++ methods from Python
- *vtkWrapPythonOverload* - maps a Python method to multiple C++ overloads
- *vtkWrapPythonMethodDef* - generates the method tables for wrapped classes
- *vtkWrapPythonTemplate* - for wrapping of C++ class templates
- *vtkWrapPythonNamespace* - for wrapping namespaces
- *vtkWrapPythonEnum* - creates type objects for enum types
- *vtkWrapPythonConstant* - adds C++ constants to Python classes, namespaces

The key states here for the key operations differs on the usage of the key operations like vtkParseString, vtkParseHierarchy, vtkWrapPythonNamespace uses states like string, vtkWrapPythonEnum, vtkWrapPythonType, vtkWrapPythonClass mainly use object instances and the others use wrapping templates, and methods.

This abstraction is similar to the VisTrails but this gives us cross-language usability of modules and libraries when needed with a simple use of wrapper functions underlying in other languages. We feel the most common scenario is the use of the *numpy module* from Python libraries which is extensively used for numerical computations of scientific data.

The major advantages of this wrapping produces cross-language usability and scalability for the user and to use the functionalities as per their requirements. And the major drawback could be the dependencies. One might not get the full usability of the libraries as they do in their native languages

Matplotlib

Matplotlib also offers the same abstraction, matplotlib itself is the comprehensive packaged library for creating and interactive visualizations. Matplotlib has its numerical mathematics extension NumPy. It provides an object-oriented API for embedding plots into applications using general-purpose GUI toolkits like Tkinter, wxPython, Qt and GTK.

A link to the top-level source file implementing the abstraction in the codebase- https://github.com/matplotlib/matplotlib/tree/main/tutorials/toolkits

Even Matplotlib has a similar abstraction as VisTrails in packages. Matplotlib exclusively uses a few toolkits axes_grid.py , axisartist.py , mplot3d.py designed to extend functionality of Matplotlib in order to accomplish specific goals. Matplotlib also has extensive text support, including support for mathematical expressions, truetype support for raster and vector outputs, newline separated text with arbitrary rotations and Unicode support.

The key operations are

- *Annotations* - A common use case of text is to annotate some feature of the plot and locations in the plot.
- *Mathtext* - One can use a subset of TeX markup in any Matplotlib text string by placing it inside a pair of dollar signs.
- *Pgf* - Using pgf backend Matplotlib can export figures as pgf drawing commands that can be processed with pdfLaTeX, xeLaTeX or luaLaTeX. XeLaTeX and LuaLaTeX have full Unicode support and can use any font that is installed in the OS, making use of advanced typographic features of OpenType, AAT and Graphite.

- *Mplot3d* - This particularly extends functionality in plotting various 3D plots like Line Plot, Scatter Plots, Wireframe plots, Surface plots, Tri-Surface plots, Contour plots, Filled Contour plots, Polygon plots, Bar plots and Quiver.
- *Axesartist* - This module contains a custom Axes class that is meant to support curvilinear grids.

The state of Annotations are the coordinate system , mathtext uses text/string, pgf uses text and axes, Mplot3d uses the axes and an extra argument *'projection=3d'*.

This is one of the commonly followed methods while using matplotlib. Python is well-known for its packaging structure and matplotlib being a library itself this gives matplotlib bi-linear way to be integrated into GUI using Tkinter, Qt, wxPython and it matplotlib itself can be used and the visualizations can be produced.

3. Dataflow Evolution

VisTrails

The third **abstraction** that we considered is about creating and updating workflows. In VisTrails this can be done by making use of VisTrail Builder GUI. This abstraction **represents** a separate window on the screen with the tools **responsible** for creating a dataflow. You can also retrieve existing workflows to examine or update them. The main purpose of the toolbar in VisTrail Builder in the window is to execute the current workflow or function and to switch between various modes.

A **link** to the top-level source file implementing the abstraction in the codebase - https://github.com/VisTrails/VisTrails/tree/9b42ca9b3550b599467b0c7dfa56f57bbd97a4f2/vistrails/core

Key state of this abstraction is a formal specification of the pipeline or workflow. **Key operations** performed by this abstraction are,

- **Pipeline:** This view shows the current workflow.
- **History:** This view shows different versions of the workflow(s) as it has progressed over time.
- **Search:** Use this mode to search for modules or sub pipelines within the current version, the current vistrail, or all vistrails.

- **Explore:** This option allows you to select one or more parameter(s) for which a set of values is created. The workflow is then executed once for each value in the set and displayed in the spreadsheet for comparison purposes.
- **Provenance:** The Provenance mode shows the user a given vistrail's execution history. When a particular execution is selected, its pipeline view with modules colored according to its associated execution result is shown.
- **Mashup:** The Mashup mode allows you to create a small application that allows you to explore different values for a selected set of parameters.
- **Execute:** Execute will either execute the current pipeline when the Pipeline, History, or Provenance views are selected, or perform the search or exploration when in Search or Exploration mode. This button is disabled for Mashup mode, or when there is not a current workflow to execute.
- The New, Open, and Save buttons will create, open, and save a vistrail, as expected.

Key information for a vistrail builder is a file consisting the dataflow information, **(use of key info)** which is then followed by a series of VTK steps to read the file, compute an isosurface and render the resulting image in spreadsheet.

