NexusVIS: A Distributed Visualization Toolkit for Mobile Applications

Carlos Lübbe*, Andreas Brodt*, Nazario Cipriani*, Harald Sanftmann†
*IPVS / †VIS, Universität Stuttgart, Germany
Email:{luebbeck, brodtas, cipriano}@ipvs.uni-stuttgart.de*, sanftmann@vis.uni-stuttgart.de†

Abstract—Many mobile pervasive applications need to visualize information about the user’s geographic surroundings combined with data from sensors, which determine the user’s context. We demonstrate NexusVIS, a distributed visualization toolkit for mobile applications. By building upon an existing data stream processing system we enable applications to define distributed visualization processes as continuous queries. This allows applications to define visualization semantics descriptively. Moreover, NexusVIS is capable of adapting the visual query at runtime, and thus allows to navigate in the visualized scene both automatically and manually through user control.

I. INTRODUCTION

Many mobile pervasive applications visualize information about the geographic surroundings depending on the user’s current context. For this purpose they leverage sensors that continuously sample the environment. A simple example is a live view of moving taxis surrounding the user’s current position superimposed on a map of the city. Each application visualizes information differently, e.g. a bus is not shown in the same way as a taxi. Nevertheless, visualizations in such a scenario share many similarities, for instance, both busses and taxis are visualized on a city map. For this reason, a common visualization toolkit for mobile applications considering pervasive sensor information is desirable.

Similar to [1], we conceptually divide the visualization process into four steps: select, map, render and display (see Figure 1). The select step extracts relevant data. Then, the map step creates abstract visual representations, such as geometries or text. The render step transforms them into a pixel-based image format. Finally, the display step presents the resulting stream of images to the user. The user can control the visualization using direct feedback links to each step. To automatically control the visualization according to some input data, e.g. when following a moving object, the control data flows over forward links to each step.

To carry out the visualization process, a common toolkit for mobile applications must satisfy four requirements:

1) The select step requires sophisticated data management and data stream processing capabilities, as it includes both static data and data streams (e.g. map data and position updates, respectively).

2) Applications need to include domain-specific visual models and mapping semantics. E.g., our taxi-application maps position updates to the visual model of a cab, whereas the bus-application uses a bus model.

3) User-driven or automatic control requires the steps to be adjustable at runtime.

4) Pervasive environments require a flexible processing architecture, which allows to distribute processing amongst heterogeneous devices. Resource-constrained devices, e.g., may require server-sided processing of computationally intensive tasks (such as rendering), while more powerful clients may perform them locally.

Both grid-based visualization tools [2], [3] and operator-based data stream processing systems [4], [5] meet requirement 4). We can reuse this functionality when implementing a visualization toolkit for mobile applications. However, grid-based visualization tools do not consider requirement 1) which is particularly important for mobile applications. As operator-based stream processing systems fulfill requirement 1), it seems attractive to add visualization functionality to an existing system. Due to its extensibility, we chose NexusDS [5] for this purpose.

We demonstrate NexusVIS, a distributed visualization toolkit for mobile applications. It enables applications to define visualization processes as continuous queries and thus combines data stream processing and visualization. To our knowledge, this is a novel feature. We address requirements 1) and 4) by building upon NexusDS [5], which manages and processes stream-based data, and flexibly distributes processing amongst heterogeneous devices. To meet requirement 2), we introduce a common data model for both spatial and visual data. This simplifies the definition of mapping semantics and enables applications to include domain-specific visual models. We meet requirement 3) by introducing dedicated command messages to modify the visualization process at runtime and thus support both user-driven and automatic control. We demonstrate the generality and usability of our approach by various visual queries, each of which focuses on a different aspect of our requirements.
A. NexusDS

NexusVIS performs the entire visualization process inside NexusDS [5], an extensible middleware for distributed stream processing. The basic entities are operators which encapsulate processing semantics and have an arbitrary number of inputs and outputs. We call an operator without inputs source and one without outputs sink, respectively. The outputs can be connected to inputs to form more complex query semantics. As shown in Figure 2, an application sends a query to NexusDS, which distributes participating operators to different nodes. The processing starts as soon as sources push data through the processing pipeline.

B. Data Model and Type Hierarchy

The visualization process incorporates three kinds of data: spatial, visual and pixel data. For pixel data, we define a binary format, carrying the images created by the render step as frames. For spatial and visual data we use an object-oriented data model, which we describe in the following.

We follow the object-oriented data model of Nexus [6]. It defines an object as a set of attribute instances. An attribute instance is a tuple consisting of an attribute name and a value. We denote an attribute instance of an object \( O \) as \( O.a \), where \( a \) corresponds to the name of the attribute instance.

An obligatory attribute instance (type), carries type information. As shown in Figure 3, NexusVIS models three major object types: DataObject, VisualObject and Command.

The DataObject models spatial data, which we further classify into static data, represented by the subtypes of StaticObject, and dynamic sensor data, modeled by the Sensor subtypes.

The VisualObject models visual scenes by means of an external modeling language, such as X3D [7]. It carries the entire visual model expressed in the respective modelling language in the model attribute.

Command instances modify the state of a data stream operator. This includes objects that are cached inside the operator and the runtime parameters of the operator. Operators may keep a local copy of large objects that are frequently needed and rarely changed. A moving object, for instance, changes its position, but keeps its visual model, which is potentially large. It is not reasonable to retransmit and reprocess the visual model whenever the object moves. Thus, a method to notify the operator of the change is required, so that the operator updates its cache accordingly. The same holds when a new moving object appears or disappears. Insert, Update and Delete objects signal the operator to insert a new object into its cache, to update, or to delete a cached object. To modify the parameters of an operator at runtime, a preceding operator sends a SetParam command containing new parameter values. This allows us to control the visualization process at runtime.

C. Data Integration

The visualization process integrates static and dynamic data. NexusVIS offers dedicated sources for both.

The static source stores rarely updated data. It contains instances of any subtype of DataObject. NexusVIS uses a static source for two purposes. First, it provides access to all static data of the geographic surroundings, such as buildings or streets. Second, it contains Sensor objects, each representing a physical sensor device. They describe device properties, such as expected data rate, or the real-world position and may also contain approximate sensor values. Applications use this information to pick the sensors of interest and include them in their continuous queries.

A sensor source manages exactly one physical sensor device. At the start, it registers a Sensor object at the static source and may then occasionally update the approximate sensor values. Applications may include sensor sources into their continuous queries at which point the sensor sources start to send a continuous stream of sensor updates to succeeding operators.

D. Operators

To run a visualization process, mobile applications use the stream operators provided by NexusVIS. In the following we outline some of them.

The selection operator \( \sigma_p \) extracts all objects from the static source that fulfill predicate \( p \). Correspondingly, the stream-based version of the selection \( f_p \) forwards all objects which fulfill predicate \( p \). The extended projection operator \( \pi_M \) maps incoming objects \( O_{in} \) to outgoing objects \( O_{out} \) by applying a set of mapping rules \( M \) to each attribute of the incoming object. Thereby, it may either just copy the attribute value or apply a primitive arithmetic function \((+,-,\)
Usually, this operator transforms DataObject instances to VisualObject instances. The render operator \( r \) inserts these instances into a virtual 3D scene. Two parameters \( v_pos \) and \( v_dir \), which correspond to the position and the direction of a camera, define the viewport of the scene. The operator renders the scene and thus creates a stream of images. Finally, the display operator \( d \) presents them to the user. The operator also integrates a GUI to control the visualization process, as shown in Figure 4.

III. DEMONSTRATION

In our demo-setting we use two different nodes to execute the visualization process: a standard laptop and an ultra mobile PC. The former is equipped with an 3D graphics chip, whereas the latter has no hardware-accelerated 3D support. Since our implementation of the render operator needs to have a 3D graphics chip to render in real-time, only two distribution scenarios are reasonable. First, the laptop executes the whole visualization including the display operator, and second, the ultra mobile executes the display operator whereas the laptop executes the rest. In those scenarios we run the visualization queries depicted in Figure 5.

The first query in Figure 5(a) visualizes a static object. It extracts the GPS instance with the id \( A \) and maps it to the visual model \( Y \), e.g., a car. The position of the model is copied from the position attribute of the GPS instance. Afterwards, it is rendered and displayed. In the same manner we map buildings or streets surrounding the user to a visual representation.

The preceding example uses the position attribute stored in the static source, which might not be up-to-date. In Figure 5(b) we feed a continuous data stream of position values \( p_i \) from the associated sensor source into the render operator which updates the position of the object in the 3D scene. Thus, we visualize the current position of the object.

As yet, the viewport of the 3D scene does not change during runtime. In our demonstration, the user can manually control the viewport by interacting with the GUI of the display operator. Figure 5(c) depicts the corresponding query. The display operator puts the user-defined values of the viewport \( p_u \) and \( d_u \) into SetParam commands and sends them to the render operator, which updates the \( v_pos \) and \( v_dir \) parameters accordingly.

In the query of Figure 5(d), the view automatically follows a moving object. The user selects the object through the GUI of the display operator which sends SetParam commands to the stream-based selection \( f \) to change the current predicate. In Figure 5(d), the user selects object \( B \). Thus, \( f \) only forwards position updates of the object chosen by the user. For each position \( p_i \), the succeeding extended projection \( \pi \) creates a SetParam command which sets the \( v_pos \) parameter to \( p_i + c \), where \( c \) is a constant offset representing the distance between the camera and the moving object. Thus, we move the camera according to the object the user wants to follow. In addition, we cause the camera to center the moving object by setting the \( v_dir \) parameter to the inverse of \( c \).

REFERENCES