Consider a **scenario,** where a different team needs a different view to a dataset. This can be achieved with the help of VisTrail Builder that consists of different views like provenance, mashup, pipeline, history. This abstraction makes such complex scenarios **simpler** by providing multi-data views so that we can compare the workflows.

VTK

There is **no closest abstraction** like a GUI in Vistrails in VTK to create and update workflows. But VTK provides a **programming interface** for creating a new pipeline.

A **link** to the top-level source file implementing the abstraction in the codebase - https://github.com/Kitware/VTK/tree/d706250a1422ae1e7ece0fa09a510186769a5fec/Common/ExecutionModel

(Key State) For programming a new pipeline, it requires a 4 key classes,

- **vtkInformation** is a map-based data structure that supports heterogeneous key-value operations with compile time type checking.

- **vtkDataObject** is supposed to store only data. It has an instance of vtkInformation that can be used to store key-value pairs in.
- **vtkAlgorithm** has a vtkInformation instance that describes the properties of the algorithm and it has information objects that describe its input and output port characteristics.
- **vtkExecutive** contains the logic of how to connect and execute a pipeline.

Key Operations of this abstraction include creating a new pipeline by making use of the classes mentioned above. It is also possible to update an already existing pipeline using vtkDataObject::Update() was a convenience method that in turn called Update() on the algorithm that produced the given data object.

(Difference) VisTrails provides a GUI to create and update workflows, whereas in case of VTK we are provided with a programmable interface. **(Pros)** Having programmable access gives more handle over setting the parameters for the pipeline, whereas in case of GUI we are confined with some strict parameters. **(Cons)** From the user perspective, a person with no prior programming knowledge could not create the pipeline. In such cases having a GUI would be much more friendly. So having a programmable interface could make complex scenarios even **harder.**

Matplotlib

The similar abstraction found in Matplotlib was the pipeline dataflow, this pipeline describes how the raw data is converted into the lines we see on the screen. From the earlier versions of Matplotlib, the system has been refactored, so the steps are discrete in the path conversion pipeline. And this enables the user to choose which parts of the pipeline to perform.

A link to the top-level source file implementing the abstraction in the codebase - https://github.com/matplotlib/matplotlib/blob/main/lib/matplotlib/transforms.py

The **key operations** are Transformation, handling missing data, Clipping.

- ·*Transformation* – The **key state** are the coordinates, and the coordinates are transformed from data coordinates to figure coordinates. If it does not involve any of the arbitrary transformations, transformation functions are called to transform coordinates into figure space or else it is just a simple matrix multiplication.
- *Handle missing data* – The **key state** here is the data array; the data array might have portions where the data is missing or invalid. The user may indicate this

either by setting those vales to ‘null or using any masked library arrays. Vector output formats, such as PDF and rendering libraries such as Agg do not often have a concept of missing data when plotting a polyline. Using MoveTo commands, the step of pipeline will skip over the missing data segments

- *Clipping* - The **key state** here is also the coordinates, the points outside of the boundaries of the figure can increase the file size by including many invisible points. Very large or very small coordinate values can cause overflow errors in the rendering of the output file, which could result in a complete garbled output. This step in the pipeline flow clips the polyline as it enters and exists the edges of the figure to prevent both the problems.

This is the closest abstraction we found in Matplotlib, using matplotlib alone as a package the only case would be to modify the dataflow pipeline itself, but when the matplotlib embedded into GUI using Tkinter, wxPython we could further enhance the comparison of data workflow pipelines with additional python code.

The pros are minimal when considered matplotlib itself, as it does not give enough flexibility to modify or update the workflows as VisTrails offer. We feel VisTrails has many extensions in this area for comparison of workflows, updating them and storing them and re-producing them for later use. VisTrails also enables the feature of recommending the user based on the stored workflows.

The other basic abstraction we felt was the usage of database, and the type of database connection. VisTrails supports relational databases and object oriented databases, VTK also supports the database implementation in the similar way, but where matplotlib has to be run over the database using completely external plugins.

References:

- http://aosabook.org/en/matplotlib.html
- http://aosabook.org/en/vistrails.html
- http://aosabook.org/en/vtk.html
- https://github.com/matplotlib
- https://github.com/VisTrails/VisTrails
- https://github.com/Kitware/VTK
- https://www.andrew.cmu.edu/user/jiaz/Papers/JiaZhang-VisTrails-HECC.pdf
- https://arxiv.org/abs/1309.1784

- https://www.vistrails.org/usersguide/v2.1/html/querying.html
- https://www.vistrails.org/usersguide/v2.0/html/VisTrails.pdf
- https://www.vistrails.org/images/Pc2.pdf
- https://vgc.poly.edu/~juliana/pub/vistrails-sigmod2008.pdf
- https://fd.kuaero.kyoto-u.ac.jp/sites/default/files/VTKUsersGuide.pdf
- https://vtk.org/doc/nightly/html/classvtkSQLQuery.html#details
- https://www.vistrails.org/usersguide/dev/html/packages.html
- https://www.vistrails.org/usersguide/v2.1/html/packages.html
- https://www.vistrails.org/usersguide/v2.2/html/VisTrails.pdf
- http://www.sci.utah.edu/~vgc/vistrails/pub/vistrails-tutorial.pdf
- https://vtk.org/Wiki/VTK/Tutorials/New_Pipeline
- https://vtk.org/Wiki/VTK/VTK_6_Migration/Removal_of_Update